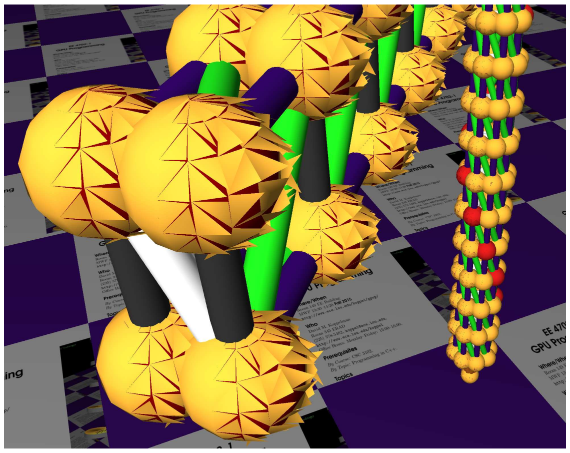
LSU EE 4702-1 Homework 4 Solution Due: 11 November 2015

Preliminary Solution

The solution has been checked into the repository, look for files that start with hw04-sol. For colorized HTML versions visit `http://www.ece.lsu.edu/koppel/gpup/2015/hw04-sol.cc.html`, `http://www.ece.lsu.edu/koppel/gpup/2015/hw04-sol-shdr.cc.html`, and `http://www.ece.lsu.edu/koppel/gpup/2015/hw04-sol-shdr-links.cc.html`.



Problem 0: Follow the instruction on the `http://www.ece.lsu.edu/koppel/gpup/proc.html` page for account setup and programming homework work flow. Compile and run the homework code unmodified. It should start with an 8-arm gyroscope. Press the `+` and `-` keys to change the number of arms. Press digits `1` through `3` to initialize different scenes, the program starts with scene 3.

Pressing 'v' cycles between shaders.

The code in this package includes the friction model asked for in Homework 2. The links however are unbreakable and there's no skin. Also, shadows are deactivated by default to make the solution to this assignment easier.

Use the arrow keys, PageUp, and PageDown to move around the scene. Use key `h` to toggle between the first (head) ball being locked in place and free. Use key `t` to do the same for the last (tail) ball. Press (lower-case) `b` and then use the arrow and page keys to move the first ball around. Press `l` to move the light around and `e` to move the eye.

Note: There is nothing to turn in for this first problem.

Problem 1: The geometry shader `gs_main_simple` in file `hw04-shdr.cc` passes triangles through unmodified. We know that when rendering spheres and cylinders (at least the cylinders that we use for links) a triangle that's not facing the user will be behind one that is facing the user. It makes sense to discard such triangles early.

(*a*) Modify the geometry shader so that it does not emit triangles based on whether they are facing the user and the value of uniform variable `tri_cull`. If `tri_cull` is zero always emit the triangle. If it is 1, emit the triangle only if the triangle front is facing the user. If `tri_cull` is 2 emit the triangle only if the back is facing the user. Base this on the triangle coordinates, not the normals. The value of `tri_cull` can be changed by pressing `c`.

First, compute the triangle normal in eye space, that's put in variable `gnorm_e` in the code below. The triangle normal should be pointing out of the sphere. Next, compute a vector pointing towards the user. In eye space the user is at the origin, so for an eye-space vertex at (x, y, z) vector $(-x, -y, -z)$ points at the user. The code assigns the dot product of the two vectors to `visibility`. If the dot product is positive then the triangle is facing the user. Finally, the code returns early based on the sign of the dot product and the value of `tri_cull`.

```
// SOLUTION -- Code in file hw04-sol-shdr.cc, routine gs_main_simple();
vec3 gnorm_e = cross
  ( In[1].vertex_e.xyz - In[0].vertex_e.xyz,
    In[2].vertex_e.xyz - In[1].vertex_e.xyz );

vec3 to_user = -In[0].vertex_e.xyz;

float visibility = dot( gnorm_e, to_user );
if ( tri_cull == 1 && visibility < 0 ) return;
if ( tri_cull == 2 && visibility > 0 ) return;
```

(*b*) Estimate the amount of work saved. Use the following symbols: Let c_v, c_g, and c_f be the amount of work done by an invocation of the vertex, geometry, and fragment shaders respectively. For example, if we rendered one triangle covering 22 pixels the total work would be $3c_v + c_g + 22c_f$ units. For your answer let n_t denote the number of triangles, and make reasonable choices for other quantities.

