

Name Solution_____

Formatted For Two-Sided Printing

GPU Programming
LSU EE 4702-1
Final Examination

Thursday, 7 December 2023 7:30-9:30 CST

Problem 1 _____ (20 pts)

Problem 2 _____ (20 pts)

Problem 3 _____ (10 pts)

Problem 4 _____ (15 pts)

Problem 5 _____ (20 pts)

Problem 6 _____ (15 pts)

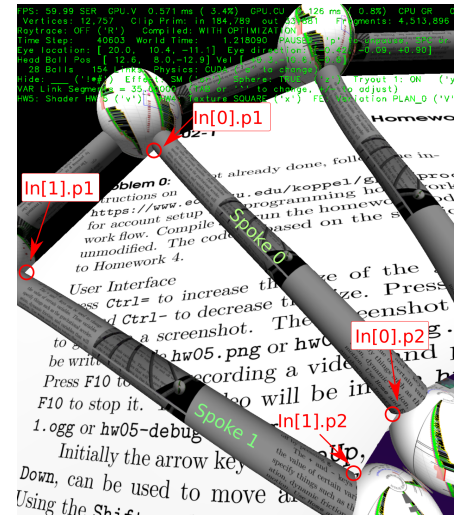
Alias ~~Whitespace-Efficient~~_____

Exam Total _____ (100 pts)

Good Luck!

Problem 1: [20 pts] The geometry shader code below is from the solution to Homework 5. Each invocation draws the area between two spokes. The screen shot shows how some inputs to the geometry shader relate to the rendered image.

(a) In the original code `EndPrimitive()` was not called until after the `s` loop. The code below calls `EndPrimitive()` inside the loop once when `variation=Plan_1` and at each iteration when `variation=Plan_s`. For `variation=Plan_0` the code is equivalent to the homework solution and it is the version used for the screenshot.



- ✓ Describe the appearance for `Plan_1`, draw a sketch. ✓ Explain.

There should be a gap in the center of the texture, see the screenshot to the lower right labeled `Plan 1`. Suppose `opt_segments/2 = 10`. At iteration 10 two vertices are emitted and then `EndPrimitive()` is called, ending a triangle strip. In iteration 11 two more vertices are emitted, and these are the first vertices of a new triangle strip, so at the end of 11 no new triangles are emitted. In iteration 12, the third and fourth vertex of the new strip is emitted, and so we get the first two triangles, which are between the iteration 11 coordinates and the iteration 12 coordinates. Because of `EndPrimivite()` there were no triangles between the iteration 10 and iteration 11 coordinates, and for that reason there is a gap.

- ✓ Describe the appearance for `Plan_s`, draw a sketch. ✓ Explain.

Nothing between the spokes will appear. That is because `EndPrimivite()` is called each iteration limiting each triangle strip to just two vertices, and so no triangles are rendered. See the screenshot labeled `Plan s`.

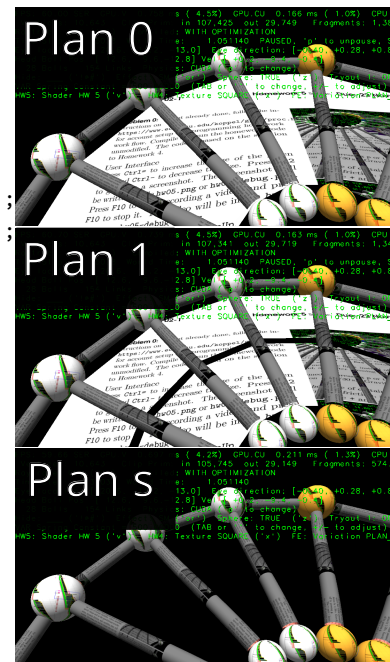
```
void gs_fe_ep() {
    for ( int s=0; s<=opt_segments; s++ ) {
        float t = float(s) / opt_segments;

        // Compute interpolated coords along spokes 0 and 1 using t.
        vec4 pos0t = bez( In[0].p1, In[0].p2, In[0].v1, In[0].v2, t );
        vec4 pos1t = bez( In[1].p1, In[1].p2, In[1].v1, In[1].v2, t );

        // Emit the vertex along spoke 0 ( In[0] )
        gl_Position = ut.clip_from_object * pos0t;
        Out.vertex_e = ut.eye_from_object * pos0t;
        Out.tex_coor = mix( In[0].tc1, In[0].tc2, t );
        EmitVertex();

        // Emit the vertex along spoke 1 ( In[1] ).
        gl_Position = ut.clip_from_object * pos1t;
        Out.vertex_e = ut.eye_from_object * pos1t;
        Out.tex_coor = mix( In[1].tc1, In[1].tc2, t );
        EmitVertex();

        switch ( variation ) {
            case Plan_0: break; // Original. Same as Homework 5 solution.
            case Plan_1: if ( s == opt_segments/2 ) EndPrimitive(); break;
            case Plan_s: EndPrimitive(); break; }
    }
    EndPrimitive();
}
```



