CMSC427 fall 2017 Project 1

Fan Yang

Requirement 1: two class files (Tetra.java and Cylinder.java) were created. Each class has the vertices, normals and texture coordinates. Keyboard events were added: when ‘c’ is pressed, the model switches to cylinder; when ‘t’ is pressed, the model switches to tetrahedron; when ‘I’ is pressed, the model switches back to the shuttle.

The mesh class and code class have been modified in many places. Anything that is related to the setup of variable “gl” was moved to the code.java file. To avoid the reloading of VBO every frame, a helper method named setupGL was created. The method is called only when the model changes.

Requirement 2: The virtual trackball is implemented according to the first link provided in the project description. Every time when the mouse moves a certain distance, the program will calculate u0 (the unit vector from the center of the sphere to the initial mouse position) and u1 (the unit vector from the center of the sphere to the final mouse position), then the angle between them is calculated using the dot product, and the rotation axis is calculated by taking the cross product followed by normalization. Then using the provided api method gl.rotate() will give us the desired rotation.

Requirement 3: In the fragment shader a point light source is created. Light direction is calculated by subtracting the vertex position from the light position. The last line of the shader is changed from color = vec4(color.rgb*cosTheta,1.0f) to color = vec4(color.rgb*cosPhi,1.0f) in order to use the light.

//The App.java file

```
public class App {

   static Code myModelView;
   static Controller myController;

   public static void main(String[] args) {
      myController = new Controller();
      myModelView = new Code(myController);
      myController.addCode(myModelView);
   }
}
```

```
// Class Controller handles all GUI events

import graphicslib3D.Vector3D;
import java.awt.event.MouseListener;
import java.awt.event.MouseMotionAdapter;
import java.awt.event.MouseEvent;
import java.awt.event.KeyListener;
import java.awt.event.KeyEvent;

// An Adapter is a default implementation of a listener, so you don't have implement all
// required methods
// An Interface Listener requires that you do
```

```
public class Controller extends MouseMotionAdapter implements KeyListener, MouseListener {
   int mx, my, width, height, radius;
   Code myView;

   // Need to add reference to ModelView object
   // So as to access redraw() and setAngle()
   public void addCode(Code c) {
      myView = c;
      width = myView.getCanvas().getWidth();
      height = myView.getCanvas().getHeight();
      radius = width/2;
   }

   public void mouseReleased(MouseEvent e) {}

   public void mousePressed(MouseEvent e) {
      mx = e.getX();
      my = e.getY();
   }

   public void mouseEntered(MouseEvent e) {}

   public void mouseExited(MouseEvent e) {}

   public void mouseClicked(MouseEvent e) {}

   // Only function event at moment
   public void mouseDragged(MouseEvent e) {
      //myView.setAngles(mx, my);
      int dx = e.getX() - mx;
      int dy = e.getY() - my;
      if(dx*dx + dy*dy < 25) return;
      if(insideCircle(mx, my) && insideCircle(e.getX(), e.getY())) {
         Vector3D u1 = setVector(mx, my);
