CS 450/550 -- Fall Quarter 2023

Project #6

100 Points

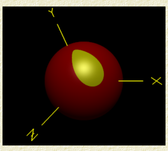
Due: November 29

Shaders

This page was last updated: November 13, 2023

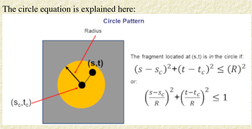
Introduction:

The goal of this project is to create an animated ellipse pattern by using an OpenGL fragment shader. The center of the ellipse will be animated using the KeyTime method you already know. The S-radius and T-radius will be animated using the Time variable and some functions that you invent. Light the object using per-fragment lighting.

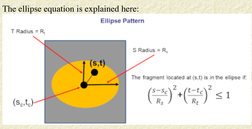


Note: the reason this pattern looks more teardrop-shaped than ellipse-shaped is that the nature of spheres is to distort patterns near the poles. That Mercator-projection map you always had in the front of the classroom in middle and high school showed this effect in how greatly it distorted the size of Greenland.

The circle equation is explained here:



The ellipse equation is explained here:



Learning Objective:

When you are done with this assignment, you will understand how to create a GPU-program, known as a *shader*. Your shader will let you dynamically alter a color pattern on an arbitrary object. Today's games and movies employ hundreds of shader programs to get the variety of special effects they need. This assignment will also set you up to succeed in CS 457/557, our shaders course which you will be able to take in the Winter Quarter.

Instructions:

1. Draw a 3D object (your choice). Be sure that it has (s,t) texture coordinates in it. All of the osu* geometry functions do.
2. Use the GLSLProgram C++ class to create a shader from the pattern.vert and pattern.frag files.
3. Write a pattern.vert vertex shader that transforms the vertex and passes, to the fragment shader, the vST texture coordinates, the vN normal vector, the vE eye vector, and the vL light vector.
4. Write a fragment shader that accepts, from the vertex shader, the vST texture coordinates, the vN normal vector, the vE eye vector, and the vL light vector.
5. The fragment shader also needs to read in, from the uniform variables, the per-fragment lighting information, uKa, uKd, uKs, uColor, uSpecularColor, and uShininess. These can be set once after the shader has been created and left alone after that. **InitGraphics( )** is a good place to do this.
6. The fragment shader also needs to read in, from the uniform variables, the ellipse center uSc and uTc, and the ellipse radii uRs and uRt. These will be set every time Display( ) is called.
7. The first half of the fragment shader needs to look at that fragment's vST texture coordinates and decide, from the ellipse equation, if that fragment gets the ellipse color or the rest-of-the-object color.
8. The second half of the fragment shader applies per-fragment lighting to that chosen color.
9. Use the SetUniformVariable( ) method to set the uniform variables.
10. Use the **in** and **out** keywords to pass 4 variabes from the vertex shader to the fragment shader. Those 4 variables will be interpolated in the rasterizer so that each fragment gets its own copy of them.
11. The choice of colors is up to you.
12. For the keytime-varying ellipse center, pick at least 4 keytime positions for Sc and Tc. Remember that they need to be between 0. and 1.
13. For the Time-varying ellipse center, some equations that will cause the Rs and Rt variables to stretch and shrink. Sine functions work well here, but you can choose anything that works. Remember that these need to be between 0. and 1.
14. You must have the ability to show the pattern animating and not animating. One way to control this is with keyboard keys:

'k'	Toggle the keytime-based animation on and off
't'	Toggle the Time-based animation on and off

The GLSLProgram C++ class

The glslprogram.h and glslprogram.cpp files are already in your Sample folder. You just need to un-comment their #include's.

See the class Shader notes for how to use them.

A Head Start

To help you get started, here are some head starts to work from:

- pattern.vert
- pattern.frag
- sample.cpp

These implement the square shader shown in the Shader notes.

