Name Solution

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 What's next?

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;

  if ( dot( normal_e, pts_e[0].xyz ) < 0 ) // SOLUTION -- Part b (the if statement)
  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];
  // SOLUTION -- Part a.
  for ( int i=0; i<4; i++ ) visible[i] = dot( face_normal_e[i], pts_e[i].xyz ) < 0;

  // SOLUTION -- Part b: Find the first visible side, set that to istart.
  int istart = -1;
  for ( int i=0; i<5; i++ ) if ( !visible[i&0x3] && visible[(i+1)&0x3] ) istart = i+1;
  if ( istart == -1 ) return;

  color = color_edge; // Color for sides.
  for ( int i=0; i<5; i++ ) {
      // int i0 = i & 0x3;          // The current edge.
      int i0 = ( i + istart ) & 0x3; // SOLUTION -- Part b: Start at istart.

      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.

      if ( !visible[i0] ) break; // SOLUTION -- Part b.
    }
 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.

The vertex shader is invoked once for each protrusion. The number of protrusions has not changed and the vertex shader has not changed so there is no change in the amount of work.

☑ 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.

The geometry shader is invoked once for each protrusion. The number of protrusions has not changed but the geometry shader sure has. The main changes to the code are: computing the visibility of each face, determining the first side to render (value of `iside`), and skipping faces that do not need to be rendered. The first two increase the amount of work and the last reduces the amount of work. The geometry shader now has to compute visibility for each face, but that's just an additional five dot products. Finding the first side to render requires a five iteration loop with a simple loop body.

The number of faces emitted drops from 5 to 2.5. The code emitting a face performs two (the sides) to four (the diamond) 4×4 matrix/vector multiplications (the coordinate transformations), each requiring $4 \times 4 = 16$ multiply/add operations, in contrast to 3 multiply/add operations for a dot product of a 3-element vector. Therefor avoiding just one transformation more than makes up for the five dot products. The original code performed $8 + 4 + 5 \times 2 = 22$ transformations, the new code roughly performs $8 + \frac{1}{2}4 + 5 \times 2 = 15$ transformations, which is more than half the amount of work. When the trigonometric functions are taken into account the amount of work is about the same.

Grading Note: An answer of half the work with a justification similar to the one above would receive full credit.

☑ 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.

The fragment shader has not been changed so the work per invocation is about the same. If half the number of faces are emitted then there will be half the number of fragment shader invocations. Assuming that the projected size of the visible faces is about the same as the invisible ones, the amount of work performed by the fragment shader will be about half.

(*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?

A view in which the projected size of the protrusions are small, say about three fragments each. A view like that can be obtained by moving the eye far away from the ring while still looking at it. In such a view for each geometry shader invocation there will be three fragment shader invocations. Let t_g be the amount of work done by an invocation of the original geometry shader and let t_f be the amount of work done for an invocation of the fragment shader (which hasn't changed). An invocation of the new geometry shader performs $1.1t_g$ work. Suppose that the original geometry shader emits six fragments for some view, and that the new one emits just three. The amount of work performed by the old one is $t_g + 6t_f$ and for the new one $1.1t_g + 3t_f$. For the rendering time to be increased $1.1t_g + 3t_f > t_g + 6t_f$, for the work to be equal $1.1t_g + 3t_f = t_g + 6t_f$, or $t_g = 30t_f$. So if the original geometry shader performs more than $30\times$ the work of the fragment shader, execution time will be increased with our new shader for views in which a protrusion covers just three fragments.

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

A view in which the projected size of the protrusions is large and so there are lots of fragments per primitive, say 1000 per primitive. In that case $1.1t_g + 1000t_f < t_g + 2000t_f$ will hold unless one geometry shader invocation does $10,000\times$ the work of a fragment shader invocation, which isn't very likely.

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?

Yes.

☑ Explain the problem with the alternative solution.

The alternative solution will emit two vertices that will then be abandoned, wasting effort.

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:

... because the driver will have no choice but to transfer v copies of each uniform, where v is the number of vertices. That will slow down execution. Also, uniform variables use constant memory, and so can be accessed faster.

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

Let u denote the size of the uniform variables to convert. For example, a 4×4 float transformation matrix consumes $4\times4\times4 = 64\,\mathrm{B}$, so if 2 transformation matrices are converted from uniforms to vertex shader inputs $u = 2\times64 = 128\,\mathrm{B}$. Let v denote the number of vertices in a rendering pass and Θ denote the CPU to GPU data bandwidth, for example, $\Theta = 500\,\mathrm{GB/s}$. Then the additional time per rendering pass would be $\frac{uv}{\Theta}$.

