Name Solution

GPU Programming
EE 4702-1
Midterm Examination

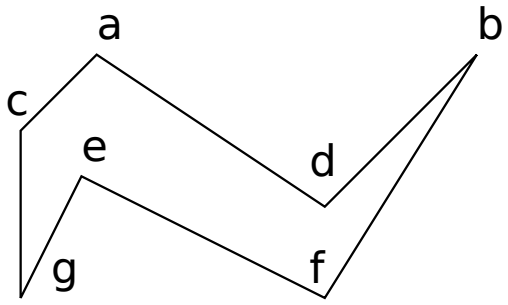
Friday, 28 October 2016 14:30–15:20 CDT

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

Alias It Begins

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 };

  /// SOLUTION – Part a
  pCoor coords[] = { pc, pg, pa, pe, pd, pf, pb };

  /// SOLUTION – Problem 1b. (Problem 1c also solves 1b.)
#if 0
  pVect vg1 = p1 - pg;
  pMatrix_Translate move(vg1);
  pMatrix m = move;
#endif

  glMatrixMode(GL_MODELVIEW);
  glPushMatrix();

  /// SOLUTION – Problem 1c. Step 1: Transform to a local space.
  //
  pMatrix_Translate to_origin(-pg);
  pMatrix_Scale scale( 1/(pb.x-pc.x), 1/(pc.y-pg.y), 1);

  // SOLUTION -- Problem 1c. Step 2: Rotate, scale, and translate for p1, p2, p3.
  //
  pVect ay(p1,p2);
  pVect v23(p2,p3);
  pVect az = cross(v23,ay);
  pNorm axn = cross(ay,az);
  float width = dot(axn,v23);
  pVect ax = width * axn;
  pMatrix_Cols rot_glo(ax,ay,az);
  pMatrix_Translate to_p1(p1);

  pMatrix m = to_p1 * rot_glo * scale * to_origin;
  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.

Solution appears above. Note that the primitive type is a triangle strip, so each vertex appears once.

(*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.

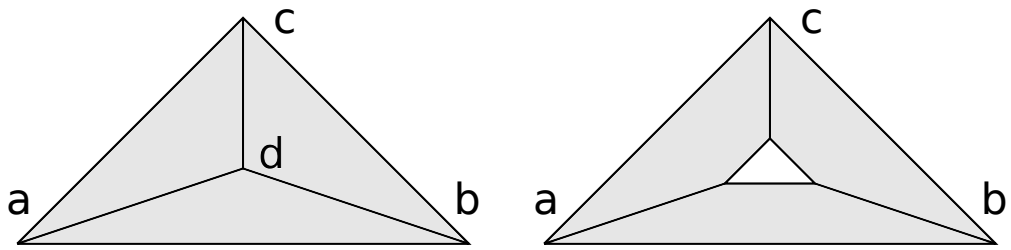
Solution appears above. A translation matrix is constructed using a vector from `pg` to `p1`. This code is guarded by an `#if 0` because part c also solves this problem.

(*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`.

The solution appears above. The first step is to transform the points to a local space in which `pg` is at the origin, the distance from `pc` to `pg` is 1 unit along the y axis, and the distance from `pc` to `pb` is 1 unit along the x axis. The second step is to rotate the coordinates using the vector from `p1` to `p2` as the new y axis, etc. The final step is to translate the figure to `p1`.

