The solution code is in the repo in file name hw04-shdr-sol.cc. If you pull and make you can run the solution by executing hw04sol. An html version is available at http://www.ece.lsu.edu/koppel/gpup/2014/hw04-shdr-sol.cc.html. Note that the git version is always the most up to date.

**Problem 0:** Follow the class account instructions for obtaining the assignment, substitute hw04 where needed. This homework code is based on Homework 2/3, and the physics simulation is identical. Like those assignments it renders a spiral. The spiral can be rendered three ways, the m switches between them. Method 0 uses render routine `render_spiral1` on the CPU side and fixed functionality in the GPU. Method 1 and 2 use routine `render_spiral2` on the CPU side and programmable shaders on the GPU side.

Routine `render_spiral2`, rather than sending actual vertex coordinates to the GPU, sends indices. The x component of a vertex “coordinate” is the ball number (the value of i in `render_spiral1`), the y component is the relative position between ball i-1 and ball i, ranging from 0 to `opt_segments-1`, and the z component indicates whether the vertex is on the inner or outer edge of the spiral. The code in vertex shader `vs_main` in file `hw04-shdr.cc` uses these values to compute the actual vertex coordinate and normal, and to compute the texture coordinates.

Notice that the indices sent by `render_spiral2` are not affected by the position of the balls. Therefore indices can be put in a buffer object and sent just once. (They do need to be re-sent if `opt_segments` changes.) The ball positions do need to be sent each frame, but that’s much less data. The code in `render_spiral2` always uses vertex shader `vs_main` and fragment shader `fs_main`, but can activate either of two geometry shaders, `gs_main_simple` (for Method 1) and `gs_main_solution` (for Method 2). Geometry shader `gs_main_simple` makes no changes to the triangles it receives, and it should not be modified. Geometry shader `gs_main_solution` is initially the same as `gs_main_simple` but should be modified in one of the problems below.

The level of detail used for the spiral is controlled by variable `opt_segments`. Its value can be modified by first pressing the TAB key until “Number of segments per spiral” appears, and then pressing + and - to adjust its value. Check out and compile the code, and make sure that it runs correctly. For this assignment only modify file `hw04-shdr.cc`.

**Problem 1:** Notice that with Method 1 and 2 pale blue is used for both sides of the spiral. In this problem modify the fragment shader, `fs_main`, so the spiral appears like the one on the image to the right. Details are given in the subproblems. Changes to the fragment shader affect Methods 1 and 2, but not zero.

(a) Modify the fragment shader so that the front surface is gold and has the texture applied and so that the back surface is purple. Do this without using the color attribute (fragment shader input), instead rely on the material properties that are set in `render_spiral2`. Note that these material properties are uniform variables.
For the names of the predefined material property uniforms available in the fragment shader see Section 7.4.1 in the OpenGL Shading Language spec version 4.5. To determine whether a fragment is from the front or the back of a primitive see Section 7.1 in the spec.

To solve this part we need to do two things: determine whether the fragment being written is for the front or the back of the triangle, and based on that we need to use the correct color. From Section 7.1 we learn that a pre-defined input to the fragment shader specifies whether the fragment is for the front or back: `gl_FrontFacing`. Looking in the `render_spiral2` routine in `hw04.cc` we find to what properties purple, gold, and gray have been assigned:

```glsl
glMaterialfv(GL_FRONT, GL_DIFFUSE, color_lsu_spirit_gold);
glMaterialfv(GL_FRONT, GL_AMBIENT, color_gray);
glMaterialfv(GL_BACK, GL_DIFFUSE, color_lsu_spirit_purple);
```

From Section 7.4.1 we find the names of the OpenGL shading language uniform variables to which these properties have been assigned. Based on that we can set the `color` variable in the fragment shader, and use that instead of the `color` input:

```glsl
vec4 color =
    is_edge ? gl_FrontMaterial.ambient :
    gl_FrontFacing ? gl_FrontMaterial.diffuse : gl_BackMaterial.diffuse;
```

(b) The surface of the spiral shows a draft this homework assignment handout, using black for the text and yellow for the background. Modify the fragment shader so that there is no spiral wherever the texture is black (or a dark color). In other words, modify it so that you can see through the spiral wherever the texture is black.

To determine if a texel is dark add up the red, green, and blue components and check if the sum is less than .05. If a texel is dark, then prevent the corresponding fragment from being written. To see how to prevent a fragment from being written see Section 6.4 in the OpenGL Shading Language spec version 4.5. (This section describes jump-like statements in the shading language.)

From Section 6.4 we learn that the `discard` keyword is used to abort the writing of a fragment, which is what we want to do if it’s dark:

```glsl
vec4 texel = is_edge ? vec4(1,1,1,1) : texture(tex_unit_0, gl_TexCoord[0].xy);
bool hole = texel.r + texel.g + texel.b < 0.05;
if ( hole ) discard;
```

**Problem 2:** The spiral’s appearance is a little unnatural because it is two-dimensional. Modify the shaders (except `gs_main_simple`) and surrounding code so that when Method 2 is active the spiral has some thickness. (See the illustration at the beginning of this assignment.)

