CS 428 - Fall 2008
Project 2: Polygon Meshes and Shading

Due electronically: Wednesday, October 29, 11:59 pm

Check this page frequently for updates and clarifications
(Significant changes or clarifications are marked with [Update])

Description

In computer graphics, surfaces are often represented with collections of polygons. There is additional structure as well; while the vertices of the mesh represent the geometry of the surface, polygon faces *share* vertices, so they not only describe where the surface is among the vertices, but also its connectivity.

Objective

This project will help you understand how to represent, create, manipulate, and render polygon meshes (in OpenGL and GLSL) in a variety of styles.

Program

You will be provided with skeleton code for this program, which supplies the necessary user interface and main program structure. A simple polygon mesh data structure is provided, along with some code in GLSL. Code which reads a polygon mesh from a file is also provided. You will be filling in the missing parts in functions (which are marked with "`// ...`"), and are listed in the `README.txt` file. You must develop the code for computing normals of the polygon mesh, drawing the mesh in various styles, and evaluating tessellated objects (an ellipsoid), in both OpenGL and GLSL.

The skeleton code for this project can be found (on the cereal machines) in the directory

```
~decarlo/428/proj2
```

Also included is a `Makefile` and a `README.txt` file describing the code structure and how to compile and run the program.

There are some example polygon mesh files in:

```
~decarlo/428/obj/
```

Some of these files are big, so watch your quota (use `quota -v`). You should probably just make a symbolic link to the objects directory contained inside your code directory, after you copy the code:

```
cp -r ~decarlo/428/proj2 .
```

Make the link like this:

```
ln -s ~decarlo/428/obj proj2/obj
```

In addition to saving your disk space, should more objects be added into that directory, you won't miss them.

GLSL

GLSL is a lot like C (well, the types are a little stronger). We'll be using just the basic features, so don't worry. The GLSL programs you'll be editing are in files that end in `.vp` for the vertex programs, and `.fp` for the fragment programs. Refer to the following GLSL materials:

- GLSL Wikipedia entry
- A GLSL tutorial at Lighthouse 3D
- The full spec, if you want all the details...

You don't have to compile the program youself; it gets compiled at run-time, and only when you use it. (The main program is designed to work fine on a machine that doesn't support GLSL, as long as you don't turn it on.) The compile-time error messages are helpful for things like syntax errors, type-checking failures, etc... Run-time errors are more problematic, as the entire program aborts. It's a good idea to make small changes to these programs, and then test them, until you feel comfortable with GLSL.

One kind of run-time error you might encounter (that causes the program to exit) happens if your GLSL program is too long. Lengths vary by machine. Make sure your code isn't drawn out -- keep it compact. You don't need to overdo it; in our own development of this project, we didn't encounter this problem. We will soon decide on a room where you should make sure your code runs (it will probably be Hill 248, since those machines are the newest).

Mathematical functions are available, and are named the same as in C (and on the whole, Java). For instance, `sin()`, `cos()`, and `pow()`. There is also a `max()` function---if you put `0` as one of the arguments instead of `0.0`, that you'll get a type error. These work with `float` type variables. There are also vector types `vec3` and `vec4` for three and four dimensional vectors. These have the expected operations: `normalize()` for vector normalization, `dot()` for dot product, etc...

Handing in

Hand in the following:

1. **all** of your java and GLSL files (no class files *please*)
2. your makefile
3. a description file `descrip.txt`

The description file should be a *very brief* description of what you changed and added. **Put your name at the top of this file**. It should briefly describe all modifications to the skeleton code you performed, as well as describe the additional code you wrote for the assignment. If you implemented any additional/optional features or anything else special, you must state what you did in this file (to get credit).

Here are the instructions on how to use the handin program. Essentially, you'll be running the following `tar` command in the directory that contains your java and GLSL files and description:

```
tar cf proj2.tar Makefile *.java *.vp *.fp descrip.txt
```

and then handing in this tar file via the web interface. We suggest checking the length of this `tar` file (type `ls -l proj2.tar`) against the number that you'll see on the screen when you hand in the file, to ensure that you've handed in the correct one.

