Name Solution _____ *Formatted For Two-Sided Printing*

Staple This Side

GPU Programming
EE 4702-1
Midterm Examination

Wednesday, 28 October 2026, 9:30-10:20 CDT

Code for this exam, including the solution, is in the repo under `gp/hw/gpup/2026/mt/`.

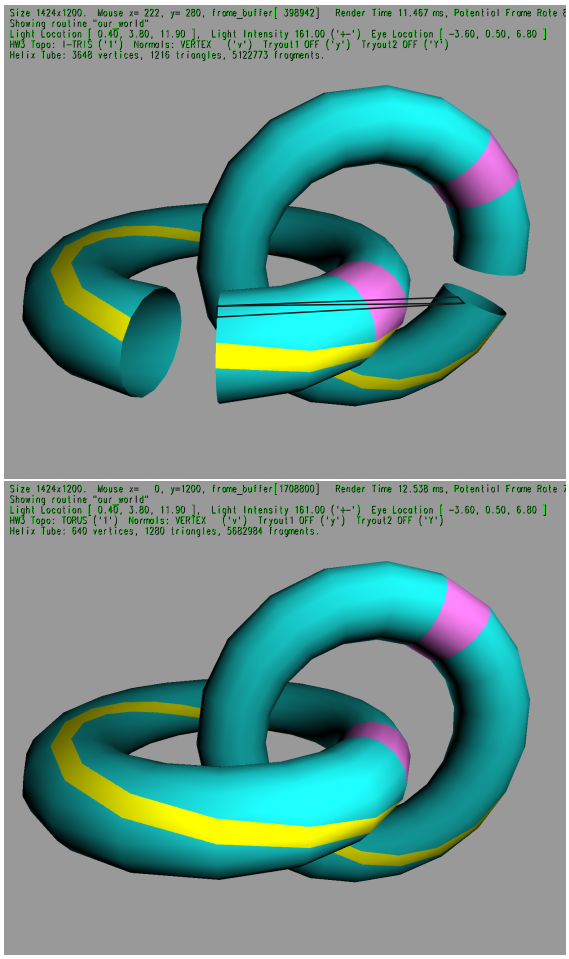
Problem 1	_____	(25 pts)
Problem 2	_____	(25 pts)
Problem 3	_____	(25 pts)
Problem 4	_____	(25 pts)
Exam Total	_____	(100 pts)

Alias They can do that? _____

Good Luck!

Staple This Side

Problem 1: [25 pts] The screenshots to the right were taken from a modified version of Homework 3 Problem 2 in which two tori are rendered in a single draw. The upper shot was rendered using the individual triangle topology and the lower shot was rendered using the new *torus* topology.



Recall that for a ring topology the number of vertices in a ring is provided. The vertices for the first ring (start of torus, say) are appended to `coors_os`, followed by the second ring, and so on. For a torus topology the number of vertices in a ring (`n_vtx_ring`) and the number of rings in a torus (`n_rings`) are provided. The contents of `coors_os` are the same as for rings, but for a torus topology triangles are rendered using the first and last rings of a torus, which is why there is no gap.

(*a*) The excerpt on the facing page is from `Our_3D::draw_rasterization`, it computes vertex indices for the ring topology from Homework 3. Variable `n_vtx_ring` is the number of vertices in a ring and `n_rings` is the number of rings in a torus.

☑ Modify the code for a torus topology.

☑ The code must work for more than one torus per draw.

☑ Don't overdo it, all that's needed is three lines of code.

(*b*) The topology used for the upper screenshot was not a ring for a reason.

☑ Show what the upper screenshot would look like if the draw were done using vertices prepared for a torus topology but `Our_3D` was set to a ring or strip topology. ☑ Don't forget that one draw renders both tori.

There would be two extra triangles, those are shown in black (simulated hand drawn) on the upper screenshot. Remember that in a strip topology a triangle is formed using the current vertex and the two most recent vertices. So the first vertex in the second torus (the one in the horizontal plane) forms a triangle using the last two vertices in the first torus, connecting the two. A torus-spanning-triangle is also emitted for the second vertex in the second torus.