         Vector3D u2 = setVector(e.getX(), e.getY());
         Vector3D axis = u1.cross(u2).normalize();
         double degrees = Math.acos(u1.dot(u2)) / Math.PI * 180.0;
         myView.setRotation(degrees, axis);
         myView.redraw();
      }
      mx = e.getX();
      my = e.getY();
   }

   public void mouseMoved(MouseEvent e) {}

   public void keyTyped(KeyEvent e) {}

   public void keyPressed(KeyEvent e) {
      char c = e.getKeyChar();
      if(c == 'i') {
         myView.setCamera(0, 0, 2);
         myView.setModel(new ImportedModel("shuttle.obj"));
      } else if(c == 't') {
         myView.setCamera(0, 0, 5);
         myView.setModel(new Tetra());
      }else if(c == 'c') {
         myView.setCamera(0, 0, 10);
         myView.setModel(new Cylinder());
      }
      myView.redraw();
```

```
    }

    /** Handle the key-released event from the text field. */
    public void keyReleased(KeyEvent e) {

    }

    private boolean insideCircle(float x, float y){
       x -= width/2;
       y -= height/2;
       return x*x + y*y < radius*radius;
    }

    private Vector3D setVector(float x, float y){
       x -= width/2;
       y -= height/2;
       float z = (float) -Math.sqrt(radius*radius - x*x - y*y);
       return new Vector3D(x,y,z).normalize();
    }
}


// ModelView class

import com.jogamp.common.nio.Buffers;
import graphicslib3D.*;
import java.io.*;
import java.nio.FloatBuffer;
import javax.swing.*;
import static com.jogamp.opengl.GL4.*;
import com.jogamp.opengl.*;
import com.jogamp.opengl.awt.GLCanvas;
import com.jogamp.opengl.util.texture.*;

public class Code extends JFrame implements GLEventListener
{  private GLCanvas myCanvas;
   private int rendering_program;
   private int vbo[] = new int[3];
   private float cameraX, cameraY, cameraZ;
   private float objLocX, objLocY, objLocZ;
   private GLSLUtils util = new GLSLUtils();

   private int texture;
   private Texture joglTexture;
   private Matrix3D rMat = new Matrix3D();
   private Mesh myObj;
    float xAngle = 0.0f;
    float yAngle = 0.0f;
   private GL4 gl;
   int mv_loc, proj_loc, camera_loc;
   boolean flag = false;

   public Code(Controller myController)
   {  setTitle("Chapter6 - program3");
      setSize(800, 800);
      // ADDED FOR MACS
      GLProfile glp = GLProfile.getMaxProgrammableCore(true);
      GLCapabilities caps = new GLCapabilities(glp);
      myCanvas = new GLCanvas(caps);
      //myCanvas = new GLCanvas();
      myCanvas.addGLEventListener(this);
```

```
    myCanvas.addMouseListener(myController);
    myCanvas.addMouseMotionListener(myController);
    myCanvas.addKeyListener(myController);
    myCanvas.setFocusable(true);
    myCanvas.requestFocus();
    getContentPane().add(myCanvas);
    this.setVisible(true);
}

public void display(GLAutoDrawable drawable)
{
    float aspect = (float) myCanvas.getWidth() / (float) myCanvas.getHeight();
    Matrix3D pMat = perspective(60.0f, aspect, 0.1f, 1000.0f);

    Matrix3D vMat = new Matrix3D();
    vMat.translate(-cameraX, -cameraY, -cameraZ);

    Matrix3D mMat = new Matrix3D();
    mMat.translate(objLocX, objLocY, objLocZ);
    mMat.rotateY(135.0f);
    // For rotation
    //mMat.rotateY(yAngle);
    //mMat.rotateX(xAngle);
    mMat.concatenate(rMat.transpose());
    mMat.scale(1.5f,1.5f,1.5f);

    Matrix3D mvMat = new Matrix3D();
    mvMat.concatenate(vMat);
    mvMat.concatenate(mMat);

    gl.glUniformMatrix4fv(mv_loc, 1, false, mvMat.getFloatValues(), 0);
    gl.glUniformMatrix4fv(proj_loc, 1, false, pMat.getFloatValues(), 0);
    gl.glUniform3f(camera_loc, cameraX, cameraY, cameraZ);
    // Object renders itself
    // Needs gl for OpenGL context
    // Could own its shader program and texture
    if(flag) {
       setupGL();
       flag = false;
    }
    render();
}


private void render()
{
    gl.glClear(GL_DEPTH_BUFFER_BIT);
    gl.glClear(GL_COLOR_BUFFER_BIT);

    gl.glActiveTexture(GL_TEXTURE0);
    gl.glBindTexture(GL_TEXTURE_2D, texture);
    // added for Macs
    gl.glUniform1i(gl.glGetUniformLocation(rendering_program, "samp"), 0);

    gl.glEnable(GL_CULL_FACE);
    //gl.glEnable(GL_DEPTH_TEST);
    gl.glFrontFace(GL_CCW);
    gl.glDrawArrays(GL_TRIANGLES, 0, myObj.vertices.length);
}

private void setupGL()
```

```
{
   gl.glGenBuffers(vbo.length, vbo, 0);

   gl.glBindBuffer(GL_ARRAY_BUFFER, vbo[0]);

   FloatBuffer vertBuf = Buffers.newDirectFloatBuffer(myObj.pvalues);
   gl.glBufferData(GL_ARRAY_BUFFER, vertBuf.limit()*4, vertBuf, GL_STATIC_DRAW);

   gl.glBindBuffer(GL_ARRAY_BUFFER, vbo[1]);
   FloatBuffer texBuf = Buffers.newDirectFloatBuffer(myObj.tvalues);
   gl.glBufferData(GL_ARRAY_BUFFER, texBuf.limit()*4, texBuf, GL_STATIC_DRAW);

   gl.glBindBuffer(GL_ARRAY_BUFFER, vbo[2]);
   FloatBuffer norBuf = Buffers.newDirectFloatBuffer(myObj.nvalues);
   gl.glBufferData(GL_ARRAY_BUFFER, norBuf.limit()*4,norBuf, GL_STATIC_DRAW);

   gl.glBindBuffer(GL_ARRAY_BUFFER, vbo[0]);
   gl.glVertexAttribPointer(0, 3, GL_FLOAT, false, 0, 0);
   gl.glEnableVertexAttribArray(0);

   gl.glBindBuffer(GL_ARRAY_BUFFER, vbo[1]);
   gl.glVertexAttribPointer(1, 2, GL_FLOAT, false, 0, 0);
   gl.glEnableVertexAttribArray(1);
}

public void init(GLAutoDrawable drawable)
{
   gl = (GL4) GLContext.getCurrentGL();
   myObj = new ImportedModel("shuttle.obj");
   rendering_program = createShaderProgram();

   cameraX = 0.0f; cameraY = 0.0f; cameraZ = 2.0f;
   objLocX = 0.0f; objLocY = 0.0f; objLocZ = 0.0f;

   joglTexture = loadTexture("spstob_1.jpg");
   texture = joglTexture.getTextureObject();

   mv_loc = gl.glGetUniformLocation(rendering_program, "mv_matrix");
   proj_loc = gl.glGetUniformLocation(rendering_program, "proj_matrix");
   camera_loc = gl.glGetUniformLocation(rendering_program, "camera_loc");

   gl.glUseProgram(rendering_program);

   setupGL();
}

private Matrix3D perspective(float fovy, float aspect, float n, float f)
{  float q = 1.0f / ((float) Math.tan(Math.toRadians(0.5f * fovy)));
   float A = q / aspect;
   float B = (n + f) / (n - f);
   float C = (2.0f * n * f) / (n - f);
   Matrix3D r = new Matrix3D();
   r.setElementAt(0,0,A);
   r.setElementAt(1,1,q);
   r.setElementAt(2,2,B);
   r.setElementAt(3,2,-1.0f);
   r.setElementAt(2,3,C);
   r.setElementAt(3,3,0.0f);
   return r;
}
```

```
// Added for MVC
// Called by controller when model changes
void redraw() {
   myCanvas.repaint();
}

// Added for MVC
// Called by controller on MouseDragged
void setAngles(int mx, int my) {
   yAngle = my/(float) myCanvas.getHeight() * 180.0f;
   xAngle = mx/(float) myCanvas.getWidth() * 180.0f;
}

