Staple This Side

Name Solution ______________________ *Formatted For Two-Sided Printing*

GPU Programming
EE 4702-1
Midterm Exam

25 October 2023, 9:30-10:20 CDT

Problem 1	______	(30 pts)
Problem 2	______	(15 pts)
Problem 3	______	(25 pts)
Problem 4	______	(30 pts)
Exam Total	______	(100 pts)

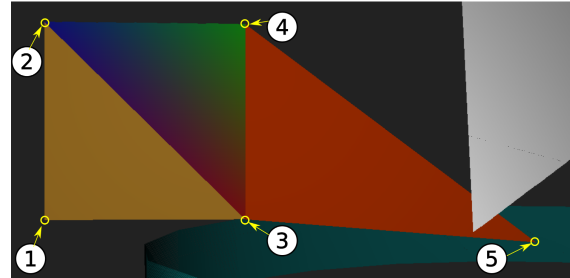
Alias Starship Launch II? ______________________

Good Luck!

Staple This Side

Problem 1: [30 pts] Appearing below is code to render three adjacent triangles (from left to right: gold, multicolored, and red), taken from the `World` class in the solution to Homework 3.

(*a*) The code below is set to render the three triangles as a triangle strip but the coordinates are inserted for an individual triangle grouping. Modify the code so that the coordinates are inserted for a triangle strip.



☑ In `World` insert coordinates for the three triangles using triangle-strip grouping.

The coordinates for the triangle strip are entered in the order shown on the screenshot above. Those same numbers are shown in the comments in the code below, to the right of `Tri Strip Num:`. Notice that coordinates and colors that are not used in a triangle strip have been crossed out below.

```
/// Code from World::render_scene()
vector<pCoor> coors_os;   vector<pColor> colors;

// Insert white and purple triangles.
coors_os << pCoor( 0, 0, 0 ) << pCoor( 9, 6, -5 ) << pCoor( 0, 7, -3 );
colors << color_white << color_white << color_white;
coors_os << pCoor(7,2,1) << pCoor(-3,3,-3.5) << pCoor(9,0,0);
colors << color_lsu_purple << color_lsu_purple << color_lsu_purple;

gc.topology_strip_set(false).vtx_normals_set();
gc.vtx_coors_set(coors_os).vtx_colors_set(colors).draw_rasterization();
coors_os.clear(); colors.clear();
// Code above correct, don't modify.

/// SOLUTION – Part a (coors_os) and Part b (colors): Cross out unneeded vertices.
// Tri Strip Num:    1                  2                  3
coors_os << pCoor(-4,0,-3) << pCoor(-4,2,-3) << pCoor(-2,0,-3);
colors << color_lsu_gold << color_lsu_gold << color_lsu_gold;

// Tri Strip Num:                        4
coors_os << pCoor(-4,2,-3) << pCoor(-2,2,-3) << pCoor(-2,0,-3);
colors << color_blue << color_green << color_red;

// Tri Strip Num:                                          5
coors_os << pCoor(-2,2,-3) << pCoor(-2,0,-3) << center + sz;
colors << color_red << color_red << color_red; // Solution: Last 2 colors unused.

// Code below correct, don't modify.
gc.topology_strip_set(true).vtx_normals_set();
gc.vtx_coors_set(coors_os).vtx_colors_set(colors).draw_rasterization();
coors_os.clear(); colors.clear();
```

In the printed code, these items are crossed out: in the first colors line, the second and third `color_lsu_gold`; in Tri Strip Num 4, `pCoor(-4,2,-3)`, `pCoor(-2,0,-3)`, `color_blue` and `color_red`; in Tri Strip Num 5, `pCoor(-2,2,-3) << pCoor(-2,0,-3)`.

Staple This Side

Staple This Side

(*b*) Modify the code in `World` and `Our_3D` so that the gold and red triangles retain their colors and so that the multicolored triangle is now green with the triangle-strip grouping. The changes for `Our_3D` should work for all triangle strips.

☑ In `World` insert colors for the three triangles to color are gold, green (instead of multi-colored), and red with triangle-strip grouping.

In a triangle strip each vertex can be part of up to three triangles. So to specify the color of a triangle use the color associated with the first vertex. (The second or third vertex could also have been used, but this solution uses the first.) The insertion of colors for the three triangles on the previous page has been modified to reflect this. Notice that the first three colors are now gold, green, and red. The last two colors of the group, also red, are not used.

☑ Modify `Our_3D` so that triangles are gold, green, and red, instead of a blurry mess. ☑ Changes should only affect triangle strips. ☑ Only a line or two of code is needed. Don't overdo it.

