LSU EE 4702-1 **Homework 5** Solution **Due:** ≈ 12 **November 2014**

The following assignment is based on the code package for Homework 4. The questions are asking about the code provided for the assignment, not about code written as part of the solution. In other words, you can answer these questions without having yet solved Homework 4.

The solution discussion refers to Homework 4 code. Parts are of the code are reproduced at the end of the solution. For HTML versions visit `https://www.ece.lsu.edu/koppel/gpup/2014/hw04.cc.html` for the CPU code and `https://www.ece.lsu.edu/koppel/gpup/2014/hw04-shdr.cc.html` for the shader code.

Problem 1: Compare the amount of data sent from CPU to GPU in Methods 0 (triangle strip, vertex contains coordinates) and 1 (triangle strip, simple geometry shader, vertex contains indices) from the Homework 4 code. (Note that a solution to Homework 4 does not change the amount of data sent from CPU to GPU.) *Note: An earlier version of the problem asked about Methods 1 and 2. The amount of data sent by 1 and 2 would be the same.*

Let n denote the number of balls (`chain_length`) and s the number of segments (`opt_segments`). Determine the amount of data, in bytes, sent to the GPU per frame for Method 0 (routine `render_spiral1`) and Method 1 (`render_spiral2`).

In the discussion below we'll ignore data sent for uniform variables and OpenGL commands. Uniform data includes explicitly declared uniforms, such as `opt_segments`, and pre-defined uniforms such as `gl_ModelViewMatrix`. Command data (which was not talked about much in class) includes whatever needs to be sent from CPU to GPU to set up a rendering pass, such as indicating the number vertices to expect. The amount of uniform and command data can be ignored because there is not much of it (though misusing uniforms and commands can slow things down). We'll also ignore texture data, since that's just sent once.

Method 0 renders using `render_spiral1`. By inspection of the code in `render_spiral1` we learn that with each vertex we are sending a texture coordinate, `glTexCoord2f`, a normal, `glNormal3fv`, and the vertex coordinate, `glVertex3fv`. Each of these is a vector of floats, a two-element vector for texture coordinates and three-element vectors for the normal and vertex coordinate. The total size per vertex is $2+3+3=8$ floats or $8\times 4 = 32\,\mathrm{B}$.

The `i` loop iterates `chain_length-1` or $n-1$ times, the `t` loop iterates `opt_segments` or s times (see the code), and a `t` loop iteration sends 2 vertices, and so the total number of vertices sent per frame is $(n-1)s2$. The amount of data is therefore $\boxed{(n-1)s2\times 32\,\mathrm{B} = 64(n-1)s\,\mathrm{B}}$.

Method 1 renders using `render_spiral2` and the programmable shaders `vs_main`, `gs_main_simple`, and `fs_main`. The vertices are specified using `glDrawArrays` rather than `glBegin`. To find the amount of data, we need to tally the amount of data sent directly into the rendering pipeline (as we did for `render_spiral1`) and the amount of non-input data read by the shaders. The non-input data is from buffer object `balls_bo` (CPU name), which is bound to `balls_pos` in the shader code.

To find the data sent through the rendering pipeline we need to look at the client arrays that are enabled. Client arrays are enabled, of course, using `glEnableClientState`. For `render_spiral2` only one array is enabled, `GL_VERTEX_ARRAY`. This array is bound to buffer object `indices_bo`. The `glVertexPointer` command tells us that the size of a vertex coordinate is `3*sizeof(int)` or 12 bytes. The number of vertices sent is, as before, $(n-1)s2$.

But these vertices are coming from a buffer object, and inspection of the code indicates that the buffer object is only updated when `opt_segments` changes. We expect that buffer object data is sent from the CPU to GPU each time it is updated on the CPU. Since `opt_segments` only changes in response to user input we can assume that the buffer object is rarely updated (at least rare with respect to frame update). The size of the indices buffer object is $(n-1)s2\times 12\,\mathrm{B}$, but because it is rarely transferred we **will not include it** in our tally of CPU to GPU data.

As mentioned earlier, the vertex shader is reading the `balls_pos` buffer object. As we can see from the code, that buffer object is updated each time we call `render_spiral2`. (It's updated because the balls move.) The buffer object is an array of `pCoor` variables (which are arranged in memory identically to `vec4` variables). The size of an element is $4 \times 4\,\mathrm{B} = 16\,\mathrm{B}$ and the total size of the buffer object is $16n\,\mathrm{B}$.

