Name _____ *Formatted For Two-Sided Printing*

Staple This Side

GPU Programming
EE 4702-1
Midterm Exam

25 October 2024, 9:30-10:20 CDT

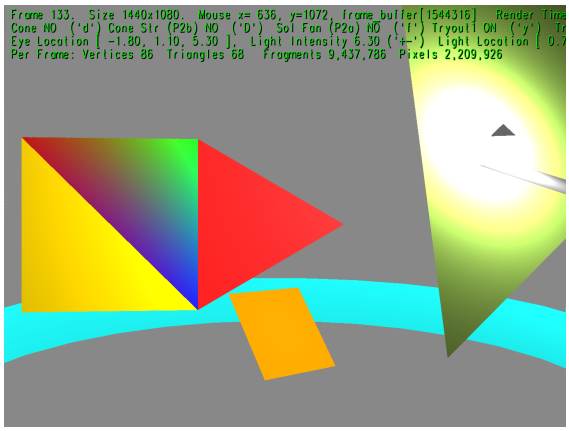
Problem 1	_____	(25 pts)
Problem 2	_____	(25 pts)
Problem 3	_____	(15 pts)
Problem 4	_____	(35 pts)
Exam Total	_____	(100 pts)

Alias _____

Good Luck!

Staple This Side

Problem 1: [25 pts] Appearing to the right is code based on Homework 3. The excerpt draws the gold and multi-colored triangle using abbreviated color names.



(*a*) Add code to draw the red equilateral triangle. One edge and two vertices match the multi-colored triangle and the third vertex is in the same plane as the others. All edges of the red triangle are the same length.

(*b*) At the bottom of the facing page is a loop that's supposed to insert normals for the triangles described by `coors_os`. Insert the normals. *Hint: this is easy.*

(*c*) Modify the code in the center of the facing page so that it draws the tan (or orange) square positioned as described below.

- ☐ The tan square center should be at coordinate `sq_ctr`.
- ☐ The length of an edge should be `sq_l`.
- ☐ The normal of the square should point towards `lt_loc` (light location).
- ☐ Use color `c_tan`.
- ☐ Two edges should be on an xz plane (parallel to the "ground").

Staple This Side

Staple This Side

```
// Insert Two More Triangles
pCoor p1(-4,0,-3), p2(-4,2,-3), p3(-2,0,-3), p4(-2,2,-3);

coors_os << p1      << p2      << p3;
colors   << c_gold  << c_gold  << c_gold;
coors_os << p2      << p4      << p3;
colors   << c_red   << c_green << c_blue;
```

☐ Add code to draw the red equilateral triangle. ☐ For full credit use `p1`,`p2`,`p3`, and `p4`, **not** their values.

☐ Add code to draw the tan square as described on the facing page.

```
pCoor sq_ctr = scene_data.square_center;
float sq_l = scene_data.square_side_length;
pCoor lt_loc = scene_data.light_location;
```

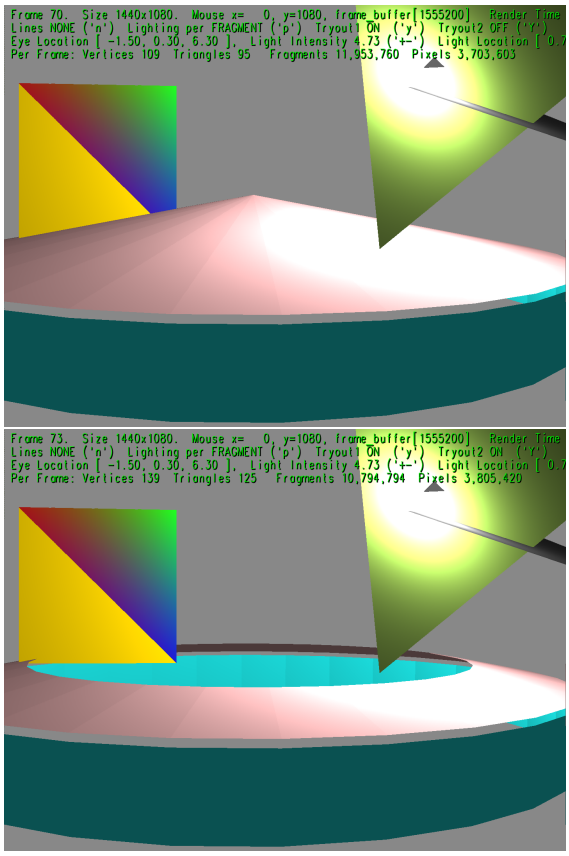
☐ Add code to the loop body below so that it inserts triangle normals into container `norms_os`.

```
for ( size_t i = 0;  i < coors_os.size();  i += 3 )
  {
    pCoor p1 = coors_os[i], p2 = coors_os[i+1], p3 = coors_os[i+2];




    norms_os <<
  }
```

