Name Solution

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

Due: 5 December 2020 at 23:59 CST

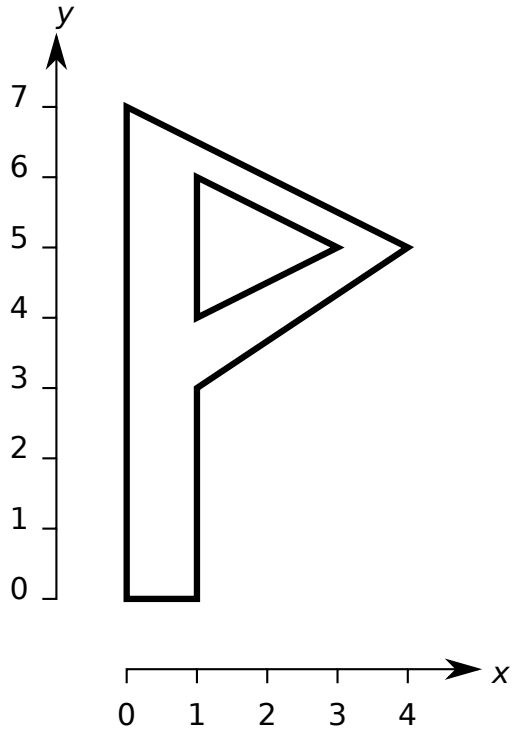
Work on this exam alone. Regular class resources, such as notes, papers, solutions, documentation, and code, can be used to find solutions. In addition outside OpenGL references and tutorials, other programming resources, and mathematical references can be consulted. Do not try to seek out references 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	_____	(20 pts)
Problem 2	_____	(20 pts)
Problem 3	_____	(30 pts)
Problem 4	_____	(30 pts)
Exam Total	_____	(100 pts)

Good Luck!

Problem 1: [20 pts] Appearing below is an illustration of a pointy p and an incomplete routine. Complete the routine so that it emits a green pointy p using a single triangle strip and `glVertex` calls.



☑ Add code to start and stop a triangle-strip rendering pass.

☑ Emit the shape using one—just one—triangle strip. Use `glV` as an abbreviation for `glVertex3f`.

☑ Use the coordinates given in the diagram. Omit the z coordinate.

☑ Set a correct normal and color.

```
void render_p() {  /// SOLUTION
  glBegin(GL_TRIANGLE_STRIP);

  glColor3f(0,.2,0);
  glNormal3f(0,0,1);

  glVertex3f(0,0,0);
  glVertex3f(1,0,0);

  glVertex3f(0,7,0);
  glVertex3f(1,6,0);

  glVertex3f(4,5,0);
  glVertex3f(3,5,0);

  glVertex3f(1,3,0);
  glVertex3f(1,4,0);

  glEnd();
}
```

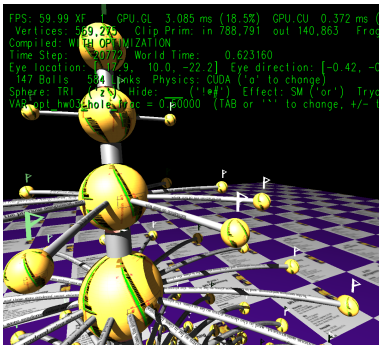
Problem 2: [20 pts] Consider `render_p` from the previous problem.

(*a*) Compute the amount of data sent from the CPU to the GPU each time `render_p` is called. If you did not solve the previous problem then assume that `render_p` calls `glVertex3f` eight times.

☑ Amount of data due to execution of `render_p`. ☑ Show units and state assumptions.

Each execution of `render_p` sends one normal, one color, and eight vertices from the CPU to the GPU and into the rendering pipeline. All of these send floats, which are 4 bytes each. Each sends three floats (as can be seen by the `3f` suffix). So the total amount of data is $(1 + 1 + 8) \times 3 \times 4 = 120\,\mathrm{B}$.

(*b*) The code below calls `render_p` for each ball. Before calling `render_p` the modelview matrix is modified so that the pointy p is rendered above the sphere. Suppose that the updated modelview matrix is sent to the GPU at the beginning of a rendering pass.



```
void World::render_prob1() {
  glMatrixMode(GL_MODELVIEW);
  for ( Ball *ball: balls ) {
      glPushMatrix();
      pNorm b_to_eye(ball->position,eye_location);
      pVect d_p = ball->omatrix * pVect(0,1,0);
      pCoor p_pos = ball->position + 1.5 * ball->radius * d_p;
      pVect az = b_to_eye;
      pNorm ax = cross(b_to_eye,pVect(0,-1,0));
      pVect ay = cross(az,ax);
      float s = 0.2 * ball->radius;
      pVect vx = s * ax, vy = s * ay, vz = s * az;
      pMatrix xform(vx,vy,vz,p_pos); // Concatenate vectors into a matrix.
      glMultTransposeMatrixf(xform);
      render_p();
      glPopMatrix();
    }}
```

Consider an alternative implementation where the vertex shader computes and uses the updated modelview matrix rather than having it computed by the routine above.

