Name ______________________

GPU Programming
EE 4702-1
Midterm Exam

21 October 2022, 9:30-10:20 CDT

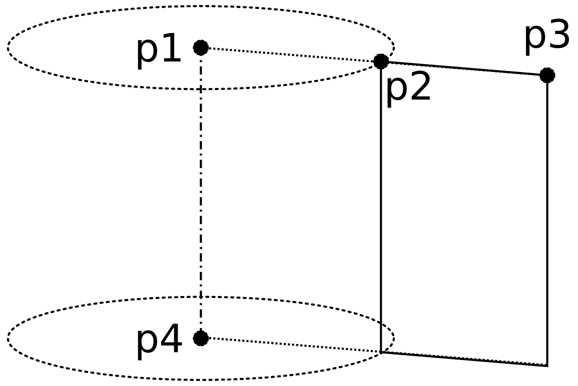
Problem 1	________	(30 pts)
Problem 2	________	(15 pts)
Problem 3	________	(15 pts)
Problem 4	________	(40 pts)
Exam Total	________	(100 pts)

Alias ______________________

Good Luck!

Problem 1: [30 pts] Appearing below is code based on the solution to Homework 2, in which a paddle wheel was to be rendered with the shape and positioning determined by given points `p1`, `p2`, `p3`, and `p4`. The code has been modified to work with the course Vulkan library and the shaders written for Homework 3.

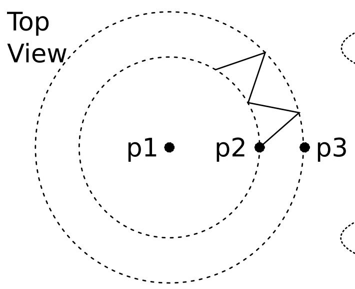
(*a*) The last lines insert normals into the buffer set. These normals are obviously wrong. Modify the code to compute the correct normal.

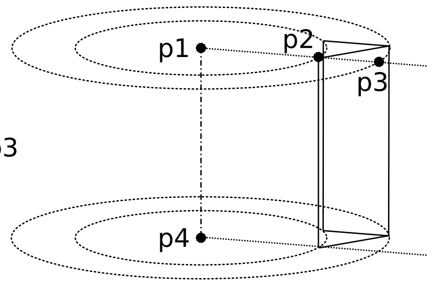


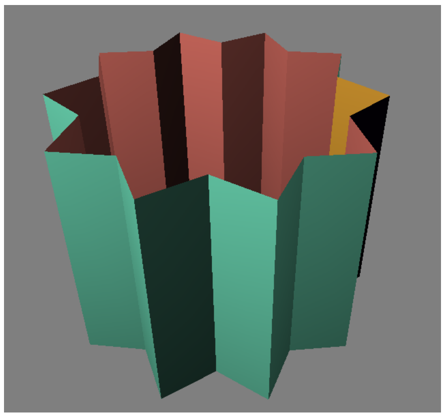
☐ Modify code to compute correct normal.

```
pVect vx(p1,p2), vz(p1,p4);
pNorm ax(vx), ay( cross(vz,ax) );
pVect vy = ay * ax.magnitude;
float r2or1 = pNorm(p1,p3).magnitude / ax.magnitude;
float delta_theta = 2 * M_PI / n_pieces;

for ( int i=0; i<n_pieces; i++ ) {
    float theta = i * delta_theta;
    pVect v = vx * cosf(theta) + vy * sinf(theta);
    pVect v2 = v * r2or1;
    bset_mta <<  p1 + v   <<  p1 + v2  <<  p4 + v;
    bset_mta <<  p1 + v2  <<  p4 + v2  <<  p4 + v;

    pNorm n = pVect(1,2,3);  // Obviously incorrect normal.
```

```
    // Insert normals into buffer set.
    bset_mta << n << n << n;
    bset_mta << n << n << n;
  }
```







(*b*) Appearing below again is the code from the Homework 2 solution. This time the goal is to render a (crude) gear positioned and oriented using the same four points, `p1-p4`. The points define two pairs of circles as illustrated above. The center illustration shows how one tooth (gear part) is constructed. There should be `n_pieces` teeth, positioned so they touch and go around the gear as shown on the screenshot to the right. Modify the code to draw the gear.

☐ Complete code so that it draws the figure. ☐ Avoid computationally wasteful code.

