Name _____

GPU Programming
LSU EE 4702-1
Solve-Home Final Examination

Wednesday 9 December to Friday 11 December 2020 16:30 CST

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.

Warning: Unlike homework assignments collaboration is not allowed on exams. Suspected copying will be reported to the dean of students. The kind of copying on a homework assignment that would result in a comment like "See ee4702xx for grading comments" will be reported if it occurs on an exam. Please do not take advantage of pandemic-forced test conditions to cheat!

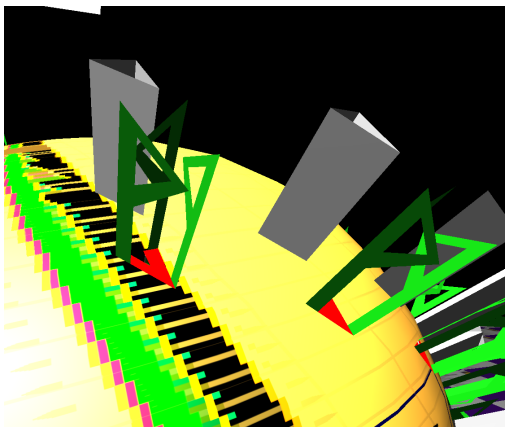
Problem 1	_____	(30 pts)
Problem 2	_____	(15 pts)
Problem 3	_____	(15 pts)
Problem 4	_____	(40 pts)
Exam Total	_____	(100 pts)

Good Luck!

Problem 1: [30 pts] The geometry shader below is based on the solution to the pre-final exam Problem 3. The geometry shader renders a prism centered on the triangle with a hole cut in it. The prism is also called a monolith in a playful reference to a recent news story. Notice that there is a `vec2` array, `pts`, declared in the shader. This array has the coordinates of the points on the pointy p from Pre-Final Exam Problem 1. The points are in the pointy p's local coordinate space and are in the right order to render the p as a triangle strip.

Modify the shader so that p's are rendered instead of monolith faces. Each p must be on the same plane as the monolith face and must be the same size. That is, the p can't extend outside where a face would be, the bottom, top and left edge must touch the respective edges of the face. The pointy part must touch or be near the right edge.

See the screenshot to the right. In the screenshot some holes have monoliths, and some have p's. In your solution just render p's.



☐ Modify the shader to render the p's where the monolith faces would go.

```
void gs_mono_p() {
  vec4 ctr_ce = AVG(vertex_e);       // Coord at triangle center.
  vec3 ctr_ne = AVG(normal_e);       // Normal at triangle center.

  // Render the triangle-with-a-hole using a triangle strip that wraps around hole.
  for ( int ii=0; ii<=3; ii++ ) {
      int i = ii % 3;
      normal_e =       f * In[i].normal_e       + (1-f) * ctr_ne;
      vertex_e =       f * In[i].vertex_e       + (1-f) * ctr_ce;
      gl_Position = gl_ProjectionMatrix * vertex_e;
      EmitVertex();     // Emit inner triangle vertex.
      normal_e = In[i].normal_e;
      vertex_e = In[i].vertex_e;
      gl_Position = In[i].gl_Position;
      EmitVertex();     // Emit original triangle vertex.
    }
  EndPrimitive();

  float size = length(In[0].vertex_e-In[1].vertex_e); // Height of monolith.
  color = vec4( 0.05, 0.5, 0.05, 1 );

  // Compute and save the coordinates of the top of the monolith, and surface normals.
  vec3 vtx_e[3], snorm_e[3];
  for ( int i=0; i<3; i++ ) {
      snorm_e[i] = size * normalize( f * In[i].normal_e + (1-f) * ctr_ne );
      vtx_e[i] = f * In[i].vertex_e.xyz + (1-f) * ctr_ce.xyz; }

  // Local x and y coordinates of the points on the pointy p.
  vec2 pts[] = { {0,0},{1,0},  {0,7},{1,6},  {4,5},{3,5},  {1,3},{1,4} };

  for ( int i=0; i<3; i++ ) {  // Iterate over faces of monolith.
      int ni = ( i + 1 ) % 3;  // Next i, possibly wrapped around.

      // Compute the normal of a face of the monolith.
      normal_e = normalize( cross( vtx_e[ni] - vtx_e[i], snorm_e[i].xyz ) );

      // Use a triangle strip to emit one face of the monolith.
      vec3 va[4] = { vtx_e[i], vtx_e[ni], vtx_e[i] + snorm_e[i], vtx_e[ni] + snorm_e[ni] };
      for ( int j=0; j<4; j++ ) {
          vertex_e.xyz = va[j];
          gl_Position = gl_ProjectionMatrix * vertex_e;
          EmitVertex();
        }
      EndPrimitive();
    }
}
```