This assignment is a bit vague so that you can make a lot of the design decisions yourself. There are several approaches to the problems here. Use your best judgement to try and get the most useful result. Ask for help or clarification when you need it.

Program use

To run the program, you either read in a polygon mesh from a file, or create one by tessellation. The files are in Wavefront OBJ format (a simplified version, actually, which only reads in the specification of vertex locations and polygon faces -- but you should still be able to use any other OBJ files you might find).

To read in a mesh from a file (for the file "cube.obj" in the objects directory):

```
java Mesh obj/cube.obj
```

To create an ellipsoid (from a 20x30 grid):

```
java Mesh -ellipsoid 20 30
```

To create an ellipsoid (using the default 24x24 grid):

```
java Mesh -ellipsoid
```

Once the program is running, you can transform the object, as well as specify how you want the object rendered (polygons, wireframe, silhouettes, smooth or faceted shading, material properties, etc...) using sliders and checkboxes.

Data structure

All shapes are represented using the `Shape` class, which contains the parameters for the shape, as well as the mesh used to represent it.

The polygon mesh contained in the `Shape` is represented as an array `vertices`, and an array `polygons`. It is assumed that all polygons are stored in a counter-clockwise fashion (normal vectors point outward), and that their normal vectors are *normalized* (unit length). The class `PolyMesh` which extends `Shape` is for arbitrary meshes (which are read from files). And the class `UVShape` extends the `Shape` class to represent a *uv*-parameterized shape (like an ellipsoid). The `Vertex` interface (a purely abstract class) represents information about a specific vertex: its location is accessed using `getPoint()` and its normal vector (averaged from neighboring polygons) is accessed using `getNormal()`. The `Polygon` interface represents information about a specific polygon in the mesh. A particular `Vertex` is accessed using `getVertex()`, the total number of vertices using `size()`, and the averaged normal vector using `getNormal()`. The `Polygon` interface is implemented by `PolygonAccess` which just handles the array of vertices in each polygon. The `Vertex` and `Polygon` classes are implemented differently by `PolyMesh` and `UVShape`. On the whole, the details of these aren't important until you start with `GLSL`, in which case, you should look how `getPoint()` and `getNormal()` work for the `UVShape`.

All access to the vertices and polygons stored in the `Shape` class should be through the `Vertex` and `Polygon` interfaces. Perhaps you can convince yourself of this by looking at the code for the methods in these interfaces. The vertices stored in each polygon are accessible through `Polygon.getVertex()`. This returns the actual `Vertex` object (not a copy of it). It doesn't return an integer index into the array of vertices -- although it could have worked that way, too.

Drawing

The polygon mesh is transformed using an object transformation (like in Project 1, although the order of rotations is reversed here, to make it more intuitive to manipulate objects like the ellipsoid, which are aligned with the Z axis). This part of the code is already written for you. After this transformation, you draw the mesh in `Shape.draw`. You will be drawing the polygon mesh using one of several styles (perhaps more than one at a time). The following are the styles:

- **polygons**: Draw each polygon in the mesh. (The OpenGL code for setting up the lighting and materials is already done for you.) Make sure that `GL_LIGHTING` is enabled when you draw the polygons (use glEnable). Draw the polygons in `Shape.drawPolygons()` using one of two different shading methods:
  - **flat shading**: Each *polygon* has a single normal vector specified for its entirety (the polygon normal).
  - **smooth shading**: Each *vertex* has a normal vector specified for it, averaged from nearby polygon normals.

  (Don't use glShadeModel to get these different effects. Instead, specify the normals with glNormal by calling it with the appropriate normal vector as described above.)