(b) The screenshot to the upper right is from the Homework 5 solution. The screenshot below shows the surface between the spokes with a sine-wavy cut. Modify the geometry shader to make that cut. Variable `width` shows how far the surface should extend. A value of 0.5 means the surface should reach halfway from one spoke to another, a value of 0.7 means the surface goes 70% of the way, etc.

- ✓ Modify the shader so that the surface extends `width` from Spoke 0 (thus forming the sine wavy shape) ✓ taking care that the texture appears as shown.
- ✓ Don't overdo it. Only a few lines are needed. The mix function might come in handy.

Two solutions appears below. The first uses `width` to scale a vector from `pos0t` to `pos1t` and adds that to `pos0t`. The second uses the mix function.

```
void gs_fe_sine() {
    for ( int s=0; s<=opt_segments; s++ ) {
        float t = float(s) / opt_segments; // delta_t
        float width = 0.5 + ampl * ( 1 + sin( t * omega ) );

        // Interpolate coordinate along each spoke.
        vec4 pos0t=bez( In[0].p1, In[0].p2, In[0].v1, In[0].v2, t);
        vec4 pos1t=bez( In[1].p1, In[1].p2, In[1].v1, In[1].v2, t);

        vec4 vx = pos1t - pos0t;
        pos1t = pos0t + width * vx;    // SOLUTION -- Without mix

        pos1t = mix( pos0t, pos1t, width ); // SOLUTION -- With mix

        // Emit the vertex along spoke 0 ( In[0] )
        gl_Position = ut.clip_from_object * pos0t;
        Out.vertex_e = ut.eye_from_object * pos0t;

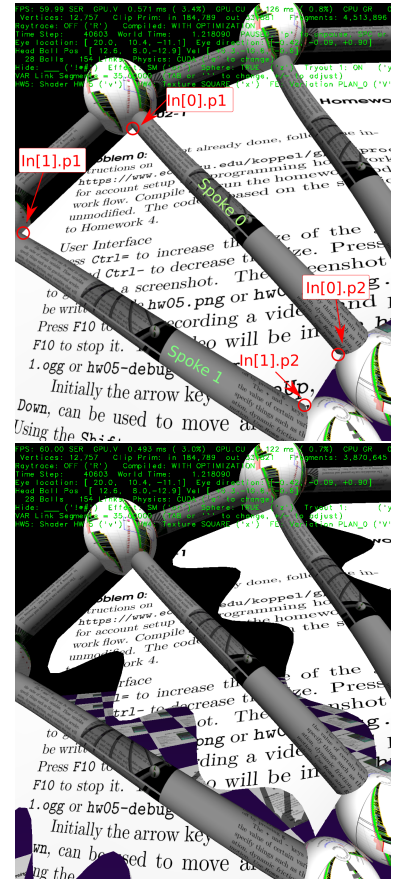
        // SOLUTION -- Keep first texture coordinate, needed to compute second.
        vec2 tc1 = mix( In[0].tc1, In[0].tc2, t );
        Out.tex_coor = tc1;

        EmitVertex();

        // Emit the vertex along spoke 1 ( In[1] ).
        gl_Position = ut.clip_from_object * pos1t;
        Out.vertex_e = ut.eye_from_object * pos1t;

        // SOLUTION -- Interpolate two texture coordinates using width.
        vec2 tc2 = mix( In[1].tc1, In[1].tc2, t );
        Out.tex_coor = mix( tc1, tc2, width );

        EmitVertex();
    }
    EndPrimitive();
}
```



Problem 2: [20 pts] The facing page shows CPU code that prepares the buffer set to draw the area between the spokes from the Homework 4 and Homework 5 solutions. The Homework 4 geometry shader is shown, the Homework 5 geometry shader code can be found in the previous problem. Texture coordinates have been omitted for simplicity.

