Name _____

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

Due: 5 December 2019 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	_____	(30 pts)
Problem 2	_____	(25 pts)
Problem 3	_____	(5 pts)
Problem 4	_____	(25 pts)
Problem 5	_____	(15 pts)
Exam Total	_____	(100 pts)

Alias _____

Good Luck!

Problem 1: [30 pts] The `gs_points` shader in the solution to Homework 3, Problem 2 renders protrusions, which are blocks attached to the ring. The shader always emits five faces: the top (which we called the diamond), and the four sides. (Since it is attached to the ring there is no need to emit a bottom face.) Normally only the outside of the block is visible so the user can see at most three faces. That means at least three faces that can't be seen are emitted nevertheless. That can't be good. In this problem shader code will be modified to avoid those faces, and in the next problem the implications of this change will be analyzed.

The code on the next page is based on (but not identical to) the Homework 3 Problem 2 solution. The intention is that this be solved by hand, though solving by editing code is allowed. The code from the next page will be provided on request or may already be available.

(*a*) In the code below (meaning the next page) there is a 4-element Boolean array `visible` that is declared but not used. Add code that sets `visible[i]` to `true` is side `i` is visible to the user. By no coincidence the declaration of `visible` is just after code that sets `face_normal_e[i]` to the eye-space normal of side `i`. *Hint: Remember that in eye space the eye is at the origin (coordinate* $(0, 0, 0)$*). Also take advantage of the fact that eye-space coordinates and normals are available. Another Hint: A correct solution to this problem consists of a loop with a single statement,* `visible[i]=.x.`*, with* `.x.` *replaced by [am I giving away too much] a dot product.*

☐ Modify code so `visible[i]` set to `true` iff face `i` is visible.

(*b*) Modify the code below so that only the visible faces are emitted. This includes the diamond on top and the four sides of the protrusion. Feel free to use the `visible` array from the previous subpart (even is that subpart has not been solved). *Hint: Only one line is needed for the diamond on top. The changes needed to the code emitting the sides are more extensive.*

☐ Modify code so that diamond is not emitted if it is not visible.

☐ Modify code so that only visible sides are emitted.

Code is on the next page.

```
void gs_points() {
  /// Note: (Code setting ctop, etc. not shown.)

  // Prepare an array of object-space coordinates of vertices
  // describing the protrusion, and use that to initialize an array of
  // eye-space coordinates.
  vec3 pts_o[8] =
    { ctop,       cr,       cbot,       cl,   ctop+nc*dr, cr+nr*dr, cbot+nc*dr, cl+nr*dr };
  vec4 pts_e[8];
  for ( int i=0; i<8; i++ ) pts_e[i] = gl_ModelViewMatrix * vec4(pts_o[i],1);
  int ord[4] = { 1, 0, 2, 3 }; // Order in which to emit vertices.

  // Emit the diamond-shaped protrusion top.
  color = color_diamond;
  normal_e = gl_NormalMatrix * nl;


  for ( int i=0; i<4; i++ ) {
      vertex_e = pts_e[ord[i]];  // Retrieve vertices in the correct order.
      gl_Position = gl_ProjectionMatrix * vertex_e;
      EmitVertex();  }
  EndPrimitive();

  // Compute the eye-space normal of each protrusion face.
  vec3 face_normal_e[4];
  for ( int i0=0; i0<4; i0++ )
    face_normal_e[i0] = cross( pts_e[(i0+1) & 0x3].xyz - pts_e[i0].xyz,
                               pts_e[i0+4].xyz - pts_e[i0].xyz );

  bool visible[4];




  color = color_edge; // Color for sides.
  for ( int i=0; i<5; i++ )
    {
      int i0 = i & 0x3;          // The current edge.



      vertex_e = pts_e[i0+4];  gl_Position = gl_ProjectionMatrix * vertex_e;
      EmitVertex();
      vertex_e = pts_e[i0];   gl_Position = gl_ProjectionMatrix * vertex_e;
      EmitVertex();
      normal_e = face_normal_e[i0]; // This normal will be used for the next two vertices.
    }
 EndPrimitive();
}
```

Problem 2: [25 pts] Assume that the previous problem has been solved correctly and so only those sides of the protrusions facing the user are emitted. Below, roughly estimate the benefit of this improvement when rendering protrusions by considering the amount of work done by the vertex, geometry, and fragment shaders. Make the estimate using the original (Homework 3 solution) vertex and fragment shaders, and the geometry shader obtained by solving the previous problem. When estimating, assume that the number of emitted faces per protrusion drops from 5 to 2.5. (The actual number of faces emitted for a particular protrusion with a particular eye location and orientation can range from zero [for example, if the viewer is in the center of the ring looking at some point on the ring] to three.)

When answering the problems below consider the change in the amount of work done by each shader (the change in the amount of work using the Homework 3 solution code to the amount of work using the geometry shader from the previous problem) invocation, and the change in the number of times the shader needs to be invoked.

(*a*) For this part assume a typical view of the ring. (Don't assume that no part of the ring is visible, or other special cases.)

☐ Compared with the Homework 3 solution code, the total work for the vertex shader stage after the change above is: ◯ *about half the amount of work,* ◯ *no change in the amount of work,* ◯ *roughly the same amount of work,* ◯ *double the amount of work.*

☐ Explain. Consider ☐ work per shader and ☐ number of shader invocations.