The two matrices in a 5000-vertex rendering pass on the $\Theta = 500\,\mathrm{GB/s}$ device would add $\frac{128\,\mathrm{B}\times5000}{500\,\mathrm{GB/s}} = 1.28\,\mu\mathrm{s}$ to the pass if sourced from the CPU.

(*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.

Answer: In the Homework 3 and 4 code all of the diamonds were the same color. So the color could have been assigned to a uniform.

Discussion: Instead it was provided using `glColor`. At least for the `glBegin/glEnd` rendering passes the driver had no way of knowing in advance that `glColor` would not be changed during the rendering pass and so it would provide it as a vertex shader input. For the `glDraw` rendering passes the driver could determine that the color would not change but it would need to recompile our programmable shaders to take advantage of that fact.

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

When the value won't be the same for every vertex. For example, the vertex normal in most of our examples. Since a uniform variable cannot be changed during a rendering pass it cannot be used for values that change.

(*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`?

In the rendering pipeline the rasterization stage follows the geometry shader. The rasterization stage assembles a primitive (using vertices emitted by the geometry or vertex shader) and then finds the pixels covered by that primitive. For each primitive it emits a fragment. To find the pixels covered by the primitive it needs clip space coordinates (which can be easily converted into pixel coordinates), which is why the geometry shader needs to emit clip space coordinates.

The geometry and fragment shaders (as well as vertex, tessellation, and compute shaders) are written by programmers using the OpenGL API. In contrast the rasterization stage is part of the fixed functionality, meaning it cannot be programmed, at least by OpenGL/OpenGL Shading Language users.

Let's suppose that the geometry shader emits coordinates (which it has to) and a color (which it "wants to") for each vertex. The color is read by the fragment shader which will do something with it, probably compute a lighted color. The geometry and fragment shaders were written by the same person or team, and so they will have agreed on a name for the variable holding the color, say `coler`. The rasterization stage only sees `coler` as a four-element vector, but has no idea what it's for and so the name is not important.

In contrast, the rasterization stage needs clip-space coordinates and so those must be written to a variable with a specific name.

(*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:

The true sphere shader renders a perfect sphere, meaning that exactly the pixels covered by the sphere are written. In contrast, when a sphere is approximated using triangles (the first method presented in class) the outline of the sphere will be a polygon rather than a circle unless the a very large number of triangles are used.

There is no similar problem when rendering a cube. A cube can be modeled using 12 triangles, and they will perfectly match the surface of a cube (if done correctly). Cube normals are easy too, there are only six of them. Since a cube can be perfectly rendered using triangles, there is no need for a true cube fragment shader.

(*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?

The are deprecated because lighting can easily be computed by vertex or fragment shaders. Those shaders can declare their own uniform variables for light positions and whatever else they need, and use those to compute lighted colors.

☑ Why were they there in the first place?

OpenGL is intended to provide a uniform API that can be used on a variety of 3D graphics accelerators.

Early graphics accelerators were not programmable (one could not write shaders) and so one had no choice but to use built-in lighting functionality. Execution of an OpenGL command such as `glLightfv(GL_LIGHT0, GL_POSITION, light_location)` might write a special hardware *register* (actually a memory location that the hardware monitors) with the coordinates on one device, but use a totally different method to set position on another device. The device might have had specialized hardware to compute lighting or might have been programmable, but only by the manufacturer.

GPUs became programmable, and over time their instruction sets became more general purpose, making specialized hardware for lighting and other graphics tasks unnecessary.

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:

To compute a filtered texel one needs to take some kind of a weighted average of texels that cover a pixel. A simple array lookup will just return one texel. There's no way to filter in advance since one can't usually predict pixel size and placement in advance, and there would be way too many possibilities anyway.

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

The code accessing `my_texel` could find the indices (values of `TexCoord.x` and `TexCoord.y`) of all texels covering the pixel of interest, retrieve those, and then compute a weighted average of them. (Some texels would only partially cover the pixel.)

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

Because each MIPMAP level would need to be a different array, at least if a single index were used.

(*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:

. . . because texel filtering is compute intensive. GPUs have specialized hardware, texture units, for filtering texels, and so driver-specific routines are needed. If a user were to write code that made use of the texture units, that code would have to be specific to a particular GPU or would have to be written for all of the GPUs that the code might run on. Using the `texture` function is far more convenient.