The questions below are about the number of shader invocations and the amount of data per frame for the these code fragments. In your answers use n_s for the value of `n_spokes` and n_g for the value of `opt_segments`. The amount of data should be given in bytes. An integer and float are each 4 bytes.

- ☒ Per frame and in terms of n_s and n_g how many times is the vertex shader invoked for the ☒ Homework 4 version and the ☒ Homework 5 of the code?

The number of vertex shader invocations is equal to the number of elements in (each array of) the buffer set. So for the Homework 4 code count the number of coordinate insertions into `bset_hw04_p2`, and for the Homework 5 code count the number of index insertions into `bset_hw05`.

Homework 4: The number of vertex shader invocations is $n_s[(n_g + 1)2 + 2]$. The $+1$ in $(n_g + 1)$ because the end condition in the `s` loop is `s<=opt_segments` rather than `s<opt_segment`.

Homework 5: The number of vertex shader invocations is $n_s + 1$.

- ☒ Per frame and in terms of n_s and n_g how many times is the geometry shader invoked for the ☒ Homework 4 version and the ☒ Homework 5 of the code?

The geometry shader is invoked for each primitive assembled from the incoming vertices using the specified topology.

The Homework 4 topology is set to a triangle strip. In this topology V vertices form $V - 2$ triangles and for each triangle the geometry shader will be invoked. So for the Homework 4 code the number of geometry shader invocations is $n_s[(n_g + 1)2 + 2] - 2$.

The Homework 5 topology is set to a line strip. In this topology V vertices form $V - 1$ lines. So for the Homework 5 code the number of geometry shader invocations is n_s .

- ☒ Per frame and in terms of n_s and n_g how many primitives are sent to the rasterizer for the ☒ Homework 4 version and the ☒ Homework 5 of the code?

The output of the geometry shader is sent to the rasterizer.

The Homework 4 geometry shader emits one triangle per invocation, so the number of triangles sent to the geometry shader is $n_s[(n_g + 1)2 + 2] - 2$. For those wondering why the number of triangles isn't just $2n_s n_g$: So that a single triangle strip can be used for the entire draw the first two triangles and the last triangle of the `s` loop have zero area.

The Homework 5 geometry shader emits $2n_g$ triangles per invocation. So the total number of triangles sent to the geometry shader is $2n_s n_g$.

- ☒ Per frame, does the Homework 4 code make ☐ fewer fragment shader invocations, ☒ about the same number of fragment shader invocations, or ☐ more fragment shader invocations than the Homework 5 code? ☒ Explain.

The same because the triangles emitted by Homework 4 and Homework 5 cover the exact same pixels, and so the number of fragment shader invocations will be identical.

- ☒ Per frame and in terms of n_s and n_g how much data is sent from the CPU to the GPU for ☒ the Homework 4 version and ☒ the Homework 5 of the code? ☒ Account for data in the buffer sets and in storage buffers `pos1`, `v1`, etc.

The Homework 4 code inserts only coordinates into the buffer set and each coordinate is $4 \times 4 \text{ B} = 16 \text{ B}$. It inserts $n_s[(n_g + 1)2 + 2]$ coordinates, and it does not use storage objects, and so the total amount of data sent to the GPU is $n_s[(n_g + 1)2 + 2]16 \text{ B}$.

The Homework 5 code inserts only integers into the buffer set, each integer is 4 B, the total amount of data due to the buffer set is $(n_s + 1)4B$. The Homework 5 code also uses storage buffers **pos1**, **pos2**, **v1**, and **v2**. Each element of each buffer is 16 B. (The vector storage buffer elements are **vec4** despite the fact that vectors have only three elements.) The number of elements in each buffer is n_s (one per spoke). Each storage buffer is $16n_s B$. The four storage buffers together are $64n_s B$. Because the links move it will be necessary to send those storage buffers to the GPU each frame.

