Geometry Shaders

COMP3015 Lab 6

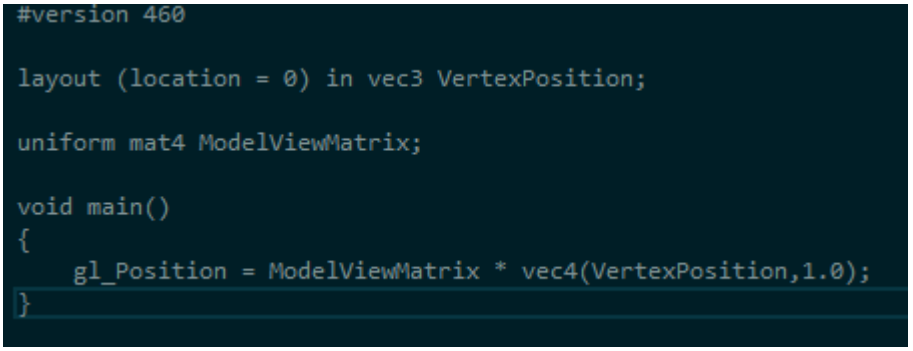
Point Sprites

Overview

Point sprites are simple quads that are aligned as such so that they always face the camera. They are very useful for particle systems in 3D. Usually, point sprites are texture mapped. The point sprint implementation result should look like the following:

Vertex Shader

Pass the vertex position transformed against the camera coordinates. Please note that we're not converting to clip coordinates just yet:



```
#version 460

layout (location = 0) in vec3 VertexPosition;

uniform mat4 ModelViewMatrix;

void main()
{
    gl_Position = ModelViewMatrix * vec4(VertexPosition,1.0);
}
```

Geometry Shader

The geometry shader emits two triangles as a triangle strip. To access the position (camera coordinates) from the vertex shader, we use the input variable ‘gl_in.’ The input primitive is available to the geometry shader via this variable. ‘gl_in’s’ type is an array of structs & each struct contains the fields ‘gl_Position,’ ‘gl_PointSize,’ & ‘gl_ClipDistance[].’ In this implementation, only ‘gl_Position’ needs to be accessed:

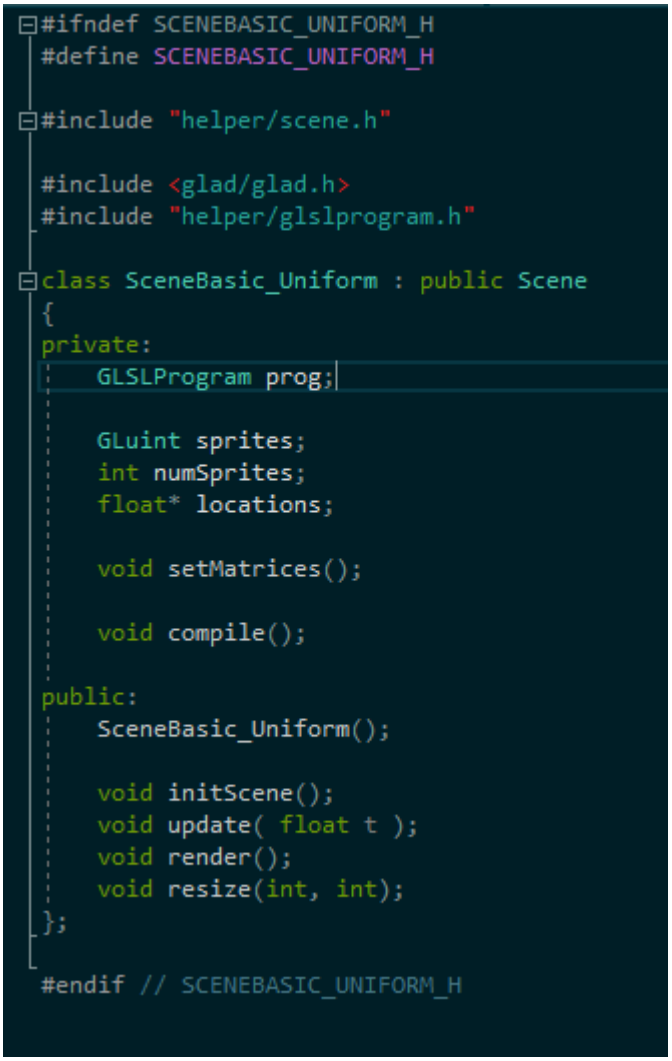
```
#version 460

layout( points ) in; //we define the type of primitives we recive - points in this case
layout( triangle_strip, max_vertices = 4 ) out; //te type of primitives we produce - triangle
                                                //strips with maximum number of vertices = 4

uniform float Size2;   // Half the width of the quad

uniform mat4 ProjectionMatrix; //for transformation into the clip space

out vec2 TexCoord; //used in our fragment shader

void main()
{
    mat4 m = ProjectionMatrix; //assign the projection matrix

    //setup vertex 1
    gl_Position = m * (vec4(-Size2,-Size2,0.0,0.0) + gl_in[0].gl_Position); //we access the information
                                                                            //through gl_in
    TexCoord = vec2(0.0,0.0); //update texture coordinates
    EmitVertex(); //pass the vertex information ot the next stage

    //setup vertex 2
    gl_Position = m * (vec4(Size2,-Size2,0.0,0.0) + gl_in[0].gl_Position);
    TexCoord = vec2(1.0,0.0);
    EmitVertex();

    //setup vertex 3
    gl_Position = m * (vec4(-Size2,Size2,0.0,0.0) + gl_in[0].gl_Position);
    TexCoord = vec2(0.0,1.0);
    EmitVertex();

    //setup vertex 4
    gl_Position = m * (vec4(Size2,Size2,0.0,0.0) + gl_in[0].gl_Position);
    TexCoord = vec2(1.0,1.0);
    EmitVertex();


    EndPrimitive(); //used to finalise the primitive and send it along the pipeline
}
```

Fragment Shader

```
#version 460

in vec2 TexCoord; //texture coordinates coming from geometry shader

uniform sampler2D SpriteTex; //texture use for our sprites

layout( location = 0 ) out vec4 FragColor; //output of the fragment shader

void main()
{
    FragColor = texture(SpriteTex, TexCoord); //look up the texture and assign the colour
}
```

Scenebasic Uniform Header

Create a pointer to a float named ‘locations’ for storing all sprite location information. Information required are the x, y & z coordinates per sprite:



```
#ifndef SCENEBASIC_UNIFORM_H
#define SCENEBASIC_UNIFORM_H

#include "helper/scene.h"

#include <glad/glad.h>
#include "helper/glslprogram.h"

class SceneBasic_Uniform : public Scene
{
private:
    GLSLProgram prog;

    GLuint sprites;
    int numSprites;
    float* locations;

    void setMatrices();

    void compile();

public:
    SceneBasic_Uniform();

    void initScene();
    void update( float t );
    void render();
    void resize(int, int);
};

#endif // SCENEBASIC_UNIFORM_H
```

Scenebasic Uniform CPP

Init Scene Function

```
void SceneBasic_Uniform::initScene()
{
    compile(); //compile, link and use shaders

    glClearColor(0.5f, 0.5f, 0.5f, 1.0f); //set up a background colour

    glEnable(GL_DEPTH_TEST); //enable depth test

    //setup 50 sprites
    numSprites = 50;
    locations = new float[numSprites * 3]; //we need an x, y and z for each sprite
    srand((unsigned int)time(0));          //assign random value, used later for location

    //for each sprite we set up the location in an array
    for (int i = 0; i < numSprites; i++) {
        vec3 p(((float)rand() / RAND_MAX * 2.0f) - 1.0f,
            ((float)rand() / RAND_MAX * 2.0f) - 1.0f,
            ((float)rand() / RAND_MAX * 2.0f) - 1.0f);
        locations[i * 3] = p.x;
        locations[i * 3 + 1] = p.y;
        locations[i * 3 + 2] = p.z;
    }

    // Set up the buffers
    GLuint handle;
    glGenBuffers(1, &handle);

    glBindBuffer(GL_ARRAY_BUFFER, handle);
    glBufferData(GL_ARRAY_BUFFER, numSprites * 3 * sizeof(float), locations, GL_STATIC_DRAW);

    delete[] locations;

    // Set up the vertex array object
    glGenVertexArrays(1, &sprites);
    glBindVertexArray(sprites);

    glBindBuffer(GL_ARRAY_BUFFER, handle);
    glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, 0, ((GLubyte*)NULL + (0)));
    glEnableVertexAttribArray(0);  // Vertex position

    glBindVertexArray(0);

    // Load texture file
    const char* texName = "../Project_Template/media/texture/flower.png";
    Texture::loadTexture(texName);

    prog.setUniform("SpriteTex", 0);
    prog.setUniform("Size2", 0.15f);
}
```

Compile Function

```
void SceneBasic_Uniform::compile()
{
    try {
        prog.compileShader("shader/basic_uniform.vert");
        prog.compileShader("shader/basic_uniform.frag");
        prog.compileShader("shader/basic_uniform.gs");
        prog.link();
        prog.use();
    } catch (GLSLProgramException &e) {
        cerr << e.what() << endl;
        exit(EXIT_FAILURE);
    }
}
```

