Name Solution

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!

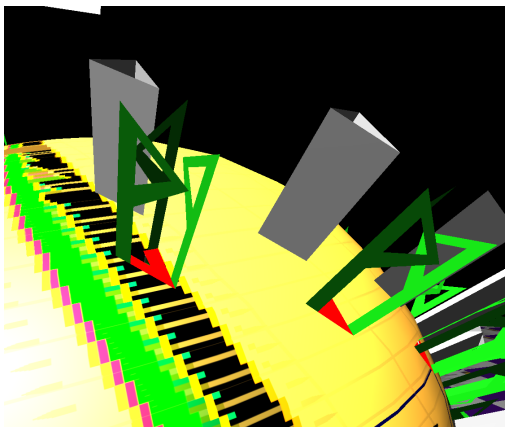
Problem 1	_____	(30 pts)
Problem 2	_____	(15 pts)
Problem 3	_____	(15 pts)
Problem 4	_____	(40 pts)
Exam Total	_____	(100 pts)

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.

The solution is on the next page.

```
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] };

      /// SOLUTION Below
      //  Compute local axes for drawing pointy p and scale them so that
      //  the pointy p fits on the monolith face.
      vec3 ax = ( vtx_e[ni] - vtx_e[i] ) / 4;
      vec3 ay = snorm_e[i] / 7;
      mat3 px = mat3(ax,ay,vec3(0));
      for ( int p=0; p<n_pts; p++ ) {  // Iterate over points describing p.
          vertex_e.xyz = vtx_e[i] + px * vec3(pts[p],0);
          gl_Position = gl_ProjectionMatrix * vertex_e;
          EmitVertex();
        }
      EndPrimitive();
      continue;
      /// SOLUTION Above
```

```
      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.

The solution appears below. Vertex output `gl_Position` is eliminated since that value can be computed in the geometry shader using `vertex_e`. There is certainly no need for both `gl_BackColor` and `gl_FrontColor` since they both carry the same value. The solution keeps `gl_BackColor`. Since `gl_FrontColor` and `gl_Position` are each of type `vec4` and a `vec4` is 16 bytes These changes reduce the vertex shader output data by 32 bytes.

```
void vs_main_basic() {  // The Vertex Shader
  // SOLUTION: REMOVE: gl_Position = gl_ModelViewProjectionMatrix * gl_Vertex;
  vertex_e = gl_ModelViewMatrix * gl_Vertex;
  normal_e = normalize(gl_NormalMatrix * gl_Normal);
  // SOLUTION: REPLACE: gl_BackColor = gl_FrontColor = gl_Color;
  gl_BackColor = gl_Color;  // SOLUTION. No need to send same color twice.
  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_BackColor =  gl_BackColorIn[i] * color_adjust;
      // SOLUTION: REPLACE gl_FrontColor = gl_FrontColorIn[i] * color_adjust;
      gl_FrontColor = gl_BackColor; // SOLUTION: Just use back color.
      // SOLUTION: REPLACE gl_Position = gl_PositionIn[i];
      gl_Position = gl_ProjectionMatrix * In[i].vertex_e; // SOLUTION
      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.

It is easier to avoid additional computation in pipeline T because there is one execution of the `i`-loop body for each vertex shader invocation and so moving a computation from the vertex shader to the `i` loop in the geometry shader does not change the amount of computation (all other things being equal).

If the shaders were used with pipeline S then for each execution of the vertex shader (in all but two cases) there would be *three* executions of the `i`-loop body (each in a different geometry shader invocation).

☑ Provide an example.

For example, consider a rectangle rendered as 10 triangles. (Yes, 2 would suffice, but suppose there were 10.) In pipeline T the vertex shader would be executed $10 \times 3 = 30$ times and the geometry shader would be executed 10 times. In each execution of the geometry shader the `i` loop executes 3 iterations, and so the loop body executes a total of $10 \times 3 = 30$ times. In the solution to the previous part the a matrix/vector multiply is eliminated from the vertex shader and one is added to the `i` loop in the geometry shader. This move does not change the total number of matrix/vector multiplications.

If that same 10-triangle rectangle were rendered in S only $10 + 2 = 12$ vertices would be sent in to the rendering pipeline and so the vertex shader would execute only 12 times. The geometry shader would still execute 10 times and the `i` loop body would still execute 30 times. So moving the matrix/vector multiplication from the vertex shader to the geometry shader would increase the amount of work by a factor of $\frac{30}{12} = 2.5$.

So with pipeline S the modifications from the previous problem impact performance in two ways. The reduction in data being sent from the vertex to geometry shader will tend to improve performance but the increase in computation will tend to reduce performance. The 4×4 matrix/vector multiply consists of 16 multiply/add (MADD) operations. In current NVIDIA GPUs the time needed for the 16 MADD instructions is much less than the time needed to move 32 bytes across the GPU chip boundary, and so even with the additional computation the change would be worthwhile.

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 geometry 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 total time used by the geometry shaders during a render pass for each method.

☑ Total rendering pass time for geometry shader using one-triangle method:

Each invocation takes t_{g1}, and a there is one invocation for each tessellated triangle, so the total time is ngt_{g1}.

☑ Total rendering pass time for geometry shader using many-triangle method:

Each invocation takes t_{gm}, and a there is one invocation for each tessellated triangle, so the total time is ngt_{gm}.

(*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.)

☑ Total rendering pass time for fragment shader using one-triangle method:

This is simply $n_{f1}t_{f1}$.

☑ Total rendering pass time for fragment shader using many-triangle method:

Since no n_{fm} has been provided it must be estimated: $n_{fm} = fn_{f1}$. So the total time is $fn_{f1}t_{fm}$.

(*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 depends on the number of pixels covered by the spheres.

☑ It can be changed when viewing a scene by:

... by moving the eye further from the scene to reduce n_{f1} or closer to the scene to increase n_{f1}.

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.

The product is computed before the rendering pass, and it is computed at most once per rendering pass. (That is because it is a uniform variable.) A rendering pass is expected to have a large number of vertex shader invocations. Suppose there are 1000 invocations. So the $4^3 = 64$ operations needed to compute the product is tiny compared to the computation performed by the vertex shaders, $32,000$ operations.

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

It would be correct if a rendering pass processed just one vertex.

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

☑ What is a view volume?

It is the part of the scene that is visible. In OpenGL it is the part of the scene inside of a cube from clip-space coordinate (-1,-1,-1) to (1,1,1).

☑ 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 in clip space if the absolute value of each component is ≤ 1 after homogenization.

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

It is easy if all three vertices are in the view volume. In that case no part of the triangle can be outside the view volume.

☑ 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.

One in which all three vertices are outside the view volume. FINISH.

(*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.

That won't work because uniform variables cannot be written by shaders, including vertex shaders. Even if the uniform were written by some other means, each vertex can have a different lighted color but the value of a uniform variable must be the same for every vertex in a rendering pass.

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

That is suitable because that would be the same for every vertex.

(*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:

Because textures are mapped on to triangles, which are two dimensional.

(*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?

Much lower overhead compared to calling `glVertex` to provide one vertex coordinate.

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

With a client array the data must be sent from the GPU to the CPU for each rendering pass, even if that data hasn't changed. A buffer object can be reused.

(*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.

Translation.

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

Scale. Also rotation.