The total amount of data from the Homework 5 code is the size of the buffer set and the storage buffers which is $4(n_s + 1)B + 64n_s B$.

```

/// Homework 4 CPU Code: Input topology is a triangle strip. Vertex attribute: pCoord
for ( size_t i = 0; i < n_spokes; i++ ) {
    size_t i2 = (i+1) % n_spokes;
    int link_idx_1 = hw04_stuff.ribs[i]->idx, int link_idx_2 = hw04_stuff.ribs[i2]->idx;
    pCoord pos11 = lis_pos1[link_idx_1], pos12 = lis_pos2[link_idx_1];
    pVect v11 = lis_v1[link_idx_1], v12 = lis_v2[link_idx_1];
    pCoord pos21 = lis_pos1[link_idx_2], pos22 = lis_pos2[link_idx_2];
    pVect v21 = lis_v1[link_idx_2], v22 = lis_v2[link_idx_2];

    for ( int s=0; s<=opt_segments; s++ ) {
        const float t = float(s) / opt_segments;
        pCoord pos1t = bez(pos11,pos12,v11,v12,t), pos2t = bez(pos21,pos22,v21,v22,t);
        if ( s == 0 ) bset_hw04_p2 << pos1t;
        bset_hw04_p2 << pos1t << pos2t;
        if ( s == opt_segments ) bset_hw04_p2 << pos2t;
    }
}
bset_hw04_p2.to_dev(); pipe_hw04_p2.record_draw( cb, bset_hw04_p2 );

/// Homework 4 Vertex Shader
void vs_main() { Out.vertex_o = in_vertex_o; }

/// Homework 4 Geometry Shader
void gs_main() {
    vec3 normal_o = cross( In[1].vertex_o.xyz - In[0].vertex_o.xyz,
                          In[2].vertex_o.xyz - In[0].vertex_o.xyz );
    vec3 normal_e = mat3( ut.eye_from_object ) * normal_o;
    for ( int i=0; i<3; i++ ) {
        gl_Position = ut.clip_from_object * In[i].vertex_o;
        Out.normal_e = normal_e;
        Out.vertex_e = ut.eye_from_object * In[i].vertex_o;
        EmitVertex();
    }
    EndPrimitive();
}

/// Homework 5 CPU Code: Input topology is a line strip. Vertex attribute: integer (link_idx)
for ( size_t i = 0; i <= n_spokes; i++ )
    bset_hw05 << hw05_stuff.spokes[i%n_spokes]->idx; // Insert link index.
bset_hw05.to_dev(); pipe_hw05.record_draw( cb, bset_hw05 );

/// Homework 5 Vertex Shader
void vs_hw05() {
    Out.p1 = pos1[in_link_idx]; Out.p2 = pos2[in_link_idx]; // Get coords from storage buffer.
    Out.v1 = v1[in_link_idx]; Out.v2 = v2[in_link_idx]; }

// See prior problem for geometry shader.