Problem 2: [15 pts] The vertex and geometry shaders below are used in rendering pipeline T, in which the input primitives are individual triangles, and rendering pipeline S, in which the input primitive is a triangle strip. In both cases the shaders work correctly, but there might be differences in performance.

(a) Modify the vertex and geometry shader to reduce the amount of data sent from the vertex shader to the geometry shader. Do so by moving some of the work performed by the vertex shader to the geometry shader. If necessary, declare new input and output variables. For this part the shaders will be run on pipeline S. Do not make changes that result in additional computation unless those changes also reduce vertex-to-geometry shader data transfer. Of course, your changes should not change what the shaders do.

☐ Modify shaders to reduce vertex-to-geometry shader data transfer.

☐ Don't make a change the has no impact on data transfer but does increase the amount of computation.

```
void vs_main_basic() {  // The Vertex Shader
  gl_Position = gl_ModelViewProjectionMatrix * gl_Vertex;
  vertex_e = gl_ModelViewMatrix * gl_Vertex;
  normal_e = normalize(gl_NormalMatrix * gl_Normal);
  gl_BackColor = gl_FrontColor = gl_Color;
  tex_coord = gl_MultiTexCoord0.xy;
}
```

```
void gs_main_basic() {  // The Geometry Shader
  const bool type_a = glPrimitiveIDIn & 1;
  vec4 color_adjust = type_a ? vec4(0.5,0.5,0.5,1) : vec4(1);

  for ( int i=0; i<3; i++ )
    {
      gl_FrontColor = gl_FrontColorIn[i] * color_adjust;
      gl_BackColor =  gl_BackColorIn[i] * color_adjust;
      gl_Position = gl_PositionIn[i];
      tex_coord = In[i].tex_coord;
      normal_e = In[i].normal_e;
      vertex_e = In[i].vertex_e;
      EmitVertex();
    }
  EndPrimitive();
}
```

(b) The requirement to *not make changes that result in additional computation unless those changes also reduce vertex-to-geometry shader data transfer* is much easier to comply with when the shaders are run on pipeline T. Explain why and include an example of such a change. The change should not change what the shaders do and should be useful. (That is, move something.)

☐ Why is it easier to avoid additional computation when the shaders are used in pipeline T than it is when the same shaders are used in pipeline S? ☐ Note: Don't compare the absolute performance of S to T, compare how much the change impacts computation on each pipeline.

☐ Provide an example.

Problem 3: [15 pts] In Homework 3 we experimented with two ways to render a triangle with a hole in it. In the one-triangle method we relied on the fragment shader to render the hole. The geometry shader just emitted one triangle with little work. In the many-triangle method we emitted several triangles, forming a triangle-with-a-hole shape.

Based on performance measurements, we found that the one-triangle method was faster. That must mean that the extra work done by the geometry shader in the many-triangle method had more of an impact than the extra work done by the fragment shader in the one-triangle method.

Let t_{g1} denote the time used by one invocation of the geometry shader for the one-triangle method, let t_{gm} denote the time used by one invocation of the fragment shader for the many-triangles method. Let t_{f1} and t_{fm} denote the times for one invocation of the respective fragment shaders.