void setRotation(double degrees, Vector3D axis) {
   rMat.rotate(degrees,axis);
}

void setCamera(float x, float y, float z){
   cameraX = x;
   cameraY = y;
   cameraZ = z;
}

void setModel(Mesh obj){
   myObj = obj;
   flag = true;
}

public void reshape(GLAutoDrawable drawable, int x, int y, int width, int height) {}
public void dispose(GLAutoDrawable drawable) {}

private int createShaderProgram()
{
   String vshaderSource[] = util.readShaderSource("code/vert.shader");
   String fshaderSource[] = util.readShaderSource("code/frag.shader");

   int vShader = gl.glCreateShader(GL_VERTEX_SHADER);
   int fShader = gl.glCreateShader(GL_FRAGMENT_SHADER);

   gl.glShaderSource(vShader, vshaderSource.length, vshaderSource, null, 0);
   gl.glShaderSource(fShader, fshaderSource.length, fshaderSource, null, 0);

   gl.glCompileShader(vShader);
   gl.glCompileShader(fShader);

   int vfprogram = gl.glCreateProgram();
   gl.glAttachShader(vfprogram, vShader);
   gl.glAttachShader(vfprogram, fShader);
   gl.glLinkProgram(vfprogram);
   return vfprogram;
}

private Texture loadTexture(String textureFileName)
{  Texture tex = null;
   try { tex = TextureIO.newTexture(new File(textureFileName), false); }
   catch (Exception e) { e.printStackTrace(); }
   return tex;
}

public GLCanvas getCanvas(){
   return myCanvas;
```

```
    }
}
```

// Cylinder class

```
import graphicslib3D.Vertex3D;

public class Cylinder extends Mesh {
    float[] verts, tcs, normals;

    public Cylinder() {
        setupVerts();
        vertices = new Vertex3D[numVertices];
        for (int i = 0; i < numVertices; i++) {
            vertices[i] = new Vertex3D();
            vertices[i].setLocation(verts[i*3], verts[i*3+1], verts[i*3+2]);
            vertices[i].setST(tcs[i*2], tcs[i*2+1]);
            vertices[i].setNormal(normals[i*3], normals[i*3+1], normals[i*3+2]);
        }
        setupVertices();
    }

    private void setupVerts() {
        int numSlices = 5, numStacks = 20;
        numVertices = numSlices*numStacks*6 + numStacks*6;
        verts = new float[numVertices*3];
        normals = new float[numVertices*3];
        tcs = new float[numVertices*2];
        for(int i = 0; i < numSlices; i++)
            for (int j = 0; j < numStacks; j++) {
                verts[(i*numStacks+j)*18] = (float) Math.cos(Math.PI*2*j/numStacks);
                verts[(i*numStacks+j)*18+1] = (float) Math.sin(Math.PI*2*j/numStacks);
                verts[(i*numStacks+j)*18+2] = i - numSlices/2.0f;
                normals[(i*numStacks+j)*18] = (float) Math.cos(Math.PI*2*j/numStacks);
                normals[(i*numStacks+j)*18+1] = (float) Math.sin(Math.PI*2*j/numStacks);
                normals[(i*numStacks+j)*18+2] = 0;
                tcs[(i*numStacks+j)*12] = 1.0f*j/numStacks;
                tcs[(i*numStacks+j)*12+1] = 1 - 1.0f*i/numSlices;

                verts[(i*numStacks+j)*18+3] = (float) Math.cos(Math.PI*2*(j+1)/numStacks);
                verts[(i*numStacks+j)*18+4] = (float) Math.sin(Math.PI*2*(j+1)/numStacks);
                verts[(i*numStacks+j)*18+5] = i - numSlices/2.0f;
                normals[(i*numStacks+j)*18+3] = (float) Math.cos(Math.PI*2*(j+1)/numStacks);
                normals[(i*numStacks+j)*18+4] = (float) Math.sin(Math.PI*2*(j+1)/numStacks);