```

Problem 3: [10 pts] Appearing below are hourglass shapes with vertices numbered. The code below inserts vertex coordinates into a buffer set as part of **an unsuccessful attempt** to render that shape. The inserted *coordinates* are correct (and the vertex numbers shown in the comments match the diagram). The problem is their order and the number of times each vertex is inserted. (Vertex 4 is inserted twice, but that's not the only problem.)

```
bset_hg
<< pCoor(0,0) // Vtx 0
<< pCoor(4,0) // Vtx 1
<< pCoor(1,1) // Vtx 2

<< pCoor(1,3) // Vtx 4
<< pCoor(2,1) // Vtx 3
<< pCoor(2,3) // Vtx 5

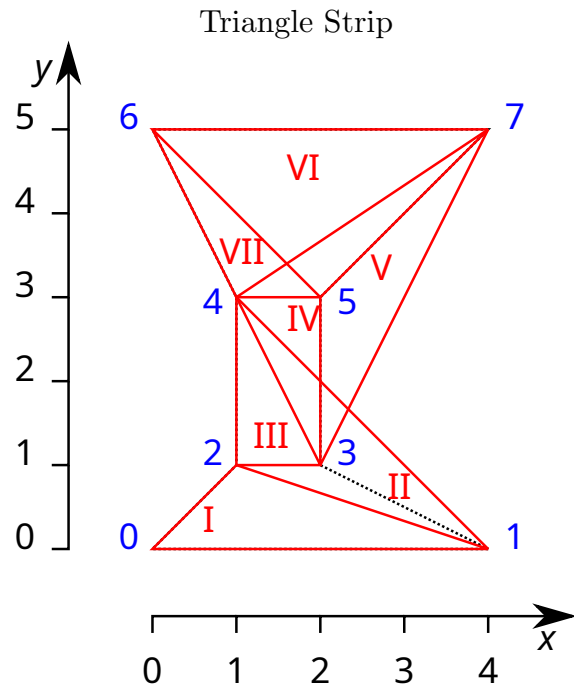
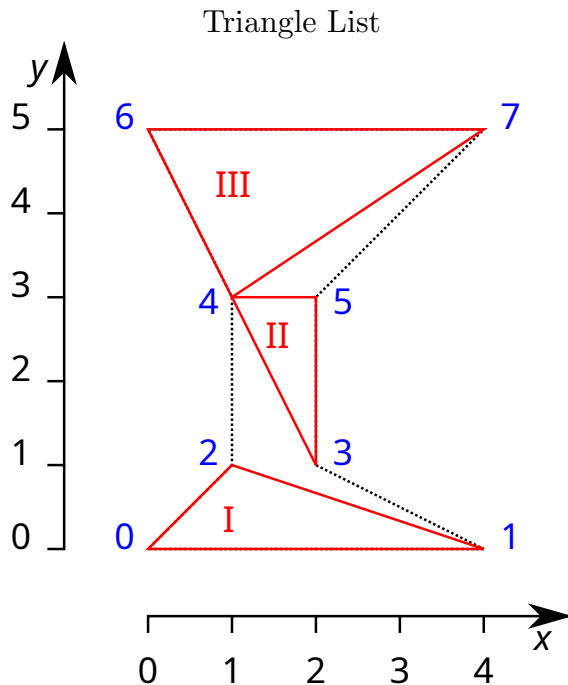
<< pCoor(4,5) // Vtx 7
<< pCoor(0,5) // Vtx 6
<< pCoor(1,3); // Vtx 4
```

Do not try to fix the code. Instead show the result of rendering with these coordinates for the topologies requested below.

☒ On the diagram on the left sketch the triangles that will be rendered by the code below when the pipeline is set up for a triangle list.

☒ On the diagram on the right sketch the triangles that will be rendered by the code below when the pipeline is set up for a triangle strip.

Solutions appear below in red. Each triangle is labeled with a Roman numeral.



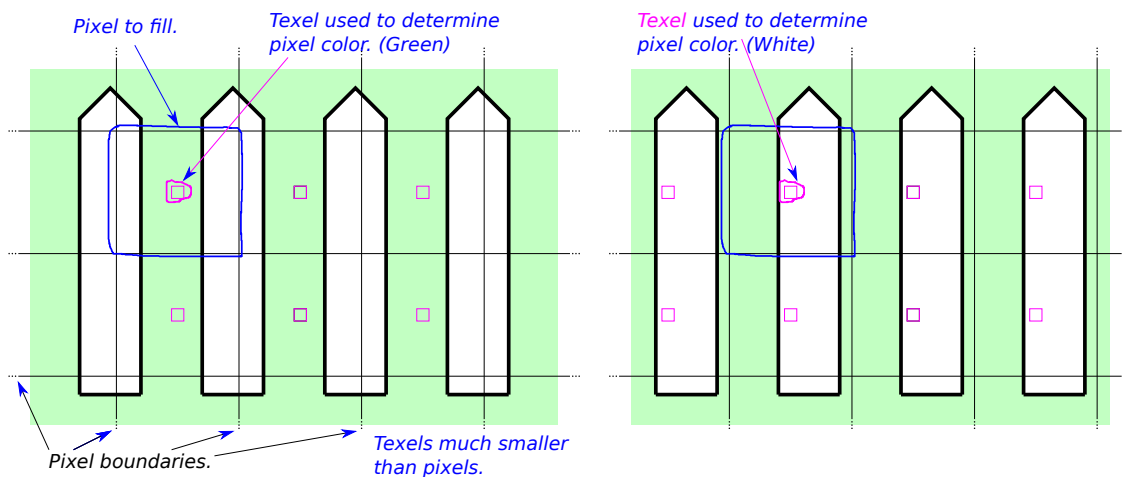
Problem 4: [15 pts] Answer the following texture questions.