The total amount of data sent from CPU to GPU for `render_spiral2` is therefore only $\boxed{16n\,\mathrm{B}\text{ per frame}}$. (Remember that we are ignoring uniform data, which for large n would not make much of a difference.)

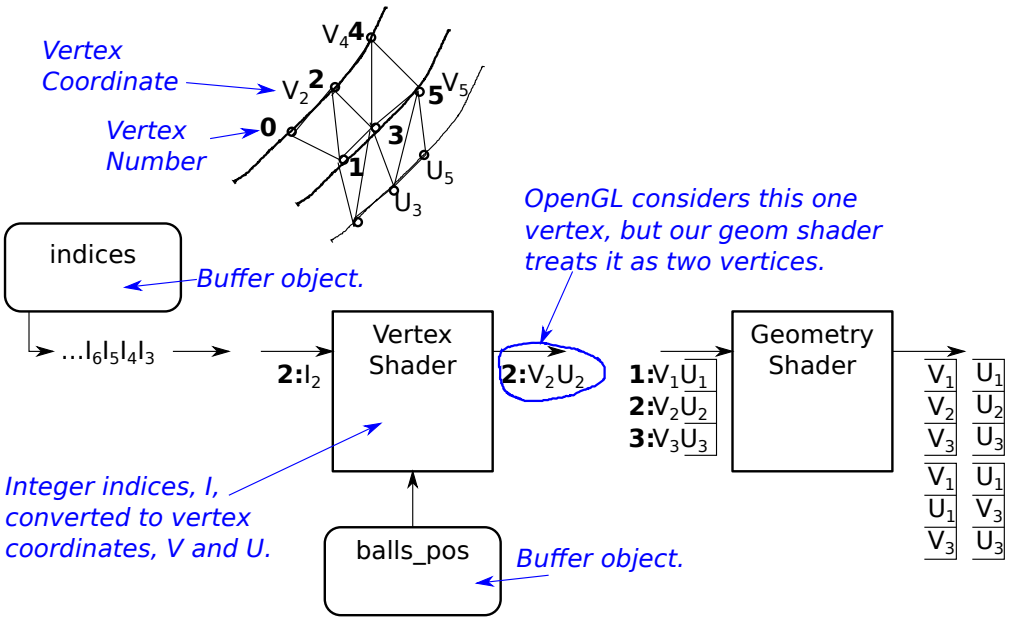
Problem 2: Consider the amount work to compute vertex coordinates and normals done by `render_spiral1` and by the vertex shader `vs_main`. The vertex shader is actually computing things multiple times that the code in `render_spiral1` computes just once. Identify such redundant computation. *Hint: It happens in two different ways, one way results in things computed twice, another way results in things computed s times.*

For the factor of two duplication by `vs_main` look at the `t` loop in `render_spiral1`. Notice that it emits two vertices. Both of these vertices are computed from the same value of `radial` and the normal for these vertices are computed from the same value of `tangial`. In contrast, the vertex shader `vs_main` is called once for each vertex. Each time it is computing `radial` and `tangial`, so it is computing those values twice as many times as `render_spiral1`.

For the factor of s duplication look at the code at the top of the `i` loop body in `render_spiral1`. Notice that values for `va` and `vb` are computed by `render_spiral1` once and then used in each of the s iterations of the `t` loop. But in the `render_spiral2` version the vertex shader computes `va` and `vb` for each vertex. So `vs_main` is computing these values $2s$ times for frequently than it needs to.

Problem 3: Explain how some of the problems above could be avoided if the CPU specified vertices using line strips and the geometry shader used lines as inputs. Of course, the geometry shader's output would be triangles. Each point on the line would correspond to a `p` in the `t` loop in the `render_spiral1` routine.