Staple This Side

Problem 2: [25 pts] The top right screenshot shows a scene with a tan cone. The code on the top of the facing page draws the tan cone using a *triangle fan*. The vertex at the top of the cone, `cone_top_ctr`, is emitted first (and just that once), then the vertices around the circumference are emitted, once per iteration.

(*a*) The the code below correctly handles the `topology` setting `Topology_Individual` (individual triangles) and `Topology_Strip` (triangle strips). Modify it so that it also works correctly for `Topology_Fan` (triangle fan).



☐ Modify code below so that it correctly renders the `Topology_Fan`, grouping and ☐ **continues to work** for `Topology_Strip` and `Topology_Individual` groupings.

```
// Main Rasterization Loop
// Set loop stride for topology (grouping).
size_t inc =
   topology == Topology_Strip ? 1 : 3;



size_t start = 0;



// Outer Loop: Iterate Over Triangles
//
for ( size_t i=start; i+2 < coors_os.size(); i += inc )
  {



    size_t i0 = i;
    size_t i1 = i + 1;
    size_t i2 = i + 2;



    // Get next triangle's object space coordinates, convert to window space.
    pCoor o0 = coors_os[i0],  o1 = coors_os[i1],  o2 = coors_os[i2];
    pCoor u0( ws_from_os * o0 ), u1( ws_from_os * o1 ), u2( ws_from_os * o2 );
    pCoor_Homogenized w0( u0 ), w1( u1 ), w2( u2 );
    //  Extract colors from list.
    pColor c0 = colors[i0], c1 = colors[i1], c2 = colors[i2];
```


Staple This Side

Staple This Side

```
/// Draw Complete Cone Using a Triangle Fan – This Code Correct, DO NOT Modify
pCoor cone_bot_ctr = cyl_center + 1.1 * cyl_axis;
pCoor cone_top_ctr = cone_bot_ctr + cyl_axis;
coors_os << cone_top_ctr;
colors << color_rosy_brown;
for ( int i=0; i<=n_segs; i++ ) {
    float theta = i * delta_theta;
    pVect v = cosf(theta) * ax + sinf(theta) * ay;
    pCoor c = cone_bot_ctr + cyl_radius * v;
    coors_os << c;
    colors << c_tan;
  }
gc.topology_set(Topology_Fan);
gc.vtx_coors_set(coors_os).vtx_colors_set(colors).draw_rasterization();
```

(*b*) The code below is similar to the code above. Modify it so that it draws the cone with the top cut off using a triangle strip. In a line from the cone top to the circumference 70% is cut off. See the lower screenshot on the facing page.

☐ Modify code so it draws ☐ cut-off cone ☐ with 70% cut off. Draw it ☐ using a triangle strip.

```
/// ☐ Please Modify.
pCoor cone_bot_ctr = cyl_center + 1.1 * cyl_axis;
pCoor cone_top_ctr = cone_bot_ctr + cyl_axis;



coors_os << cone_top_ctr;
colors << c_tan;



for ( int i=0; i<=n_segs; i++ )
  {
    float theta = i * delta_theta;
    pVect v = cosf(theta) * ax + sinf(theta) * ay;
    pCoor c = cone_bot_ctr + cyl_radius * v;



    coors_os << c;
    colors << c_tan;



  }







gc.topology_set(Topology_Strip); // Note, uses a strip. No need to change this.
gc.vtx_coors_set(coors_os).vtx_colors_set(colors).draw_rasterization();
```

