Name _____

GPU Programming
EE 4702-1
Take-Home Pre-Final Examination

Due: 30 November 2018 at 16:30

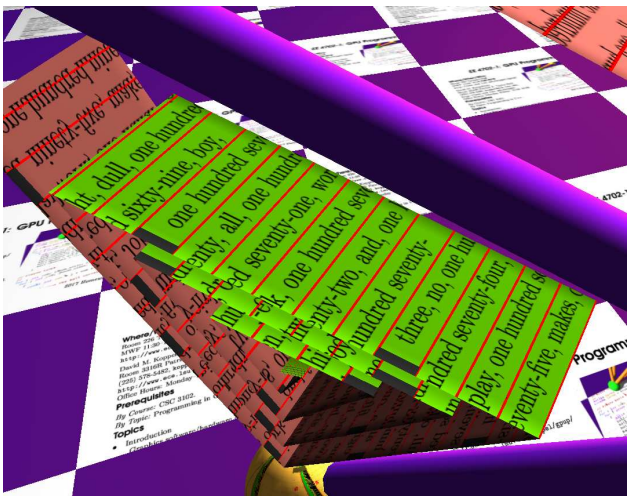
Work on this exam alone. Regular class resources, such as notes, papers, solutions, documentation, and code, can be used to find solutions. In addition outside OpenGL references and tutorials, other programming resources, and mathematical references can be consulted. Do not try to seek out references that specifically answer any question here. **Do not discuss this exam with classmates or anyone else**, except questions or concerns about problems should be directed to Dr. Koppelman.

Problem 1	_____	(33 pts)
Problem 2	_____	(34 pts)
Problem 3	_____	(33 pts)
Exam Total	_____	(100 pts)

Alias _____

Good Luck!

Problem 1: [33 pts] Appearing on the next page is the fragment shader code taken from the solution to Homework 3, in which a texture is applied to the triangular spiral in such a way that it looks like the texture image was cut into strips and glued together.



(*a*) Modify the shader code so that the text is rotated by 90°, as shown in the screenshot above. The text aspect ratio must be consistent along segments. As with the homework, the text must be readable without skipped or repeated sections. **Do not** assume or make any changes to the host code. In particular, the fragment shader should work with the same primitives as emitted for the Homework 3 code.

(*b*) Modify the shader code so that red lines are drawn separating the lines of text.

It is okay to try out your solution on the Homework 3 package, but put the solution in this exam. That is, the solution **will not** be collected by the TA-bot.

See the Homework 3 solution for detailed comments explaining the shader code.

Use next page for solution.

☐ Modify shader so text rotated 90°.

☐ Draw red lines between groups of text.

☐ Text aspect ratio and size must be consistent on all segments and ☐ there must be no gaps or skipped sections.

```
void fs_main_hw03() {
  float line_num = floor(tex_coord.x);
  vec2 tc = vec2( tex_coord.x, ( line_num + tex_coord.y ) * tex_ht );

  vec4 color =
   gl_FrontFacing ? gl_FrontMaterial.diffuse : gl_BackMaterial.diffuse;

  // No need to modify code below.
  vec4 texel = texture(tex_unit_0,tc);
  vec3 nne = normalize(normal_e);
  float edge_dist = tex_coord.y;
  vec3 bnorm = tryout.x ? nne : edge_dist > 0.9
    ? normalize(mix(nne,spiral_normal,2*(edge_dist-0.9))) : edge_dist < 0.1
    ? normalize(mix(-spiral_normal,nne,0.8+2*edge_dist)) : nne;
  vec4 lighted_color =
    generic_lighting( vertex_e, color, bnorm, gl_FrontFacing );
  gl_FragColor = texel * lighted_color;
  gl_FragDepth = gl_FragCoord.z;
}
```

Problem 2: [34 pts] Recall that the true sphere code covered in class emits a square (two triangles) for each sphere to be rendered. The square is chosen so that it perfectly frames the sphere as seen by the viewer. The square is perfect in the sense that if it were made any smaller parts of the sphere would not be visible. But it is still wasteful because the sphere is not visible in the corners of the square and so those fragment shader invocations are wasted.

(*a*) Modify the true-sphere geometry shader below so that it emits a $2s$-sided polygon rather than a square. Show the code for computing the polygon coordinates, **do not** use the assumed functions from the next problem. Note that when $s = 2$ the code should emit a bounding square, which is what the original code does.

Use next page for solution.