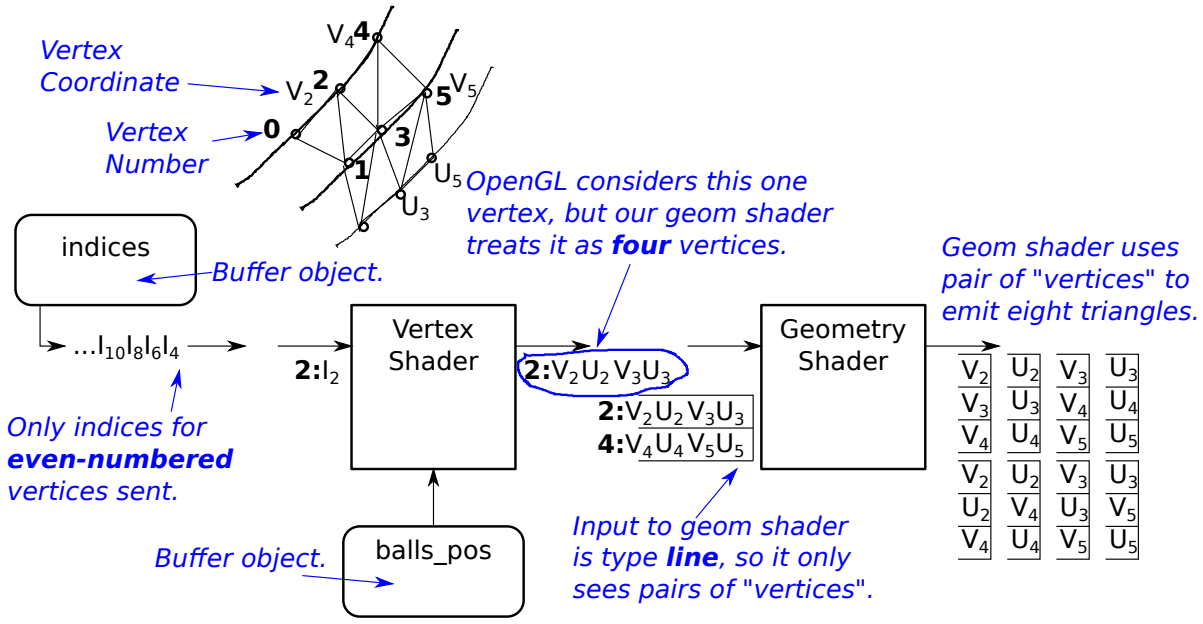
First, consider the operation of `render_spiral2`:



The buffer object is supplying indices for all of the vertices on one spiral, including the inner edge (even-numbered in illustration) and outer edge (odd numbered). The vertex shader computes the corresponding upper spiral coordinate, U, but as far as OpenGL is concerned the output of the vertex shader is still just one vertex, to be collected in to triangles

and passed to the geometry shader, which emits the upper and lower triangles, and the edge triangles. That's Homework 4.

In the solution to this problem we will only send the inner-rim vertices (even-numbered) to the vertex shader, organized as line strips:



The vertex shader will generate coordinates for the outer-rim vertices along with the corresponding upper vertices. So here the vertex shader output is four vertices (but still considered one vertex by OpenGL). The duplication of computation is avoided by having the vertex shader compute the inner and outer vertex coordinates together.

Our geometry shader just needs two of these 4-vertex vertices to construct two triangles each on the upper and lower spiral and two triangles each on the inner and outer edge. Line strips rather than triangle strips are being used because the geometry shader only needs two of our vertices.

