Name Solution

GPU Programming

EE 4702-1

Midterm Exam

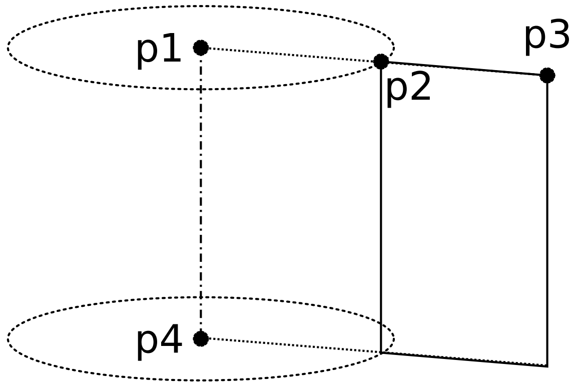
21 October 2022, 9:30-10:20 CDT

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

Alias Artemis I?

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.

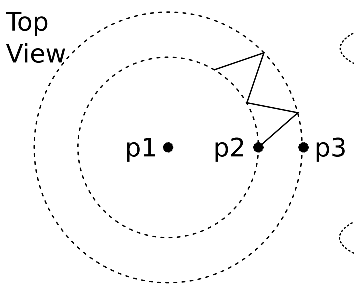


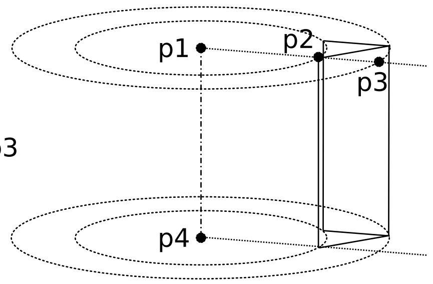
(*a*) The last lines insert normals into the buffer set. These normals are obviously wrong. Modify the code to compute the correct normal.

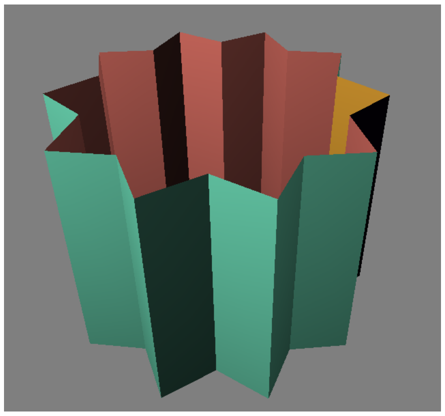
☑ Modify code to compute correct normal.

The solution appears below. The normal is the cross product of the vz and v. One might think of v as a rotated version of vx. A common mistake was to take the cross product of vz and vx, which would only work for i=0.

```
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.

    // SOLUTION
    pNorm n = cross(vz,v);
    //
    // That's it, just one line.









    // 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.

Solution appears on the next page.

```
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 <<
  }
```

Solution to Problem 1b appears below.

If there are to be the same number of gear teeth as there were paddle wheel blades then we will need to iterate over twice the number of positions around the circle. For that reason compute a delta theta that is based on `2*n_pieces` and use `2*n_pieces` as the loop bound.

A gear tooth consists of two rectangles. (The two rectangles are shown using solid lines in the center illustration on the top of the previous page.) Each iteration of the loop below emits one rectangle. An iteration starts by constructing a vector from `p1` to the inner circle, `vc` (c is for current). If `i` is odd then that vector is lengthened so that it reaches from `p1` to the outer circle. The just-computed vector, `vc`, and the vector computed in the previous iteration, `vp`, are used to compute the four vertices of the rectangle, `pp_bot`, `pc_bot`, `pp_top`, and `pc_top`. Two triangles are emitted using these vertices. After that a normal is computed and emitted. Note that the same normal is used for all six vertices.

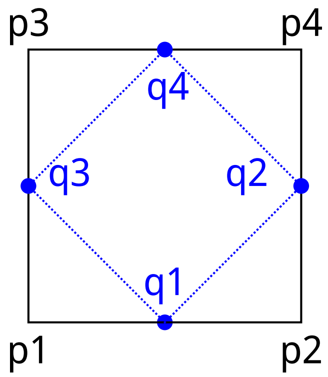
```
pVect vx(p1,p2), vz(p1,p4);
pNorm ax(vx), ay( cross(vz,ax) );
pVect vy = ay * ax.magnitude;
float r2or1 = pNorm(p1,p3).magnitude / ax.magnitude;
float delta_theta = 2 * M_PI / n_pieces;