Let n denote the number of spheres rendered, and let g denote the number of triangles in one sphere tessellation. Finally, let f denote the fraction of the triangle covered by the hole.

(*a*) Based on these, find an expression for the time used by the geometry shaders for each method.

☐ Time for geometry shader using one-triangle method:

☐ Time for geometry shader using many-triangle method:

(*b*) Find an expression for the time used by the fragment shaders. Use n_{f1} for the total number of fragment shader invocations in the one-triangle method. (But use it for both expressions.)

☐ Time for fragment shader using one-triangle method:

☐ Time for fragment shader using many-triangle method:

(*c*) What does n_{f1} depend on? How can n_{f1} be made larger or smaller when viewing a scene without changing the primitives sent into the rendering pipeline. That is, one can't send fewer spheres or more spheres into the rendering pipeline.

☐ Quantity n_{f1} depends on:

☐ It can be changed when viewing a scene by:

Problem 4: [40 pts] Answer each question below.

(*a*) The two shaders below do the same thing, though slightly differently.

```
void vs_plan_a() {
  vertex_e = gl_ModelViewMatrix * gl_Vertex;
  gl_Position = gl_ProjectionMatrix * vertex_e;
}

void vs_plan_b() {
  vertex_e = gl_ModelViewMatrix * gl_Vertex;
  gl_Position = gl_ModelViewProjectionMatrix * gl_Vertex;
}
```

OpenGL is provided a modelview matrix and a projection matrix at the beginning of a rendering pass. In both `vs_plan_a` and `vs_plan_b` there are two matrix/vector multiplies, which require $4^2 = 16$ multiply/add operations each. But `vs_plan_b` uses `gl_ModelViewProjectionMatrix`, which is the product of the modelview and projection matrices. The product of these two matrices is computed using $4^3 = 64$ multiply/add operations. That brings the total to $16 + 16 + 64 = 96$ operations, much more than 32 for `vs_plan_a`, right?

☐ Describe the flaw with this argument.

☐ Describe a case when the argument is correct, but explain why this case does not reflect typical use.

(*b*) Answer the following questions about view volumes.

☐ What is a view volume?

☐ It is easy to determine whether a vertex is in the view volume by using its coordinate in ◯ *object space*, ◯ *eye space*, or ◯ *clip space* (check one).

☐ Given the coordinate in that space, how can one tell whether it is inside or outside the view volume?

☐ It is easy to determine whether some triangles are in the view volume. ☐ Provide an example of such a triangle and ☐ explain why.

☐ Provide an example of a triangle for which it is not so easy to determine if it is in the view volume. ☐ Illustrate with a diagram.

(*c*) Describe how suitable an OpenGL uniform variable is for the following:

☐ Explain whether this a suitable use for a uniform variable: To hold the lighted color computed by a vertex shader.

☐ Explain whether this a suitable use for a uniform variable: To hold the location of a light source.

(*d*) Vertex coordinates are usually three dimensional but texture coordinates are usually two dimensional. Why? (Ignore the w component in your answer.)

☐ Texture coordinates have two, not three, dimensions because:

(*e*) In many of our sphere examples we put the coordinates into a buffer object.

☐ What are the advantages of a buffer object over using individual `glVertex` calls to feed a rendering pipeline?

☐ What are the advantages of a buffer object over using a client (CPU) array to feed a rendering pipeline?

(*f*) A homogeneous coordinate consists of four components, compared to just three for ordinary Cartesian coordinates. Homogeneous coordinates increase the amount of work needed for a matrix/vector multiply from 9 to 16 multiplications. Transformations are realized by multiplying a transformation matrix by a coordinate.

☐ Describe a transformation that cannot be done without homogeneous coordinates.

☐ Describe a transformation that can be done using ordinary Cartesian coordinates.