                normals[(i*numStacks+j)*18+5] = 0;
                tcs[(i*numStacks+j)*12+2] = 1.0f*(j+1)/numStacks;
                tcs[(i*numStacks+j)*12+3] = 1 - 1.0f*i/numSlices;

                verts[(i*numStacks+j)*18+6] = (float) Math.cos(Math.PI*2*(j+1)/numStacks);
                verts[(i*numStacks+j)*18+7] = (float) Math.sin(Math.PI*2*(j+1)/numStacks);
                verts[(i*numStacks+j)*18+8] = i - numSlices/2.0f + 1;
                normals[(i*numStacks+j)*18+6] = (float) Math.cos(Math.PI*2*(j+1)/numStacks);
                normals[(i*numStacks+j)*18+7] = (float) Math.sin(Math.PI*2*(j+1)/numStacks);
                normals[(i*numStacks+j)*18+8] = 0;
                tcs[(i*numStacks+j)*12+4] = 1.0f*(j+1)/numStacks;
                tcs[(i*numStacks+j)*12+5] = 1 - 1.0f*(i+1)/numSlices;

                verts[(i*numStacks+j)*18+9] = (float) Math.cos(Math.PI*2*j/numStacks);
                verts[(i*numStacks+j)*18+10] = (float) Math.sin(Math.PI*2*j/numStacks);
                verts[(i*numStacks+j)*18+11] = i - numSlices/2.0f;
```

```
            normals[(i*numStacks+j)*18+9] = (float) Math.cos(Math.PI*2*j/numStacks);
            normals[(i*numStacks+j)*18+10] = (float) Math.sin(Math.PI*2*j/numStacks);
            normals[(i*numStacks+j)*18+11] = 0;
            tcs[(i*numStacks+j)*12+6] = 1.0f*j/numStacks;
            tcs[(i*numStacks+j)*12+7] = 1 - 1.0f*i/numSlices;

            verts[(i*numStacks+j)*18+12] = (float) Math.cos(Math.PI*2*(j+1)/numStacks);
            verts[(i*numStacks+j)*18+13] = (float) Math.sin(Math.PI*2*(j+1)/numStacks);
            verts[(i*numStacks+j)*18+14] = i - numSlices/2.0f + 1;
            normals[(i*numStacks+j)*18+12] = (float) Math.cos(Math.PI*2*(j+1)/numStacks);
            normals[(i*numStacks+j)*18+13] = (float) Math.sin(Math.PI*2*(j+1)/numStacks);
            normals[(i*numStacks+j)*18+14] = 0;
            tcs[(i*numStacks+j)*12+8] = 1.0f*(j+1)/numStacks;
            tcs[(i*numStacks+j)*12+9] = 1 - 1.0f*(i+1)/numSlices;

            verts[(i*numStacks+j)*18+15] = (float) Math.cos(Math.PI*2*j/numStacks);
            verts[(i*numStacks+j)*18+16] = (float) Math.sin(Math.PI*2*j/numStacks);
            verts[(i*numStacks+j)*18+17] = i - numSlices/2.0f + 1;
            normals[(i*numStacks+j)*18+15] = (float) Math.cos(Math.PI*2*j/numStacks);
            normals[(i*numStacks+j)*18+16] = (float) Math.sin(Math.PI*2*j/numStacks);
            normals[(i*numStacks+j)*18+17] = 0;
            tcs[(i*numStacks+j)*12+10] = 1.0f*j/numStacks;
            tcs[(i*numStacks+j)*12+11] = 1 - 1.0f*(i+1)/numSlices;
        }


    for(int k = 0; k < numStacks; k++){
        int p = numSlices*numStacks*6 + k*3;
        verts[3*p] = (float) Math.cos(Math.PI*2*k/numStacks);
        verts[3*p+1] = (float) Math.sin(Math.PI*2*k/numStacks);
        verts[3*p+2] = numSlices/2.0f;
        normals[3*p] = 0.0f;
        normals[3*p+1] = 0.0f;
        normals[3*p+2] = 1.0f;
        tcs[2*p] = (float) (Math.cos(Math.PI*2*k/numStacks) + 1)/2;
        tcs[2*p+1] = (float) (Math.sin(Math.PI*2*k/numStacks) + 1)/2;

        verts[3*p+3] = (float) Math.cos(Math.PI*2*(k+1)/numStacks);
        verts[3*p+4] = (float) Math.sin(Math.PI*2*(k+1)/numStacks);
        verts[3*p+5] = numSlices/2.0f;
        normals[3*p+3] = 0.0f;
        normals[3*p+4] = 0.0f;