☐ Compared with the Homework 3 solution code, the total work for the geometry shader stage after the change above is: ◯ *about half the amount of work,* ◯ *no change in the amount of work,* ◯ *roughly the same amount of work,* ◯ *double the amount of work.*

☐ Explain. Consider ☐ work per shader and ☐ number of shader invocations.

☐ Compared with the Homework 3 solution code, the total work for the fragment shader stage after the change above is: ◯ *about half the amount of work,* ◯ *no change in the amount of work,* ◯ *roughly the same amount of work,* ◯ *double the amount of work.*

☐ Explain. Consider ☐ work per shader and ☐ number of shader invocations.

(*b*) Suppose again that the ring is visible, and suppose that each invocation of the new geometry shader requires 10% more work than the geometry shader from the Homework 3 solution. (The extra time is needed to detect invisible faces.) What sort of view would make that extra work increase the total time needed for a rendering pass, and would make the extra work offset by a large reduction elsewhere?

☐ Characteristics of a view that would increase rendering time even with invisible faces omitted?

☐ Characteristics of a view that would sharply reduce rendering time with invisible faces omitted?

Problem 3: [5 pts] Appearing below is a correct solution to Homework 3 Problem 1, followed by a possible alternative solution. Indicate whether the alternative solution will render the strip as intended (the same way as the first solution) and whether there is any problem with it.

```
void gs_strip_plus() {  // Homework 3 Solution
  if ( In[2].normal_e == vec3(0,0,0) ) return;
  for ( int i=0; i<3; i++ )
    {
      normal_e = In[i].normal_e;
      vertex_e = In[i].vertex_e;
      color = In[i].color;
      gl_Position = In[i].gl_Position;
      EmitVertex();
    }
  EndPrimitive();
}
```

```
void gs_strip_plus() {  // A possible alternative solution.
  for ( int i=0; i<3; i++ )
    {
      if ( In[2].normal_e == vec3(0,0,0) ) return;
      normal_e = In[i].normal_e;
      vertex_e = In[i].vertex_e;
      color = In[i].color;
      gl_Position = In[i].gl_Position;
      EmitVertex();
    }
  EndPrimitive();
}
```

☐ Will the alternative solution render the strip correctly?

☐ Explain the problem with the alternative solution.

Problem 4: [25 pts] Answer each question below.

(*a*) Any uniform variable that a vertex shader uses can be replaced by a vertex shader input. Why is that not a good idea? Explain the impact on performance.

☐ Uniform variables should not be replaced by shader inputs because:

☐ How might one estimate the impact on performance for such a replacement?

(*b*) In some cases a vertex shader input can be replaced by a uniform variable. Describe a situation in which a vertex shader input can be replaced by a uniform, and in which a shader input cannot be replaced by a uniform. An example can be found in the shaders for Homework 3 and 4.

☐ Situation where vertex shader input can be replaced by a uniform. ☐ Explain.

☐ Situation where vertex shader input **can not** be replaced by a uniform. ☐ Explain.

(*c*) The geometry shader must write clip-space coordinates to output variable `gl_Position`. Anything else the geometry shader needs to write, such as a surface normal, can be written to variables of the programmer's choosing. Why does OGSL require clip-space coordinates in that way?

☐ Why must the geometry shader write clip-space coordinates?

☐ Why must it write them to a variable with a specific name, `gl_Position`?

(*d*) The true sphere shader presented in class rendered a sphere perfectly. Why would there be no advantage in a true cube shader?

☐ No advantage to a true cube shader because:

(*e*) The shadow volume code and our lighting code took advantage of built in variables describing how user code set lighting state such as
`gl_LightSource[i].position`, which provides the position of light `i`. Light position and other lighting variables are deprecated, meaning they may be removed in future versions of OpenGL.

☐ Why are they deprecated?

☐ Why were they there in the first place?

Problem 5: [15 pts] Many of our fragment shaders make library function calls such as `texture(tex_unit_0,TexCoord)` to obtain a filtered texel located at coordinate `TexCoord`. Coordinate `TexCoord` is something the shader code provided. Consider an alternative method of obtaining a texel in which the user prepares an array of texels bound to a buffer object named `my_texture`. The fragment shader might obtain the texel using code such as `my_texel[twid*TexCoord.x + twid*tht*TexCoord.y]`, where `twid` is the width of the texture image and `tht` is the height of the image.

(*a*) Explain why the code fragment `my_texel[twid*TexCoord.x + twid*tht*TexCoord.y]` might return a texel, but not a filtered texel. In particular, what would need to be done for it to be filtered, and why could it not provide the benefit of mipmap levels. Remember that `my_texel` is an ordinary array of RGBA values from a texture image (called texels). Each element is a `vec4`, the `vec4` members are `.r`, `.g`, `.b`, and `.a`.

☐ `my_texel[twid*TexCoord.x + twid*tht*TexCoord.y]` could not return a filtered texel because:

☐ Roughly, how could code accessing the texel array `my_texel` return a filtered texel?

☐ Explain why accessing the single texel array could not provide the benefit of multiple mipmap levels.

(*b*) The library function `texture` is part of the OpenGL shading language. Unlike `gl_LightSource`, `texture` is not deprecated, meaning that it won't be removed from OpenGL anytime soon. Why is it that lighting-related features are deprecated while texture accesses and filtering are not deprecated?

☐ The `texture` function and most other texture-related features are not deprecated because: