Name _____

GPU Programming
EE 4702-1
Midterm Examination

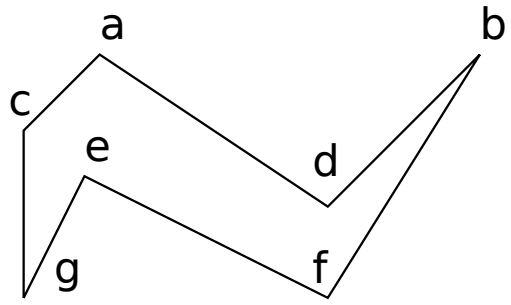
Friday, 28 October 2016 14:30–15:20 CDT

Problem 1	_____	(20 pts)
Problem 2	_____	(30 pts)
Problem 3	_____	(15 pts)
Problem 4	_____	(35 pts)
Exam Total	_____	(100 pts)

Alias _____

Good Luck!

Problem 1: [20 pts] Appearing below is a figure with labeled vertices. In the code below `pa`, `pb`, ... are initialized to the coordinates of the corresponding vertices. Two arrays are declared, `coords_wrong` and `coords`.




```
void World::render_p1(pCoor p1, pCoor p2, pCoor p3) {
  pCoor pa(2,4,0);
  pCoor pb(7,4,0);
  pCoor pc(1,3,0);
  pCoor pd(5,2,0);
  pCoor pe(1.8,2.4,0);
  pCoor pf(5,0.8,0);
  pCoor pg(1,0.8,0);

  pCoor coords_wrong[] = { pa, pb, pc, pd, pe, pf, pg };

  pCoor coords[] = {                                    } ; //  ☐  Fill In

  glMatrixMode(GL_MODELVIEW);
  glPushMatrix();









  pMatrix m =                                             ; //  ☐  Fill In

  glMultTransposeMatrixf(m);


  glBegin(GL_TRIANGLE_STRIP);
  for ( int i=0; i<coords.size(); i++ ) glVertex3fv( coords[i] );
  glEnd();    glPopMatrix();  }
```

(*a*) Initialize array `coords` so that the figure is rendered properly.

☐ Put `pa` ... `pg` into the `coords` initialization above in the correct order.

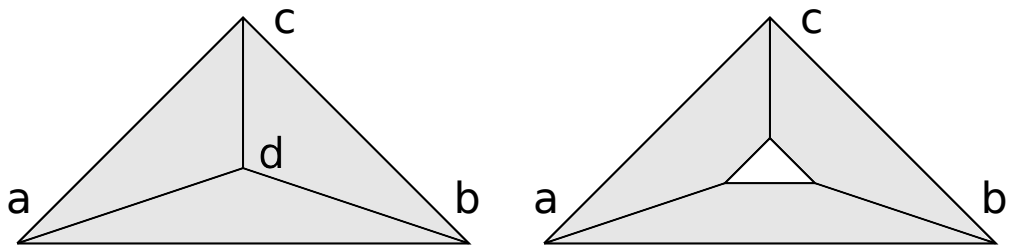
(*b*) Complete the code for the modelview matrix so that the whole figure is moved such that vertex `g` is at coordinate `p1`. (Note: the problem below also solves this problem.)

☐ Assign the correct value for `m` in the code above and make any other necessary changes.

(*c*) Set the modelview matrix so that the figure is rendered such that vertex `g` is at coordinate `p1`, vertex `c` is at `p2`, and all points are in the plane formed by `p1`, `p2`, and `p3`. This will move and rotate the figure, but won't change its shape. The matrix constructors `pMatrix_Rows rot1(v1,v2,v3)` and `pMatrix_Cols rot2(v1,v2,v3)` may come in handy. *Note: In the original version of the exam* `b` *was to be moved to* `p3`.

☐ Assign `m` to move the entire figure so that `g` is at `p1`, `c` is at `p2`, and `b` is at `p3`.

Problem 2: [30 pts] Illustrated below are two possible outputs of a geometry shader processing input primitive *abc*. The geometry shader code shown below, when completed, renders the figure on the left (this page's problem) or right (next page's problem).




```
layout (   triangle_strip,   max_vertices = 3   ) out;          //  ☐   Fix.

void gs_main_1() {
  vec3 pt_sum = vec3(0);    vec3 n_sum = vec3(0);
  for ( int i=0; i<3; i++ ) { pt_sum += In[i].vertex_e.xyz; n_sum += In[i].normal_e; }
  vec4 center_e = vec4( pt_sum/3, 1 );
  vec3 center_norm_e = n_sum/3;




  for ( int i=0; i<=3; i++ ) {
      int idx = i%3;
      vertex_e = center_e;   normal_e = center_norm_e;
      gl_Position = In[idx].gl_Position;
      EmitVertex();

      normal_e = In[idx].normal_e;   vertex_e = In[idx].vertex_e;
      gl_Position = In[idx].gl_Position;
      EmitVertex(); }
  EndPrimitive(); }
```

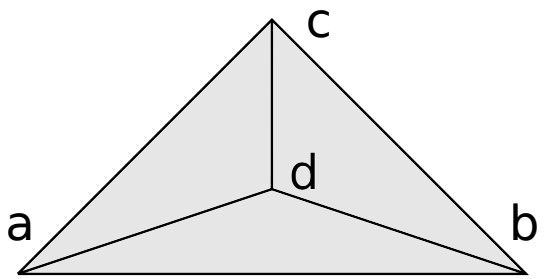
(*a*) Fix the layout declaration and modify the code so that `gl_Position` is assigned a correct value for the center point (d in the diagram). Assume that `center_e` is already correct. *Hint: (Don't just look at the figure for the layout declaration fix.).*

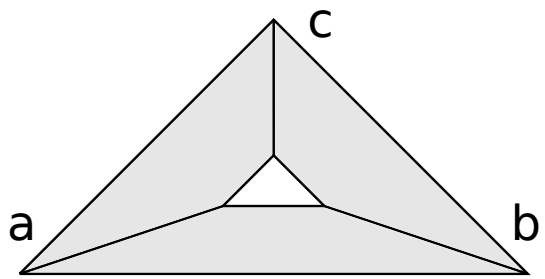
☐ Fix the layout declaration. ☐ Assign `gl_Position` correctly for the center point.