☑ How much data (in bytes) would this vertex shader invocation need to read in order compute the new matrix?

The vertex shader would need to read the values of the following variables: `ball->position`, `ball->omatrix`, `ball->radius`, `eye_location`, and implicitly, the current value of the modelview matrix.

Variable `ball->position` and `eye_location` are each four-element vectors, and so $4 \times 4 = 16\,\mathrm{B}$ each. Variable `ball->omatrix` is a 3×3 vector of floats, its total size is $3 \times 3 \times 4 = 36\,\mathrm{B}$. The modelview matrix is a 4×4 vector of floats, its total size is $4 \times 4 \times 4 = 64\,\mathrm{B}$. Finally, `ball->radius` is one float which is $4\,\mathrm{B}$.

Not all of this data needs to be sent. Only one column of `ball->omatrix` is used, reducing its contribution to $12\,\mathrm{B}$. Assuming the w component of `ball->position` and `eye_location` can assumed to be 1, only three floats each need to be sent. Presumably the modelview matrix will be sent anyway, so it would not be in the spirit of the problem to count it.

Taking these into account the amount of data is $12 + 12 + 12 + 4 = 40\,\mathrm{B}$.

Uniform variables could be used for `eye_location` and the modelview matrix since they are the same for every vertex in a rendering pass. The shading language compatibility profile automatically puts the modelview matrix in uniform variable `gl_ModelviewMatrix`.

Since `render_p` performs an entire rendering pass then uniform variables could be used for the other variables (`ball->position`, etc.) too. But if `render_p` did not stop and start the rendering pass, but instead that was done by code outside the `ball` loop, then the other variables would have to be shader inputs because they would be different at each iteration.

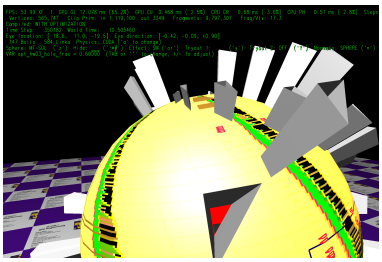
☑ Explain why the total amount of computation using this vertex shader would be larger.

In `render_prob1` the computation used to compute `xform` is done once per iteration (ball). But if `xform` were computed in the vertex shader then it would be done once per vertex, which is eight times as often for `render_p`.

☑ Explain why, even if the total computation were larger, execution time might still be faster.

Because the GPU is highly parallel so it can perform the computation more quickly. (Though in this case with only eight vertices in a rendering pass it won't make much of a difference.)

Problem 3: [30 pts] The shader code in routine `gs_monolith` below is based on the solution to Homework 3, in which the sphere was to be rendered with triangular holes.



(*a*) Modify the code so that there is a prism-shaped gray monolith over each hole. Render only the sides of the prisms, not the ends. A viewer looking down the axis of a prism should be able to see through the prism and into the sphere, perhaps to check whether it's full of stars. See the screenshot to the right. (Sorry, no stars.)

☑ Emit the prism faces. (The coordinates of one end of the prism are the same as the coordinates of the hole.)

☑ Determine the normal of each prism face, and emit the normal. ☑ All fragments of a face must have the same normal.

☑ Correctly set `max_vertices`. Don't set it larger than needed.

(*b*) The problem above could be solved using one triangle strip to render the prism, or using three triangle strips (one for each face). For the one-triangle-strip solution there would be a problem with the normals. Explain what that problem is, and how it might be fixed by modifying the fragment shader. (Not shown.)

☑ Problem with normals with a one-triangle-strip prism.

Normals are associated with vertices. In the prism shape rendered using a single triangle strip a vertex belongs to two faces of the prism. The normal can't be correct for both of them. The value of the normal seen by the fragment shader will be a blend of the normals provided at each edge.

☑ Key ideas (but not complete code) to fix normals by modifying fragment shader.