☐ Write coordinates and normals. Don't write colors.

```
pVect vx(p1,p2), vz(p1,p4);
pNorm ax(vx), ay( cross(vz,ax) );
pVect vy = ay * ax.magnitude;
float r2or1 = pNorm(p1,p3).magnitude / ax.magnitude;
float delta_theta = 2 * M_PI / n_pieces;


for ( int i=0; i<n_pieces; i++ ) {
    float theta = i * delta_theta;
    pVect v = vx * cosf(theta) + vy * sinf(theta);
    pVect v2 = v * r2or1;







    bset_mta <<







    bset_mta <<
  }
```

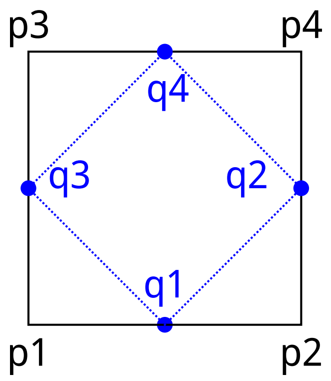
Problem 2: [15 pts] Coordinates `w0`, `w1`, and `w2` are the window-space coordinates of a triangle, call it `w`, ready to be rasterized. As pointed out in class, a triangle can be rasterized more efficiently if an edge is parallel to the x axis. Split triangle `w` into two triangles, `a` and `b`, such that `a` and `b` both have an edge parallel to the x axis and of course `a` and `b` cover the same points as `w`. To make things easier, assume that `w0.y > w1.y > w2.y`. Vectors between `w`'s vertices have been computed for convenience.

☐ Assign `a0` through `b2` so that `a` and `b` each have an edge parallel to the x axis and ☐ `a` and `b` together form `w`.

```
pCoor w0 = coors_w[i++];   // Assume w0.y > w1.y > w2.y
pCoor w1 = coors_w[i++];
pCoor w2 = coors_w[i++];

pVect v02(w0,w2);
pVect v01(w0,w1);
pVect v12(w1,w2);
```

```
// Triangle a
//
pCoor a0 =

pCoor a1 =

pCoor a2 =

// Triangle b
//
pCoor b0 =

pCoor b1 =

pCoor b2 =
```

Problem 3: [15 pts] The points `p1` through `p4` and points `q1` through `q4` in the diagram to the right each form a square, and both squares are in the same plane. Compute transformation matrix `m` (see below) that will transform points `p1`-`p4` to points `q1`-`q4`. The code at the bottom of the page shows how the matrix will be used. Some transformations are shown for reference.



☐ Compute transformation `m` that rotates points as described.

```
pCoor p1 = get_point(),  p2 = get_point();
pCoor p3 = get_point(),  p4 = get_point();

// The declarations below are for reference.
pNorm axis;
float angle, scale_factor;
pVect vec;
pMatrix_Rotation mat_r(axis,angle);
pMatrix_Translate mat_t(vec);
pMatrix_Scale mat_s(scale_factor);
// The declarations above are for reference.
```

```
pMatrix m =                                                           ;


pCoor q1 = m * p1;
pCoor q2 = m * p2;
pCoor q3 = m * p3;
pCoor q4 = m * p4;
```

Problem 4: [40 pts] Answer each question below.

(*a*) The vertex shader code below is getting the clip-from-object transformation matrix from a uniform variable. It is possible to supply this matrix as a vertex shader input. What would be the disadvantage of doing so?

```
// Conventional Approach
layout (location = LOC_IN_POS) in vec4 in_vertex_o;
void vs_main() {
  // Transform the vertex object-space coordinate to clip space.
  gl_Position = ut.clip_from_object * in_vertex_o;


// Alternative Approach
layout (location = LOC_IN_POS) in vec4 in_vertex_o;
layout (location = LOC_IN_MAT) in mat4 in_clip_from_object;
void vs_main() {
  // Transform the vertex object-space coordinate to clip space.
  gl_Position = in_clip_from_object * in_vertex_o;
```

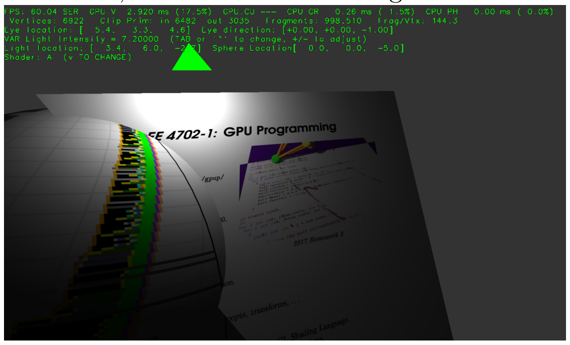
☐ Disadvantage of sending matrix as shader input.