Work is saved because we only have to render one half or less of the sphere or cylinder. Since we are culling triangles in the geometry shader we are reducing the amount of work in the fragment shader (since it's after the geometry shader), but we aren't reducing the amount of work performed by the vertex shader (since it's before the geometry shader). We are actually creating more work for the geometry shader since it needs to test whether to cull a triangle.

To determine how much work is saved we need to know the average number of fragments per triangle. Let n_f denote the average number of fragments per triangle. Without culling the fragment shader will be invoked $n_t n_f$ times, performing $n_t n_f c_f$ work. Assuming that we cull half the triangles (which is conservative–we'll usually cull more) and that the average number of fragments per triangle is the same on culled triangles (which is a little unfair, there will usually be less since they are farther away) the amount of work saved for the will be $\frac{1}{2} n_t n_f c_f$.

To see how much of a performance impact that has we need to look at the other shaders. With or without culling the amount of work done by the vertex shader is $n_t c_v$, assuming that triangle strips are used and ignoring the two-vertex startup per sphere. The geometry shader does a little more work to check whether a triangle can be culled, but if a triangle is culled it does less work. Assume that overall, the amount of work does not change. Then the work done is $n_t c_g$ with or without culling.

The absolute amount of work saved is $\frac{1}{2} n_t n_f c_f$. The speedup with culling (the time without culling divided by the time with culling) is given by

$$\frac{n_t c_v + n_t c_g + \frac{1}{2} n_t n_f c_f}{n_t c_v + n_t c_g + n_t n_f c_f}.$$

If $n_f c_f$ is large compared to c_v and c_g a substantial amount of work will be saved.

(*c*) Try to determine actual values for c_v, c_g, and c_f by running your code. Pause the simulation to make sure the physics isn't hogging CPU. The number of vertices is shown on the green text near the top, the total number is at the end of the line. Triangle strips are used for both the balls and the links. Pressing the `+` and `-` keys when setup 3 is visible will change the number of arms on the gyroscope, use this to vary the number of balls and links.

Problem 2: The instance shader code rendering links shows all links the same color.

(*a*) Modify the shaders in `hw04-shdr-links.cc` so that that the fragment shader obtains the color using a value based on `glVertex_ID` rather than reading the color itself from on of its inputs. Only the vertex shader has access to `glVertex_ID`, define shader inputs and outputs as needed to send its value to the fragment shader.

Modify other shaders in this file consistently to remove the old code passing color data through the pipeline.

Note: The original version of the problem was to use pre-defined fragment shader variable `glVertex_ID`*. As a student pointed out, there is no such variable in the fragment shader.*

In the unmodified code, `color` travels from the CPU, through the fragment shader, geometry shader, and finally reaching the fragment shader where it is used. It is unmodified in its journey from CPU to fragment shader, meaning that the vertex and geometry shader pass its value unmodified. The color is type **vec4**, which is 16 bytes. In our solution, rather than passing a 16-byte color through the pipeline we'll pass a 4-byte vertex ID (sort of a link number) through the pipeline, and have the fragment shader look up the color values from an array using the vertex ID as an array index.

To implement this we need to modify both the CPU code, in **hw04.cc**, and the shader code, in **hw04-shdr-links.cc**. (In the solution those file names are **hw04-sol.cc** and **hw04-sol-shdr-links.cc**.)

In the shader code interface blocks all occurrences of **vec4 color** are replaced with **int vtx_id**. For example,

```
// Interface block for vertex shader output / geometry shader input.
out Data_to_GS {
  ivec2 indices;
  /// SOLUTION -- Problem 2
  //  Replace vec4 color with int vtx_id;
  // vec4 color;
  int vtx_id;
};
```

Similar changes would be made to the other interface blocks. The solution above shows **vec4 color** commented out. In real life we would just delete the line and rely on the repositories diff function to tell us what was there before.

In the vertex shader we need to assign **vtx_id**, and in the fragment shader we need to use it. The changed routines are shown below.

```
void vs_main() {
  indices = gl_Vertex;
  vtx_id = gl_VertexID;
}

void fs_main() {
  /// SOLUTION -- Problem 2
  //  Use vtx_id to retrieve color.
  //
  vec4 color = links_color[vtx_id];

  gl_FragColor = generic_lighting(vertex_e, color, normalize(normal_e));
  gl_FragDepth = gl_FragCoord.z;
}
```

The code above reads the color from array **links_color**, which is a buffer object. In the shader code we need to declare the array, providing the data type (**vec4**) and also a binding point (3). The interface block name, **Links_Color** is ignored in this code.

```
 /// SOLUTION -- Problem 2
layout ( binding = 3 ) buffer Links_Color { vec4 links_color[]; };
```