  Here are examples of each of these shading methods:

Flat shading

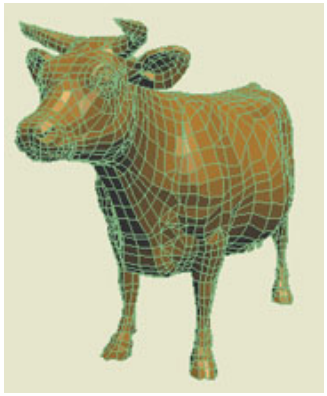
Smooth shading

- **wireframe**: Draw only the lines (the boundary) for each polygon in the mesh. The lines should be drawn in some solid color of your choice. (This means that `GL_LIGHTING` should be disabled.)

  While you could draw this using lines directly (i.e. `GL_LINE_LOOP`), you must instead draw them as polygons, where you use an OpenGL feature that renders only the boundaries of polygons as lines (use glPolygonMode with `GL_FRONT_AND_BACK` and `GL_LINE`). If you don't do it this way, you won't get silhouettes working (below).

  If drawing polygons is not enabled, you should see the entire mesh (as seen below, on the left). Otherwise, the wireframe is only seen on the visible part of the mesh (as on the right).


Wireframe

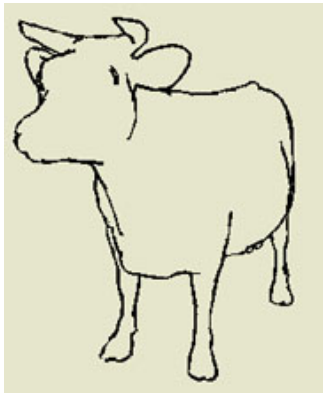

Wireframe + polygons

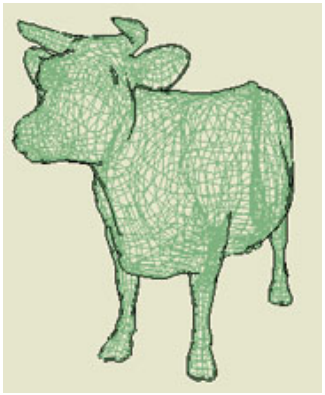
Note that the wireframe rendering in the picture on the right also implements one of the optional extensions, which prevents it from appearing "flickery".

- **silhouettes**: Draw the "silhouettes" of the model as black lines.

  There is a sneaky way of doing this in OpenGL. If you first draw the polygons in a solid color (unshaded), and then draw the back-facing polygons (selected using glCullFace) as thickened wireframe (the line width is set using glLineWidth), then the parts of the wireframe only show through where there is a silhouette. This way, you don't even need to figure out where the silhouette actually is!

  If you are not drawing any polygons, you can actually draw the first pass in a way that doesn't affect the resulting image directly, but does still update the Z-buffer (use glColorMask) -- this is how you get the result as in the image below, on the left. The rest of the images show how silhouettes are drawn with other styles (wireframe, or smoothly shaded polygons).


Silhouettes only


Silhouettes + wireframe


Silhouettes + polygons

- **Phong shading**: Implement per-pixel (Phong) shading using GLSL.

  You're given the vertex program for this in `illum.vp`. You'll see that it's essentially a pass-through shader, which additionally sets varying variables that hold the position and normal in eye coordinates. So these quantities will be available in the fragment shader (after they are interpolated for each pixel in the polygon), even though the geometry will have been projected by then. (It also sets `gl_FrontColor` so that when we bypass the fragment shader when drawing the wireframe, the color of the wireframe still gets through.) We will be using the Phong reflection model without attenuation, and no global ambient light. We will also use the Blinn-Phong specular model (the Phong specular model is below in the extra credit). Thus, you must compute the following in your fragment shader in `illum.fp`:

$$L_a k_a + L_d k_d \max(0, \hat{n} \cdot \hat{l}) + \underbrace{L_s k_s \max(0, \hat{n} \cdot \hat{h})^{\alpha}}_{\text{skip when } \hat{n} \cdot \hat{l} < 0}$$

  This uses the halfway vector *h*, which is an equivalent computation to what OpenGL is using in this program. So when you check *GLSL Illumination* the appearance of the model should look exactly the same when you're doing flat shading. For smooth shading, the appearance of the specularities should be improved, as in the following.

