Name Solution _____ *Formatted For Two-Sided Printing*

Staple This Side

GPU Programming
EE 4702-1
Midterm Exam

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

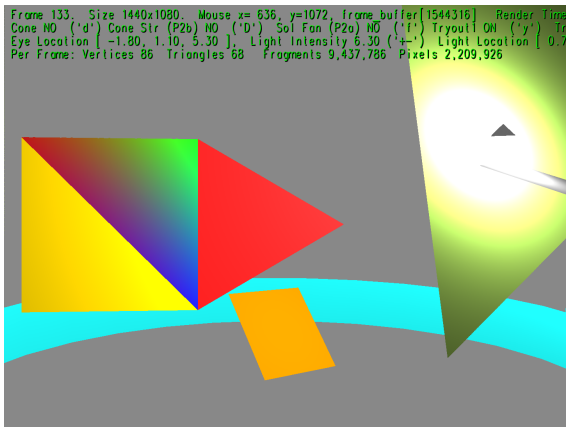
Problem 1	_____	(25 pts)
Problem 2	_____	(25 pts)
Problem 3	_____	(15 pts)
Problem 4	_____	(35 pts)
Exam Total	_____	(100 pts)

Alias Mechanical "Mouse?" _____

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").

Solution to Part (a): Solution appears on the facing page. Obviously, points `p3` and `p4` form the vertical edge of the red equilateral triangle. The code writes the coordinates of the third vertex to `px`. Let v_{34} be the "vertical" vector from `p3` to `p4` and v_{24} be the "horizontal" vector from `p2` to `p4`. (The words vertical and horizontal are quoted to emphasize that our method of computing `px` does not rely on v_{34} actually being vertical [aligned with the y axis], nor on v_{24} being horizontal.) To reach `px` start at `p2` and move "up" by $\frac{1}{2}v_{34}$ and then across by $\sqrt{\frac{3}{4}}\,v_{24}$. The factor $\sqrt{\frac{3}{4}}$ was found by solving the Pythagorean Theorem $1^2 = \left(\frac{1}{2}\right)^2 + x^2$ for x.

Solution to Part (b): Solution appears on the facing page. The normal can be found by taking the cross product of vectors formed from the vertices. Note that a normal must be inserted for each vertex.

Solution to Part (c): Solution appears on the facing page. Start by computing the square normal, `sq_az`, which is a vector from the square center to the light. Using that compute `sq_ax` so that it is a vector orthogonal to `sq_az` and the y axis. From those, compute `sq_ay` and use `sq_ax` and `sq_ay` to compute the square's vertices, and emit them along with the color.

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.

```
// SOLUTION -- Problem 1(a)
pVect v_34(p3,p4);  // Compute "horizontal" vector.
pVect v_24(p2,p4);  // Compute "vertical" vector.
pCoor px = p3 + sqrt(0.75) * v_24 + 0.5 * v_34;
coors_os << p4 << p3 << px;          // Emit vertices.
colors << c_red << c_red << c_red;   // Emit colors.
```

☑ 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;

// SOLUTION -- Problem 1(c)
// Compute axes for drawing the square. Square is along sq_ax and sq_ay.
pNorm sq_az = pVect( sq_ctr, lt_loc );       // Normal to square.
pNorm sq_ax = cross( pVect(0,1,0), sq_az );  // Parallel to xz plane.
pVect sq_ay = cross( sq_az, sq_ax );

// Compute vectors from square center to +x and +y edges.
pVect sq_vx = sq_l/2 * sq_ax,            sq_vy = sq_l/2 * sq_ay;

// Compute square's vertices.
pCoor sq11 = sq_ctr + sq_vx + sq_vy,     sq01 = sq_ctr - sq_vx + sq_vy;
pCoor sq00 = sq_ctr - sq_vx - sq_vy,     sq10 = sq_ctr + sq_vx - sq_vy;

// Emit square's vertices and colors.
coors_os << sq11  << sq10  << sq01       << sq01  << sq10  << sq00;
colors   << c_tan << c_tan << c_tan      << c_tan << c_tan << c_tan;
```

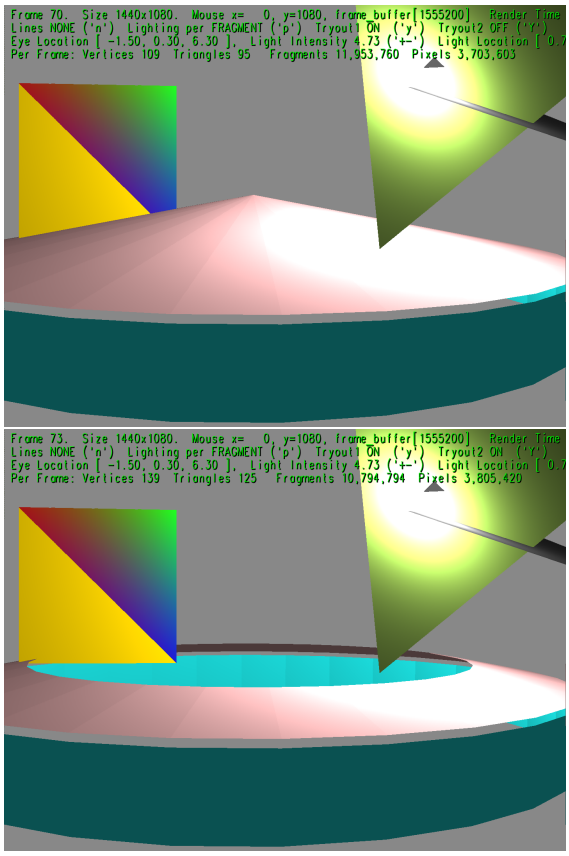
☑ Add code to the loop body below so that it inserts triangle normals into container norms_os.