Solution appears below. Use `c0` if a strip is being rendered. Notice that since one color is used there is no need to use the barycentric coordinates, `b0`, `b1`, `b2`, to blend colors.

```
/// Code from Our_3D::draw_rasterization()
for ( size_t i=0; i+2<coors_os.size(); i += ( topology_strip ? 1 : 3 ) ) {
    // Get next triangle's object- and window-space coordinates.
    pCoor o0 = coors_os[i+0],  o1 = coors_os[i+1],  o2 = coors_os[i+2];
    pCoor_Homogenized w0( ws_from_os*o0 ), w1( ws_from_os*o1 ), w2( ws_from_os*o2 );
    pColor c0 = colors[i+0],  c1 = colors[i+1],  c2 = colors[i+2];

    pNorm tn = cross(o0,o1,o2); // Compute triangle normal (for lighting).
    pVect4 n0, n1, n2;
    if ( vtx_normals ){ n0=vtx_normals[i]; n1=vtx_normals[i+1]; n2=vtx_normals[i+2]; }
    pVect v20(w2,w0), v21(w2,w1);
    float db0 = 1/max(fabs(v20.x),fabs(v20.y)), db1 = 1/max(fabs(v21.x),fabs(v21.y));

    // Iterate over triangle using barycentric coordinates.
    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;  // Fragment window-space coordinate.
        if ( uint(wf.x) >= win_width || uint(wf.y) >= win_height ) continue;
        const size_t idx = wf.x + int(wf.y) * win_width;  // Frame buffer index.

        // Depth (Z) Test
        if ( wf.z < 0 || wf.z > 1 || zbuffer[ idx ] < wf.z ) continue;
        zbuffer[ idx ] = wf.z;
        pCoor of = b0*o0 + b1*o1 + b2*o2;  // Approx object-space coord of fragment.

        /// SOLUTION - Part b: Just use c0 if rendering a triangle strip.
        pColor color = topology_strip ? c0 : b0*c0 + b1*c1 + b2*c2;

        // Compute Lighted Color
        pVect4 n = vtx_normals ? b0*n0 + b1*n1 + b2*n2 : tn;
        pNorm f_to_l(of,light_location);
        float phase = fabs(dot(n,f_to_l));
        pColor lighted_color = phase / f_to_l.mag_sq * light_color * color;

        frame_buffer[ idx ] = lighted_color.int_rgb(); // Write the frame buffer.
      }}
```

Problem 2: [15 pts] The code below is condensed from the Homework 2 solution, in which an inner ring had to be added to an outer ring of balls. Unlike the homework solution the code below computes the position (coordinate) of a ball on the inner ring using transformation matrix `m` on the position of an outer ring ball. Add code to compute `m` so that the inner ball is placed in the correct position. *Hint: This is not that difficult.*

☑ Add code so that `m` is set to a transformation matrix that finds an inner-ball position given an outer-ball position.

```
float r_outer = si.ring_outer_radius;        // Radius of outer ring.
pCoor ctr_outer = si.center_pos;             // Center of outer ring.
pVect vx = r_outer * ax,  vy = r_outer * ay;

for ( int i=0; i<n_balls; i++ ) {            // Place balls on outer ring.
    pCoor pos = ctr_outer + vx * cosf(Δθ * i) + vy * sinf(Δθ * i);
    balls += new Ball( pos, ball_default );} // Construct ball, add to ball list.

float r_inner = 0.5f * p1p2.magnitude;       // Radius of inner ring.
pCoor ctr_inner = si.p1 + r_inner * p1p2;    // Center of inner ring.
pVect vx2 = r_inner * ax, vy2 = r_inner * ay;

if ( false ) for ( int i=0; i<n_balls; i++ ){
    // Compute Inner-Ball Position.  THIS LOOP DOESN'T EXECUTE. SHOWN FOR REFERENCE.
    pCoor pos = ctr_inner + vx2 * cosf(Δθ * i) + vy2 * sinf(Δθ * i);
    balls += new Ball( pos, ball_default ); }

// The declarations below are for reference.
pNorm axis;
float angle, scale_factor;
pVect vec;
pMatrix_Rotation mat_r(axis,angle);   // Abbreviation: pM_Rot
pMatrix_Translate mat_t(vec);         // Abbreviation: pM_Tra
pMatrix_Scale mat_s(scale_factor);    // Abbreviation: pM_Sca
// The declarations above are for reference.


/// SOLUTION
//
pM_Tra m_to_origin( -ctr_outer );   // Move center of outer circle to origin.
pM_Sca m_scale( r_inner/r_outer );  // Change size. (Must be done with center at origin.)
pM_Tra m_from_origin( ctr_inner );  // Move to inner circle position.

pMatrix m = m_from_origin * m_scale * m_to_origin;        ; // ☑  Compute m.

for ( int i=0; i<n_balls; i++ ) // "New" For This Exam
  {
    pCoor pos_outer = balls[i]->position;       // DO NOT MODIFY THIS LOOP.
    pCoor pos_inner = m * pos_outer;
    balls += new Ball( pos_inner, ball_default );
  }
```