Gouraud shading
(OpenGL)

Phong shading
(GLSL)

In addition to `pos` and `norm` which are the interpolated values from the vertices, the following variables are also going to be useful:

- `gl_LightSource[0].position`: the position of the light (already in eye coordinates)
- `gl_FrontLightProduct[0].ambient`: $L_a\, k_a$
- `gl_FrontLightProduct[0].diffuse`: $L_d\, k_d$
- `gl_FrontLightProduct[0].specular`: $L_s\, k_s$
- `gl_FrontMaterial.shininess`: *&alpha*

Note: You need to compute the vectors *l* and *v* yourself. The vector *l* points from the surface point (`pos`) to the position of the light (`gl_LightSource[0].position`). Everything is in eye-coordinates (the coordinate system of the OpenGL camera), so you just subtract the and normalize. The vector *v* proceeds similarly, except that the vector points from the surface point to the camera center (where is this in eye coordinates?). So again, subtract two points and normalize the result. (Note that *v* is not the vector (0,0,1); the book says this, but it also says this is for a "fixed viewing direction", such as an orthographic camera. That isn't what we're doing here, so compute *v* as describe above.)

Note: Do not use `gl_LightSource[0].halfVector` at all, but instead compute *h* yourself from *l* and *v*, as above. (This particular value is constant, and requires strong assumptions to be true for it to be useful.)

The result of the illumination computation should be stored in `gl_FragColor`. The light source is a positional (not directional) light, and using the above variables, this gives you enough information to compute the illumination equation above.

We need to make sure that the fragment shader doesn't interfere with the other drawing styles. For instance, the lighting will be applied to all drawing (including the wireframe) if we're not careful. We actually only want the fragment shader operating when we're drawing the shaded polygons. As a result, before drawing these polygons, you should call `enableFragShader()`, and afterwards you should call `disableFragShader()`. These functions set the variable `useFragShader` in `illum.fp`.

- **toon shading**: Shade the model using only a few colors to give it a cartoon-like appearance.

Toon shading + silhouettes

Given the following terms from above:

$$d = \max(0, \hat{n} \cdot \hat{l}) \qquad s = \underbrace{\max(0, \hat{n} \cdot \hat{h})^{\alpha}}_{0 \text{ when } \hat{n} \cdot \hat{l} < 0}$$

we can compute a toon-shaded version by quantizing the shading. There are two parameters in `illum.fp` (from the GUI) called `toonLow` and `toonHigh`. A diffuse toon shader simply replaces *d* by a constant (of your choice). A different constant is used when *d* is less than `toonLow`, greater than `toonHigh`, or in between. Given the appropriate constant *c*, the resulting color is simply ($L_a\, k_a + L_d\, k_d$ c). We can combine this with a specular toon shader by first checking if *s* is greater than `toonHigh`. If it is, then set the resulting color to a bright value. Otherwise, use *d*, the diffuse toon shading result. This separates the effects of the specularities from the diffuse shading, and thus preserves the shape and color (white) of the

specularities. This computation should be done in the function `toonShade()` in `illum.fp` which is passed the values of *d* and *s*, and returns the resulting color. For this part, you'll need to come up with a number of constants; choose them to get the most attractive result.

Each of these drawing styles is individually specified through the user interface and are defined as `BooleanParameters` in `Shape`.

Normal vectors

Given a polygon mesh, we must compute both the polygon normals (those returned by `Polygon.getNormal()`), and the vertex normals (those returned by `Vertex.getNormal()`), which are averaged from the adjoining polygon normals, weighted by area. The code for computing the polygon normals for a is provided for you: see `PolyMesh.PolygonPM.computeNormal()`. Is uses Newell's method, but does *not* normalize the normal vectors---the length of the vector is proportional to the area of the polygon. You'll need this when computing the vertex normals. For a `UVShape`, all normals are computed analytically; more on this below.