Use geometry shader `gs_main_solution` to render multiple triangles as follows. Let $V_1$, $V_2$, and $V_3$ denote the vertices of a triangle at the input to the geometry shader. The geometry shader should emit $V_1 V_2 V_3$ (as it already does) and also $U_1 U_2 U_3$ where $U_i = V_i + \vec{d}$ and $\vec{d}$ is a vector of length 0.1 pointing from ball $i-1$ to $i$ (the balls that the spiral segment is between). The geometry shader should also emit triangles for an inner and outer edge which joins the two spirals. The edge should be the gray color set in `render_spiral2`. See the illustration.

Solution to this problem will require modifying all three shaders, and modifying the shaders’ interface blocks (such as `Data_to_GS`). **No changes** should be necessary in file `hw04.cc`. It’s okay to modify this file to help in debugging your code, but the solution itself should only be in `hw04-shdr.cc`. If you believe otherwise, please contact the instructor. Divide work appropriately between the vertex shader and geometry shader.
Here are some things to watch out for:

- Be sure to adjust the maximum number of vertices specified for the geometry shader output. If this number is too low execution will end with an error.

- When you modify an interface block (such as Data_to_GS) be sure to modify the other interface block with the same name (one is a shader output, the other is a shader input).

Here are some debugging tips:

- Check for shader code compilation errors when your program starts. The errors are sent to stdout and should appear in the shell or in gdb, depending on how you started the program.

- If execution ends with an OpenGL error, you can get a more detailed error message by turning on OpenGL debugging. To do this change false to true in popengl_helper.ogl_debug_set(false); near the end of hw04.cc.

- Three UI-controlled variables are available for debugging. They are debug_bool.x, debug_bool.y, and debug_float. The Booleans can be toggled with d and D. The float can be adjusted by pressing Tab until “Debug Float” appears and then press + and - to adjust the value.

In the vertex shader compute both the original vertex and a new upper vertex which is offset by 0.1. Send both the original vertex and the new upper vertex to the geometry shader, along with a radial and vertex index information, which the geometry shader will need.

Here is the vertex shader code that computes the new vertex and writes its coordinate in eye and clip space, a radial vector (which points outward from the axis of the spiral, and indices):

```cpp
// Solution code at the end of the vertex shader.
vec3 v12n = normalize(v12);
vec3 depth_vector = 0.1f * v12n;
vec4 position_upper = position + vec4(depth_vector,0);
Position_upper = gl_ModelViewProjectionMatrix * position_upper;
vertex_e_upper = gl_ModelViewMatrix * position_upper;
indices = gl_Vertex;
radial_e = gl_NormalMatrix * radial;

// New vertex shader outputs. (Added to end of Data_to_GS.)
vec4 vertex_e_upper;
vec4 Position_upper;
vec3 radial_e;
ivec3 indices;
```

With the “upper” vertices it is easy to emit the regular and upper triangle in a simple loop nest:

```cpp
// Solution code in gs_main_solution
for ( int level=0; level<2; level++ )
{
  const bool upper = level == 1;

  for ( int i=0; i<3; i++ )
  {  
```
The geometry shader will construct the triangles making up the edge using two of the three vertices at its input. Depending on how the input triangle is oriented, either two of its vertices are on the inner edge, or two are on the outer edge. The code below puts the identity of the two vertices in a short array to make it easier for subsequent code to operate on them:

```cpp
// Solution code in gs_main_solution
int idx[2];
if ( In[0].indices.z == In[1].indices.z ) { idx[0] = 0; idx[1] = 1; }
else if ( In[0].indices.z == In[2].indices.z ) { idx[0] = 0; idx[1] = 2; }
else { idx[0] = 1; idx[1] = 2; }

bool is_inner = In[idx[0]].indices.z == 1;

With the vertices that we need placed in array idx emitting the edge triangles is a simple matter:

```cpp
for ( int i=0; i<2; i++ )
{
    const int v = idx[i];
    vertex_e = In[v].vertex_e;
    gl_Position = In[v].gl_Position;
    normal_e = is_inner ? -In[v].radial_e : In[v].radial_e;
    is_edge = true;  // Identify these as edge primitives for coloring.
    EmitVertex();
    vertex_e = In[v].vertex_e_upper;
    gl_Position = In[v].gl_Position_upper;
    normal_e = is_inner ? -In[v].radial_e : In[v].radial_e;
    is_edge = true;
    EmitVertex();
}
EndPrimitive();
```

Also note that a new geometry shader output/fragment shader input has been added, is_edge. That of course is used so that the fragment shader knows when to use the gray color.

One final thing: notice that the edge triangles use a different surface normal, the radial, and that the sign of the normal needs to be flipped for the inner edge.