One solution would be to add two geometry shader outputs, `bool mono`, and `flat vec3 nflat_e`. When `mono` is false the fragment shader would use the usual normal, `normal_e`, but when `mono` is true it would use `nflat_e`. Because `nflat_e` is declared `flat` it won't be interpolated. Its value will be the one specified by the provoking (last) vertex of the triangle. The geometry shader can set `nflat_e` appropriately.

```
layout ( triangle_strip, max_vertices = 8 ) out;
void gs_monolith() {
  vec4 ctr_ce = AVG(vertex_e);       // Coord at triangle center.
  vec3 ctr_ne = AVG(normal_e);       // Normal at triangle center.
  vec2 ctr_tx = AVG(gl_TexCoord[0]); // Texture coord at triangle center.

  const float f = tryoutf.y; // Relative hole size.
  color = In[0].color;

  // Render the triangle-with-a-hole using a triangle strip that wraps
  // around the hole in the triangle.
  for ( int ii=0; ii<=3; ii++ ) {
      int i = ii % 3;  // Values of i:  0, 1, 2, 0.  (0 appears twice.)
      vec3 n_e = In[i].normal_e;

      // Prepare and emit inner vertex by blending outer vertex and center.
      normal_e =       f * n_e                  + (1-f) * ctr_ne;
      vertex_e =       f * In[i].vertex_e       + (1-f) * ctr_ce;
      gl_TexCoord[0] = f * In[i].gl_TexCoord[0] + (1-f) * ctr_tx;
      gl_Position = gl_ProjectionMatrix * vertex_e;
      EmitVertex();

      // Prepare and emit vertex of original triangle.
      normal_e = n_e;
      vertex_e = In[i].vertex_e;
      gl_TexCoord[0] = In[i].gl_TexCoord[0];
      gl_Position = In[i].gl_Position;
      EmitVertex();
    }
  EndPrimitive();

  color = vec4(0.5,0.5,0.5,1); // Change color to gray.
```

Solution code on next page.

```
  /// SOLUTION
  vec3 vtx_e[3], snorm_e[3];

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

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

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

      vertex_e.xyz = vtx_e[i];
      gl_Position = gl_ProjectionMatrix * vertex_e;
      EmitVertex();

      vertex_e.xyz = vtx_e[ni];
      gl_Position = gl_ProjectionMatrix * vertex_e;
      EmitVertex();

      vertex_e.xyz = vtx_e[i] + snorm_e[i];
      gl_Position = gl_ProjectionMatrix * vertex_e;
      EmitVertex();

      vertex_e.xyz = vtx_e[ni] + snorm_e[ni];
      gl_Position = gl_ProjectionMatrix * vertex_e;
      EmitVertex();

      EndPrimitive();
    }
}
```

Problem 4: [30 pts] Answer each question below.

(*a*) A program uses OpenGL and includes shader code. That program runs fine on a system using a GPU. Is it possible in principle to run that same code on a system without a GPU? The system still has a display which can show graphics. Explain why or why not.

☑ Can the program run on the system without a GPU?

☑ Explain why it can run or why it cannot ever run.

Yes it can run on the system without a GPU. That includes shader code, which can be compiled for the CPU just as easily as it can be compiled for a GPU. Though a GPU can execute many threads (or shader invocations) in parallel, that is not necessary for correctness. So the CPU could just have one thread compute everything.

(*b*) In class we described how vertices could be assigned different kind of colors (material properties) such as diffuse and specular using OpenGL calls such as `glMaterial`. From these a lighted color would be computed, accounting for the material properties and also light properties. Do current GPUs have special hardware to compute this lighted color? Explain.

☑ Do current GPUs (not just Ampere) have hardware to compute lighted colors? ☑ Explain.

No. Lighting would be computed by programmer-written shaders, or if the programmer did not provide shaders, in default shaders provided by the OpenGL implementation. There is no special hardware to compute lighting.

☑ Why was it necessary for OpenGL to have calls such as `glMaterial`, `glLight`, and so on?

Early GPUs were not programmable and had special hardware to compute lighting. The OpenGL lighting-related calls provided the data that this special hardware needed.

(*c*) Appearing below are three variations on a geometry shader. They are part of a rendering pipeline that also includes a fragment shader. The shader labeled `Original` is written clearly. A bored programmer came up with the `Stunt A` and `Stunt B` versions. Suppose that all variables are of type `vec4`, and that the names of the input variables accurately describe their contents. Explain whether each of the Stunt variations can be made to work by modifying the fragment shader.

```
void gs_typical() {  // Original
  for ( int i=0; i<3; i++ ) {
      gl_FrontColor = In[i].color;
      gl_BackColor = In[i].color;
      gl_Position = In[i].gl_Position;
      gl_TexCoord[0] = In[i].gl_TexCoord[0];
      EmitVertex();
    }
  EndPrimitive();
}
```

```
void gs_typical() { // Stunt A
  for ( int i=0; i<3; i++ ) {
      gl_FrontColor = In[i].color;
      gl_BackColor = In[i].gl_TexCoord[0];
      gl_Position = In[i].gl_Position;
      gl_TexCoord[0] = In[i].color;
      EmitVertex();
    }
  EndPrimitive();
}
```

```
void gs_typical() {  // Stunt B
  for ( int i=0; i<3; i++ ) {
      gl_FrontColor = In[i].color;
      gl_BackColor = In[i].gl_TexCoord[0];
      gl_Position = In[i].color;
      gl_TexCoord[0] = In[i].gl_Position;
      EmitVertex();
    }
  EndPrimitive();
}
```

☑ `Stunt A` ⊗ *can be made to work* or ◯ *cannot be made to work* .