Render Function

```
void SceneBasic_Uniform::render()
{
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT); //we clear colour and depth buffer

    //set up camera
    vec3 cameraPos(0.0f, 0.0f, 3.0f);
    view = glm::lookAt(cameraPos,
        vec3(0.0f, 0.0f, 0.0f),
        vec3(0.0f, 1.0f, 0.0f));

    model = mat4(1.0f);
    setMatrices(); //setup ypour matrices and send to shaders

    glBindVertexArray(sprites);
    glDrawArrays(GL_POINTS, 0, numSprites);

    glFinish();
}
```

Set Matrices Function

```
void SceneBasic_Uniform::setMatrices()
{
    mat4 mv = view * model; //model view matrix
    prog.setUniform("ModelViewMatrix", mv);
    prog.setUniform("ProjectionMatrix", projection); //send the projection matrix
}
```

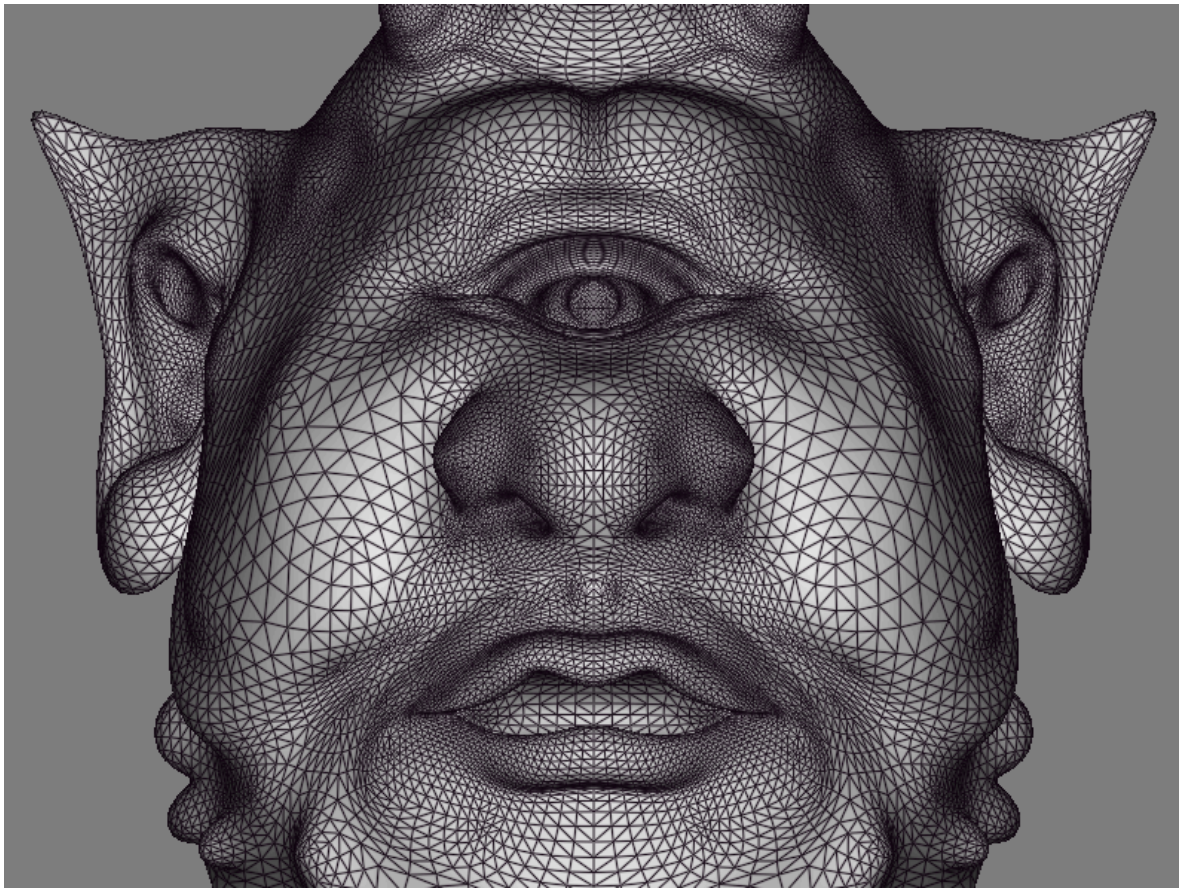
Resize Function

```
void SceneBasic_Uniform::resize(int w, int h)
{
    //setup the ciewport and the projection matrix
    glViewport(0, 0, w, h);
    width = w;
    height = h;
    projection = glm::perspective(glm::radians(70.0f), (float)w / h, 0.3f, 100.0f);
}
```

Wireframes

Overview

In the wireframe implementation, the geometry shader is used to compute additional information about the given triangle in order to allow the fragment shader to draw the edges of each polygon directly over the shaded mesh. Notably, the triangle is not modified at all in order to achieve this; the geometry shader passes the primitive along unmodified. The end result of this implementation should be the following:



Vertex Shader

Use the vertex shader from the point sprite implementation, however the following must be modified:

- Pass the ‘vertexPosition’ & ‘vertexNormal’ to the next stage
- Transform ‘vertexPosition’ & ‘vertexNormal’ into camera coordinates
- Pass ‘vertexPosition’ to the clipping stage, multiply ‘vertexPosition’ with ‘MVP’ & assign the result to ‘gl_Position’

The code should look like the following:

```
#version 460

layout (location = 0) in vec3 VertexPosition;
layout (location = 1 ) in vec3 VertexNormal;

out vec3 VNormal;
out vec3 VPosition;

uniform mat4 ModelViewMatrix;
uniform mat3 NormalMatrix;
uniform mat4 MVP;

void main()
{
    VNormal = normalize( NormalMatrix * VertexNormal);
    VPosition = vec3(ModelViewMatrix * vec4(VertexPosition,1.0));
    gl_Position = MVP * vec4(VertexPosition,1.0);
}
```

Geometry Shader

Input & Output

The geometry of the triangle does not need to change. Therefore the input & output types remain unchanged & the received triangle should be sent unmodified with ‘noperspective out vec3 GEdgeDistance:’

No Perspective Qualifier

The ‘noperspective’ qualifier indicates that the values are to be interpolated linearly, instead of making use of the default perspective correct interpolation. These distances are in screen space, so it would be incorrect to interpolate them in a non-linear fashion.

Global Variables

```
#version 460

//declare the primitives for the geometry shader
//we don't change the geometry of the triangle just pass the data through
//to the fragment shader
layout( triangles ) in;
layout( triangle_strip, max_vertices = 3 ) out;

out vec3 GNormal;    //normal used by fragment shader
out vec3 GPosition;  //position used for fragment shader

noperspective out vec3 GEdgeDistance;  //the distance from the edge

in vec3 VNormal[];
in vec3 VPosition[];

uniform mat4 ViewportMatrix;  // Viewport matrix
```

Main Function

```
void main()
{
    // Transform each vertex into viewport space from clip coordinates
    //divide by the w coordinate as the clip coordinates are homogeneous
    //and need to be converted back to true Cartesian coordinates
    vec2 p0 = vec2(ViewportMatrix * (gl_in[0].gl_Position / gl_in[0].gl_Position.w));
    vec2 p1 = vec2(ViewportMatrix * (gl_in[1].gl_Position / gl_in[1].gl_Position.w));
    vec2 p2 = vec2(ViewportMatrix * (gl_in[2].gl_Position / gl_in[2].gl_Position.w));

    //calculate the edges of the triangle
    float a = length(p1 - p2);
    float b = length(p2 - p0);
    float c = length(p1 - p0);

    //calculate the angles
    float alpha = acos( (b*b + c*c - a*a) / (2.0*b*c) );
    float beta = acos( (a*a + c*c - b*b) / (2.0*a*c) );

    //calculate the distances
    float ha = abs( c * sin( beta ) );
    float hb = abs( c * sin( alpha ) );
    float hc = abs( b * sin( alpha ) );

    //pass the distance and position of the new vertex to the fragment
    //also pass the normal, position of the mesh
    GEdgeDistance = vec3( ha, 0, 0 );
    GNormal = VNormal[0];
    GPosition = VPosition[0];
    gl_Position = gl_in[0].gl_Position;
    EmitVertex();

    GEdgeDistance = vec3( 0, hb, 0 );
    GNormal = VNormal[1];
    GPosition = VPosition[1];
    gl_Position = gl_in[1].gl_Position;
    EmitVertex();

    GEdgeDistance = vec3( 0, 0, hc );
    GNormal = VNormal[2];
    GPosition = VPosition[2];
    gl_Position = gl_in[2].gl_Position;
    EmitVertex();

    EndPrimitive();
}
```

Fragment Shader

Global Variables

```
#version 460

struct LightInfo {
  vec4 Position;  // Light position in eye coords.
  vec3 Intensity; // A,D,S intensity
};
uniform LightInfo Light;

struct MaterialInfo {
  vec3 Ka;            // Ambient reflectivity
  vec3 Kd;            // Diffuse reflectivity
  vec3 Ks;            // Specular reflectivity
  float Shininess;    // Specular shininess factor
};
uniform MaterialInfo Material;

//line struct for rendering the wireframe
uniform struct LineInfo {
  float Width;
  vec4 Color;
} Line;

//in variables coming from geometry shader
in vec3 GPosition;
in vec3 GNormal;
noperspective in vec3 GEdgeDistance;

//out colour
layout( location = 0 ) out vec4 FragColor;
```