(*b*) In Homework 3 a set of colors was placed in a uniform variable. Would it be feasible to put vertex coordinates in a uniform variable also for typical rendering?

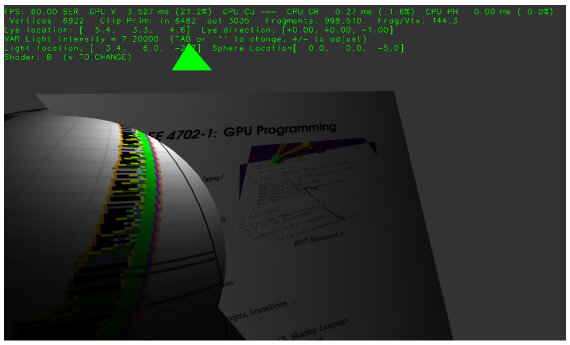
☐ Is putting vertex coordinates in a uniform variable ○ *feasible* or ○ *not feasible* ?

☐ Explain.

(*c*) The two screenshots below show the same scene rendered with two different sets of shaders. The scene consists of one big triangle and a sphere (consisting of many triangles). In one image lighting is done in the vertex shader, in the other in the fragment shader.



```
FPS: 60.04 SER  GPU.V  2.920 ms (17.5%)  GPU.CU ---  CPU GR   0.26 ms ( 1.5%)  CPU PH   0.00 ms ( 0.0%)
 Vertices: 6922   Clip Prim: in 6482  out 3035   Fragments: 998,510   Frag/Vtx: 144.3
Eye location: [  5.4,   3.3,   4.6]  Eye direction: [+0.00, +0.00, -1.00]
VAR Light Intensity = 7.20000  (TAB or '`' to change, +/- to adjust)
Light location: [  3.4,   6.0,  -2.7]  Sphere Location[  0.0,   0.0,  -5.0]
Shader: A  (v TO CHANGE)
```



```
FPS: 60.00 SER  GPU.V  3.527 ms (21.2%)  GPU.CU ---  CPU GR   0.27 ms ( 1.6%)  CPU PH   0.00 ms ( 0.0%)
 Vertices: 6922   Clip Prim: in 6482  out 3035   Fragments: 998,510   Frag/Vtx: 144.3
Eye location: [  5.4,   3.3,   4.6]  Eye direction: [+0.00, +0.00, -1.00]
VAR Light Intensity = 7.20000  (TAB or '`' to change, +/- to adjust)
Light location: [  3.4,   6.0,  -2.7]  Sphere Location[  0.0,   0.0,  -5.0]
Shader: B  (v TO CHANGE)
```

☐ In which image is lighting being performed in the fragment shader? ◯ *Shader A (left)* or ◯ *Shader B (right)* . ☐ Explain.

☐ Performing lighting in the vertex shader ◯ *looks better* ◯ *uses less computation* . ☐ Explain.

☐ Performing lighting in the fragment shader ◯ *looks better* ◯ *uses less computation* . ☐ Explain.

(*d*) Suppose the big triangle in the images above were split into 64 triangles.

☐ How many additional vertices would there be when rendered using a triangle list.

☐ How many additional fragments would there be?

☐ How would that affect the two images above?

(*e*) One thing our CPU-only ray tracing code did not do was check for shadows. But it would be easy to do so. Suppose ray tracing code included a routine `cast_ray(ray_origin,ray_direction)` that returned the triangle closest to the ray origin. Using that routine, how can one check for shadows?

☐ With ray tracing can check for shadows by:

(*f*) In our CPU-only rasterization routines we had our own rasterization code:

```
for ( float b0=0; b0<=1; b0 += db0 )
  for ( float b1=0; b1<=1-b0; b1 += db1 ) {
      pCoor c = w2 + b0 * v20 + b1 * v21;  // Window-space coordinate.
      if ( uint(c.x) >= win_width || uint(c.y) >= win_height ) continue;
      demo_frame_buffer[ c.x + int(c.y) * win_width ] = color;
    }
```

The routine above was kept simple for teaching purposes, but we could have tuned it for the CPU or GPU the code was to run on. However in both Vulkan and OpenGL rasterization is a fixed-function stage, meaning we can't write our own specially tuned routine. Doesn't that hurt—not help—performance?

☐ Why is rasterization fixed-function? ☐ How does preventing us from tuning help performance?