(*c*) Answer the following about the ring topology versus the strip topology used to render the helix from Homework 3.

☑ The number of vertices using a ring topology is

⊗ *less than*, ◯ *same as*, ◯ *more than* a strip topology. ☑ Explain.

To render the helix (HW3) or the torus (this problem) using a strip each vertex would need to be sent twice. With the ring topology each vertex is sent once, so there are about half as many.

☑ The number of fragments using ring topology is

⊗ *less than*, ◯ *same as*, ◯ *more than* a strip topology. ☑ Explain.

Staple This Side
Staple This Side

Staple This Side

When the strip topology is used to render the helix or the torus, a segment (an example being the purple bands in the screenshots) would be perfectly rendered by a strip if the entire draw consisted of one segment. But the helix and the tori consist of many segments. The first two vertices in each segment (except for the first) result in triangles connecting to the previous segment. These extraneous triangles were not visible because they were inside the helix. But they still resulted in extra fragments, and so the ring topology rasterizes to fewer fragments.

```
size_t n_vtx = coors_os.size();
size_t inc = topo_tris ? 3 : 1;
size_t stop_ring = 2 * ( n_vtx - n_vtx_ring ) - 1;
size_t stop_torus = 2 * n_vtx - 1;
size_t stop = topo_ring ? stop_ring : topo_torus ? stop_torus : n_vtx - 3;

for ( size_t i=start; i<=stop; i += inc )
  {
    size_t i0, i1, i2;

    if ( topo_ring || topo_torus )
      {
        bool triLL = ( i & 1 ) == 0;  // If true, a lower-left triangle.
        size_t ir0 = i >> 1;
        size_t ir1m = ir0 + 1;  // Value before checking for wrap-around.
        size_t ir1 = ir1m % n_vtx_ring  ?  ir1m  :  ir1m - n_vtx_ring;

        i0 = triLL ?  ir0  :  ir1;
        i1 = triLL ?  ir1  :  ir1 + n_vtx_ring;
        i2 = ir0 + n_vtx_ring;

        // The values below may come in handy.
        size_t n_vtx = coors_os.size();
        size_t vtx_per_torus = n_vtx_ring * n_rings;

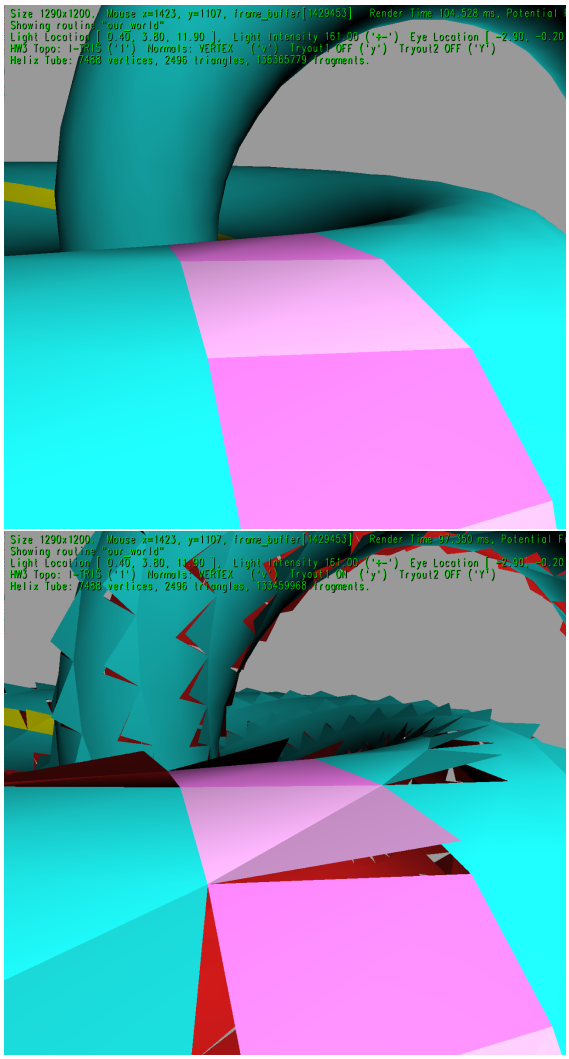
        // SOLUTION
        if ( topo_torus ) {
            // Compute the index of the first vertex in the current torus.
            size_t torus_base = i0 / vtx_per_torus * vtx_per_torus;
            i1 = torus_base + i1 % vtx_per_torus;  // Fix any wraparound.
            i2 = torus_base + i2 % vtx_per_torus;  // Fix any wraparound.
          }

      }
    else
      {
        i0 = i;  i1 = i+1;  i2 = i+2;  // For strips or indiv. triangles.
      }

    // Get next triangle's object space coordinates ..
    pCoor o0 = coors_os[i0],  o1 = coors_os[i1],  o2 = coors_os[i2];
```