Main Function

```
void main()
{
    //calculate the color
    vec4 color = vec4( phongModel(GPosition, GNormal), 1.0 );

    // Find the smallest distance for the fragment
    float d = min( GEdgeDistance.x, GEdgeDistance.y );
    d = min( d, GEdgeDistance.z );

    float mixVal;
    if( d < Line.Width - 1 )
    {
        mixVal = 1.0;
    }
    else if( d > Line.Width + 1 )
    {
        mixVal = 0.0;
    }
    else
    {
        float x = d - (Line.Width - 1);
        mixVal = exp2(-2.0 * (x*x));
    }

    FragColor = mix( color, Line.Color, mixVal );
}
```

Scenebasic Uniform Header

```
private:
    GLSLProgram prog;

    std::unique_ptr<ObjMesh> ogre;

    glm::mat4 viewport;

    void setMatrices();

    void compile();
```

Scenebasic Uniform CPP

Init Scene Function

```
void SceneBasic_Uniform::initScene()
{
    compile();

    glClearColor(0.5f, 0.5f, 0.5f, 1.0f);

    glEnable(GL_DEPTH_TEST);

    float c = 1.5f;
    projection = glm::ortho(-0.4f * c, 0.4f * c, -0.3f * c, 0.3f * c, 0.1f, 100.0f);

    ////////////// Uniforms ////////////////////
    prog.setUniform("Line.Width", 0.75f);
    prog.setUniform("Line.Color", vec4(0.05f, 0.0f, 0.05f, 1.0f));
    prog.setUniform("Material.Kd", 0.7f, 0.7f, 0.7f);
    prog.setUniform("Light.Position", vec4(0.0f, 0.0f, 0.0f, 1.0f));
    prog.setUniform("Material.Ka", 0.2f, 0.2f, 0.2f);
    prog.setUniform("Light.Intensity", 1.0f, 1.0f, 1.0f);
    prog.setUniform("Material.Ks", 0.8f, 0.8f, 0.8f);
    prog.setUniform("Material.Shininess", 100.0f);
}
```

Compile Function

```
void SceneBasic_Uniform::compile()
{
    try {
        prog.compileShader("shader/basic_uniform.vert");
        prog.compileShader("shader/basic_uniform.frag");
        prog.compileShader("shader/basic_uniform.gs");
        prog.link();
        prog.use();
    } catch (GLSLProgramException &e) {
        cerr << e.what() << endl;
        exit(EXIT_FAILURE);
    }
}
```

Render Function

```
void SceneBasic_Uniform::render()
{
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

    vec3 cameraPos(0.0f, 0.0f, 3.0f);
    view = glm::lookAt(cameraPos,
        vec3(0.0f, 0.0f, 0.0f),
        vec3(0.0f, 1.0f, 0.0f));

    model = mat4(1.0f);
    setMatrices();
    ogre->render();

    glFinish();
}
```

Set Matrices Function

```
void SceneBasic_Uniform::setMatrices()
{
    mat4 mv = view * model;
    prog.setUniform("ModelViewMatrix", mv);
    prog.setUniform("NormalMatrix",
        glm::mat3(vec3(mv[0]), vec3(mv[1]), vec3(mv[2])));
    prog.setUniform("MVP", projection * mv);
    prog.setUniform("ViewportMatrix", viewport);
}
```

Resize Function

```
void SceneBasic_Uniform::resize(int w, int h)
{
    glViewport(0, 0, w, h);

    float w2 = w / 2.0f;
    float h2 = h / 2.0f;
    viewport = mat4(vec4(w2, 0.0f, 0.0f, 0.0f),
        vec4(0.0f, h2, 0.0f, 0.0f),
        vec4(0.0f, 0.0f, 1.0f, 0.0f),
        vec4(w2 + 0, h2 + 0, 0.0f, 1.0f));
}
```

Silhouette Lines

Result

Vertex Shader

Use the vertex shader from the wireframe implementation.

Geometry Shader

Overview

A single triangle (the original triangle) & at most one quad for each edge will be produced. This corresponds to a maximum of 15 vertices than can be produced. This value should be indicated within the output layout directive.

Variable declaration isFrontFacing() & emitEdgeQuad() functions.

Global Variables

```
//input primitives
layout( triangles_adjacency ) in;
layout( triangle_strip, max_vertices = 15 ) out;

//output primitives
out vec3 GNormal;
out vec3 GPosition;

// Which triangle edges are silhouette edges
//0 not edge, 1 is an edge
flat out int GIsEdge;

in vec3 VNormal[];
in vec3 VPosition[];

uniform float EdgeWidth;
uniform float PctExtend;
```

Is Front Facing Function

```
bool isFrontFacing( vec3 a, vec3 b, vec3 c )
{
    return ((a.x * b.y - b.x * a.y) + (b.x * c.y - c.x * b.y) + (c.x * a.y - a.x * c.y))
            > 0;
}
```

Emit Edge Quad Function

```
void emitEdgeQuad( vec3 e0, vec3 e1 )
{
    vec2 ext = PctExtend * (e1.xy - e0.xy);
    vec2 v = normalize(e1.xy - e0.xy);
    vec2 n = vec2(-v.y, v.x) * EdgeWidth;

    GIsEdge = 1;   // This is part of the sil. edge

    gl_Position = vec4( e0.xy - ext, e0.z, 1.0 ); EmitVertex();
    gl_Position = vec4( e0.xy - n - ext, e0.z, 1.0 ); EmitVertex();
    gl_Position = vec4( e1.xy + ext, e1.z, 1.0 ); EmitVertex();
    gl_Position = vec4( e1.xy - n + ext, e1.z, 1.0 ); EmitVertex();

    EndPrimitive();
}
```

Main Function

```
void main()
{
    vec3 p0 = gl_in[0].gl_Position.xyz / gl_in[0].gl_Position.w;
    vec3 p1 = gl_in[1].gl_Position.xyz / gl_in[1].gl_Position.w;
    vec3 p2 = gl_in[2].gl_Position.xyz / gl_in[2].gl_Position.w;
    vec3 p3 = gl_in[3].gl_Position.xyz / gl_in[3].gl_Position.w;
    vec3 p4 = gl_in[4].gl_Position.xyz / gl_in[4].gl_Position.w;
    vec3 p5 = gl_in[5].gl_Position.xyz / gl_in[5].gl_Position.w;

    if( isFrontFacing(p0, p2, p4) ) {
        if( ! isFrontFacing(p0,p1,p2) ) emitEdgeQuad(p0,p2);
        if( ! isFrontFacing(p2,p3,p4) ) emitEdgeQuad(p2,p4);
        if( ! isFrontFacing(p4,p5,p0) ) emitEdgeQuad(p4,p0);
    }

    // Output the original triangle

    GIsEdge = 0;   // This triangle is not part of an edge.

    GNormal = VNormal[0];
    GPosition = VPosition[0];
    gl_Position = gl_in[0].gl_Position;
    EmitVertex();

    GNormal = VNormal[2];
    GPosition = VPosition[2];
    gl_Position = gl_in[2].gl_Position;
    EmitVertex();

    GNormal = VNormal[4];
    GPosition = VPosition[4];
    gl_Position = gl_in[4].gl_Position;
    EmitVertex();

    EndPrimitive();
}
```

Fragment Shader

Ensure the ‘Light’ & ‘Material’ structs are used for light calculation in respect of this implementation:

```
uniform vec4 LineColor;

in vec3 GPosition;
in vec3 GNormal;

flat in int GIsEdge;

layout( location = 0 ) out vec4 FragColor;

//toon shading levels
const int levels = 3;
const float scaleFactor = 1.0 / levels;

vec3 toonShade( )
{
    vec3 s = normalize( Light.Position.xyz - GPosition.xyz );
    vec3 ambient = Material.Ka;
    float cosine = dot( s, GNormal );
    vec3 diffuse = Material.Kd * ceil( cosine * levels ) * scaleFactor;

    return Light.Intensity * (ambient + diffuse);
}

void main() {
    if( GIsEdge == 1 ) {
        FragColor = LineColor;
    } else {
        FragColor =  vec4( toonShade(), 1.0 );
    }

}
```

Scenebasic Uniform Header

The implementation should be similar to the wireframe implementation. If desired, you can make the camera rotate using the angle variable:

```
private:
    GLSLProgram prog;

    std::unique_ptr<ObjMesh> ogre;
    float angle, tPrev, rotSpeed;

    void setMatrices();

    void compile();
```

Scenebasic Uniform CPP

Constructor

```
#include <glm/gtc/matrix_transform.hpp>
#include <glm/gtc/constants.hpp>
using glm::vec3;
using glm::mat4;
using glm::vec4;
using glm::mat3;

//constructor for torus
SceneBasic_Uniform::SceneBasic_Uniform() : angle(0.0f), tPrev(0.0f), rotSpeed(glm::pi<float>() / 8.0f)
{
    ogre = ObjMesh::loadWithAdjacency("../Project_Template/media/bs_ears.obj");
}
```