(a) A texture at MIPMAP level 4 is 8192 bytes. What is the size of the image at MIPMAP level 5?

☒ The size of the image at level 5 is:

The size is $\frac{8192}{2 \times 2^{5-4}} = \frac{8192}{4} = 2048$.

(b) The images below illustrate a problem that can occur when a texel size is much smaller than a pixel size. The texture image shows a white picket fence with green behind the fence (probably grass).



Suppose the eye in the scene illustrated above is slowly moving from left to right. What would be the major problem with the appearance of the scene?

☒ Major problem with appearance:

The object would change from green to white as the texel used to determine the pixel color moved from the grass to the fence post.

The solution to the problem is to use the average color of all the texels covering each pixel.

☒ When and where are texels averaged together? ☒ Why doesn't this averaging have an impact on rendering speed?

The pixels are averaged together in advance. Perhaps they are averaged right after the images are loaded when the program starts running. Since this is before the first frame is rendered there is no impact on performance (other than start-up).

Problem 5: [20 pts] Answer the following questions about low-level rendering code.

(a) The loop nest below iterates over the barycentric coordinates of a triangle to find fragments to write into the frame buffer. Variables `w0`, `w1`, and `w2` hold the window-space coordinates of the triangle vertices. Consider increments `db0` and `db1`.

```
for ( float b0=0; b0<=1; b0 += db0 )
  for ( float b1=0; b1<=1-b0; b1 += db1 ) {
    const float b2 = 1 - b0 - b1;
    pCoord wf = b0*w0 + b1*w1 + b2*w2; // Window-space coordinate of fragment.
    frame_buffer[ wf.x + int(wf.y) * win_width ] = color; }
```

- ☒ Describe the consequences if `db0` and `db1` are too small. Consider both ☒ correctness ☒ and performance.

If they are too small the same pixel will be written several times with the same color. That does not impact correctness but it does slow down execution.

- ☒ Describe the consequences if `db0` and `db1` are too large. Consider both ☒ correctness ☒ and performance.

If they are too large then some pixels will not be written. The triangle may have a striped or checkered appearance. This will speed up execution, which is normally a good thing but not here because the triangle is not shown correctly.

(b) The rasterization code below uses either eye-space (perspective correct) barycentric coordinates `bc0`, `bc1`, and `bc2` or window-space barycentric coordinates to interpolate texture coordinates `tc0`, `tc1`, and `tc2`. Suppose a scene consists of two triangles forming a rectangle, and a texture is applied to the rectangle. The texture image shows a ruler (for measuring lengths).

```
for ( float b0=0; b0<=1; b0 += db0 )
  for ( float b1=0; b1<=1-b0; b1 += db1 ) {
    const float b2 = 1 - b0 - b1;
    pCoord wf = b0*w0 + b1*w1 + b2*w2; // Window-space coordinate of fragment.

    float bc0 = 1/(f.w0w2z*(1-b0)/b0+1),    bc1 = 1/(f.w1w2z*(1-b1)/b1+1),    bc2 = 1-bc0-bc1;

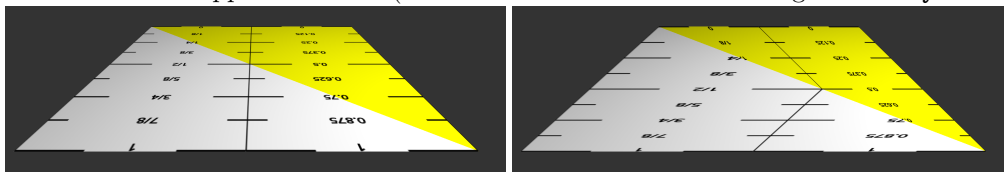
    if ( use_eye ) tc = bc0*tc0 + bc1*tc1 + bc2*tc2; // Eye-space interpolation.
    else          tc = b0*tc0 + b1*tc1 + b2*tc2;    // Window-space interpolation.
    texel = texture2D(tc); // Look up and filter texel.

    frame_buffer[ wf.x + int(wf.y) * win_width ] = color * texel; }
```

