Name _____ *Formatted For Two-Sided Printing*

Staple This Side

GPU Programming
EE 4702-1
Midterm Examination

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

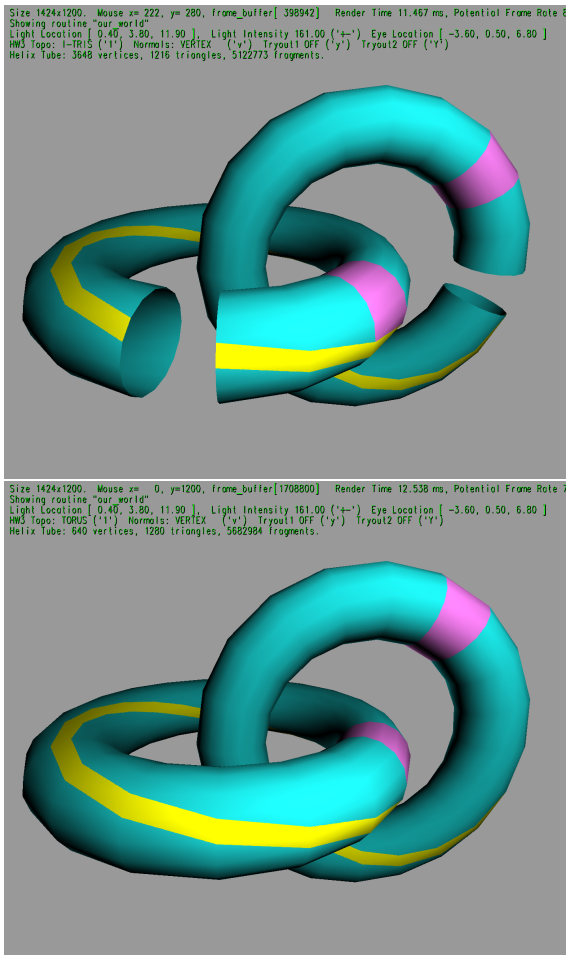
Problem 1	_____	(25 pts)
Problem 2	_____	(25 pts)
Problem 3	_____	(25 pts)
Problem 4	_____	(25 pts)
Exam Total	_____	(100 pts)

Alias _____

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.

(*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.

☐ 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

```
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 to part a goes here.







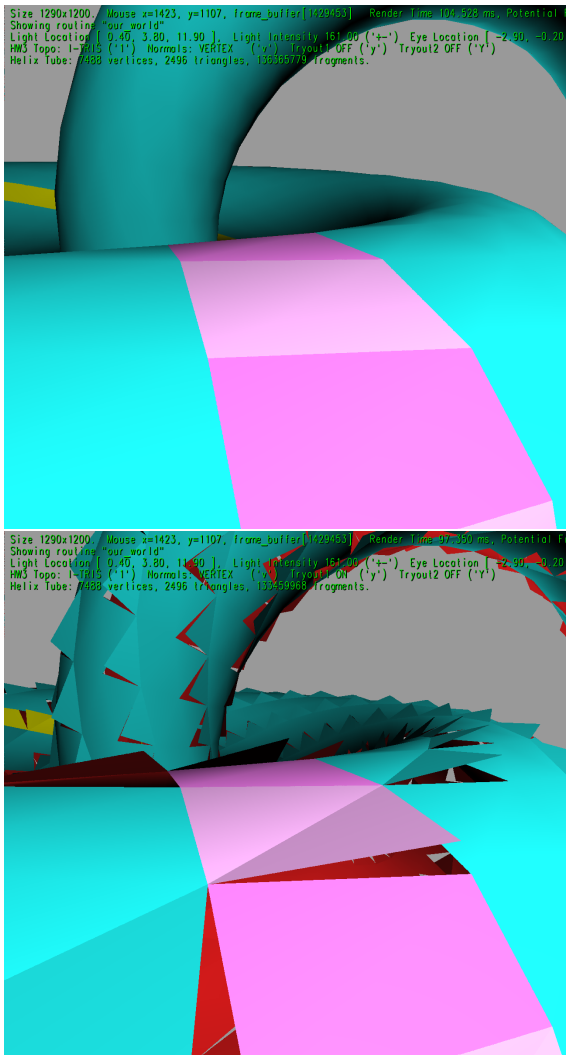


      }
    else
      {
        i0 = i;  i1 = i+1;  i2 = i+2;  // For strips or indiv. triangles.
      }

    // 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.

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

☐ 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?

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

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
       // Put solution for rotated triangle here. Can abbreviate.




       pNorm axis =                                      ;  //  ☐  Assign value
       pMatrix_Rotation rot( axis, numbers::pi/5 );        //  Or use something else.




       pMatrix m =                                       ; //  ☐  Set to transform.
       //  ☐  Use m to transform coords. Insert into coors_os, normals_os.

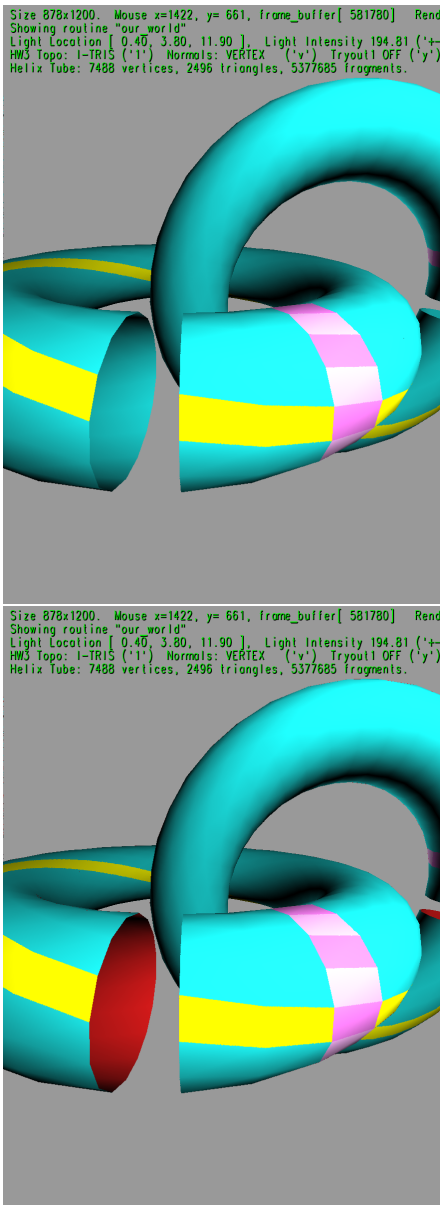


  }} break;
```

Staple This Side

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.

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

Staple This Side
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]; //



    bool clip =                                                    ; //  Assign value.
    if ( clip ) continue; // Skip rasterization loop.



    pVect f_dir =                                                  ; //  Assign value.
    pCoor& eye = eye_location_os; // Handy abbreviation.



    bool front_facing =                                            ; //  Assign value.


    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;
```

Staple This Side

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.

(*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.

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

(*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?

Staple This Side

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.

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

(*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`?

Staple This Side