Init Scene Function

```
void SceneBasic_Uniform::initScene()
{
    compile();

    glClearColor(0.5f, 0.5f, 0.5f, 1.0f);

    glEnable(GL_DEPTH_TEST);

    angle = glm::half_pi<float>();

    prog.setUniform("EdgeWidth", 0.015f);
    prog.setUniform("PctExtend", 0.25f);
    prog.setUniform("LineColor", vec4(0.05f, 0.0f, 0.05f, 1.0f));
    prog.setUniform("Material.Kd", 0.7f, 0.5f, 0.2f);
    prog.setUniform("Light.Position", vec4(0.0f, 0.0f, 0.0f, 1.0f));
    prog.setUniform("Material.Ka", 0.2f, 0.2f, 0.2f);
    prog.setUniform("Light.Intensity", 1.0f, 1.0f, 1.0f);
}
```

Render Function

```
void SceneBasic_Uniform::render()
{
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

    //update camera position
    vec3 cameraPos(1.5f * cos(angle), 0.0f, 1.5f * sin(angle));
    view = glm::lookAt(cameraPos,
        vec3(0.0f, -0.2f, 0.0f),
        vec3(0.0f, 1.0f, 0.0f));

    model = mat4(1.0f);
    setMatrices();
    ogre->render();

    glFinish();
}
```

Set Matrices Function

```
void SceneBasic_Uniform::setMatrices()
{
    mat4 mv = view * model;
    prog.setUniform("ModelViewMatrix", mv);
    prog.setUniform("NormalMatrix",
        glm::mat3(vec3(mv[0]), vec3(mv[1]), vec3(mv[2])));
    prog.setUniform("MVP", projection * mv);
}
```

Resize Function

Notice the orthographic perspective for the camera:

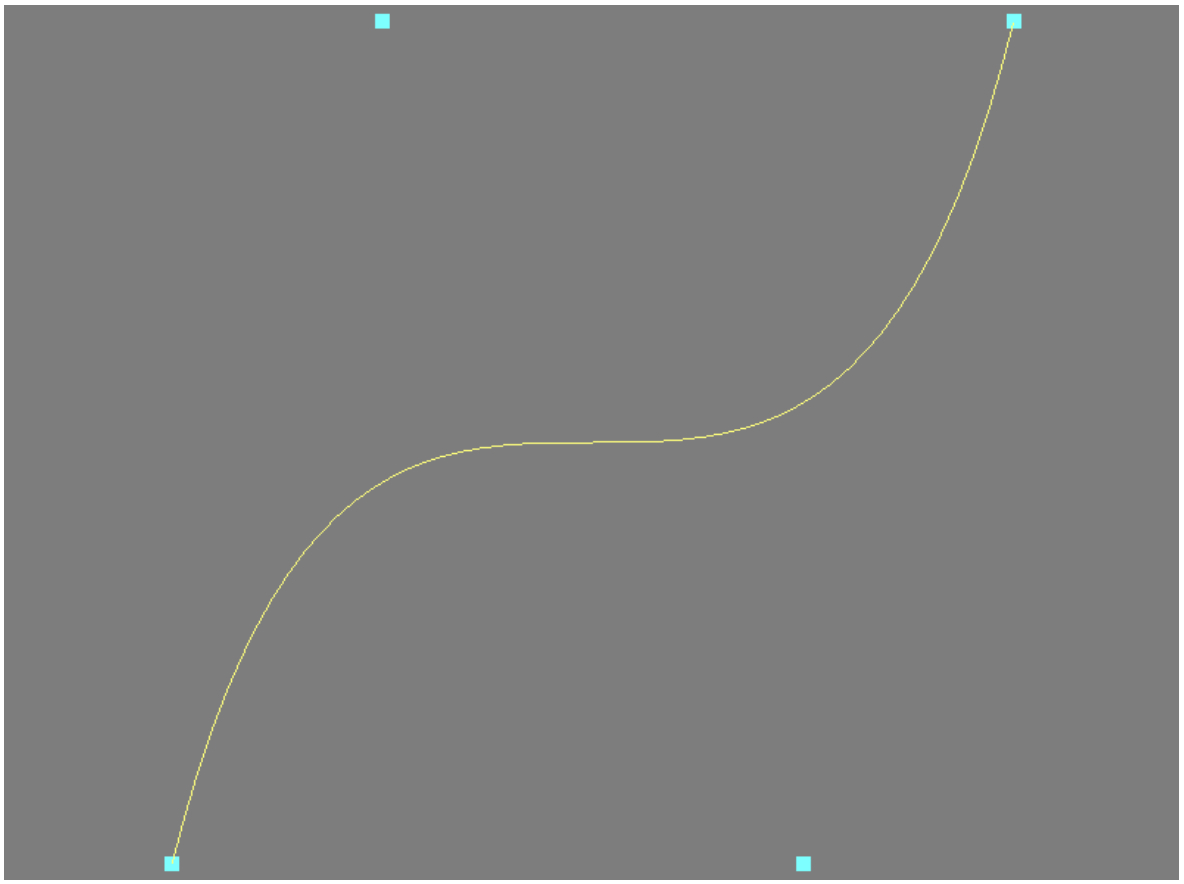
```
void SceneBasic_Uniform::resize(int w, int h)
{
    glViewport(0, 0, w, h);
    float c = 1.5f;
    //using an orthographic camera
    projection = glm::ortho(-0.4f * c, 0.4f * c, -0.3f * c, 0.3f * c, 0.1f, 100.0f);
}
```

Other Functions

The compile() function remains unchanged, however update the angle in the update() function using delta time. For reference, use examples that you implemented earlier in the module or the diffuse example that was given to you.

Curve Tessellation

Result



Vertex Shader

Notice that information is passed with no transformations applied:

```
#version 460

layout (location = 0 ) in vec2 VertexPosition;

void main()
{
    gl_Position = vec4(VertexPosition, 0.0, 1.0);
}
```

Tessellation Control Shader (TCS)

Inputs & Outputs

The main method within the TCS passes the input position (of the patch vertex) to the output position without modification. The ‘gl_out’ and ‘gl_in’ arrays contain the input and output information associated with each vertex in the patch.

GL Invocation Identifier

Note that we assign & read from ‘gl_InvocationID’ in the input & output arrays. The ‘gl_InvocationID’ variable defines the output patch vertex for which this invocation of the TCS is responsible. The TCS can access all of the elements of the ‘gl_in’ array, however it should only write to the location in ‘gl_out’ corresponding to ‘gl_InvocationID.’ The other indices will be written by other invocations of the TCS.

Tessellation Levels

Next, the TCS sets the tessellation levels by assigning to the ‘gl_TessLevelOuter’ array. Note that the values for ‘gl_TessLevelOuter’ are floating point numbers rather than integers. They will be rounded up to the nearest integer & clamped automatically by the OpenGL system.

Tessellation Array

The first element in the array defines the number of isolines that will be generated. Each isoline will have a constant value for ‘v.’ Since we need to create a single curve, the value of ‘gl_TessLevelOuter[0]’ should be one. The second one defines the number of line segments that will be produced in the line strip. Each vertex in the strip will have a value for the parametric ‘u’ coordinate that will vary from zero to one. See the lecture slides for an explanation on what the ‘u’ coordinate is):

Code

```
#version 460

layout( vertices=4 ) out;

uniform int NumSegments;
uniform int NumStrips;

void main()
{
    // Pass along the vertex position unmodified
    gl_out[gl_InvocationID].gl_Position = gl_in[gl_InvocationID].gl_Position;

    // Some drivers (e.g. Intel) treat these levels incorrectly.  The OpenGL spec
    // says that level 0 should be the number of strips and level 1 should be
    // the number of segments per strip.  Unfortunately, not all drivers do this.
    // If this example doesn't work for you, try switching the right
    // hand side of the two assignments below.
    gl_TessLevelOuter[0] = float(NumStrips);
    gl_TessLevelOuter[1] = float(NumSegments);
}
```

Tessellation Evaluation Shader (TES)

Isolines Layout Input

This indicates the type of subdivision that is performed by the tessellation primitive generator. Other possibilities here include quads and triangles.

GL Tessellation Coordinates

Within the main function of the TES, the ‘gl_TessCoord’ variable contains the tessellation's ‘u’ & ‘v’ coordinates for this invocation. As we are only tessellating in one dimension, we only need the ‘u’ coordinate, which corresponds to the ‘x’ coordinate of ‘gl_TessCoord.’

Patch Primitive Points

The next step accesses the positions of the four control points (all the points in our patch primitive). These are available in the ‘gl_in’ array.

Bernstein Polynomials, Interpolation & Position Output

The cubic Bernstein polynomials are then evaluated at ‘u’ and stored in ‘b0,’ ‘b1,’ ‘b2,’ & ‘b3.’ Next, we compute the interpolated position using the Bezier curve equation. The final position is converted to clip coordinates and assigned to the ‘gl_Position’ output variable.