```
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];

    // SOLUTION -- Problem 1(b)
    pNorm n = cross( pVect(p1,p2), pVect(p1,p3) );
    norms_os << n << n << n;
  }
```

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.

Solution appears below. The `inc` (increment) is now set to 1 for both fans and strips. Also, for a fan `i0` is always 0.

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


size_t start = 0;

/// SOLUTION – Problem 2(a)
//  For both fan and strip increment should be 1.
inc = topology == Topology_Individual ? 3 : 1;
//
// Both a triangle fan and triangle strip create a new triangle for
// each new vertex.


//  Outer Loop: Iterate Over Triangles
//
for ( size_t i=start; i+2 < coors_os.size(); i += inc )
  {
    /// SOLUTION – Problem 2(a)
    // size_t i0 = i;  // Comment out this line.
    size_t i0 = topology == Topology_Fan ? 0 : i;  // For fan need vertex 0 in every triangle.
    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.

Solution appears below. To draw the cut-off cone the code computes a vector, `vtc`, from the circle perimeter to the cone top. It then uses `vtc` to find a point 30% the way from the perimeter to the top. Note also that each iteration emits two vertices (`c` and `c+0.3*vtc`) and that the initial cone top point is removed.

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

/// SOLUTION – Problem 2(b)
// coors_os << cone_top_ctr;  // Central point of fan not needed for a strip.
// 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;

    /// SOLUTION – Problem 2(b)
    pVect vtc( c, cone_top_ctr);  // Vector from circle perimeter (c) to top of cone.
    // Emit circle perimeter point, and a point 1/3 way to top of cone.
    coors_os << c      << c + 0.3 * vtc;
    colors   << c_tan  << 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

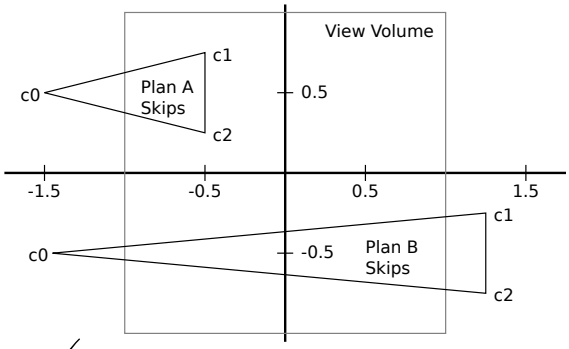
Problem 3: [15 pts] Shown below is another excerpt from `Our_3D`. The intent of the Plan A and Plan B code variations is to avoid rasterizing a triangle that's outside the view volume.

```
  // 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.

The Plan A code only checks one vertex, so such a triangle is still visible if the other two vertices are in the view volume. See the upper triangle in the diagram on the right.



☑ 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.

The Plan B code does not account for the possibility that though all three vertices are outside the view volume some points inside the triangle can still be in the view volume. See the lower triangle in the diagram to the right.

(*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.

The code appears below. The test `c0.x < -1` checks whether point `c0` is outside the view volume based on the $x = -1$ plane. A triangle is definitely not visible if *all three* of its vertices are on the outside side of the *same plane*. For example, the first line in the code below uses the $x = -1$ plane, the second line uses the $y = -1$ plane, etc.

```
      // SOLUTION
      if (  c0.x < -1  &&  c1.x < -1  &&  c2.x < -1  ) continue;
      if (  c0.y < -1  &&  c1.y < -1  &&  c2.y < -1  ) continue;
      if (  c0.x > +1  &&  c1.x > +1  &&  c2.x > +1  ) continue;
      if (  c0.y > +1  &&  c1.y > +1  &&  c2.y > +1  ) continue;
```