☐ Modify code to emit $2s$-sided bounding polygon. ☐ Take advantage of the fact that $2s$ is an even number.

```
const int s = 8;     // Don't assume that s is always 8!!
layout ( points ) in;
layout ( triangle_strip, max_vertices = 4 ) out;

void gs_main() {
  vertex_id = In[0].vertex_id;
  mat4 mvp = trans_proj * gl_ModelViewMatrix;
  vec4 pos_rad = sphere_pos_rad[vertex_id];
  vec3 ctr_o = pos_rad.xyz;
  float r = pos_rad.w;
  vec3 e_o = vec3(gl_ModelViewMatrixInverse*vec4(0,0,0,1));  // Eye location in object space.

  vec3 ec_o = ctr_o - e_o;  // Eye to Center (of sphere).
  float ec_len = length(ec_o);
  vec3 ne = ec_o / ec_len;

  // Find vectors orthogonal to ec_o.
  vec3 atr_o = abs(ec_o);
  int min_idx = atr_o.x < atr_o.y ? ( atr_o.x < atr_o.z ? 0 : 2 )
    : ( atr_o.y < atr_o.z ? 1 : 2 );
  vec3 nx_raw = min_idx == 0 ? vec3(0,-ec_o.z,ec_o.y)
    : min_idx == 1 ? vec3(-ec_o.z,0,ec_o.x) : vec3(-ec_o.y,ec_o.x,0);
  vec3 nx = normalize(nx_raw),  ny = cross(ne,nx);

  // Compute center of the limb, lmb_o. (The limb is the sphere outline.)
  float sin_theta = r / ec_len;
  float a = r * sin_theta;
  vec3 lmb_o = ctr_o - a * ne;

  // Compute axes to draw the limb.
  float b = r * sqrt( 1 - sin_theta * sin_theta );
  vec3 ax = b * nx,  ay = b * ny;

  // Emit a bounding square for the limb.  --- Solution should start here.
  for ( int i = -1; i < 2; i += 2 ) for ( int j = -1; j < 2; j += 2 ) {
        vertex_o = lmb_o + ax * i + ay * j;
        gl_Position = mvp * vec4(vertex_o,1);
        EmitVertex();  }
  EndPrimitive();









}
```

(*b*) Let t_f denote the execution time of 1 invocation of the true-sphere fragment shader when the sphere covers the fragment, and let t_n denote the execution time of 1 invocation when the sphere does not cover the fragment. Assume that $t_f > t_n$. Let $t_{g0} + st_{g1}$ denote the execution time of one invocation of the true-sphere geometry shader that emits a $2s$-sided polygon.

In general, emitting a polygon with more sides reduces the work performed by the fragment shader but increases the work done by the geometry shader, improving overall performance. But there will be a point at which increasing s *reduces* performance.

Using the symbols defined above **plus other information**, find an expression for the time needed to render a sphere. The other information is needed to compute a sphere rendering time using the timings above. Of course, no credit will be given if that other information *is* the execution time. Also determine the value of s that will minimize the total time used by the geometry and fragment shaders for rendering a sphere. When computing the s that minimizes time it is okay to apply simplifications based on s being large. (Otherwise a closed-form solution can't easily be found.) *Hint: Yes, the solution requires calculus.*

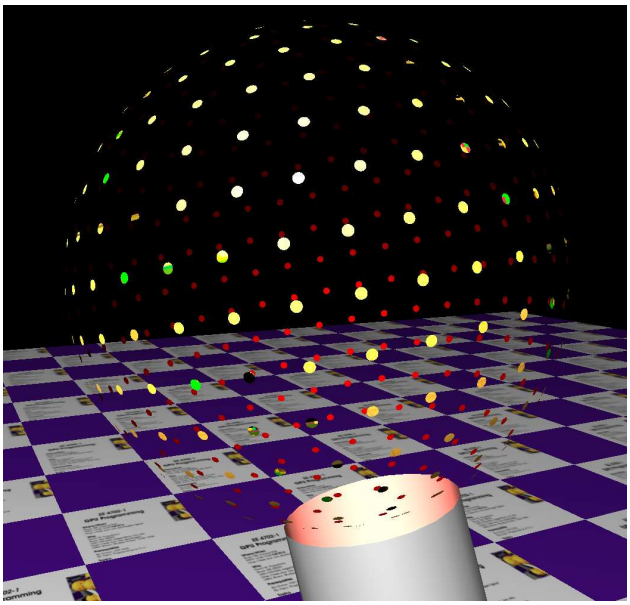
☐ Indicate the other information that is needed.