Code

```
#version 460

layout( isolines ) in;

uniform mat4 MVP;

void main()
{
    float u = gl_TessCoord.x;

    vec3 p0 = gl_in[0].gl_Position.xyz;
    vec3 p1 = gl_in[1].gl_Position.xyz;
    vec3 p2 = gl_in[2].gl_Position.xyz;
    vec3 p3 = gl_in[3].gl_Position.xyz;

    float u1 = (1.0 - u);
    float u2 = u * u;

    // Bernstein polynomials
    float b3 = u2 * u;
    float b2 = 3.0 * u2 * u1;
    float b1 = 3.0 * u * u1 * u1;
    float b0 = u1 * u1 * u1;

    // Cubic Bezier interpolation
    vec3 p = p0 * b0 + p1 * b1 + p2 * b2 + p3 * b3;

    gl_Position = MVP * vec4(p, 1.0);

}
```

Fragment Shader

```
#version 460

uniform vec4 LineColor;

layout ( location = 0 ) out vec4 FragColor;

void main()
{
    FragColor = LineColor;
}
```

Solid Shaders

Overview

The next 2 shaders are used for the control points:

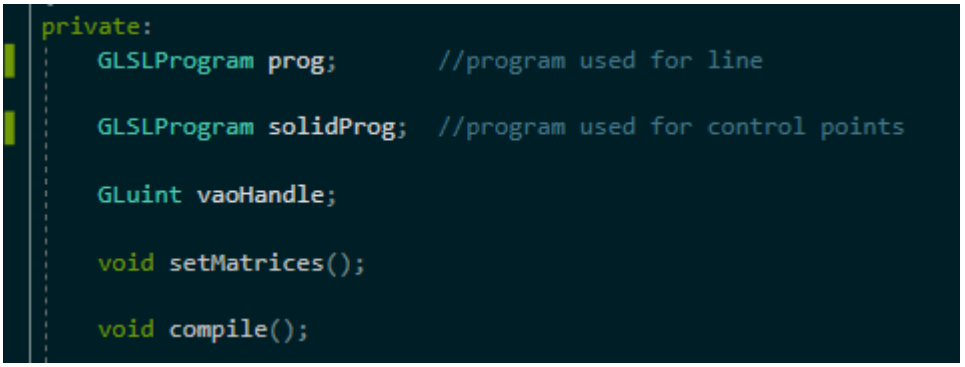
Solid Vertex Shader

```
#version 460

layout (location = 0 ) in vec2 VertexPosition;

uniform mat4 MVP;

void main()
{
    gl_Position = MVP * vec4(VertexPosition, 0.0, 1.0);
}
```

Solid Fragment Shader

```
#version 460

uniform vec4 Color;

layout ( location = 0 ) out vec4 FragColor;

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

Scenebasic Uniform Header



Scenebasic Uniform CPP

```cpp
void SceneBasic_Uniform::initScene()
{
    compile();

    glEnable(GL_DEPTH_TEST);

    float c = 3.5f;
    projection = glm::ortho(-0.4f * c, 0.4f * c, -0.3f * c, 0.3f * c, 0.1f, 100.0f);
    glPointSize(10.0f);

    // Set up patch VBO
    float v[] = { -1.0f, -1.0f, -0.5f, 1.0f, 0.5f, -1.0f, 1.0f, 1.0f };

    GLuint vboHandle;
    glGenBuffers(1, &vboHandle);

    glBindBuffer(GL_ARRAY_BUFFER, vboHandle);
    glBufferData(GL_ARRAY_BUFFER, 8 * sizeof(float), v, GL_STATIC_DRAW);

    // Set up patch VAO
    glGenVertexArrays(1, &vaoHandle);
    glBindVertexArray(vaoHandle);

    glBindBuffer(GL_ARRAY_BUFFER, vboHandle);
    glVertexAttribPointer(0, 2, GL_FLOAT, GL_FALSE, 0, 0);
    glEnableVertexAttribArray(0);

    glBindVertexArray(0);

    // Set the number of vertices per patch.  IMPORTANT!!
    glPatchParameteri(GL_PATCH_VERTICES, 4);

    // Segments and strips may be inverted on NVIDIA
    prog.use();
    prog.setUniform("NumSegments", 50);
    prog.setUniform("NumStrips", 1);
    prog.setUniform("LineColor", vec4(1.0f, 1.0f, 0.5f, 1.0f));

    solidProg.use();
    solidProg.setUniform("Color", vec4(0.5f, 1.0f, 1.0f, 1.0f));
}
```

Compile Function

```
void SceneBasic_Uniform::compile()
{
    try {
        prog.compileShader("shader/basic_uniform.vert");
        prog.compileShader("shader/basic_uniform.frag");
        prog.compileShader("shader/basic_uniform.tes");
        prog.compileShader("shader/basic_uniform.tcs");
        prog.link();
        prog.use();

        solidProg.compileShader("shader/solid.vs");
        solidProg.compileShader("shader/solid.fs");
        solidProg.link();

    } catch (GLSLProgramException &e) {
        cerr << e.what() << endl;
        exit(EXIT_FAILURE);
    }
}
```

Render Function

```
void SceneBasic_Uniform::render()
{
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

    vec3 cameraPos(0.0f, 0.0f, 1.5f);
    view = glm::lookAt(cameraPos,
        vec3(0.0f, 0.0f, 0.0f),
        vec3(0.0f, 1.0f, 0.0f));

    model = mat4(1.0f);

    glBindVertexArray(vaoHandle);
    setMatrices();

    // Draw the curve
    prog.use();
    glDrawArrays(GL_PATCHES, 0, 4);

    // Draw the control points
    solidProg.use();
    glDrawArrays(GL_POINTS, 0, 4);

    glFinish();
}
```

SetMatrices Function

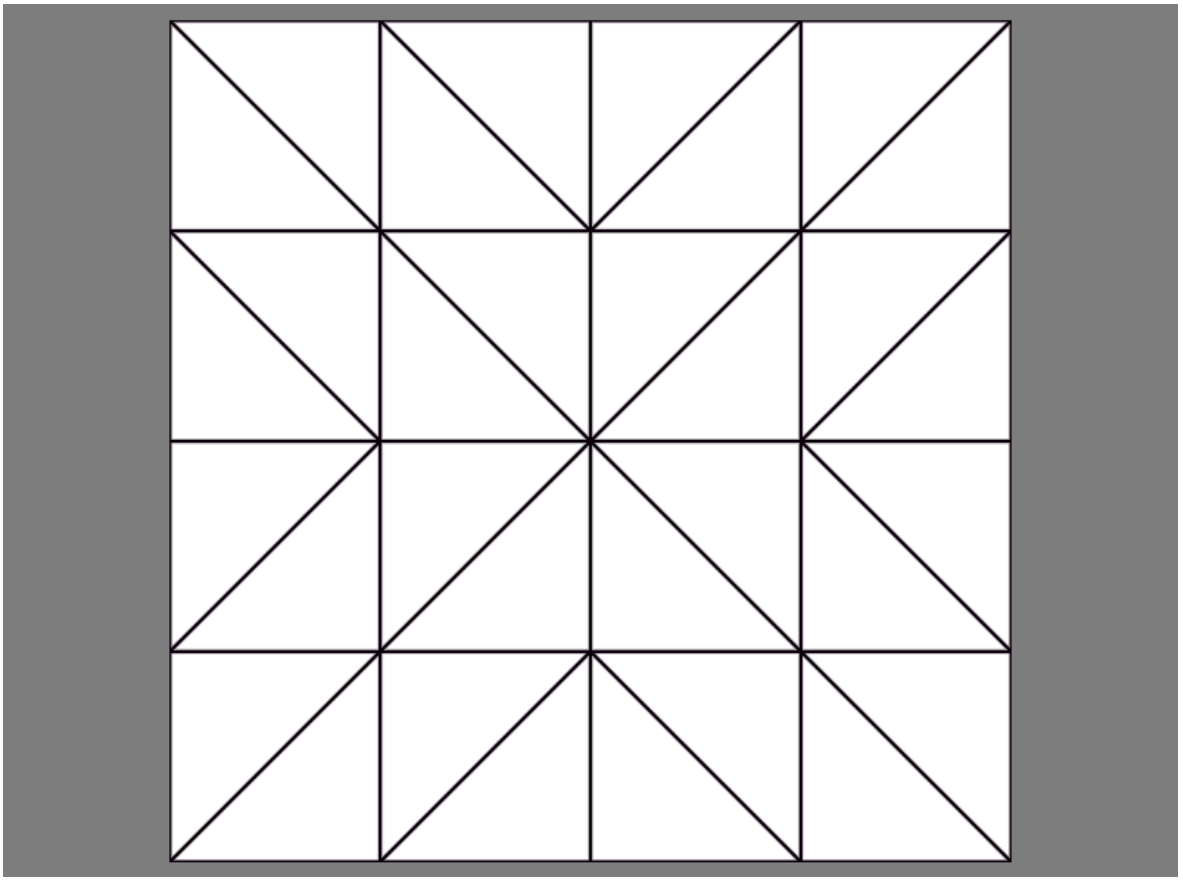
```
void SceneBasic_Uniform::setMatrices()
{
    mat4 mv = view * model;
    prog.use();
    prog.setUniform("MVP", projection * mv);
    solidProg.use();
    solidProg.setUniform("MVP", projection * mv);
}
```

Resize Function

```
void SceneBasic_Uniform::resize(int w, int h)
{
    glViewport(0, 0, w, h);
}
```

2D Quad Tessellation

Result



Vertex Shader

Use the vertex shader from curve tessellation implementation.

Tessellation Control Shader (TCS)

In the main function, the TCS passes the position of the vertex without modification & sets the inner & outer tessellation levels. All four of the outer tessellation levels are set to the value of ‘Outer’ & both of the inner tessellation levels are set to ‘Inner.’