Staple This Side

Problem 2: [25 pts] The screenshot to the upper right was generated with the code on the facing page, based on Homework 3 Problem 1 in which pairs of triangles are inserted for an individual triangle ordering. In the lower screenshot one triangle of a pair is placed normally, the other is rotated around their common edge.



(*a*) Rotate the triangle:

☑ Set `m` to a matrix that will transform coordinate(s) to the rotated position ▷ rotating by $\pi/5$ radians around the triangles' common edge.

The rotation axis is set to the common edge (between `p10` and `p01`. Remember that in the rotation transform constructed by `pMatrix_Rotation` the rotation is around the origin in the direction specified by `axis`. The translation transforms moves the axis so that it passes through the origin, and then returns it.

☑ Use `m` to transform the coordinates ☑ and insert them in `coors_os` to form the rotated triangle.

Notice that only one coordinate, `p11`, needs to be transformed.

☑ Set the normal of the rotated triangle to the triangle's own normal, not to the `n0`, `n1`, `n2` values.

(*b*) In the solution above `m` should be the product of several matrices.

☑ From an efficiency point of view for this code fragment why is matrix-matrix multiplication **not** a good idea?

Because there are easier ways to transform the **one vertex**. The two matrix multiplies require 128 multiply/add operations. See the solution to the next subpart for a better way to transform the vertex.

☑ Show an efficient way for this situation to transform the coordinates that would not require matrix-matrix multiplication.

Instead of using a translation transform just add or subtract vectors. The two vector operations require just six additions (or subtractions), much fewer than 128 multiply/add operations. See the code below.

```
// SOLUTION - Problem 2b.
pVect to_origin( p01, pCoor(0,0,0) );  // Zero computation.
pCoor p11ro = p11 + to_origin;          // Three additions.
pCoor p11rro = rot * p11ro;             // Sixteen mult/adds.
p11r = p11rro - to_origin;              // Three subtractions.
```