On the CPU `render_spiral3` would be very similar to `render_spiral2` except for the following changes: The number of vertices sent, `num_elts`, would be half the amount. That is, rather than sending a vertex for the inner and outer part of the spiral, a vertex would only be sent for the inner part of the spiral.

```
//    CPU RENDER SPIRAL ROUTINES, FROM FILE hw04.cc
void
World::render_spiral1()
{
  glDisable(GL_COLOR_MATERIAL);
  glMaterialfv(GL_FRONT,GL_AMBIENT_AND_DIFFUSE,color_lsu_spirit_purple);
  glMaterialfv(GL_BACK,GL_AMBIENT_AND_DIFFUSE,color_lsu_spirit_gold);

  glActiveTexture(GL_TEXTURE0);
  glEnable(GL_TEXTURE_2D);
  glBindTexture(GL_TEXTURE_2D,texid_syl);

  float theta = 0;
  const float spiral_radius = 0.5;
  const float delta_t = 1.0 / opt_segments;
  const float omega = 10;

  glBegin(GL_TRIANGLE_STRIP);

  for ( int i=1; i<chain_length; i++ )
    {
      Ball *const ball1 = &balls[i-1];
      Ball *const ball2 = &balls[i];
      pCoor pos1 = ball1->position;
      pCoor pos2 = ball2->position;

      //
      // Render a spiral slide from position of ball1 to ball2.
      //

      pVect v12 = pos2 - pos1;

      // Find a vector that's orthogonal to v12.
      //
      pNorm ax = v12.x == 0 ? pVect(0,v12.z,-v12.y) : pVect(v12.y,-v12.x,0);
      pNorm ay = cross(v12,ax);

      pVect vx = ax * spiral_radius;
      pVect vy = ay * spiral_radius;

      for ( float t=0; t<0.999; t += delta_t )
        {
          pCoor p = pos1 + t * v12;
          theta += delta_t * omega;

          // Compute a vector from the spiral axis to a point on the spiral.
          //
          pVect radial = vx * cos(theta) + vy * sin(theta);

          // Find a vector in the "direction" of the spiral motion as
          // if t were time.
```

```
            //
            pVect tangial = -omega * vx * sin(theta) + omega * vy * cos(theta);
            //
            // Note: expression above is derivative of radial w.r.t. t.

            pCoor p_outer = p + radial;
            const float inner_frac = 0.5;
            pCoor p_inner = p + inner_frac * radial;
            pVect tang = v12 + tangial;
            pVect tang_inner = v12 + inner_frac * tangial;

            // Compute a vector pointing up.
            //
            pNorm norm = cross(radial,tang);
            pNorm norm_inner = cross(radial,tang_inner);

            const float du = 0.5 / chain_length;
            const float u = float(i) * du;

            glTexCoord2f(t,u+du);
            glNormal3fv(norm_inner);
            glVertex3fv(p_inner);

            glTexCoord2f(t,u);
            glNormal3fv(norm);
            glVertex3fv(p_outer);
          }
      }

  glEnd();

  vs_fixed->use();

  glEnable(GL_COLOR_MATERIAL);
}

void
World::render_spiral2()
{
  glDisable(GL_COLOR_MATERIAL);
  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);

  glActiveTexture(GL_TEXTURE0);
  glEnable(GL_TEXTURE_2D);
  glBindTexture(GL_TEXTURE_2D,texid_syl);

  if ( !balls_bo ) glGenBuffers(1, &balls_bo);
  glBindBuffer(GL_ARRAY_BUFFER, balls_bo);
  for ( int i=0; i<chain_length; i++ ) balls_pos[i] = balls[i].position;
```

```
  glBufferData
    (GL_ARRAY_BUFFER,chain_length*sizeof(balls_pos[0]),
     balls_pos, GL_DYNAMIC_DRAW);
  glBindBuffer(GL_ARRAY_BUFFER, 0);
  const int num_elts = ( chain_length - 1 ) * opt_segments * 2;

  if ( !indices_bo ) glGenBuffers(1, &indices_bo );

  if ( indices_segments != opt_segments )
    {
      indices_segments = opt_segments;
      struct ivec3 { int i, j, k; ivec3(){}; ivec3(int ip, int jp, int kp):i(ip),j(jp),k(kp){}};
      ivec3* const indices = new ivec3[ num_elts ];
      ivec3 *ip = indices;
      for ( int i=1; i<chain_length; i++ )
        for ( int t=0; t<opt_segments; t++ )
          for ( int inner=0; inner<2; inner++ )
            *ip++ = ivec3(i,t,inner);
      glBindBuffer(GL_ARRAY_BUFFER, indices_bo);
      glBufferData
        (GL_ARRAY_BUFFER,
         num_elts * sizeof(indices[0]),
         indices,
         GL_STATIC_DRAW);
      delete indices;
    }

  if ( opt_vtx_method == 1 )
    vs_geo_simple->use();
  else
    vs_geo_sol->use();

  glColor3fv(color_powder_blue);

  glBindBufferBase(GL_SHADER_STORAGE_BUFFER,1,balls_bo);
  glUniform1i(1,opt_segments);
  glUniform1i(2,chain_length);
  glUniform2i(3,opt_debug_x,opt_debug_y);
  glUniform1f(4,opt_debug_float);

  glBindBuffer(GL_ARRAY_BUFFER, indices_bo);
  glVertexPointer(3,GL_INT,3*sizeof(int),0);
  glEnableClientState(GL_VERTEX_ARRAY);
  glBindBuffer(GL_ARRAY_BUFFER, 0);

  glDrawArrays(GL_TRIANGLE_STRIP, 0, num_elts );

  glDisableClientState(GL_VERTEX_ARRAY);
  vs_fixed->use();
  glEnable(GL_COLOR_MATERIAL);
}
```