```
#version 460

layout( vertices=4 ) out;

uniform int Outer;
uniform int Inner;

void main()
{
    // Pass along the vertex position unmodified
    gl_out[gl_InvocationID].gl_Position = gl_in[gl_InvocationID].gl_Position;

    gl_TessLevelOuter[0] = float(Outer);
    gl_TessLevelOuter[1] = float(Outer);
    gl_TessLevelOuter[2] = float(Outer);
    gl_TessLevelOuter[3] = float(Outer);

    gl_TessLevelInner[0] = float(Inner);
    gl_TessLevelInner[1] = float(Inner);
}
```

Tessellation Evaluation Shader (TES)

Layout Input Parameters

There are 3 required layout into parameters: The ‘quads’ parameter that indicates that the tessellation-primitive generator should tessellate the parameter space using quad tessellation, the ‘equal_spacing’ parameter that indicates that the tessellation should be performed such that all subdivisions have equal length & the ‘ccw’ parameter that indicates that the primitives should be generated with counter clockwise winding

Coordinate & Position Retrieval

The main function in the TES starts by retrieving the parametric coordinates for this vertex by accessing the ‘gl_TessCoord’ variable. Then we move on to read the positions of the four vertices in the patch from the ‘gl_in’ array. We store them in temporary variables to be used in the interpolation calculation.

Interpolation & Output

The built-in ‘gl_Position’ output variable then gets the value of the interpolated point using the preceding equation. Finally, we convert the position into clip coordinates by multiplying by the model-view-projection matrix:

Code

```
#version 460

layout( quads, equal_spacing, ccw ) in;

uniform mat4 MVP;

void main()
{
    float u = gl_TessCoord.x;
    float v = gl_TessCoord.y;

    vec4 p00 = gl_in[0].gl_Position;
    vec4 p10 = gl_in[1].gl_Position;
    vec4 p11 = gl_in[2].gl_Position;
    vec4 p01 = gl_in[3].gl_Position;

    // Linear interpolation
    gl_Position =
        p00 * (1.0-u) * (1.0-v) +
        p10 * u * (1.0-v) +
        p01 * v * (1.0-u) +
        p11 * u * v;

    // Transform to clip coordinates
    gl_Position = MVP * gl_Position;
}
```

Geometry Shader

```
#version 460

layout( triangles ) in;
layout( triangle_strip, max_vertices = 3 ) out;

noperspective out vec3 EdgeDistance;

uniform mat4 ViewportMatrix;  // Viewport matrix

void main()
{
    // Transform each vertex into viewport space
    vec3 p0 = vec3(ViewportMatrix * (gl_in[0].gl_Position / gl_in[0].gl_Position.w));
    vec3 p1 = vec3(ViewportMatrix * (gl_in[1].gl_Position / gl_in[1].gl_Position.w));
    vec3 p2 = vec3(ViewportMatrix * (gl_in[2].gl_Position / gl_in[2].gl_Position.w));

    float a = length(p1 - p2);
    float b = length(p2 - p0);
    float c = length(p1 - p0);
    float alpha = acos( (b*b + c*c - a*a) / (2.0*b*c) );
    float beta = acos( (a*a + c*c - b*b) / (2.0*a*c) );
    float ha = abs( c * sin( beta ) );
    float hb = abs( c * sin( alpha ) );
    float hc = abs( b * sin( alpha ) );

    EdgeDistance = vec3( ha, 0, 0 );
    gl_Position = gl_in[0].gl_Position;
    EmitVertex();

    EdgeDistance = vec3( 0, hb, 0 );
    gl_Position = gl_in[1].gl_Position;
    EmitVertex();

    EdgeDistance = vec3( 0, 0, hc );
    gl_Position = gl_in[2].gl_Position;
    EmitVertex();

    EndPrimitive();
}
```

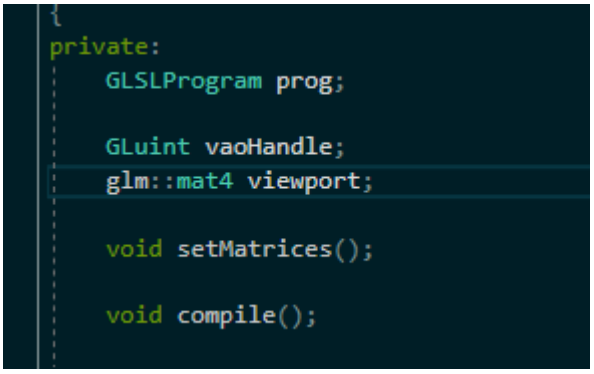
Fragment Shader

The code is similar to that used in the wireframe implementation, however with the omission of the Blinn Phong calculations:

```
#version 460

uniform float LineWidth;
uniform vec4 LineColor;
uniform vec4 QuadColor;

noperspective in vec3 EdgeDistance;

layout ( location = 0 ) out vec4 FragColor;

float edgeMix()
{
    // Find the smallest distance
    float d = min( min( EdgeDistance.x, EdgeDistance.y ), EdgeDistance.z );

    if( d < LineWidth - 1 ) {
        return 1.0;
    } else if( d > LineWidth + 1 ) {
        return 0.0;
    } else {
        float x = d - (LineWidth - 1);
        return exp2(-2.0 * (x*x));
    }
}

void main()
{
    float mixVal = edgeMix();

    FragColor = mix( QuadColor, LineColor, mixVal );
}
```

Scenebasic Uniform Header

The code is similar to that used in the curve tessellation implementation, however with the addition of a viewport matrix for drawing the wireframe:



Scenebasic Uniform CPP

Init Scene Function

```cpp
void SceneBasic_Uniform::initScene()
{
    compile();

    glClearColor(0.5f, 0.5f, 0.5f, 1.0f);

    glEnable(GL_DEPTH_TEST);

    float c = 3.5f;
    projection = glm::ortho(-0.4f * c, 0.4f * c, -0.3f * c, 0.3f * c, 0.1f, 100.0f);

    ////////////// Uniforms ////////////////////
    prog.setUniform("Inner", 4);
    prog.setUniform("Outer", 4);
    prog.setUniform("LineWidth", 1.5f);
    prog.setUniform("LineColor", vec4(0.05f, 0.0f, 0.05f, 1.0f));
    prog.setUniform("QuadColor", vec4(1.0f, 1.0f, 1.0f, 1.0f));
    ////////////////////////////////////////////

    // Set up patch VBO
    float v[] = { -1.0f, -1.0f, 1.0f, -1.0f, 1.0f, 1.0f, -1.0f, 1.0f };

    GLuint vboHandle;
    glGenBuffers(1, &vboHandle);

    glBindBuffer(GL_ARRAY_BUFFER, vboHandle);
    glBufferData(GL_ARRAY_BUFFER, 8 * sizeof(float), v, GL_STATIC_DRAW);

    // Set up patch VAO
    glGenVertexArrays(1, &vaoHandle);
    glBindVertexArray(vaoHandle);

    glBindBuffer(GL_ARRAY_BUFFER, vboHandle);
    glVertexAttribPointer(0, 2, GL_FLOAT, GL_FALSE, 0, 0);
    glEnableVertexAttribArray(0);

    glBindVertexArray(0);

    // Set the number of vertices per patch.  IMPORTANT!!
    glPatchParameteri(GL_PATCH_VERTICES, 4);

    GLint maxVerts;
    glGetIntegerv(GL_MAX_PATCH_VERTICES, &maxVerts);
    printf("Max patch vertices: %d\n", maxVerts);
}
```

Render Function

```
void SceneBasic_Uniform::render()
{
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

    vec3 cameraPos(0.0f, 0.0f, 1.5f);
    view = glm::lookAt(cameraPos,
        vec3(0.0f, 0.0f, 0.0f),
        vec3(0.0f, 1.0f, 0.0f));

    model = mat4(1.0f);
    setMatrices();

    glBindVertexArray(vaoHandle);
    glDrawArrays(GL_PATCHES, 0, 4);

    glFinish();
}
```

Set Matrices Function

```
void SceneBasic_Uniform::setMatrices()
{
    mat4 mv = view * model;
    prog.setUniform("MVP", projection * mv);
    prog.setUniform("ViewportMatrix", viewport);
}
```

Resize Function