☑ Explain why or why not.

It could only be made to work if only the back of every primitive faced the user. In that case the fragment shader would use `gl_Color` (which would get the value of `gl_BackColor` as the texture coordinate and `gl_TeXCoord` as the color. But it is not realistic to expect that.

☑ `Stunt B` ◯ *can be made to work* or ⊗ *cannot be made to work* .

☑ Explain why or why not.

It cannot be made to work because the rasterizer stage, which is between the geometry and fragment stages, uses `gl_Position`, and so it must be set correctly. In contrast, nothing between the geometry and fragment stages uses `gl_TeXCoord` nor `gl_BackColor` so swapping there values would go unnoticed.

(*d*) The code fragment below, taken from the classroom demo-7 code, updates the buffer object when `gpu_buffer_stale` is `true`. Variable `gpu_buffer_stale` is set to `true` initially, and again only when something about the sphere changes (for example, the number of triangles used to approximate it). The line commented `DISASTER` was added for this problem. Explain what will go wrong. *Hint: The problem occurs when* `gpu_buffer_stale` *is frequently true. Note: This problem also appeared on the 2019 final exam.*

```
if ( gpu_buffer_stale ) {
    gpu_buffer_stale = false;

    // Generate buffer id (name), if necessary.
    if ( !gpu_buffer ) glGenBuffers(1,&gpu_buffer); // ORIGINAL
    glGenBuffers(1,&gpu_buffer);                    // DISASTER

    // Tell GL that subsequent array pointers refer to this buffer.
    glBindBuffer(GL_ARRAY_BUFFER, gpu_buffer);

    // Copy data into buffer.
    glBufferData
      (GL_ARRAY_BUFFER,           // Kind of buffer object.
       coords_size*sizeof(pCoor), // Amount of data (bytes) to copy.
       sphere_coords.data(),      // Pointer to data to copy.
       GL_STATIC_DRAW);           // Hint about who, when, how accessed.

    // Tell GL that subsequent array pointers refer to host storage.
    glBindBuffer(GL_ARRAY_BUFFER, 0);
  }
```

☑ Explain the disaster that the `DISASTER` line causes.

Buffer leak. Each time block is executed a new buffer object is allocated and filled, but the old one is kept around and so memory or some other resource will eventually be used up.

(*e*) A vertex shader appears below, it is based on code from `links-shdr.cc` in the links demo and is used to render spheres using an instanced rendering pass. If there are 20 spheres in the scene and each sphere has 300 vertices, the vertex shader is called $20 \times 300 = 6000$ times. Label the variables in the routine below as follows:

☑ Label each OpenGL defined uniform variable with a U.

☑ Label each OpenGL defined input variable with an I.

☑ Label each OpenGL defined output variable with an O.

```
void vs_main_instances_sphere() {
  vec4 pos_rad = sphere_pos_rad[gl_InstanceID];
  float rad = pos_rad.w;
  mat4 rot = transpose(sphere_rot[gl_InstanceID]);
  vec4 normr = rot * gl_Vertex;
  normal_o = normr.xyz;
  vertex_o = vec4( pos_rad.xyz + rad * normal_o, 1 );

  gl_Position = gl_ModelViewProjectionMatrix * vertex_o;
  gl_TexCoord[0] = gl_MultiTexCoord0;
  color = sphere_color[gl_InstanceID];
}
```

(*f*) The storage type of `sphere_pos_rad` in `vs_main_instances_sphere` from the previous subproblem could be a shader input, a uniform variable, or a buffer object. Assume that all could be made to work correctly under some circumstances.

☑ Would it be ◯ *good* , ◯ *doable* , or ⊗ *bad* for `sphere_pos_rad` to be a shader input? ☑ Explain. If bad, how bad.

If the entire array were a vertex shader input that would likely be too large and even if it could fit it would be wasteful because the amount of data sent down the rendering pipeline would be the size of `sphere_pos_rad` **times** the number of vertices.

☑ Would it be ⊗ *good* , ◯ *doable* , or ◯ *bad* for `sphere_pos_rad` to be a uniform variable? ☑ Explain. If bad, how bad.

There is limited uniform space. If it would fit then it might work well. But, it would just be sent once per rendering pass, which is much better than sending it as a vertex shader input.

☑ Would it be ⊗ *good* , ◯ *doable* , or ◯ *bad* for `sphere_pos_rad` to be a buffer object? ☑ Explain. If bad, how bad.

This would work best because it could be any size and there would be less wasted data transfer.