☐ Show an expression for the time needed to render a sphere using a $2s$-sided bounding polygon in terms of t_f, etc., and the other information.

☐ Find an expression for s that will minimize time in terms of t_f, etc., and the other information.

Problem 3: [33 pts] The screenshot below is based on the code from Homework 4. The code is in *lenses* mode, meaning that the sphere is visible only at certain points. Unlike Homework 4, the lenses are much further apart from each other, and therefore much less of the sphere is visible. If the code in the Homework 4 solution were used then most invocations of the fragment shader would be wasted because they end up discarding the fragment.

There would be much less waste if a separate primitive were emitted for each lens, rather than one primitive for the entire sphere.



Suppose that each sphere has the same number of lenses, call that number h, and refer to the lenses as lens 0, lens 1, etc. Assume that a library of functions is available for computing positions for each lens. Among these is a function `lens_get_ctr_o(i,sphere_info)` which returns the object-space coordinates of the center of lens `i`. Variable `sphere_info` holds the value of the sphere center, radius, and orientation. Assume that `sphere_info` has already been prepared. You may assume that related functions exist such as `lens_get_ctr_e(i,sphere_info)` for eye-space coordinates. Also convenient for this problem is function `lens_get_bb_o(i,j,n,sphere_info)` which returns the object-space coordinates of point `j` of an `n`-sided bounding polygon for lens `i`. The bounding polygon is like the bounding square used in the true-sphere code: it tightly fits around the lens as seen by the viewer (or equivalently, when projected on to the frame buffer). The points wrap around the lens in a clockwise direction with increasing `j`.

Show changes needed to the geometry and fragment shaders on the following pages to render the lenses efficiently for cases in which h is relatively small, say on the order of 10.

Problem 3, continued:

(*a*) Appearing below is a shortened version of the geometry shader from the Homework 4 solution. The comments have been removed, but feel free to look at comments in the posted solution. Modify the geometry shader below so that it emits a primitive for each lens and provides information needed by the fragment shader (see next problem). Use the functions described above to find coordinates of bounding polygons for the lenses.

☐ Modify so that it emits a primitive for each lens.

☐ If necessary, change layout and output declarations.

☐ Provide information needed by fragment shader (see next subproblem).

```
out Data_to_FS // Data to fragment shader
{
  flat int vertex_id;
  vec3 vertex_o;


};

const int h = 8; // Number of holes. Don't assume it is always 8.
layout ( points ) in;
layout ( triangle_strip, max_vertices = 4 ) out;

void gs_main() {
  vertex_id = In[0].vertex_id;
  vec4 pos_rad = sphere_pos_rad[vertex_id];
  vec3 ctr_o = pos_rad.xyz;
  float r = pos_rad.w;
  Sphere_Info sphere_info = make_sphere_info(pos_rad,sphere_rot[vertex_id]);

  // Eye location in object space.
  vec3 e_o = vec3(gl_ModelViewMatrixInverse * vec4(0,0,0,1));

  // Vectors from eye to sphere center.
  vec3 ec_o = ctr_o - e_o;  // Eye to Center (of sphere).
  float ec_len = length(ec_o);
  vec3 ne = ec_o / ec_len;

  // Vectors orthogonal to ec_o.
  vec3 atr_o = abs(ec_o);
  int min_idx = atr_o.x < atr_o.y ? ( atr_o.x < atr_o.z ? 0 : 2 )
    : ( atr_o.y < atr_o.z ? 1 : 2 );
  vec3 nx_raw = min_idx == 0 ? vec3(0,-ec_o.z,ec_o.y)
    : min_idx == 1 ? vec3(-ec_o.z,0,ec_o.x) : vec3(-ec_o.y,ec_o.x,0);
  vec3 nx = normalize(nx_raw);
  vec3 ny = cross(ne,nx);

  // Compute center of the limb, lmb_o, the circle formed by the most
  // distant visible parts of the sphere surface.
```

```
  float sin_theta = r / ec_len;
  float a = r * sin_theta;
  vec3 lmb_o = ctr_o - a * ne;

  // Compute axes to draw the limb.
  float b = r * sqrt( 1 - sin_theta * sin_theta );
  vec3 ax = b * nx, ay = b * ny;
  mat4 mvp = trans_proj * gl_ModelViewMatrix;

  // Emit a bounding square for the limb.
  for ( int i = -1; i < 2; i += 2 ) for ( int j = -1; j < 2; j += 2 ) {
        vertex_o = lmb_o + ax * i + ay * j;
        gl_Position = mvp * vec4(vertex_o,1);
        EmitVertex();
      }
  EndPrimitive();

}
```

(*b*) Appearing below is the fragment shader from the Homework 4 solution. The comments have been removed for brevity, feel free to look at the full Homework 4 solution. Show the changes needed to the fragment shader. The shader should be substantially simpler.