// SOLUTION
float delta_theta_h = 2 * M_PI / ( 2 * n_pieces );

pVect vp = vx;  // Previous axis-to-cylinder vector.

for ( int i=1; i<=2*n_pieces; i++ ) {

    float theta = i * delta_theta_h;

    pVect vc = vx * cosf(theta) + vy * sinf(theta);
    if ( i & 1 ) vc *= r2or1;

    // Compute vertices of rectangle.
    pCoor pp_bot = p4 + vp,  pc_bot = p4 + vc;
    pCoor pp_top = p1 + vp,  pc_top = p1 + vc;

    // Emit two triangles making up the rectangle.
    bset_mta << pp_bot << pp_top << pc_top;
    bset_mta << pp_bot << pc_top << pc_bot;

    // Compute the normal of the rectangle.
    pNorm np = cross(pc_bot,pc_top,pp_top);

    // Emit the normals.
    bset_mta << np << np << np   << np << np << np;

    // Save vc for use in the next iteration.
    vp = vc;
  }
```

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);

// SOLUTION
pCoor w02 = w0 + v02 * v01.y / v02.y;
// A slightly more efficient solution: just one division.
pCoor w02b(  w0.x + v02.x * v01.y / v02.y,  w1.y,  w1.z );

pCoor a0 = w0;
pCoor a1 = w1;
pCoor a2 = w02;

pCoor b0 = w02;
pCoor b1 = w1;
pCoor b2 = w2;
```

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.

The solution appears below. To transform the p points so that they are at the position of the q points we need to perform a rotation by 45° and then scale the points so that the square is smaller. The rotation should be around the center of the square, but typical rotation matrices rotate around the origin. So before applying such a rotation we need to translate the points so that the center of the square is over the origin. It is also important that the scaling be applied to points in which the center is at the origin.

So we need to apply the transformations in this order: (1) translate center to origin, (2) rotate, scale, (3) translate back to original location.

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

// Compute the coordinate of the center of the square.
pCoor ctr = p1 + pVect(p1,p4)/2;

// Prepare translation matrices that moves center to the origin and back.
pMatrix_Translate to_origin( -ctr );
pMatrix_Translate from_origin( ctr );

// Compute the rotation axis for use in rotating the points.
pNorm rot_axis = cross(p1,p2,p3);

// Prepare a rotation matrix that rotates by 45 deg  ( pi/4 ).
pMatrix_Rotation rot( rot_axis, M_PI/4 );

// Prepare a scale matrix that shrinks the p square to the q square's size.
pMatrix_Scale scale( 1/sqrt(2) );

// Combine these to form one transformation matrix.
//
pMatrix m = from_origin * scale * rot * to_origin;
//            Fourth      Third   2nd   First
//
// Note that transformations are applied from RIGHT to LEFT.

// Use m to compute the position of the q points.
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.

Answer: That would more than double the amount of data going into the vertex shader, which could hurt performance.

Explanation: Each matrix consists of 16 floats. Suppose the vertex shader inputs already consist of a coordinate, a color, and a normal. Those total $4 + 4 + 3 = 11$ floats. So sending the matrix along with them would more than double the amount of data.