Staple This Side
Staple This Side

Staple This Side

```
case Topology_Individual:
 for ( int j=0;  j < tr_co_prev.size()-1;  j++ ) {
   pColor& color = colors_ring[j];  // For convenience.
   pCoor p00 = tr_co_prev[j], p01 = tr_co_prev[j+1];
   pCoor p10 = tr_co_curr[j], p11 = tr_co_curr[j+1];
   pVect4 n00 = tr_n_prev[j],  n01 = tr_n_prev[j+1];
   pVect4 n10 = tr_n_curr[j],  n11 = tr_n_curr[j+1];

   coors_os   << p00 << p01 << p10;  // Triangle I
   normals_os << n00 << n01 << n10;  // Triangle I

   if ( ( j + i ) & 1) {
       // Place second triangle along the tube.  Not rotated. DON'T MODIFY
       coors_os   << p01 << p11 << p10;   // Triangle II
       normals_os << n01 << n11 << n10;   // Triangle II
     } else {
#if 0
       // Examples of Transfromation Matrices.  Abbreviations okay.
       // Scale by sx along x axis, etc.
       pMatrix_Scale scale(2, 4, 6 ); // Scale by 2 along x, 4 along y, ..
       //
       // Rotate so local x axis now points in direction ax, etc.
       pMatrix_Cols rot_to_global(global_ax, global_ay, global_az);
       // Rotate by 1.23 radians around global_ax.
       pMatrix_Rotation rot( global_ax, 1.23 );
       //
       // Translate so that local origin (0,0,0) is moved to global_origin.
       pMatrix_Translate to_global( global_origin );
#endif

        // SOLUTION -- Set axis to direction of the common edge.
        pNorm axis = pVect( p10, p01 );

        pMatrix_Rotation rot( axis, numbers::pi/5 );

        pMatrix_Translate to_origin( -p01 );  // Move axis to origin.
        pMatrix_Translate from_origin( p01 ); // And back from origin.

        // Multiply the three transformations in the correct order.
        pMatrix m = from_origin * rot * to_origin;

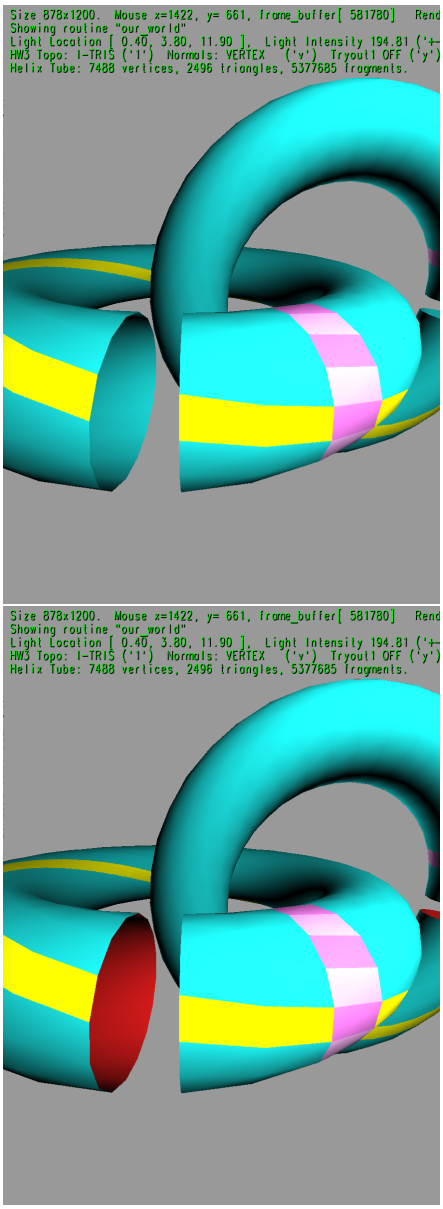
        // Transform the one coordinate that's not on the torus.
        pCoor p11r = m * p11;

        // Compute the rotated triangle normal.
        pVect4 nr = pNorm( cross( axis, pVect( p10, p11r ) ) );

        // Insert vertices and normals.
        coors_os << p01 << p11r << p10;  // Triangle II
        normals_os << nr << nr << nr;    // Triangle II
  }} break;
```

Staple This Side

Problem 3: [25 pts] For the upper screenshot the same color is used for both sides of each triangle, that color is an attribute of the vertex. For the lower screenshot the back of each triangle is red. On the facing page is a simplified version of the main loop from `Our_3D::draw_rasterization`.



(*a*) Recall that for OpenGL and Vulkan a point is *seeing the front* of a triangle with vertices `o0`, `o1`, and `o2` (using the variable names in the code) if movement from `o0` to `o1` to `o2` is counterclockwise as seen from the point (this is the same as applying the right-hand rule).

☑ Compute vector `f_dir` so that it points out the front of the triangle. ☑ Don't assume that `n0`, etc. point out the front of the triangle.

☑ In the code on the facing page set `front_facing` to true if the eye sees the front of the triangle. ▷ Use `f_dir`, even if you could not compute it. ▷ Variable `eye` is the eye's object-space coordinates.