Staple This Side

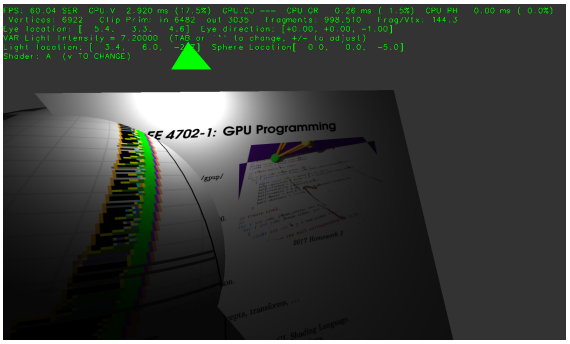
Staple This Side

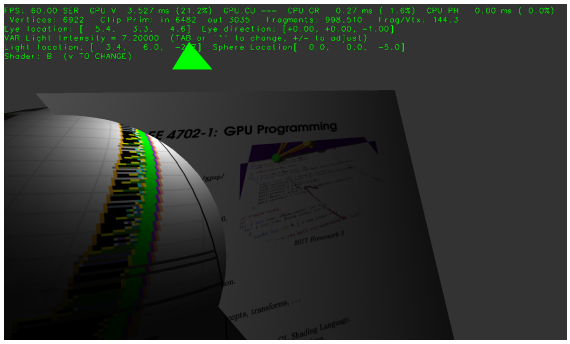
Problem 4: [35 pts] Answer each question below.

(*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.

Solution appears below. The screenshot on the lower left was rendered with per-fragment lighting and the one on the right with per-vertex lighting. In the left image a bright circle of light is clearly shown on the syllabus image because the distance to the light is taken into account for each fragment. In contrast for the screenshot on the right the lighting is computed only at the vertices and so the lighted color at a fragment is based on a weighted average of the lighted colors at each vertex, and the weighted average can never be brighter than the most brightly lit vertex, which will be dimmer than the most brightly lit fragment.





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

In per-vertex lighting lighting will be computed three times for each (individual triangle), once per vertex, no matter what. In per-fragment lighting it is computed once per fragment. The further the eye is from the triangle the fewer fragments a triangle will have, and so per-vertex lighting computes more fragments if the eye is far enough away so that the projected size of the triangle falls on one or two fragments.

(*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.

Rasterization uses a *depth buffer* (sometimes called a *z buffer*) to prevent a fragment from being written to the frame buffer if it is more distant from the eye than the fragment that's already written there. For each pixel in the frame buffer there is a storage location in the depth buffer which stores the distance from the eye (or something equivalent to distance) of whatever fragment is in the color buffer (the buffer where the pixels are written). If a fragment being processed is closer to the user than its lighted color is written to the color buffer and its distance to the eye is written into the depth buffer.

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

In ray tracing a ray is cast from the eye for each pixel in the frame buffer. For a particular ray the ray tracing code will find, in a *cast ray operation*, every triangle that the ray intersects. For each of those triangles the distance from the eye is found, and the code will keep track of the triangle that is closest to the eye. There is no depth buffer because the cast ray operation works on a single ray, and a simple scalar variable (as opposed to an array) can keep track of the distance.

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.

Full-Credit Answer: Our ray-tracing code does more computation because the inner loop iterates through *every triangle* whereas "real" ray-tracing systems use a data structure like a *bounding volume hierarchy (BVH)* to quickly rule out intersections with most triangles.

Explanation for those studying, not needed as part of the answer: For example, if a plane can be found such that almost half of the triangles are to the left of the plane and almost half are to the right of the plane, then if the ray from the eye is on the left side, almost half the triangles can be skipped. (A few triangles will be on both sides.) In a BVH and similar structures are hierarchical so there will be many planes which can be used to iteratively rule out more and more triangles.

(*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?

Three matrices are multiplied for each of the three assignments in each iteration of the loop. Since the matrices don't change inside the loop a product can be computed before the loop. See the fix below.

☑ Fix the problem.

```
// SOLUTION
pMatrix window_from_object = window_from_clip * clip_from_eye * eye_from_object;
//  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_object * coors_os[i];
    pCoor u1 = window_from_object * coors_os[i+1];
    pCoor u2 = window_from_object * coors_os[i+2];
```

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

Multiplying two 4×4 matrices requires $4^3 = 64$ multiply/add (MADD) operations. A matrix/vector product requires $4^2 = 16$ MADD operations. If the loop executes for T iterations, the total work is $(4^3 \times 2 + 4^2)3T = 432T$ MADD operations for the code above. If the matrix product is pre-computed the work is $4^3 \times 2 + 3 \times 4^2 T = 128 + 48T$ MADD operations, which is much fewer.

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.

The eye location and monitor (eye) direction are needed. The transformation will include a translation that moves the eye location to the origin and a rotation that rotates the monitor (projection plane) so that its normal is in the $-z$ direction.

☑ 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.

The distance from the eye to the monitor is needed (the *near distance*, and the width and height of the monitor. In the frustum transform used in class the width is specified by two variables, l (left) and r (right), the height is specified by two variables t (top) and b (bottom), the monitor distance by n (near) and the maximum view able distance f (far). These values define a view volume, and eye-to-clip transformation maps coordinates inside the view volume to the range ± 1 for x and y axes and $[0, 1]$ for the z axis.