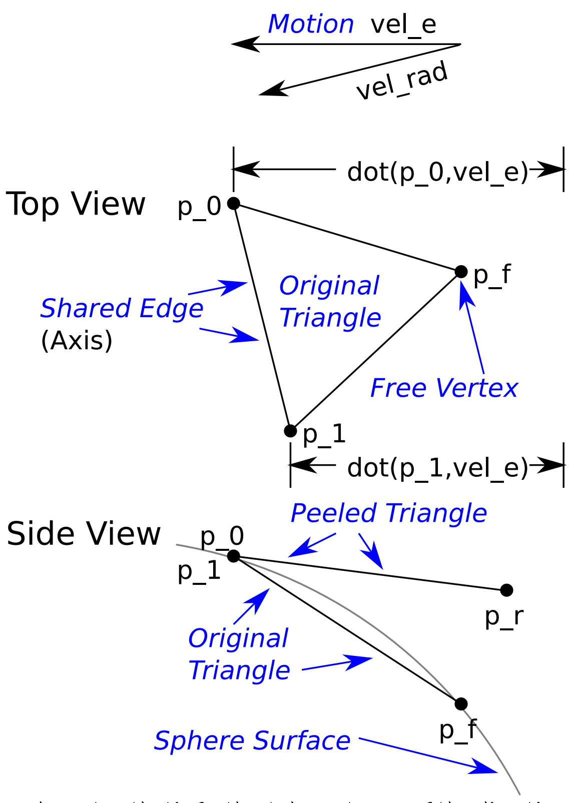
In the CPU code we need to create a buffer object for the colors, prepare an array of link colors, copy the array to the buffer object, and then bind the buffer object to binding point 3 (the number used in the layout declaration above). The added code is shown below with surrounding code removed. In other words the additions below are in the declaration of the **World** class and in routine **render_objects**.

```
  //  In class World declare buffer object name, and array for colors.
  //
  GLuint links_color_bo;
  pColor *links_color;
```

```
// In World::render_objects: Just once, generate a buffer object name.
glGenBuffers(1,&links_color_bo);

// In World::render_objects: Whenever more space is needed, allocate.
delete links_color;
links_color = new pColor[links_size];

// In World::render_objects: Whenever links change, update color array.
if ( link_change ) for ( LIter link(links); link; ) {
  links_indices[link].x = link->ball1->idx;
  links_indices[link].y = link->ball2->idx;
  links_color[link] = link->color;      /// SOLUTION -- Problem 2
}

// In World::render_objects: Whenever links change, update buffer object.
glBindBuffer(GL_ARRAY_BUFFER,links_color_bo);
glBufferData
  (GL_ARRAY_BUFFER,
   links.size() * sizeof(links_color[0]), // Amount of data (bytes) to copy.
   links_color,
   GL_STATIC_DRAW); // Hint about who, when, how accessed.

// In World::render_objects:
//    Just before rendering pass, bind buffer object to point 3.
glBindBufferBase(GL_SHADER_STORAGE_BUFFER,3,links_color_bo);
```

Possible test question: Describe what would happen if each change above (in the solution) were omitted. In particular indicate whether it would be a compile error, a runtime error, or just a wrong answer. For example, if the declaration `GLuint links_color_bo` were omitted there would be a compile error.

Problem 3: Modify `gs_main_simple` so that it can emit a second triangle that looks like it is being peeled off a moving sphere by the rushing air. The second triangle should be the same shape as the original triangle and should share one edge with the original triangle. See the illustration at the beginning of this assignment. The choice of shared edge and the angle of the triangle should be based in some way on the velocity of the ball.



There are three parts to this solution: getting velocity values to the shader, identifying the shared edge (the one that remains on the surface of the sphere, and determining a position for the free vertex.

The changes needed to get the ball velocity to the shader are similar to the changes needed to get link colors to the shader in the previous problem. About the only significant difference is that when we rendered links, each link was rendered as a point (the geometry shader converted the "point" into a cylinder), and so the vertex ID could be used to lookup the color. Spheres are rendered using an instanced draw. For an instanced draw the vertex ID indicates a point on the sphere's surface, while the primitive ID indicates which sphere is being rendered. Therefore will use the primitive ID to look up velocity.