        normals[3*p+5] = 1.0f;
        tcs[2*p+2] = (float) (Math.cos(Math.PI*2*(k+1)/numStacks) + 1)/2;
        tcs[2*p+3] = (float) (Math.sin(Math.PI*2*(k+1)/numStacks) + 1)/2;

        verts[3*p+6] = 0.0f;
        verts[3*p+7] = 0.0f;
        verts[3*p+8] = numSlices/2.0f;
        normals[3*p+6] = 0.0f;
        normals[3*p+7] = 0.0f;
        normals[3*p+8] = 1.0f;
        tcs[2*p+4] = 0.5f;
        tcs[2*p+5] = 0.5f;

        int q = numSlices*numStacks*6 + numStacks*3 + k*3;
        verts[3*q] = (float) Math.cos(Math.PI*2*(k+1)/numStacks);
        verts[3*q+1] = (float) Math.sin(Math.PI*2*(k+1)/numStacks);
        verts[3*q+2] = -numSlices/2.0f;
        normals[3*q] = 0.0f;
        normals[3*q+1] = 0.0f;
```

```
            normals[3*q+2] = -1.0f;
            tcs[2*q] = (float) (Math.cos(Math.PI*2*(k+1)/numStacks) + 1)/2;
            tcs[2*q+1] = (float) (Math.sin(Math.PI*2*(k+1)/numStacks) + 1)/2;

            verts[3*q+3] = (float) Math.cos(Math.PI*2*k/numStacks);
            verts[3*q+4] = (float) Math.sin(Math.PI*2*k/numStacks);
            verts[3*q+5] = -numSlices/2.0f;
            normals[3*q+3] = 0.0f;
            normals[3*q+4] = 0.0f;
            normals[3*q+5] = -1.0f;
            tcs[2*q+2] = (float) (Math.cos(Math.PI*2*k/numStacks) + 1)/2;
            tcs[2*q+3] = (float) (Math.sin(Math.PI*2*k/numStacks) + 1)/2;

            verts[3*q+6] = 0.0f;
            verts[3*q+7] = 0.0f;
            verts[3*q+8] = -numSlices/2.0f;
            normals[3*q+6] = 0.0f;
            normals[3*q+7] = 0.0f;
            normals[3*q+8] = -1.0f;
            tcs[2*q+4] = 0.5f;
            tcs[2*q+5] = 0.5f;
        }
    }
}



// Tera class
import graphicslib3D.Vertex3D;

public class Tetra extends Mesh {
    float[] verts, tcs, normals;

    public Tetra() {
        numVertices = 12;
        setupVerts();
        vertices = new Vertex3D[numVertices];
        for (int i = 0; i < numVertices; i++) {
            vertices[i] = new Vertex3D();
            vertices[i].setLocation(verts[i * 3], verts[i * 3 + 1], verts[i * 3 + 2]);
            vertices[i].setST(tcs[i * 2], tcs[i * 2 + 1]);
            vertices[i].setNormal(normals[i * 3], normals[i * 3 + 1], normals[i * 3 + 2]);
        }
        setupVertices();
    }

    private void setupVerts() {
        float u = (float) Math.sqrt(3);
        float v = (float) Math.sqrt(6);
        float w = (float) Math.sqrt(2);

        float[] verts =
        {
            1.0f, -u/3, 0.0f,       -1.0f,  -u/3, 0.0f,   0.0f, u*2/3, 0.0f,
            -1.0f,  -u/3, 0.0f,     1.0f, -u/3, 0.0f,     0.0f, 0.0f, v*2/3,
            -1.0f,  -u/3, 0.0f,     0.0f, 0.0f, v*2/3,    0.0f, u*2/3, 0.0f,
            1.0f,  -u/3, 0.0f,      0.0f, u*2/3, 0.0f,    0.0f, 0.0f, v*2/3
        };

        float[] normals =
        {
```

```
            0.0f, 0.0f, -1.0f,      0.0f, 0.0f, -1.0f,      0.0f, 0.0f, -1.0f,
            0.0f, -w*2/3, 1/3,      0.0f, -w*2/3, 1/3,      0.0f, -w*2/3, 1/3,
            -v/3, w/3, 1/3,         -v/3, w/3, 1/3,         -v/3, w/3, 1/3,
            v/3, w/3, 1/3,          v/3, w/3, 1/3,          v/3, w/3, 1/3
        };

        float[] tcs =
        {
            0.0f, 1.0f,         1.0f, 0.0f,         0.0f, 0.0f,
            1.0f, 0.0f,         0.0f, 1.0f,         1.0f, 1.0f,