(*b*) Suppose that the goal is to split *abc* into three triangles that each have $\frac{1}{3}$ the fragments that *abc* would have had. Does the code above (assuming `gl_Position` is set correctly) achieve that goal? If not, briefly explain how to modify the code so that there will be three triangles with $\frac{1}{3}$ the vertices each.

☐ Are fragments equally split? If no, explain how to fix.

Problem 2, continued:






```
layout (   triangle_strip,   max_vertices = 3   ) out;          //  ☐   Fix.

void gs_main_1() {
  vec3 pt_sum = vec3(0);    vec3 n_sum = vec3(0);
  for ( int i=0; i<3; i++ ) { pt_sum += In[i].vertex_e.xyz; n_sum += In[i].normal_e; }
  vec4 center_e = vec4( pt_sum/3, 1 );
  vec3 center_norm_e = n_sum/3;









  for ( int i=0; i<=3; i++ ) {
      int idx = i%3;









      vertex_e = center_e;  normal_e = center_norm_e;
      gl_Position = In[idx].gl_Position;
      EmitVertex();

      normal_e = In[idx].normal_e;   vertex_e = In[idx].vertex_e;
      gl_Position = In[idx].gl_Position;
      EmitVertex(); }
  EndPrimitive(); }
```

(*c*) Modify the geometry shader so that it splits the triangle into the shape on the right, which is like the original triangle with a triangle-shaped hole in the center. The rendered primitives should only cover the area shown in gray. The exact area of the hole is not important, as long as it's not zero.

☐ Modify shader to render hole as shown in the shape on the right.

Problem 3: [15 pts] The incomplete code below is supposed to test whether pairs of balls *interpenetrate*, meaning that there is a volume of space that both balls occupy, and if so apply a separation force. The separation force works like an ideal spring pushing apart the balls. *Note: in the original exam the phrase defining interpenetration read "they occupy the same space".*

```
for ( int i=0; i<chain_length; i++ ) for ( int j=i+1; j<chain_length; j++ )
  {
    Ball* const ball_i = &balls[i];
    Ball* const ball_j = &balls[j];
    pCoor pos_i = ball_i->position;
    pCoor pos_j = ball_j->position;
    float rad_i = ball_i->radius;
    float rad_j = ball_j->radius;









    bool interpenetrating =                                    ; //  ☐  Fill in.

    if ( ! interpenetrating ) continue;









    ball_i->force +=                                           ; //  ☐  Fill in.

    ball_j->force +=                                           ; //  ☐  Fill in.
  }
```

(*a*) Write code to determine whether the two balls are interpenetrating, and assign the result to `interpenetrating`.

☐ Set `interpenetrating` to true if balls are intersecting.

(*b*) Write code to add on a separation forces to the balls. Use `sep_spring_constant` for the spring constant.

☐ Write code to add the separation force to balls' `force` members. ☐ Don't forget that force is a vector quantity.

Problem 4: [35 pts] Answer each question below.

(*a*) Convert homogeneous coordinate $\begin{bmatrix} 1 \\ 2 \\ 3 \\ 4 \end{bmatrix}$ into an ordinary Cartesian coordinate.

☐ Cartesian coordinate:

(*b*) Explain why `gl_Vertex`, the object-space vertex coordinate vertex shader input, is only in the compatibility profile (meaning that it's not really needed) but `gl_Position`, the output of the vertex and geometry shader stages is in the core profile (meaning that it is really needed).

☐ Input `gl_Vertex` **is not** really needed because:

☐ Output `gl_Position` **is** really needed because:

(*c*) The modelview matrix is predefined as a uniform variable. What would be the disadvantage of making it a vertex shader input?

☐ Disadvantage of making modelview a vertex shader input.

(*d*) The code below updates a buffer object whenever its old contents becomes outdated. Two lines have been accidentally commented out. What are the consequences of each.

```
//      if ( gpu_buffer_stale )  // LINE A, ACCIDENTALLY Commented out.
   {
//          if ( !gpu_buffer )   // LINE B, ACCIDENTALLY Commented out.
            glGenBuffers(1,&gpu_buffer);
          glBindBuffer(GL_ARRAY_BUFFER, gpu_buffer);
          glBufferData
            (GL_ARRAY_BUFFER, coords_size*sizeof(float),
             coords, GL_STATIC_DRAW);

          glBindBuffer(GL_ARRAY_BUFFER, 0);
          gpu_buffer_stale = false;
        }
```

☐ Consequences of LINE A commented out, but LINE B not commented out, in particular on ☐ performance and ☐ stability.

☐ Consequences of LINE B commented out, but LINE A not commented out, in particular on ☐ performance and ☐ stability..

(*e*) What are the typical operations performed in a fragment shader? What must the fragment shader do?

☐ Typical operations for fragment shader.

☐ Mandatory operation for fragment shader.