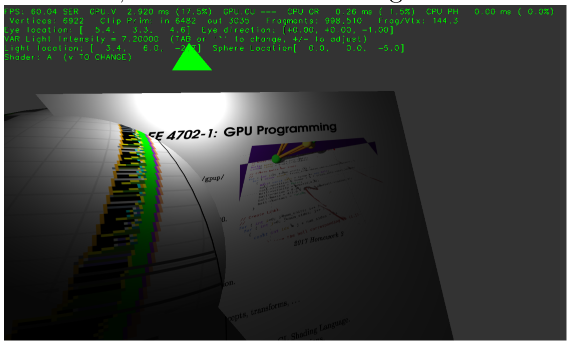
(*b*) In Homework 3 a set of colors was placed in a uniform variable. Would it be feasible to put vertex coordinates in a uniform variable also for typical rendering?

☑ Is putting vertex coordinates in a uniform variable ◯ *feasible* or [✓ *not feasible*] ?

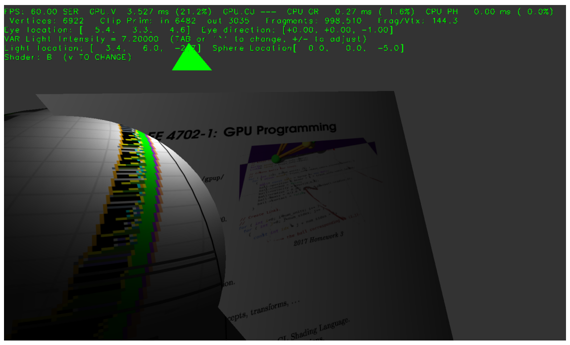
☑ Explain.

It is not feasible because in a typical draw hundreds or more vertices are streamed in. There is a limit to the size of uniform variables (which varies by device) that size would be exceeded by using vertices.

(*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.

Shader A. Because the lighting is brightest *between* the big triangle's upper vertices. If lighting were done in the vertex shader a lighted color would have been computed for each of those two vertices and interpolated values would be used for the fragments between them. That's whats happening in the right figure, where the upper-left vertex looks whitish, but the upper-right vertex is dark, and the colors gradually change from whitish to dark from the left vertex to the right vertex.

[✓] Performing lighting in the vertex shader ◯ *looks better* [✓ *uses less computation*] . [✓] Explain.

Answer: Less computation, because there typically fewer vertices than fragments.

Explanation: A triangle has three vertices, but can cover thousands or more fragments. The vertex shader is executed for each vertex, the fragment shader is executed for each fragment. The work needed to compute lighting would be done three times in the fragment shader but many times (hundreds of thousands for the images above, as those with sharp vision can see) so clearly there is much less work computing lighting in the vertex shader.

[✓] Performing lighting in the fragment shader [✓ *looks better*] ◯ *uses less computation* . [✓] Explain.

Looks better because the lighting that is used is computed at the fragment, rather than an interpolation of values computed at the three vertices (for a triangle). See the answer to the In-which-image question.

(*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.

Since a triangle list is being used, there would be $64 \times 3 = 192$ vertices which is 189 more than is used by the original triangle.

[✓] How many additional fragments would there be?

Zero. The number of fragments does not change. (A fragment is generated if the center of a pixel is within the triangle, so there will not be fragments that are part of both triangles. Unless two triangles' edges pass over the same fragment centers. Because these are individual triangles there would be a fragment for each triangle. But that would require a perfect alignment. Since we're on the topic, if the two triangles were part of a triangle strip or fan then a fragment would be generated along the edges of just one of those triangles.)

[✓] How would that affect the two images above?

The code using shader B (the vertex shader) would generate an image that would look more like shader A. The spot of light would look more like a polygon.

(*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:

Let S denote the coordinates of the intersection of the eye ray with the closest triangle. Let L denote the coordinates of the light. Call `cast_ray(S,SL)`, where `SL` is the vector from S to L. If the closest intersection is the light itself, then S is illuminated, otherwise it is in the shade.

(*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?

The fixed-function rasterization is usually part of the GPU driver and has been written by the GPU manufacturer for the particular device it is running on. Our OpenGL or Vulkan code will then be using a good rasterizer wherever it is run. Had we taken the time to write a rasterizer for a particular device, it would only work well on that device, and would eventually become outdated. Unless we spent all of our time writing rasterizers for each new device that came on the market. But we rather be concentrating on more interesting parts of our real-time 3D application.