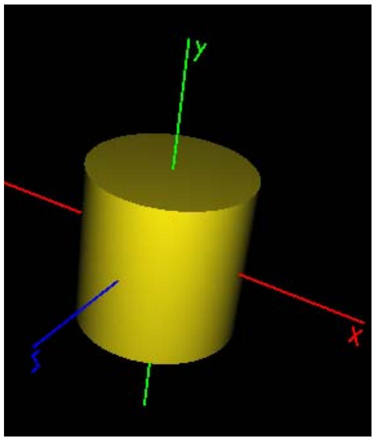
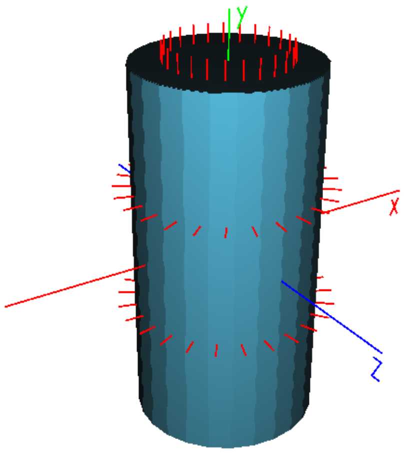
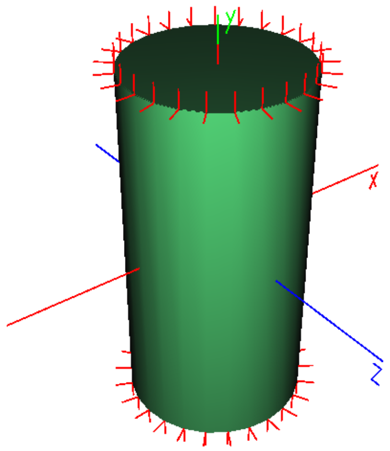
Programing Assignment #4

In this assignment you will generate faces and normals for a primitive object in order to enable shading in two different modes: flat shading and smooth shading.







Requirement 1 (60%) - Implement and display your shaded primitive object

You will build a new primitive object following the same guidelines as in the previous PA, where the new class will have its own methods init(), build(), and draw(). You may re-use your SoCapsule object and build a SoSmoothCapsule object, or you may as well just make a simpler SoCylinder class. The only requirements are that 1) you implement your primitive object with your own code and 2) you allow for controlling its resolution similarly to the capsule PA. Below are more details:

a) You will have to create at least an array of vertices and an array of normals. As a simplification you do not need to send one color per vertex as before and instead you may specify colors as materials passed to the vertex shader as uniforms.

b) Your build method will now need to organize points not as lines but as triangles. When draw() is called, it will draw the arrays with parameter GL_TRIANGLES.

c) In order to use the illumination equation we saw in lecture you will need to define light parameters and material parameters. Example classes for this are given at the end of this document.

d) You will need to write new shaders in order to achieve smooth shading. For this PA it will be **enough to use Gouraud shading**, and example shaders are given at the end of this document.

e) Note that you will need to create a new glsl program using the two new shaders. But since you will also be displaying some colored lines in order to visualize your normals, the previous shaders (for flat colors) will also be needed. The support code already has all the functionality needed to load new shaders. Just create a program with them, and attach them to your new object.

f) Controls: use the following key mappings to change the shading mode:
'z' - Flat Shading 'x' - Smooth Shading

Also, to demonstrate that your object is correct, leave all the original functionality controls for changing the resolution of your object and for rotating the entire object.:

'q' : increment the number of faces (by one)
'a' : decrement the number of faces (by one)
arrows : rotate object

Grading:
- correct flat shading as resolution changes: 30%
- correct smooth shading as resolution changes: 30%

Requirement 2 (25%) – Visualization of Normals

You will also need to draw lines showing each normal vector you are sending to OpenGL. You should draw the normal vectors as segments originating from each vertex in smooth shading, and from the center of each face in flat shading mode. See the pictures above for examples.

You are free to decide how to generate the segments for visualization of the normals. For example, you may create a new SoSegments class. Recall that the normal vector of a polygon is obtained with the cross product between two vectors lying on the plane of the polygon.

Visualizing the normals is a good way to be sure you are achieving correct renderings. If the tube does not look correct then your normals may be not correct. Remember that when using colors (for drawing the normal vectors) you will have to use the shaders for flat rendering, and when using materials (for drawing the smooth tube) you will then have to use shaders for smooth rendering

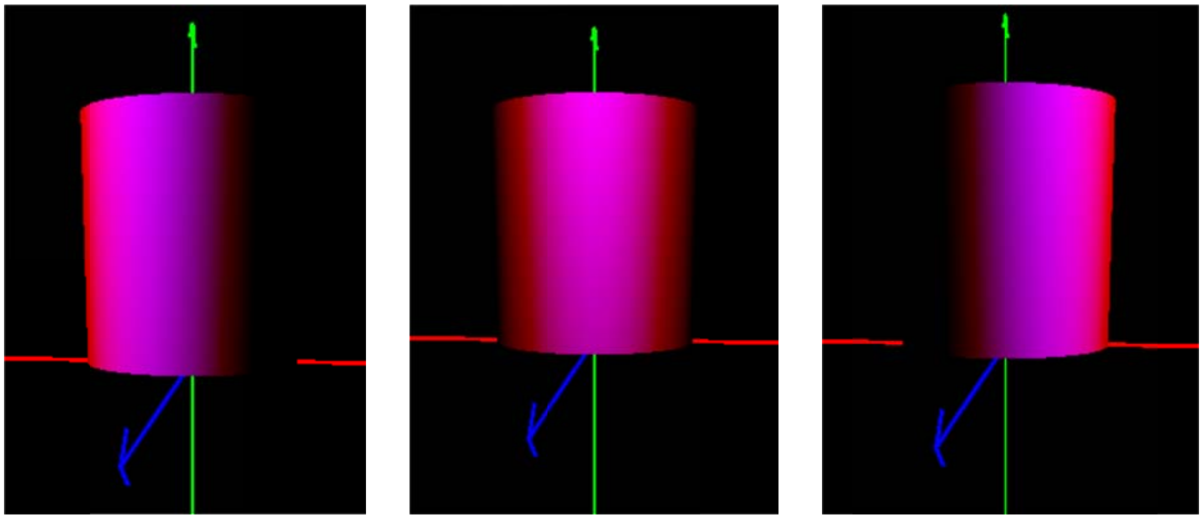
Controls: use the following keys to turn on and off the display of the normals:
'c' – normals on
'v' – normals off

