LSU EE 4702-1 Homework 3 Solution Due: 16 November 2012

Follow the Programming Homework Work Flow *instructions on the procedures page, substituting* `hw3` *for* `hw1`*. Also, update the include directory.*
Use `http://www.ece.lsu.edu/gp/refs/GLSLangSpec.4.30.6.pdf`*, the OpenGL 4.3 Shading Language spec, as a reference. The code in this assignment requires OpenGL 4.3, which should be installed on* `snow`*,* `ice`*, and* `frost`*.*

The solution has been checked into the repository as new files hw3-sol.cc and hw3-shdr-sol.cc. The solution is also in the Web at `http://www.ece.lsu.edu/koppel/gpup/2012/hw3-shdr-sol.cc.html` *and* `http://www.ece.lsu.edu/koppel/gpup/2012/hw3-sol.cc.html`*.*

Problem 0: Compile and run the code unmodified, a yellow helix should be displayed. The arrow keys initially move the eye position, familiarize yourself with these keys, the others in the comments of `hw3.cc`, and the keys described in this problem.

The code is set up to run two different shader programs, called "SP Geo Shade1" and "SP Geo Shade2." The name of the active shader program is displayed on the lowest green line, pressing "s" will switch from one to the other. The two shader programs use the same vertex and fragment shader, but different geometry shaders. The problems indicate which geometry shader to modify.

For this assignment there are three user-controlled variables of interest. (See the Variables in the Keyboard Commands section of the comments in hw3.cc.) *Segs Per Helix Rev* controls the number of segments in one revolution of the helix. Larger numbers mean smaller triangles. *Segs Per Wire Rev* controls the number of segments in one revolution of the wire. Larger numbers mean smaller triangles. *Door Angle* See Problem 3.

Problem 1: The unmodified version of the code does everything needed to display a texture on the helix, except for sending texture coordinates. (The texture is this assignment.) Rather than generating texture coordinates on the CPU and sending them to the GPU, the goal here is to compute the texture coordinates on the GPU using a vertex shader. The vertex shader in `hw3-shder.cc` currently just copies the texture coordinates from the CPU unmodified. Since the CPU never sends texture coordinates these will hold some initial value, perhaps $(0, 0)$.

Modify the vertex shader in `hw3-shdr.cc` so that it computes texture coordinates, instead of reading them from `gl_MultiTexCoord0`.

- Display the texture so that it appears right-side up, undistorted, and is recognizable.
- Optional: The size and position of the texture should not change when number of segments per helix or wire are changes.

Note: This problem is easy. Texture coordinates can be computed in terms of the values in `helix_index`. For the optional part new uniform variables need to be defined.

As the statement problem came close to explaining, the texture coordinates can be found by multiplying the helix indices by some constants. In the solution these are computed on the CPU and put in a two-element uniform vector, `texture_scale`:

```
const float scale_x = - 20.0 / seg_per_helix_revolution;
const float scale_y = 1.0f / seg_per_wire_revolution;
glUniform2f(4,scale_x,scale_y); GE();
```

In the vertex shader these are used to compute the texture coordinates:

```
gl_TexCoord[0] = texture_scale * helix_index;
```

See the code for details.

Problem 2: Both geometry shaders assign a variable named `type_a`, but the variable is not used. Modify `gs_main_helix` and other code so that textures are not displayed for triangles for which `type_a` is true. Do this by declaring new variables for communicating the value of `type_a` from the geometry shader to the fragment shader. Use this variable in the fragment shader to avoid doing a texture lookup.

- Declare a new variable for communicating between the geometry shader and the fragment shader. **Do not** use an existing variable, tempting though it may be.
- Set the appropriate interpolation for the new variable.
- Use the variable in the fragment shader to control whether textures are used.

Note: The fragment shader already avoids texture lookup for the back side of primitives, it also applies a reddish tint to such primitives. Make sure that reddish tint is not applied to the front of primitives for which texture lookup has been suppressed.

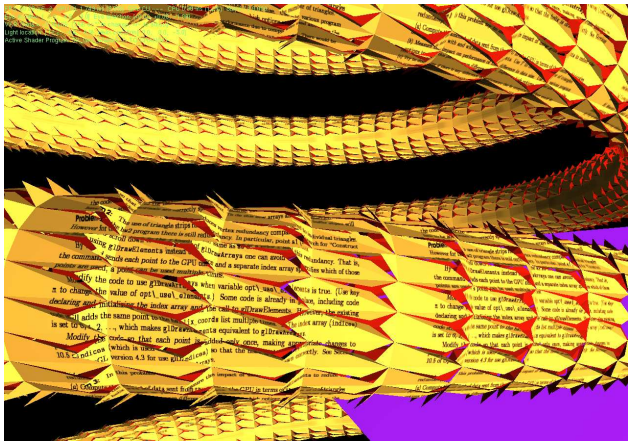
See Section 4.3.4 and 4.3.6 of the OpenGL 4.3 specification for details on how to declare shader input and output variables.

This problem is also fairly easy, but it's not a one-liner.