☑ When `front_facing` is `true` set the triangle color to `color_red` instead of `c0`, etc.

(*b*) (This problem is not about color.) If `clip` is `true` the current triangle won't be rasterized. Ideally `clip` is set `true` iff zero fragments are within the window. To make it easy:

☑ In the code fragment set `clip` to `true` when the triangle is completely to the left of the user's monitor. *Hint: See the scissoring line in the rasterization loop.*

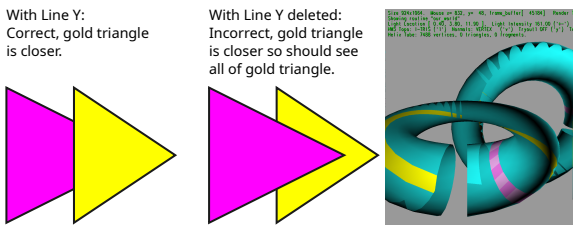
(*c*) Near the bottom of the facing page are lines labeled `Line X` and `Line Y`.

☑ Describe what would happen if `Line X` were deleted.

If `Line X` were deleted then triangles on the viewer side of the monitor would be visible, including triangles behind the viewer. Also, the frame buffer would be written for triangles that were very far away.

☑ Describe what would happen if `Line Y` were deleted. ☑ Illustrate with a sketch.

If `Line Y` were deleted then the last fragment at a particular pixel location would be the one visible, even if it were further from the eye than other fragments that were earlier written to that pixel location. See the purple and gold triangles to the right. The screenshot on the far right shows the torus rendered with `Line Y` effectively deleted. The back of the second (left-side) torus should be behind the first (right-side) torus, but it's in front.



Staple This Side
Staple This Side

```
for ( size_t i=0; i<=n_vtx-3; i += 3 ) {
    size_t i0=i, i1=i+1, i2=i+2;
    // Get next triangle's object space coordinates ..
    pCoor o0 = coors_os[i0],  o1 = coors_os[i1],  o2 = coors_os[i2];
    // .. and transform them to window space.
    pCoor_Homogenized_Keep_w w0( ws_from_os * o0 );
    pCoor_Homogenized_Keep_w w1( ws_from_os * o1 );
    pCoor_Homogenized_Keep_w w2( ws_from_os * o2 );

    pColor c0 = colors[i0], c1 = colors[i1], c2 = colors[i2];

    auto n0 = vtx_normals[i0], n1 = vtx_normals[i1], n2 = vtx_normals[i2];

    // SOLUTION
    bool clip = w0.x < 0  &&  w1.x < 0  &&  w2.x < 0;
    // Note that to be sure the triangle is not in the window
    // *all* three vertices must be to the left (w.x < 0 ).
    if ( clip ) continue;

    // SOLUTION
    pVect f_dir = cross( pVect(o0,o1), pVect(o0,o2) ); // Front of right-handed triangle.

    pCoor& eye = eye_location_os; // Handy abbreviation.

    // SOLUTION
    bool front_facing = dot( f_dir, pVect( o0, eye_location ) ) > 0;
    // Note: This works for triangles, but not strips because in a
    // strip the front side alternates between right- and left-handed.

    // SOLUTION
    if ( !front_facing ) { c0 = c1 = c2 = color_red; }


    pVect w20(w2,w0), w21(w2,w1), w01(w0,w1);
    const float db0 = 0.5/ min(max(fabs(w20.x),fabs(w20.y)), max(fabs(w01.x),fabs(w01.y)));
    const float db1 = 0.5/max(fabs(w21.x),fabs(w21.y));

    // The Rasterization Loop
    for ( float b0=0; b0<=1; b0 += db0 )
      for ( float b1=0; b1<=1-b0; b1 += db1 ) {
          const float b2 = 1 - b0 - b1;
          pCoor wf = b0 * w0 + b1 * w1 + b2 * w2;

          // Scissor (discard) fragment if it's outside the window.
          if ( uint(wf.x) >= win_width || uint(wf.y) >= win_height ) continue;
          // Compute the index into the frame and depth buffers.
          const size_t idx = wf.x + int(wf.y) * win_width;

          float zinv = wf.z;
          if ( zinv < 0 || zinv > 1 ) continue;   // <------- Line X
          if ( zbuffer[ idx ] < zinv ) continue;  // <------- Line Y
          zbuffer[ idx ] = zinv;
```

Problem 4: [25 pts] Answer each question below.