```
void SceneBasic_Uniform::resize(int w, int h)
{
    glViewport(0, 0, w, h);

    float w2 = w / 2.0f;
    float h2 = h / 2.0f;
    viewport = mat4(vec4(w2, 0.0f, 0.0f, 0.0f),
        vec4(0.0f, h2, 0.0f, 0.0f),
        vec4(0.0f, 0.0f, 1.0f, 0.0f),
        vec4(w2 + 0, h2 + 0, 0.0f, 1.0f));
}
```

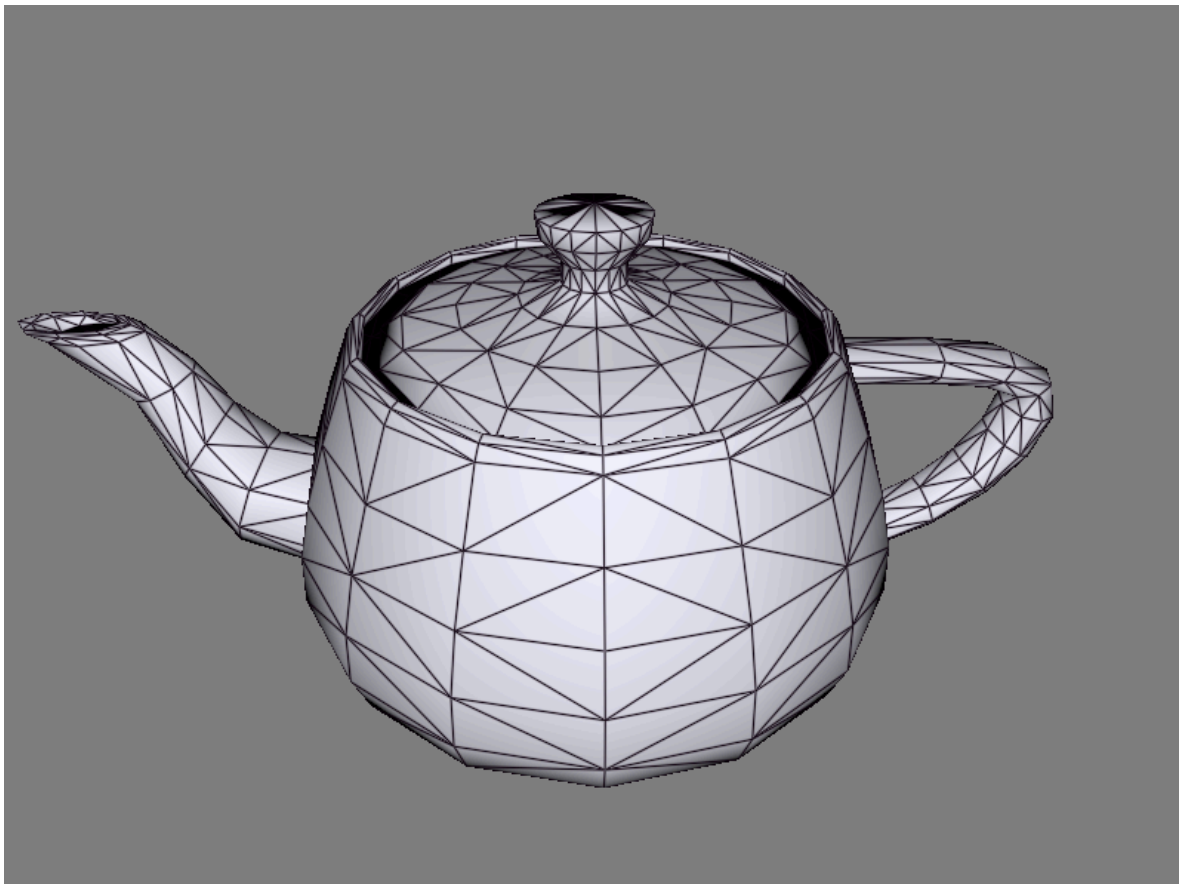
Compile Function

```
void SceneBasic_Uniform::compile()
{
    try {
        prog.compileShader("shader/basic_uniform.vert");
        prog.compileShader("shader/basic_uniform.frag");
        prog.compileShader("shader/basic_uniform.gs");
        prog.compileShader("shader/basic_uniform.tes");
        prog.compileShader("shader/basic_uniform.tcs");
        prog.link();
        prog.use();

    } catch (GLSLProgramException &e) {
        cerr << e.what() << endl;
        exit(EXIT_FAILURE);
    }
}
```

Tessellating a 3D Surface

Result

Vertex Shader



Vertex Shader

```
#version 460

layout (location = 0 ) in vec3 VertexPosition;

void main()
{
    gl_Position = vec4(VertexPosition, 1.0);
}
```

Tessellation Control Shader (TCS)

The code is similar to that used in the 2D quad tessellation implementation, but instead a tessellation level is used:

```
#version 460

layout( vertices=16 ) out;

uniform int TessLevel;

void main()
{
    // Pass along the vertex position unmodified
    gl_out[gl_InvocationID].gl_Position = gl_in[gl_InvocationID].gl_Position;

    gl_TessLevelOuter[0] = float(TessLevel);
    gl_TessLevelOuter[1] = float(TessLevel);
    gl_TessLevelOuter[2] = float(TessLevel);
    gl_TessLevelOuter[3] = float(TessLevel);

    gl_TessLevelInner[1] = float(TessLevel);
    gl_TessLevelInner[0] = float(TessLevel);
}
```

Tessellation Evaluation Shader (TES)

Global Variables

```
#version 460

layout( quads ) in;

out vec3 TENormal;
out vec4 TEPosition;


uniform mat4 MVP;
uniform mat4 ModelViewMatrix;
uniform mat3 NormalMatrix;
```

Basis Functions Function

Evaluate the Bernstein polynomials & their derivatives for a given value of the ‘t’ parameter. The results should be returned in the ‘b’ and ‘db’ output parameters:

```
void basisFunctions( out float[4] b, out float[4] db, float t )
{
    float t1 = (1.0 - t);
    float t12 = t1 * t1;

    b[0] = t12 * t1;
    b[1] = 3.0 * t12 * t;
    b[2] = 3.0 * t1 * t * t;
    b[3] = t * t * t;

    db[0] = -3.0 * t1 * t1;
    db[1] = -6.0 * t * t1 + 3.0 * t12;
    db[2] = -3.0 * t * t + 6.0 * t * t1;
    db[3] = 3.0 * t * t;
}
```

Main Function

Tessellation Coordinate & Patch Vertices Reassignment

Assign the tessellation coordinates to the ‘u’ & ‘v’ variables, & reassign all 16 of the patch vertices to variables with shorter names in order to shorten the code that is written after.

```
void main()
{
    float u = gl_TessCoord.x;
    float v = gl_TessCoord.y;

    // Reassign
    vec4 p00 = gl_in[0].gl_Position;
    vec4 p01 = gl_in[1].gl_Position;
    vec4 p02 = gl_in[2].gl_Position;
    vec4 p03 = gl_in[3].gl_Position;
    vec4 p10 = gl_in[4].gl_Position;
    vec4 p11 = gl_in[5].gl_Position;
    vec4 p12 = gl_in[6].gl_Position;
    vec4 p13 = gl_in[7].gl_Position;
    vec4 p20 = gl_in[8].gl_Position;
    vec4 p21 = gl_in[9].gl_Position;
    vec4 p22 = gl_in[10].gl_Position;
    vec4 p23 = gl_in[11].gl_Position;
    vec4 p30 = gl_in[12].gl_Position;
    vec4 p31 = gl_in[13].gl_Position;
    vec4 p32 = gl_in[14].gl_Position;
    vec4 p33 = gl_in[15].gl_Position;
```

Bernstein Polynomials & Derivatives

Call the basisFunctions function to compute the Bernstein polynomials & their derivatives at ‘u’ and ‘v.’ Store the results in ‘bu,’ ‘dbu,’ ‘bv,’ & ‘dbv:’

```
// Compute basis functions
float bu[4], bv[4];   // Basis functions for u and v
float dbu[4], dbv[4]; // Derivitives for u and v
basisFunctions(bu, dbu, u);
basisFunctions(bv, dbv, v);
```

Evaluation

Evaluate the sums from the preceding equations for the position (TEPosition), the partial derivative with respect to ‘u’ (‘du’), & the partial derivative with respect to ‘v’ (‘dv’):

```
// Bezier interpolation
TEPosition =
 p00*bu[0]*bv[0] + p01*bu[0]*bv[1] + p02*bu[0]*bv[2] + p03*bu[0]*bv[3] +
 p10*bu[1]*bv[0] + p11*bu[1]*bv[1] + p12*bu[1]*bv[2] + p13*bu[1]*bv[3] +
 p20*bu[2]*bv[0] + p21*bu[2]*bv[1] + p22*bu[2]*bv[2] + p23*bu[2]*bv[3] +
 p30*bu[3]*bv[0] + p31*bu[3]*bv[1] + p32*bu[3]*bv[2] + p33*bu[3]*bv[3];
```

Normalisation

Compute the normal vector as the cross product of ‘du’ & ‘dv:’

```
// Compute the normal vector
vec4 du =
 p00*dbu[0]*bv[0] + p01*dbu[0]*bv[1] + p02*dbu[0]*bv[2] + p03*dbu[0]*bv[3] +
 p10*dbu[1]*bv[0] + p11*dbu[1]*bv[1] + p12*dbu[1]*bv[2] + p13*dbu[1]*bv[3] +
 p20*dbu[2]*bv[0] + p21*dbu[2]*bv[1] + p22*dbu[2]*bv[2] + p23*dbu[2]*bv[3] +
 p30*dbu[3]*bv[0] + p31*dbu[3]*bv[1] + p32*dbu[3]*bv[2] + p33*dbu[3]*bv[3];

vec4 dv =
 p00*bu[0]*dbv[0] + p01*bu[0]*dbv[1] + p02*bu[0]*dbv[2] + p03*bu[0]*dbv[3] +
 p10*bu[1]*dbv[0] + p11*bu[1]*dbv[1] + p12*bu[1]*dbv[2] + p13*bu[1]*dbv[3] +
 p20*bu[2]*dbv[0] + p21*bu[2]*dbv[1] + p22*bu[2]*dbv[2] + p23*bu[2]*dbv[3] +
 p30*bu[3]*dbv[0] + p31*bu[3]*dbv[1] + p32*bu[3]*dbv[2] + p33*bu[3]*dbv[3];

vec3 n = normalize( cross(du.xyz, dv.xyz) );
```

Conversion to Clip & Camera Coordinates & Output

Convert the position (‘TEPosition’) to clip coordinates & assign the result to ‘gl_Position.’ Convert it to camera coordinates before passing it to the fragment shader.