Requirement 3 (15%) – Change the location of the light(s)

In this final requirement you will need to use two keys to move the light source location, and as you move the light position you should be able to see the corresponding shading effects.

Controls: use (at least) the following keys to move the light source:
'w' – light source moves in one direction
's' – light source moves in the opposite direction

The details of this final requirement are open, we basically want you to manipulate the light position and generate meaningful changes in the shading. Below are examples obtained by moving the light source from left to right:



Example of Light and Material classes with minimal parameters:

```
class Light
 { public :
    GsVec pos;
    GsColor amb, dif, spe; // ambient, diffuse, specular
    Light ( const GsVec& p, const GsColor& a, const GsColor& d, const GsColor& s )
      : pos(p), amb(a), dif(d), spe(s) {}
 };

class Material
 { public :
    GsColor amb, dif, spe; // ambient, diffuse, specular
    float sh; // shininess
    Material ( const GsColor& a, const GsColor& d, const GsColor& s, float sn )
      : amb(a), dif(d), spe(s), sh(sn) {}
 };
```

Example of a possible draw method:

```
void SoSmoothObj::draw ( const GsMat& tr, const GsMat& pr, const Light& l, const Material& m )
 {
   // Draw Lines:
   glUseProgram ( prog );
   glBindVertexArray ( va[0] );

   glBindBuffer ( GL_ARRAY_BUFFER, buf[0] ); // positions
   glEnableVertexAttribArray ( 0 );
   glVertexAttribPointer ( 0, 3, GL_FLOAT, GL_FALSE, 0, 0 );

   glBindBuffer ( GL_ARRAY_BUFFER, buf[1] ); // normals
   glEnableVertexAttribArray ( 1 );
   glVertexAttribPointer ( 1, 3, GL_FLOAT, GL_FALSE, 0, 0 ); // false means no normalization
   glUniformMatrix4fv ( uniloc[0], 1, GL_FALSE, tr.e );
   glUniformMatrix4fv ( uniloc[1], 1, GL_FALSE, pr.e );

   float f[4]; // we convert below our color values to be in [0,1]
   glUniform3fv ( uniloc[2], 1, l.pos.e );
   glUniform4fv ( uniloc[3], 1, l.amb.get(f) );
   glUniform4fv ( uniloc[4], 1, l.dif.get(f) );
   glUniform4fv ( uniloc[5], 1, l.spe.get(f) );

   glUniform4fv ( uniloc[6], 1, m.amb.get(f) );
   glUniform4fv ( uniloc[7], 1, m.dif.get(f) );
   glUniform4fv ( uniloc[8], 1, m.spe.get(f) );
   glUniform1fv ( uniloc[9], 1, &m.sh );

   glDrawArrays ( GL_TRIANGLES, 0, _numpoints );
 }
```

Note: for the above code to work, method GsColor::get(float f[4]) in glutapp3d.7z support code has been updated to:

```
float* GsColor::get ( float f[4] ) const
 {
   f[0] = float(r) / 255.0f;
   f[1] = float(g) / 255.0f;
   f[2] = float(b) / 255.0f;
   f[3] = float(a) / 255.0f;
   return f;
 }
```

Example vertex shader for Gouraud shading (vsh_smtl_gouraud.glsl):

```
# version 400

layout (location = 0) in vec3 vPos;
layout (location = 1) in vec3 vNorm;

uniform mat4 vTransf;
uniform mat4 vProj;
uniform vec3 lPos;
uniform vec4 la;
uniform vec4 ld;
uniform vec4 ls;
uniform vec4 ka;
uniform vec4 kd;
uniform vec4 ks;
uniform float sh;

out vec4 Color;

vec4 shade ( vec4 p )
 {
   vec3 n = normalize ( mat3(vTransf)*vNorm ); // vertex normal
   vec3 l = normalize ( lPos-p.xyz );          // light direction
   vec3 r = reflect ( -l, n );                 // reflected ray
   vec3 v = vec3 ( 0, 0, 1.0 );                // view point

   vec4 amb = la*ka;
   vec4 dif = ld*kd*max(dot(l,n),0.0);
   vec4 spe = ls*ks*pow(max(r.z,0.0),sh);      // r.z==dot(v,r)
   if ( dot(l,n)<0 ) spe=vec4(0.0,0.0,0.0,1.0);

   return amb + dif + spe;
 }

void main ()
 {
   vec4 p = vec4(vPos,1.0) * vTransf; // vertex pos in eye coords

   Color = shade ( p );

   gl_Position = p * vProj;
 }
```

Example fragment shader for Gouraud shading (fsh_gouraud.glsl):

```
# version 400

in  vec4 Color;
out vec4 fColor;

void main()
 {
   fColor = Color;
 }
```

-Nothing special here since for Gouraud shading we are, as before, letting OpenGL interpolate the colors computed in the vertex shader for each vertex.

(note: also take a look at “gltutors-shading.7z)