☐ Changes needed, based on geometry shader from previous part.

☐ The fragment shader should render lenses only, not full spheres.

☐ Simplify based on the fact that geometry shader is providing lens information.

```
void fs_main()
{
  vec4 pos_rad = sphere_pos_rad[vertex_id];
  vec3 ctr_o = pos_rad.xyz;
  float rsq = pos_rad.w * pos_rad.w;
  vec3 e_o = gl_ModelViewMatrixInverse[3].xyz;
  vec3 ef = vertex_o - e_o;   // Eye to fragment.
  vec3 ce = e_o-ctr_o;
  float qfa = dot(ef,ef), qfb = 2 * dot(ef,ce), qfc = dot(ce,ce) - rsq;
  float discr = qfb*qfb - 4 * qfa * qfc;

  if ( discr < 0 ) discard;

  const float pi = 3.14159265359;
  const float two_pi = 2 * pi;
  const bool holes = opt_holes == OHO_Holes;    // If true sphere has holes
  const bool lenses = opt_holes == OHO_Lenses;
  const bool solid = !holes && !lenses;         // If true show complete sphere.

  vec4 sur_o;  // Global surface cordinates (front or back) of sphere.
  vec3 normal_o, sur_l;
  bool front;          // If true, cordinates are of front of sphere.
  float theta, eta;    // Local angular cordinates of sphere.
  bool found_hole = false;

  for ( float dir = -1;  dir <= 1;  dir+=2 ) {
      front = dir == -1;
      float t = ( -qfb + dir * sqrt( discr ) ) / ( 2 * qfa );

      sur_o = vec4(e_o + t * ef, 1);
      normal_o = normalize(sur_o.xyz - ctr_o);

      if ( !front ) normal_o = -normal_o;

      sur_l = normalize(mat3(sphere_rot[vertex_id]) * normal_o);
      theta = atan(sur_l.x,sur_l.z);
      eta = acos(sur_l.y);

      if ( solid ) break;

      const float hole_frac = 0.2;
      const float radians_per_hole_eq = two_pi / opt_n_holes_eqt;
```

```
      const float hole_radius = 0.5 * hole_frac * radians_per_hole_eq;

      float eta_hole = round( eta / radians_per_hole_eq ) * radians_per_hole_eq;
      float r = sin(eta_hole);
      const float n_holes = floor( two_pi * r / radians_per_hole_eq );
      if ( n_holes < -1 )
        {
          if ( holes ) break;
          if ( dir == 1 ) discard;
        }

      const float radians_per_hole = two_pi / n_holes;
      float theta_hole = round( theta / radians_per_hole ) * radians_per_hole;
      vec3 hole_dir_l =
        vec3( r * sin(theta_hole), cos(eta_hole), r * cos(theta_hole) );

      float dist = distance(hole_dir_l,sur_l);
      found_hole = dist < hole_radius;
      if ( lenses && found_hole  holes && !found_hole ) break;
      if ( !front ) discard;
    }

  vec3 normal_ee = normalize(gl_NormalMatrix * normal_o);
  vec4 sur_ee = gl_ModelViewMatrix * sur_o;

  // Compute clip-space depth. Take care so that compiler avoids full
  // matrix / vector multiplication.
  vec4 sur_e = gl_ModelViewMatrix * sur_o;
  vec4 sur_c = trans_proj * vec4(0,0,2*sur_e.z,1);
  gl_FragDepth = sur_c.z / sur_c.w;

  vec2 tcoord = vec2( ( 1.5 * pi + theta ) / two_pi, eta / pi );
  vec4 texel = front ? texture(tex_unit_0,tcoord) : vec4(1);
  vec4 color2 = front ? sphere_color[vertex_id] : vec4(0.3,0,0,1);
  gl_FragColor = texel * generic_lighting(sur_ee, color2, normal_ee);
}
```