Per-Fragment Lighting

You need to do per-fragment lighting.
Here is what needs to be added to the **vertex shader** code to enable per-fragment lighting:

```
// make this 120 for the mac:
#version 330 compatability

// out variables to be interpolated in the rasterizer and sent to each fragment shader:

out  vec3  vN;          // normal vector
out  vec3  vL;          // vector from point to light
out  vec3  vE;          // vector from point to eye
out  vec2  vST;         // (s,t) texture coordinates

// where the light is:

const vec3 LightPosition = vec3(  0., 5., 5. );

void
main( )
{
        vST = gl_MultiTexCoord0.st;
        vec4 ECposition = gl_ModelViewMatrix * gl_Vertex;
        vN = normalize( gl_NormalMatrix * gl_Normal );  // normal vector
        vL = LightPosition - ECposition.xyz;            // vector from the point
                                                        // to the light position
        vE = vec3( 0., 0., 0. ) - ECposition.xyz;       // vector from the point
                                                        // to the eye position
        gl_Position = gl_ModelViewProjectionMatrix * gl_Vertex;
}
```

Here is what needs to be added to the **fragment shader** code to enable per-fragment lighting:

```
// make this 120 for the mac:
#version 330 compatability

// lighting uniform variables -- these can be set once and left alone:
uniform float   uKa, uKd, uKs;          // coefficients of each type of lighting -- make sum to 1.0
uniform vec3    uColor;                 // object color
uniform vec3    uSpecularColor;         // light color
uniform float   uShininess;             // specular exponent

// ellipse-equation uniform variables -- these should be set every time Display( ) is called:

uniform float   uSc, uTc;
uniform float   uRs, uRt;

// in variables from the vertex shader and interpolated in the rasterizer:

in  vec3  vN;                   // normal vector
in  vec3  vL;                   // vector from point to light
in  vec3  vE;                   // vector from point to eye
in  vec2  vST;                  // (s,t) texture coordinates


void
main( )
{
        vec3 Normal = normalize(vN);
        vec3 Light  = normalize(vL);
        vec3 Eye    = normalize(vE);
        float s = vST.s;
        float t = vST.t;

        // determine the color using the ellipse equation:

        vec3 myColor = uColor;
        if( ??? )
        {
                myColor = ???;
        }

        // apply the per-fragmewnt lighting to myColor:

        vec3 ambient = uKa * myColor;

        float d = max( dot(Normal,Light), 0. );       // only do diffuse if the light can see the point
        vec3 diffuse = uKd * d * myColor;

        float s = 0.;
        if( dot(Normal,Light) > 0. )                  // only do specular if the light can see the point
        {
                vec3 ref = normalize(  reflect( -Light, Normal )  );
                s = pow( max( dot(Eye,ref),0. ), uShininess );
        }
        vec3 specular = uKs * s * uSpecularColor;
        gl_FragColor = vec4( ambient + diffuse + specular,  1. );
}
```

So, the first half of the fragment shader determines the color to use based on if that fragment is inside the pattern or not. The second half of the fragment shader code uses that color in the per-fragment lighting. The final value assigned to gl_FragColor will be an RGBA that has both the pattern color and the lighting intensity.

Turning Animation Effects On and Off

I recommend that you pass in your pattern parameters as uniform variables and just control when you give them new values. Thus, in Display( ), you might say:

```
        Pattern.Use( );
        if( KeytimePatternOn )
        {
                float sc = Sc.GetValue( Time );
                float tc = Tc.GetValue( Time );
        }
        if( TimePatternOn )
        {
                float rs = << some function of the Time variable >>
                float rt = << some function of the Time variable >>
        }
        Pattern.SetUniformVariable( "uSc", sc );
        Pattern.SetUniformVariable( "uTc", tc );
        Pattern.SetUniformVariable( "uRs", rs );
        Pattern.SetUniformVariable( "uRt", rt );

        << do the drawing >>

        Pattern.UnUse( );       // Pattern.Use(0);  also works
```

Turn-in:

Use the Teach system to turn in your:

1. .cpp file
2. .vert file
3. .frag file
4. A PDF report containing:
   - Project number and title
   - Your name and email address
   - A description of what you did to get the display you got
   - A table showing your keytime values for USc and uTc. It is OK just to include the lines of code that set them.
   - A couple of cool-looking screen shots from your program.
   - Tell us what convinces you that your animation is indeed doing what you set it up to do.
   - The link to your video demonstrating that your project does what the requirements ask for. If you can, we'd appreciate it if you'd narrate your video so that you can tell us what it is doing.

Grading:

Item	Points
Ellipse pattern is drawn	25
Ellipse center is keytime-animated	25
Ellipse radii are Time-equation-animated	25
Per-fragment lighting works	25
Potential Total	100