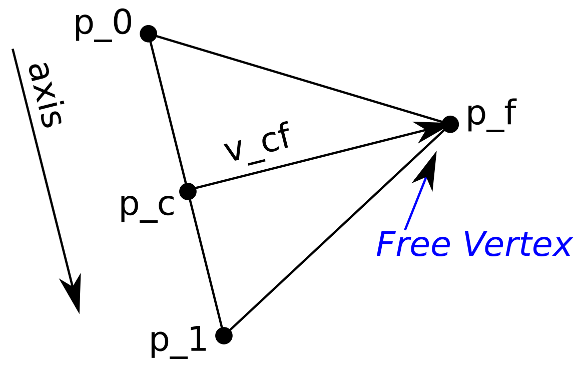
The bulk of the solution is in routine `gs_peel`, which is called by `gs_main_simple`. Routine `gs_main_simple` calls `gs_peel` before it renders a triangle on the sphere surface. If `gs_peel` returns true it means that it has emitted a peeled triangle, in which case `gs_main_simple` omits the original triangle colored red. Otherwise, `gs_main_simple` omits the original triangle colored yellow.

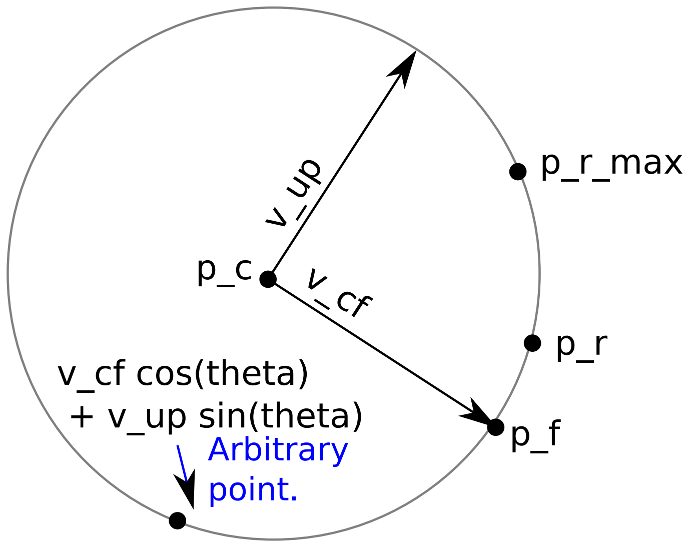
Routine `gs_peel` identifies the shared edge by finding the triangle vertex that's furthest downstream of the direction of motion. That's done by taking the dot product of each vertex with the velocity vector and choosing the smallest value (see the illustration above). (To get a feel for how this works consider velocity vector $v = [-2.1, 0, 0]$. The dot product of v with coordinate $[x, y, z]$ is $-2.1x$. The vertex with the largest x component is the furthest downstream.) Since eye-space vertex coordinates are available, the object-space velocity vector read from the array, `vel_o`, is converted to eye space coordinates, `vel_e`. All of that is done by the simplified code below:

```
vec3 vel_o = balls_velocity[In[0].inst_id].xyz;
vec3 vel_e = gl_NormalMatrix * vel_o;

float upstr_amt[3];
for ( int i=0; i<3; i++ ) upstr_amt[i] = dot(In[i].vertex_e.xyz,vel_e);
int min01 = upstr_amt[0] < upstr_amt[1] ? 0 : 1;
int free_i = upstr_amt[min01] < upstr_amt[2] ? min01 : 2;

// Get the indices of the other two vertices.
int a0 = ( free_i + 1 ) % 3;
int a1 = ( free_i + 2 ) % 3;

vec3 p_0 = In[a0].vertex_e.xyz;
vec3 p_1 = In[a1].vertex_e.xyz;
vec3 p_f = In[free_i].vertex_e.xyz;
```

Because we don't want the triangle to change shape, the free vertex will be constrained to move in a circle with the center being the closest point on the shared edge to the free vertex, that point is called `p_c` in the code. The shared edge is called `axis`. We need to compute two vectors that we can use to find points on this circle, those vectors are called `v_cf` and `v_up` in the code:

```
vec3 axis = p_1 - p_0;  // Shared edge.
vec3 axis_n = normalize(axis);
vec3 v_0f = p_f - p_0;
vec3 p_c = p_0 + axis_n * dot(axis_n,v_0f);
vec3 v_cf = p_f - p_c;
float len_cf_sq = dot(v_cf,v_cf);
float len_cf = sqrt(len_cf_sq);
vec3 v_up = cross(axis_n,v_cf);
```


p_0
axis
p_f
v_cf
p_c
Free Vertex
p_1


v_up
p_r_max
p_c
v_cf
p_r
v_cf cos(theta)
+ v_up sin(theta)
Arbitrary
point.
p_f