You must write the code that computes the *vertex* normals in `PolyMesh.computeAllNormals()`. This computation is performed on the entire mesh at once (using the algorithm described in class), so that the normal at a vertex is the *area-weighted average* of the polygon normals over all polygons in which it is contained.

Tessellation

To form the mesh of a parametric shape, you start from a grid of vertices over a 2D domain, and compute its geometry (both points and normal vectors) from analytic equations. So for each point *(u,v)* in the domain, we can compute its corresponding point on the surface using a function *p(u,v)* and its (unnormalized) normal vector using *n(u,v)*. For instance, one possible parameterization of an ellipsoid (with three parameters: a_x, a_y and a_z - the axis lengths of the ellipsoid in the x, y, and z directions) has the equation:

$$p_{\text{ellipsoid}}(u,v) = \begin{pmatrix} a_x \cos u \cos v \\ a_y \sin u \cos v \\ a_z \sin v \end{pmatrix} \quad u \in [0, 2\pi), v \in \left[-\frac{\pi}{2}, \frac{\pi}{2}\right]$$

$$n_{\text{ellipsoid}}(u,v) = \begin{pmatrix} a_y a_z \cos u \cos v \\ a_x a_z \sin u \cos v \\ a_x a_y \sin v \end{pmatrix}$$

Here, *u* is like longitude, and *v* is like latitude. The grid of points in this domain is already constructed for you---see `UVShape.buildUVGrid()`. It creates a grid which includes the grid points at *u=2&pi* as well. But since we are computing normal vectors analytically, this isn't a problem. You must do the following:

- Implement the analytic functions in `Ellipsoid.evalPosition()` and `Ellipsoid.evalNormal()`. These are used by a ordinary OpenGL implementation.
- Implement the analytic functions in the vertex program `ellipsoid.vp`. When GLSL is enabled, you'll see in the methods `UVShape.VertexUV.getPoint()`, `UVShape.VertexUV.getNormal()`, and `UVShape.PolygonUV.getNormal()`, that the appropriate *u,v* values are stuffed into the 3D vertex or normal (with the z component always being zero). So the application sends *u,v* to the vertex program, and it must compute the rest. You are to complete the vertex program so that the point and normal of the ellipsoid are the output of the vertex program. (Most of it is already there.)
  Note that the *u,v* values for the vertex and normal can be different (i.e. flat shading), so be sure to use the appropriate *u,v* values when computing each!

Extensions

The following is a list of optional extensions; difficulties are marked.

- [**Required** for CS grad students] (Easy)
  Implement a torus as a tesselated primitive. The torus should have 2 parameters: *inner-radius* (the radius of the hole, *r*) and *outer-radius* (the radius of the entire shape, *R*). Here is the equation of a torus with slight different parameters (where *a* is the tube-radius, and *c* is the distance from the center of the tube to the center of the shape):

$$s_{\text{torus}}(u,v) = \begin{pmatrix} (c + a\cos v)\cos u \\ (c + a\cos v)\sin u \\ a\sin v \end{pmatrix} \quad u \in [0, 2\pi), v \in [0, 2\pi)$$

$$n_{\text{torus}}(u,v) = \begin{pmatrix} \cos u\ \cos v \\ \sin u\ \cos v \\ \sin v \end{pmatrix}$$

See the Mathworld torus page for details. Work from the class `Torus` which looks very much like `Ellipsoid` (there is already some code in `Mesh.main()` that lets you specify "`-torus`" on the command line.)

- [**Required** for CS grad students] (Easy)
  Simply drawing the wireframe on top of the polygons mostly gets the desired result (thanks to the Z-buffer). It does have some precision difficulties, where sporadic pieces of the wireframe are invisible. One solution to this adds a tiny offset to the wireframe (at each vertex, in the normal direction), so it is drawn outside the shape. A better solution uses the OpenGL features that add tiny displacements to the Z-buffer values when drawing polygons (use glPolygonOffset). Implement this better solution. Offset the filled polygons, not the wireframe.