            1.0f, 0.0f,         0.0f, 0.0f,         1.0f, 1.0f,
            0.0f, 1.0f,         1.0f, 1.0f,         0.0f, 0.0f,
        };

        this.verts = verts;
        this.normals = normals;
        this.tcs = tcs;
    }
}


// Mesh class

import graphicslib3D.Vertex3D;

public class Mesh
{
    // Model vertices
   protected Vertex3D [] vertices;
   protected int numVertices;
   public float[] pvalues;
   public float[] tvalues;
   public float[] nvalues;


   public void setupVertices()
   {
      pvalues = new float[numVertices*3];
      tvalues = new float[numVertices*2];
      nvalues = new float[numVertices*3];

      for (int i=0; i<numVertices; i++)
      {  pvalues[i*3]   = (float) (vertices[i]).getX();
         pvalues[i*3+1] = (float) (vertices[i]).getY();
         pvalues[i*3+2] = (float) (vertices[i]).getZ();
         tvalues[i*2]   = (float) (vertices[i]).getS();
         tvalues[i*2+1] = (float) (vertices[i]).getT();
         nvalues[i*3]   = (float) (vertices[i]).getNormalX();
         nvalues[i*3+1] = (float) (vertices[i]).getNormalY();
         nvalues[i*3+2] = (float) (vertices[i]).getNormalZ();
      }
   }
}
```

//vertex shader program

```
#version 410

// Input from VB0s 0, 1 and 2
```

```
layout (location = 0) in vec3 position;
layout (location = 1) in vec2 tex_coord;
layout (location = 2) in vec3 normal;

// Output to fragment shader
out vec2 tc;
out vec3 n;
out vec4 varyingColor;
out vec3 varyingViewDir;

// Input constant across all vertices
uniform mat4 mv_matrix;
uniform mat4 proj_matrix;
uniform sampler2D samp;

void main(void)
{  // Standard matrix transformation of vertex position
    gl_Position = proj_matrix * mv_matrix * vec4(position,1.0);

       // Pass texture coordinate on to fragment shader
       tc = tex_coord;

       // Transform normal to camera coordinates
       n = (mv_matrix*vec4(normal,1.0f)).xyz;
       // Create a view vector to the camera location (0,0,2)
       varyingViewDir = (mv_matrix*vec4(position-vec3(0.0f,0.0f,2.0f),1.0f)).xyz;

       // Create a color based on vertex position
       varyingColor = vec4(position*0.4,1.0) + vec4(0.5, 0.5, 0.5, 0);
}
```

```
// fragment shader program
#version 410

// Input from vertex shader
in vec2 tc;
in vec3 n;
in vec4 varyingColor;
in vec3 varyingViewDir;

// Output to screen
out vec4 color;

uniform mat4 mv_matrix;
uniform mat4 proj_matrix;
uniform sampler2D samp;

void main(void)
{
    // Create a directional light
    vec3 lightDirection = -normalize(vec3(1.0f, 1.0f, 2.0f));

    // Options for starting color
    // Color option 1: fixed color
    //color = vec4(1.0f, 0.0f, 1.0f, 1.0f);
    // Color option 2: texture
   color = texture(samp,tc);
   // Color option 3: varying by vertex option
```

```
   //color =  varyingColor;
   // Color option 4: normal
    //color = vec4(n,1.0f);

   // Computations for shading model
   // Compute dot between light and normal
    float cosTheta = dot(lightDirection,n);
    // Compute reflection vector and dot with view vector
    vec3 reflectVector = normalize(reflect(-lightDirection,n));
   float cosPhi = dot(reflectVector,varyingViewDir);

   // Compute final color
   // Only mulitiply rgb by cosine, since we don't want to change
   // the alpha channel
   color = vec4(color.rgb*cosTheta,1.0f);
}
```