```
// Transform to clip coordinates
gl_Position = MVP * TEPosition;
```

Conversion of Normal to Camera Coordinates

Convert the normal vector to camera coordinates by multiplying it with NormalMatrix. The result is normalised & passed to the fragment shader via ‘TENormal:’

```
    // Convert to camera coordinates
    TEPosition = ModelViewMatrix * TEPosition;
    TENormal = normalize(NormalMatrix * n);
}
```

Geometry Shader

```
#version 460

layout( triangles ) in;
layout( triangle_strip, max_vertices = 3 ) out;

in vec3 TENormal[];
in vec4 TEPosition[];
noperspective out vec3 EdgeDistance;

out vec3 Normal;
out vec4 Position;

uniform mat4 ViewportMatrix;  // Viewport matrix

void main()
{
    // Transform each vertex into viewport space
    vec3 p0 = vec3(ViewportMatrix * (gl_in[0].gl_Position / gl_in[0].gl_Position.w));
    vec3 p1 = vec3(ViewportMatrix * (gl_in[1].gl_Position / gl_in[1].gl_Position.w));
    vec3 p2 = vec3(ViewportMatrix * (gl_in[2].gl_Position / gl_in[2].gl_Position.w));

    float a = length(p1 - p2);
    float b = length(p2 - p0);
    float c = length(p1 - p0);
    float alpha = acos( (b*b + c*c - a*a) / (2.0*b*c) );
    float beta = acos( (a*a + c*c - b*b) / (2.0*a*c) );
    float ha = abs( c * sin( beta ) );
    float hb = abs( c * sin( alpha ) );
    float hc = abs( b * sin( alpha ) );

    EdgeDistance = vec3( ha, 0, 0 );
    Normal = TENormal[0];
    Position = TEPosition[0];
    gl_Position = gl_in[0].gl_Position;
    EmitVertex();

    EdgeDistance = vec3( 0, hb, 0 );
    Normal = TENormal[1];
    Position = TEPosition[1];
    gl_Position = gl_in[1].gl_Position;
    EmitVertex();

    EdgeDistance = vec3( 0, 0, hc );
    Normal = TENormal[2];
    Position = TEPosition[2];
    gl_Position = gl_in[2].gl_Position;
    EmitVertex();

    EndPrimitive();
}
```

Fragment Shader

```
#version 460

uniform float LineWidth;
uniform vec4 LineColor;
uniform vec4 LightPosition;
uniform vec3 LightIntensity;
uniform vec3 Kd;

noperspective in vec3 EdgeDistance;
in vec3 Normal;
in vec4 Position;

layout ( location = 0 ) out vec4 FragColor;

vec3 diffuseModel( vec3 pos, vec3 norm )
{
    vec3 s = normalize(vec3(LightPosition) - pos);
    float sDotN = max( dot(s,norm), 0.0 );
    vec3 diffuse = LightIntensity * Kd * sDotN;

    return diffuse;
}

float edgeMix()
{
    // Find the smallest distance
    float d = min( min( EdgeDistance.x, EdgeDistance.y ), EdgeDistance.z );

    if( d < LineWidth - 1 ) {
        return 1.0;
    } else if( d > LineWidth + 1 ) {
        return 0.0;
    } else {
        float x = d - (LineWidth - 1);
        return exp2(-2.0 * (x*x));
    }
}

void main()
{
    float mixVal = edgeMix();
    vec4 color = vec4( diffuseModel( Position.xyz, Normal ), 1.0);
    color = pow( color, vec4(1.0/2.2) );
    FragColor = mix( color, LineColor, mixVal );
}
```

Scenebasic Uniform Header

```
private:
    GLSLProgram prog;

    GLuint vaoHandle;
    glm::mat4 viewport;

    TeapotPatch teapot;

    float angle, tPrev, rotSpeed;

    void setMatrices();

    void compile();
```

Scenebasic Uniform CPP

Init Scene Function

```
void SceneBasic_Uniform::initScene()
{
    compile();

    glClearColor(0.5f, 0.5f, 0.5f, 1.0f);

    glEnable(GL_DEPTH_TEST);

    angle = glm::pi<float>() / 3.0f;

    ////////////// Uniforms ////////////////////
    prog.setUniform("TessLevel", 4);
    prog.setUniform("LineWidth", 0.8f);
    prog.setUniform("LineColor", vec4(0.05f, 0.0f, 0.05f, 1.0f));
    prog.setUniform("LightPosition", vec4(0.0f, 0.0f, 0.0f, 1.0f));
    prog.setUniform("LightIntensity", vec3(1.0f, 1.0f, 1.0f));
    prog.setUniform("Kd", vec3(0.9f, 0.9f, 1.0f));
    ////////////////////////////////////////////

    glPatchParameteri(GL_PATCH_VERTICES, 16);
}
```

Render Function

```
void SceneBasic_Uniform::render()
{
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

    vec3 cameraPos(4.25f * cos(angle), 3.0f, 4.25f * sin(angle));
    view = glm::lookAt(cameraPos,
        vec3(0.0f, 0.0f, 0.0f),
        vec3(0.0f, 1.0f, 0.0f));

    model = mat4(1.0f);
    model = glm::translate(model, vec3(0.0f, -1.5f, 0.0f));
    model = glm::rotate(model, glm::radians(-90.0f), vec3(1.0f, 0.0f, 0.0f));
    setMatrices();
    teapot.render();
    glFinish();
}
```

Set Matrices Function

```
void SceneBasic_Uniform::setMatrices()
{
    mat4 mv = view * model;
    prog.setUniform("ModelViewMatrix", mv);
    prog.setUniform("NormalMatrix",
        mat3(vec3(mv[0]), vec3(mv[1]), vec3(mv[2])));
    prog.setUniform("MVP", projection * mv);
    prog.setUniform("ViewportMatrix", viewport);
}
```

Resize Function

```
void SceneBasic_Uniform::resize(int w, int h)
{
    glViewport(0, 0, w, h);
    float w2 = w / 2.0f;
    float h2 = h / 2.0f;
    viewport = mat4(vec4(w2, 0.0f, 0.0f, 0.0f),
        vec4(0.0f, h2, 0.0f, 0.0f),
        vec4(0.0f, 0.0f, 1.0f, 0.0f),
        vec4(w2 + 0, h2 + 0, 0.0f, 1.0f));
    projection = glm::perspective(glm::radians(60.0f), (float)w / h, 0.3f, 100.0f);
}
```

Appendix

2D Quad Tessellation

Tessellation Evaluation Shader (TES)

```
#version 460

layout( quads, equal_spacing, ccw ) in;

uniform mat4 MVP;

void main()
{
   float u = gl_TessCoord.x;
   float v = gl_TessCoord.y;

   vec4 p00 = gl_in[0].gl_Position;
   vec4 p10 = gl_in[1].gl_Position;
   vec4 p11 = gl_in[2].gl_Position;
   vec4 p01 = gl_in[3].gl_Position;

   // Linear interpolation
   gl_Position =
      p00 * (1.0-u) * (1.0-v) +
      p10 * u * (1.0-v) +
      p01 * v * (1.0-u) +
      p11 * u * v;

   // Transform to clip coordinates
   gl_Position = MVP * gl_Position;
}
```

Geometry Shader

```
#version 460

layout( triangles ) in;
layout( triangle_strip, max_vertices = 3 ) out;

noperspective out vec3 EdgeDistance;

uniform mat4 ViewportMatrix;  // Viewport matrix
```

```
void main()
{
   // Transform each vertex into viewport space
   vec3 p0 = vec3(ViewportMatrix * (gl_in[0].gl_Position / gl_in[0].gl_Position.w));
   vec3 p1 = vec3(ViewportMatrix * (gl_in[1].gl_Position / gl_in[1].gl_Position.w));
   vec3 p2 = vec3(ViewportMatrix * (gl_in[2].gl_Position / gl_in[2].gl_Position.w));

   float a = length(p1 - p2);
   float b = length(p2 - p0);
   float c = length(p1 - p0);
   float alpha = acos( (b*b + c*c - a*a) / (2.0*b*c) );
   float beta = acos( (a*a + c*c - b*b) / (2.0*a*c) );
   float ha = abs( c * sin( beta ) );
   float hb = abs( c * sin( alpha ) );
   float hc = abs( b * sin( alpha ) );

   EdgeDistance = vec3( ha, 0, 0 );
   gl_Position = gl_in[0].gl_Position;
   EmitVertex();

   EdgeDistance = vec3( 0, hb, 0 );
   gl_Position = gl_in[1].gl_Position;
   EmitVertex();

   EdgeDistance = vec3( 0, 0, hc );
   gl_Position = gl_in[2].gl_Position;
   EmitVertex();

   EndPrimitive();
}
```