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).




```
/// SOLUTION – Problem 2a – Change max_vertex from 3 to 8.
layout ( triangle_strip, max_vertices = 8 ) out;

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;

  /// SOLUTION – Problem 2b
  //  Transform coordinate of center point from eye space to clip space.
  vec4 center_c = gl_ProjectionMatrix * center_e;

  for ( int i=0; i<=3; i++ ) {
      int idx = i%3;
      vertex_e = center_e;   normal_e = center_norm_e;

      /// SOLUTION – Problem 2b
      //  Assign clip-space coordinate to geometry shader output.
      gl_Position = center_c;

      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.

Solution appears above. The loop iterates four times and `EmitVertex` appears twice in the loop body, and so eight vertices are emitted.

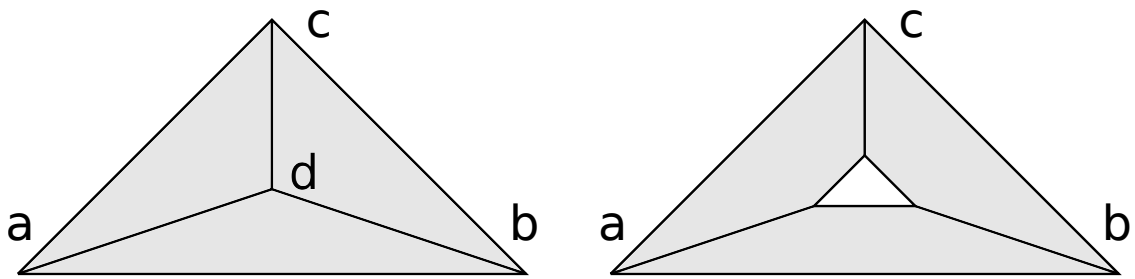
(*b*) Suppose that the goal is to split abc into three triangles that each have $\frac{1}{3}$ the fragments that abc would have had. Does the code above (assuming `gl_Position` is set correctly) achieve that goal? If not, briefly explain how to modify the code so that there will be three triangles with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

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

Short answer: They are not necessarily equally split because the number of fragments generated for a triangle depends upon its clip-space coordinates, and so it doesn't matter that abc has been split into three triangles of equal area in eye space (plus three triangles of zero area). It can be fixed by computing the center point using the clip space coordinates of a, b, and c.

More details: The shader above emits six triangles: three cover zero fragments (such as triangle dcd) while the number of fragments covered by the remaining three triangles depends on their clip space coordinates (computed using the projection matrix) and whether they are within the view volume. Even assuming that the entire triangle were in the view volume, the number of fragments could vary, for example, when the eye is closer to vertex a than b or c, triangle bdc will have fewer fragments than acd. To fix this problem the center point should be chosen based on clip space coordinates rather than eye space coordinates. Then the eye-space coordinate of the center point would be computed using the inverse of the projection matrix.

Problem 2, continued:




```
/// SOLUTION – Problem 2c
layout ( triangle_strip, max_vertices = 8 ) out;

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;



      /// SOLUTION – Problem 2c
      vec3 ctr_to_vtx = In[idx].vertex_e.xyz - center_e.xyz;
      vertex_e = vec4( center_e + 0.5 * ctr_to_vtx, 1 );
      normal_e = ( center_norm_e + In[idx].normal_e ) * 0.5;
      gl_Position = gl_ProjectionMatrix * vertex_e;
      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.

Solution appears above. Rather than emit the center point, the code emits a vertex halfway between the center point and a triangle vertex. The normal is also scaled.

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;

    /// SOLUTION – Part a
    pNorm pos_i_to_j(pos_i,pos_j);
    float ip_dist = rad_i + rad_j - pos_i_to_j.magnitude;
    bool interpenetrating = ip_dist > 0 ;

    if ( ! interpenetrating ) continue;

    /// SOLUTION – Part b
    pVect sep_force = ip_dist * sep_spring_constant * pos_i_to_j;

    ball_i->force += -sep_force;
    ball_j->force += sep_force;
  }
```

(*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.

Solution appears above. The code computes an interpenetration distance, `ip_dist`, the length of the overlapping region on the line between their centers. The balls are intersecting if this quantity is positive.

(*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.

Solution appears above. Apply a force along the line connecting their centers. The force is proportional to the amount of intersection.

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:

The Cartesian coordinate is $\begin{bmatrix} 1/4 \\ 1/2 \\ 3/4 \end{bmatrix}$.

(*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:

Because `gl_Vertex` is only used by shader code, not by fixed functionality. So if the vertex shader code needs the object-space vertex coordinate, then that can be declared as an input (with whatever name the coder wants). There is no need for a pre-defined name for the object space vertex coordinate because none of the fixed functionality uses is.

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

Because `gl_Position`, the vertex's clip-space coordinate is used by the fixed functionality, in particular by the rasterizer to perform clipping and to generate fragments. Therefore, the name of the shader output must be a name known to the rasterizer (something pre-defined) and not something made up by the person who wrote the shaders.

(*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.

As a vertex shader input it would have to be copied and moved so that each shader invocation had its own value. And remember that the value consists of 16 floats. Consuming perhaps communication bandwidth and time. That's a waste because the value would be the same for an entire rendering pass. Furthermore, uniforms are probably placed in a higher speed memory than shader inputs.

(*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.

The code would execute every time, not just when the data was stale. This would hurt performance because of the need to send more data from the CPU to the GPU. It would not affect stability.

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

A new buffer object would be created each time, but without discarding the old buffer object. This would consume more and more memory, slowing execution and ultimately causing an execution error, or if the programmer was careful, an apology and an early exit.

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

☑ Typical operations for fragment shader.

Fragment shaders typically read texels and blend them with the incoming color. Fragment shaders can also perform lighting.

☑ Mandatory operation for fragment shader.

The fragment shader must write the final color and depth (z value) to the fragment shader outputs.