Staple This Side

Problem 3: [15 pts] Shown below is another excerpt from `Our_3D`. The intent of the Plan A and Plan B code variations is to avoid rasterizing a triangle that's outside the view volume.

```
  // Main Rasterization Loop -- Outer Loop: Iterate Over Triangles
  for ( size_t i=0; i+2 < coors_os.size(); i += inc ) {
      // Get object space, convert to unhomoegenized clip-space, then homogenize.
      pCoor o0 = coors_os[i+0],  o1 = coors_os[i+1],  o2 = coors_os[i+2];
      pCoor u0( cs_from_os * o0 ), u1( cs_from_os * o1 ), u2( cs_from_os * o2 );
      pCoor_Homogenized c0( u0 ), c1( u1 ), c2( u2 ); // Clip Space Coords

#ifdef    PLAN_A    /// Plan A
      //  If clip-space coords outside of view volume skip. (Not correct.)
      if ( c0.x < -1 || c0.y < -1 || c0.x > 1 || c0.y > 1 ) continue;

#elifdef  PLAN_B    /// Plan B
      //  If clip-space coords outside of view volume skip. (Not correct.)
      if (    ( c0.x < -1 || c0.y < -1 || c0.x > 1 || c0.y > 1 )
           && ( c1.x < -1 || c1.y < -1 || c1.x > 1 || c1.y > 1 )
           && ( c2.x < -1 || c2.y < -1 || c2.x > 1 || c2.y > 1 ) ) continue;
#endif
```

(*a*) Show an example of a triangle that should be visible but that Plan A would skip. Note that coordinates are in clip space. You may use any coordinate space for your answer as long as you clearly show the view volume (a 2D slice of the view volume is okay).

☐ Draw a diagram of the view volume (2D is fine) and use it to ☐ show a triangle that should be visible but Plan A skips.

☐ Draw a diagram of the view volume (2D is fine) and use it to ☐ show a triangle that should be visible but Plan B skips.

(*b*) Show code that will correctly detect whether triangle is completely outside of the view volume based on the x and y dimensions.

☐ Code that correctly checks if triangle is outside of view volume ☐ that both Plan A and Plan B miss.

Staple This Side

Staple This Side

Problem 4: [35 pts] Answer each question below.

(*a*) One disadvantage of computing lighting per-vertex rather than per-fragment is that lighting does not look as good.

☐ Illustrate that effect by showing lighting with per-fragment lighting and with per-vertex lighting.

☐ Show how eye position can make the computation needed for per-vertex lighting *greater than* that needed for per-fragment lighting. ☐ Explain.

(*b*) When rendering a scene we need to make sure that we don't see the parts of a triangle that are behind another triangle.

☐ How does rasterization rendering prevent a blocked triangle part from being visible? ☐ What storage or object is specifically used to compare distance from eye? ☐ Illustrate using two triangles.

☐ How does ray tracing prevent a blocked object from being visible? ☐ Illustrate using two triangles.

Staple This Side

(*c*) The CPU-only course code includes demos for rasterization and ray tracing. The amount of computation performed by the rasterization demos, such as `demo-04-z-light.cc`, roughly matches that needed by rasterization done by real systems, such as by Vulkan rasterization pipelines, say on an AMD video card. However, the amount of computation performed by the ray tracing demo, `demo-05-ray-tracing.cc`, is much larger than that performed by Vulkan ray-tracing pipelines on an AMD video card. Explain why using variables T for the number of triangles, A, the average number of fragments per triangle, and $w \times h$, the number of pixels in the frame buffer.

☐ Why does our CPU-only ray tracing do more computation? ☐ Why do real ray tracing systems perform less computation? ☐ Note that this question asks about the amount of computation, not the time needed to do it.

(*d*) The code below is shockingly inefficient.

```
//  Outer Loop: Iterate Over Triangles
for ( size_t i=start; i+2 < coors_os.size(); i += inc ) {
    // Compute window-space coordinates.
    pCoor u0 = window_from_clip * clip_from_eye * eye_from_object * coors_os[i];
    pCoor u1 = window_from_clip * clip_from_eye * eye_from_object * coors_os[i+1];
    pCoor u2 = window_from_clip * clip_from_eye * eye_from_object * coors_os[i+2];
```

☐ What about the code above would make one interested in efficiency gasp in horror?

☐ Fix the problem.

☐ Indicate how much less computation is done. ☐ Provide a number.

Staple This Side

(*e*) To render a scene we need to know something about the user and the user's monitor. (Elsewhere this would collectively be called the camera.)

☐ What about the user and the user's monitor do we need to know to transform from world space to eye space?
☐ Be specific, for example, show the information in transformation matrix constructors.

☐ What about the user and the user's monitor do we need to know to transform from eye space to clip space?
☐ Be specific, for example, show the information in transformation matrix constructors.

Staple This Side