- [**Required** for CS grad students] (Easy to Moderate)
  Instead of using the Blinn-Phong specular model (which dots the halfway vector and the normal), implement the Phong model (which computes the reflected light vector *r* and dots it with the view vector *v*):
  $$L_a k_a + L_d k_d \max(0, \hat{n}\cdot\hat{l}) + \underbrace{L_s k_s \max(0, \hat{r}\cdot\hat{v})^{\alpha'}}_{\text{skip when } \hat{n}\cdot\hat{l} < 0} \qquad \text{where } \hat{r} = 2(\hat{n}\cdot\hat{l})\hat{n} - \hat{l}$$
  You might want to use the GLSL function
  ```
  reflect(I,N) = I - 2(N•I)N
  ```
  which reflects the vector `I` across the plane specified by the normal vector `N` (which must be normalized). Here, α' is the shininess value, but it doesn't correspond exactly with the α shininess value in the Blinn-Phong model.

  Use the checkbox labeled "Phong model" to enable this feature, which corresponds to the variable `phong` in `illum.fp`. When you implement it, you'll see how the shininess values don't correspond when you switch between models. Put in an adjustment that makes the degree of shininess be *comparable* but not exactly equal, by determining α' in terms of α. See the Wikipedia Blinn-Phong shading model page for details. Explain how you came up with your adjustment in a comment, or perhaps in a separate file. The "right way" involves doing some math: I suggest doing it in Maple.

- (Moderate)
  Implement a different parameterized shape -- perhaps your own! In case you want to know how to compute normal vectors, here is maple code that can be used to derive it. (However, it might not simplify so nicely.) You might even try a non-orientable surface like a Möbius strip. This has the added difficulty such that its normal vectors cannot be used so easily; some of them are always pointing the wrong way. Try (in a vertex or fragment shader) to flip the normal vectors (just in this case) so that they always face the camera. You'll need to add a variable to control this in the fragment program, as well as something to `IllumProgram.bindUniform()`.

- (Moderate)
  Draw the *vertex* normal vectors on the surface by drawing a short line (of appropriate length) from each vertex, to a point that has been offset by a scaled version of the normal vector. Here is an example:

Flat shading + vertex normals

This is easy for a normal polygon mesh. But for a uv-parameterized shape, it gets tricky, as the geometry is computed by the vertex shader. (There is a solution that is quite easy to implement.) Take care that you don't "break" any of the other drawing styles.

Add a method `Shape.drawNormals()` and call it from `Shape.draw()`. Your code will need to act differently when GLSL is on and you're drawing a `UVShape`; it's fine to use a boolean expression like the following instead of a more object-oriented solution:

```
useGLSL() && !getClass().getName().equals("PolyMesh")
```

- Feel free to add additional features other than those listed here. The book has some nice material that isn't too difficult to code up. Or you can put extra effort into implementing the required features (beyond what is required).

Hints

The following are some suggestions...

- A reasonable order to proceed is first to draw the meshes. Work on drawing the wireframe first, then the polygons (you'll have to compute vertex normal vectors for the smooth shading). Then work on silhouettes. Once you're done with this, then work on the tessellations. Finally, work on phong and toon shading.
- Don't forget... Vectors and points represented as `Vector3d` and `Point3d` are *objects* in Java. This means that if you do `b=a;` then `b` refers to the same object as `a`: if you change `b` then `a` changes too. If you want a separate copy, do this: `b = new Vector3d(a);`.
- Don't worry about the case when the average normal vector comes out to zero.
- If you specify the normal vectors incorrectly, your object will be a dark featureless blob. You should still be able to see the object as wireframe, though.
- A guideline for cleaner code: when you enable an OpenGL feature, disable it when you're done with it.
- The order that you draw the polygons, wireframe, etc... matters! Think about this carefully.
- Refer to the OpenGL documentation provided on the course home page.
- If you're stuck, ask for help! Don't waste hours of your time trying to find some strange bug. Go get a beer or something. Or ice cream. Or maybe just do your homework for another class!

---

428 Home