(*a*) We are trying to improve the performance of a rasterization application.

☑ Should we pay more attention to the number of fragments or the number of vertices? ☑ Explain.

We should pay more attention to the number of fragments because there are more of them. Suppose there were 1000 vertices and 100000 fragments. Suppose the vertex shader and fragment shader each perform 70 units of work per invocation. (A unit of work might be an arithmetic operation or moving a piece of data.) Plan A: we tune our vertex shader to only need 7 units of work! The total work is now $7 \times 1000 + 70 \times 100000 = 7007000 = 7.007 \times 10^6$ units. Plan B: we tune our fragment shader to need 35 units of work. The total amount of work is now $70 \times 1000 + 35 \times 100000 = 3570000 = 3.57 \times 10^6$ units. So effort put into fragments yields more performance.

In reality, by the way, on a typical NVIDIA high-end GPU if an arithmetic operation, such as a multiply/add, is 1 unit of work, then moving a unit of data might be 4, 40, or 200 units of work depending where it was moved from or to.

(*b*) Suppose the window manager tells our application it only needs to update the left half of its window.

☑ Explain how a ray tracing routine would only update the left half of the window. ☑ Be specific.

Ray tracing starts by casting a ray through each pixel. If the window manager specifies that only half the pixels need to be repainted then the ray tracing outer loop would only cast a ray through those pixels, and so it would do about half the work.

☑ Why might a rasterization routine do more than half the amount of computation to update the left half?

A rasterization routine could take into account visibility when constructing the frustum transform (requiring application support) or when clipping and scissoring fragments of a triangle (requiring only driver support). During clipping we might discover that a particular triangle is entirely on the right half of the window, and so it would skip doing anything further with it. But, if the triangle were in both parts then it might have to proceed to the rasterization loop and must scissor out the pixels that are on the right side. So it might take more than half the time.

(*c*) Moving data is expensive, so why include a normal with each vertex in a draw when it's easy to calculate from triangle vertices?

☑ Reason for normal as a vertex attribute, despite cost of moving it?

Because for many draws triangles are approximating a smooth surface, such as a sphere. To get better lighting results the application sets the vertex normal to the sphere normal rather than the triangle normal. So a general-purpose geometry shader could not compute a sphere normal. (One written for spheres could.)

Staple This Side

Staple This Side

(*d*) Three common storage qualifiers in OGSL shader code are `in`, `uniform`, and `out`.

☑ Provide an example of a variable that would be declared `in` for a vertex shader.

Examples include the vertex coordinate and the vertex normal.

☑ Provide an example of a variable that would be declared `uniform` for a vertex shader.

An example is an clip-from-object transformation matrix.

(*e*) In shader code X, a variable that would normally be declared `uniform` is declared `in`.

☑ Assuming that code X works correctly, what is the disadvantage of declaring the variable `in`?

The disadvantage is that much more data would need to be moved and stored somewhere. For example, if the clip-from-object transformation were a vertex attribute and there were 1000 vertices, then $1000 \times 16 \times 4\,\mathrm{B} = 64000\,\mathrm{B}$ would be moved instead of just $16 \times 4\,\mathrm{B} = 64\,\mathrm{B}$. A typical vertex might have a coordinate, normal, color, and a couple of texture coordinates, in total consuming about $4 \times 4 \times 4 = 64\,\mathrm{B}$. So also sending the transform would double the amount of data.

Note: a couple means exactly two.

Staple This Side
Staple This Side