The solution is to add a new variable to the output of the geometry shader and the input to the fragment shader. The variable is type bool and is given the interpolation qualifier, `flat`. Some sample changes are shown below, see the code for details.

```
#ifdef _GEOMETRY_SHADER_
out Data
{
  vec3 normal_e;
  vec4 vertex_e;
  flat bool no_texture;  /// SOLUTION - Problem 2
};
#endif
#ifdef _FRAGMENT_SHADER_
in Data
{
  vec3 normal_e;
  vec4 vertex_e;
  flat bool no_texture;    /// SOLUTION - Problem 2
};
#endif
```

Problem 3: Modify geometry shader `gs_main_helix2` so that each `type_a` helix triangle is split into three triangles under control of CPU variable `opt_door_angle`, call the value of that variable θ. Each new triangle consists of two vertices from the original triangle, and (initially) the original triangle center. These new triangles will rotate, the amount based on θ, around an axis formed by the original edge. Though the triangles rotate, their shape should not change. The appearance might be of doors opening up, see the screen shot to the right.



More precisely, let p_0, p_1, and p_2 denote the vertices of the original triangle, and let $c = (p_0 + p_1 + p_2)/3$ denote the centroid, $\overrightarrow{n} = \overrightarrow{p_0p_1} \times \overrightarrow{p_0p_2}$ denote the triangle normal, and $\hat{n}$ be the normalized normal. The original triangle, $p_0p_1p_2$, is to be split into new triangles $p_0p_1c_2$, $p_0c_1p_2$, and $c_0p_1p_2$ where c_0 is given by

$$c_0 = q_0 + \overrightarrow{q_0c} \cos\theta + l_0\hat{n} \sin\theta,$$

where q_0 is the point on line p_1p_2 closest to c and l_0 is the distance from q_0 to c. Points c_1 and c_2 are defined similarly.

Geometry shader `gs_main_helix2` should emit these three triangles.

- Using an appropriate method, provide a value of CPU variable `opt_door_angle` to the shader.
- Calculation of triangle coordinates should not be performed on clip-space coordinates.
- The `normal_e` output of the geometry shader should be for the respective triangle, not the wire.
- Note that the output of the geometry shader should be both clip-space coordinates (for the fixed-functionality) and eye-space coordinates (for our fragment shader).

See the OpenGL Shading Language 4.3 specification for information on the shader language, including functions for dot product, cross product, and normalization.

Here are some tips that apply to all problems:

- Shader program error messages appear when the main program starts, not when code is compiled.
- The error message line number might be off by one.
- The following shader code error message can be ignored, (but don't ignore other messages that occur with it): "warning C7547: extension GL EXT geometry shader4 not supported in profile gp5fp."

An excerpt from the geometry shader code appears below. See the code for additional changes.

```
void
gs_main_helix2()
```

```
{
  /// SOLUTION - Problem 3

  // Select which triangles to split into "trap doors".
  //
  const bool trap_door = In[0].hidx.x >= In[2].hidx.x;

  // Set door angle (0 is fully closed, pi/2 is straight out, etc.
  //
  float th = trap_door ? theta : 0;

  // Find center of triangle (in eye-space coordinates).
  //
  vec3 pctr = (In[0].vertex_e + In[1].vertex_e
               + In[2].vertex_e ).xyz * 0.333333f;

  // Find texture coordinate of center of triangle.
  //
  vec4 tctr = ( gl_TexCoordIn[0][0] + gl_TexCoordIn[1][0]
                + gl_TexCoordIn[2][0] ) * 0.333333f;

  // Generate the three "trap door" triangles.
  //
  for ( int tri=0; tri<3; tri++ )
    {
      // Note that calculations performed in eye-space coordinates. If
      // the calculations were performed in clip-space coordinates
      // (for example, using gl_PositionIn) results would be distorted
      // due to the perspective projection.

      // Select two "fixed" vertices.
      //
      vec3 p0 = In[(0+tri)%3].vertex_e.xyz;
      vec3 p1 = In[(1+tri)%3].vertex_e.xyz;

      // Select the "swinging" vertex.
      //
      int p2_idx = (2+tri)%3;
      vec3 p2 = pctr;

      // Compute the two axes defining the plane that point new_p2
      // (below) will move along.

      // Start by finding the point between p0 and p1 that's closest to p2.
      //
      vec3 v01_norm = normalize( p0 - p1 );
      vec3 v02 = p2 - p0;
      vec3 p01_nearest = p0 + v01_norm * dot(v01_norm,v02);

      // Axis 2 is from p2 to the p0-p1 line.
      //
```

```
      vec3 axis2 = p2 - p01_nearest;
      float axis2_len = length(axis2);
      vec3 axis2_norm = axis2/axis2_len;

      // Axis 1 is a vector orthogonal to both axis 2 and the p0-p1 line.
      //
      vec3 axis1_norm = cross(v01_norm,axis2_norm);
      vec3 axis1 = axis2_len * axis1_norm;

      // Use a simple circle formula to find new_p2.
      //
      vec4 new_p2 = vec4(p01_nearest + cos(th) * axis2 + sin(th) * axis1,1);

      // Finish by computing the clip-space coordinate of new_p2_c and
      // an eye-space normal for the new triangle.
      //
      vec4 new_p2_c = gl_ProjectionMatrix * new_p2;
      vec3 new_norm = normalize(cross(v01_norm,new_p2.xyz-p0));
      normal_e = -new_norm;

      // Emit the three vertices of this triangle.
      //
      for ( int i=0; i<3; i++ )
        {
          gl_FrontColor = gl_FrontColorIn[i];
          gl_Position = i == p2_idx ? new_p2_c : gl_PositionIn[i];
          gl_TexCoord[0] = i == p2_idx ? tctr : gl_TexCoordIn[i][0];
          vertex_e = i == p2_idx ? new_p2 : In[i].vertex_e;
          no_texture = false;
          EmitVertex();
        }
      EndPrimitive();
    }
}
```