Staple This Side

Staple This Side

Problem 3: [25 pts] A scene consisting of $T = 30$ triangles is to be rendered on to a $w \times h$ pixel frame buffer with $w = h = 100$. Suppose that each triangle has an area of $A = 40$ pixels in window space. **All parts of all triangles are in the view volume.**

(a) In a rasterization draw for this scene, what are the maximum and minimum number of times the depth (z) and color buffer can be read and written. (The color buffer is where the pixel is written.) Explain.

☑ Answers below should be in terms of T, A, w, and h.

☑ Max number of depth buffer reads: Sol: TA ☑ Min number of depth buffer reads: Sol: TA

The depth buffer must be read for every fragment, and there are TA total fragments.

☑ Max number of depth buffer writes: Sol: TA ☑ Min number of depth buffer writes: Sol: A

The depth buffer is only written if the fragment being considered is closer to the eye than the one in the depth buffer or if there is nothing in the depth buffer. For non-overlapping triangles that works out to TA writes (the max), for perfectly overlapping triangles with the closest one first that works out to A writes.

☑ Max number of color buffer reads: Sol: 0 ☑ Min number of color buffer reads: Sol: 0

The color buffer is not read during a rendering pass, so zero in both cases. (The color buffer is later read when sending it to a display device [on simpler systems]).

☑ Max number of color buffer writes: Sol: TA ☑ Min number of color buffer writes: Sol: A

The answers and reasoning are the same as for the depth buffer writes.

☑ Briefly explain difference between maximum and minimum cases.

For the all of the minimum cases the triangles must perfectly overlap. For the minimum depth buffer and color buffer write cases the first triangle must be the one closest to the eye. For all of the maximum cases it is sufficient that the triangles do not overlap, not even one pixel.

(b) In a ray tracing draw, how many times will the color (frame) buffer be read and written for this scene? Explain.

☑ Answers below should be in terms of T, A, w, and h.

☑ Max number of color buffer reads: Sol: 0 ☑ Min number of color buffer reads: Sol: 0

☑ Max number of color buffer writes: Sol: TA ☑ Min number of color buffer writes: Sol: A

The number of fragments per triangle, A, is the same in rasterization and ray tracing. So the answer and reasoning used for color buffer writes in the rasterization applies here too.

(c) Our CPU-only ray-tracing code included a ray/triangle intersection test:

```
pVect v01(e0,e1), v02(e0,e2); // Find where ray intercepts plane defined by triangle.
pVect nt = cross(v01,v02);
float t = dot( pVect(e0), nt ) / dot( ray, nt );
// t indicates distance from eye to intercept point in units of ray lengths.
```

```
// Skip this triangle if a closer triangle already found.
if ( t >= tmin ) continue;
```

☑ For the scene above, how many times is this intersection test done by *our* CPU-only code in terms of T, A, w, and h.

In ray tracing a ray is cast from the eye through each pixel, so wh rays are cast. For each ray one must find the closest triangle intersection. In our code, `cpu-only/demo-05-ray-tracing.cc`, when looking for the closest triangle for a ray the intersection test above is applied to *every* triangle. So the total number of intersection tests in our code is $\boxed{whT}$. In practical ray tracing systems triangles are organized, perhaps into a *bounding volume hierarchy*, and using the organization fewer intersection tests are needed.

Staple This Side

Staple This Side

Staple This Side

Problem 4: [30 pts] Answer each question below.

(a) The code below from our CPU-only ray tracing demo finds the intersection of `ray` with the plane defined by a triangle. Show the number of arithmetic operations for each line, using the three categories: add, multiply & multiply-add, and divide. (Count subtract as an add, and count multiply-subtract as a multiply add.)

Solution shown below in blue. Note that the constructor `pVect v01(e0,e1)` constructs a vector, `v01`, using two coordinates, `e0` and `e1`. It does so by subtracting the corresponding x, y, and z components (ignoring w which would be a problem if $w \neq 1$).

```
float find_intercept(pCoor e0, pCoor e1, pCoor e2, pVect ray) {

// Number of  ☑ Add & sub:  6    ☑ Mult & Mult-Add:  0    ☑ Divisions:  0
  pVect v01(e0,e1), v02(e0,e2);

// Number of  ☑ Add & sub:  0    ☑ Mult & Mult-Add:  6    ☑ Divisions:  0
  pVect nt = cross(v01,v02);

// Number of  ☑ Add & sub:  0    ☑ Mult & Mult-Add:  6    ☑ Divisions:  1
  float t = dot( pVect(e0), nt ) / dot( ray, nt );

  return t; }
```

(b) Show the result of assignment of each line below.

The solution is shown below. Note that the component names after the `.` can be drawn from three sets, `xyzw` (convenient for coordinates and vectors), `rgba` (convenient for colors), and `stpq` (convenient for texture coordinates). Components can appear in any order and can be duplicated (as in `qb`). The size of the resulting vector is the number of components, which is two in each case below.

```
vec4 p = vec4(1,2,3,4);  // ☑  p = {1,2,3,4}

vec2 qa = p.xy;          // ☑  qa = {1,2}

vec2 qb = p.xx;          // ☑  qb = {1,1}

vec2 qc = p.ba;          // ☑  qc = {3,4}
```

(c) In graphics (classic OpenGL) terminology there is a distinction between a material property and a lighted color.

☑ What is the difference between a material property and a lighted color?

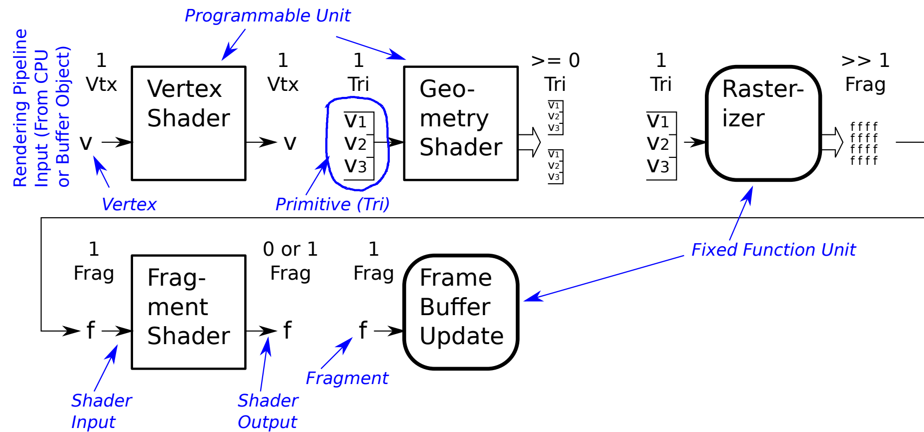
A material property is what is popularly known as a color. A leaf is green even if nobody is looking at it, or if it is completely dark outside. A lighted color is computed using the material property and information about a light source such as its location, brightness, etc, and the geometry of the light location, vertex location and normal, and possibly the eye.

☑ The frame buffer should be written with one of these two items. How could one tell if the wrong one of these two were written?

The lighted color should be written. If the material property were written instead an entire primitive would be the same color. Consider a sphere. If the material property were written for the triangles approximating the sphere the sphere would look like a circle.

Staple This Side

(*d*) Illustrated below is a diagram of a Vulkan/OpenGL graphics (rendering) pipeline with many labels omitted.



☑ Label each stage of the pipeline.

Solution appears above.

☑ At the start of the pipeline there is a label "Rendering Pipeline Input (From)". The words after *From* are omitted. What should follow the word *From*?

Solution appears above.

☑ There is a label "Fixed Function Unit" in the diagram. What does that mean?

That means the stage can't be programmed by the user. For example, a user (meaning a person writing, say, a video game) can write a vertex shader for the vertex stage but the user can't write a custom frame buffer update routine. The code in the fixed-function units is written by the developer of the Vulkan implementation or graphics device driver, which might be the manufacturer of the GPU or the writer of the operating system.

(*e*) Answer the following questions about shader stages used in a triangle list (not strip) vertex ordering.

☑ Which usually gets executed more frequently, a ○ *vertex shader* or a ⊗ *fragment shader* ?

☑ Explain.

In a triangle list grouping, there are three vertices for each triangle. In typical cases the triangle will cover many pixels, perhaps hundreds or more. For each covered pixel the fragment shader is run. Therefore, the fragment shader is run more often.

☑ A scene has one triangle. Explain a situation, perhaps in terms of the eye location, in which the number of vertex shader invocations will be three times the number of fragment shader invocations.

Lets assume that the triangle is completely within the view volume. It must be then that the eye is so far away from the triangle that the triangle is projected on to just one pixel.

Staple This Side
Staple This Side