- ☒ Should `use_eye` be ☒ `true` or ☐ `false`? ☒ Explain.

It should be `true` because distances in window-space coordinates are distorted by the perspective foreshortening done by the conversion to clip-space coordinates by the frustum transform. This can be seen in the two screenshots below, which show a ruler-like texture applied to a square formed by a white and yellow triangle. The left screenshot is the equivalent of `use_eye=true`, the right `use_eye=false`. In the left image the 0.5 mark is at the halfway point in eye space, which is what one would expect. In the right image the 0.5 mark is half way in window space which is wrong. One can verify that the 0.5 mark is at the half-way point in window space by using a real ruler (technically a scale) to measure the screenshot on a printed copy of this page.

- ☒ Show an image of the rectangle rendered with `use_eye=true` and ☒ with `use_eye=false`. ☒ Choose a view in which the two will appear different (but the same view for both settings of `use_eye`).



(c) Consider two scenes. Scene I has T triangles, Scene II has $3T$ triangles. Scene II was created by splitting each Scene I triangle into three triangles that cover the same area so that Scenes I and II appear to be identical.

- ☒ How would the time to render Scene II compare to the time to render Scene I on a typical rasterization pipeline (not the ray tracing pipeline below)? ☐ About the same, ☒ a little more time, or ☐ a lot more time. ☒ Explain.

In a typical rendering pass most of the time is spent in the fragment shader. The number of fragments in the two scenes are identical, so the total fragment shader time of each scene will be the same. Scene II has three times the number of vertices, so it will take three times as long to run the vertex shaders. If the fraction of time to run the vertex shaders in Scene 1 were small, say 5%, then Scene II would only take 10% longer.

Appearing below is our super-simple CPU-only ray-tracing code.

```
// Outer Loops (yw, xw): Iterate over each pixel.
for ( uint yw = 0; yw < win_height; yw++ )
  for ( uint xw = 0; xw < win_width; xw++ )
  {
    pCoor px_e = window_ll_e + window_dx_e * xw + window_dy_e * yw;
    pVect ray(px_e); // Ray From Eye to Pixel.

    float tmin = 1e10; // Distance to closest triangle found so far.

    // Inner Loop (it): Iterate over each triangle.
    for ( auto it = coors_es.begin(); it != coors_es.end(); ) {
      pCoor e0 = *it++, e1 = *it++, e2 = *it++; // Get Triangle

      // Find where ray intercepts plane defined by triangle.
      pVect v01(e0,e1), v02(e0,e2);
      pVect nt = cross(v01,v02);
      float t = dot( pVect(e0), nt ) / dot( ray, nt );
      /// Remaining code not shown.
    }
  }
}
```

- ☒ How would the time to render Scene II compare to the time to render Scene I on the ray tracing code above? ☐ About the same, ☐ a little more time, or ☒ a lot more time. ☒ Explain.

It would take a lot more time because our super-simple ray tracing code iterates over *all* triangles, and since Scene II has three times as many triangles it will take three times as long.

- ☒ Explain the difference between “real” ray-tracing code and the code above which makes the previous question about Scene I and Scene II unfair.

Real ray tracing code does not iterate through every triangle when casting a ray. It might place triangles into different boxes in a bounding volume hierarchy. Only triangles in the intersected boxes need be examined, which can greatly reduce the number of triangles to examine.

Problem 6: [15 pts] Answer each question below.

(a) Compute the dot product of vectors $\begin{bmatrix} 1 \\ 2 \\ 3 \end{bmatrix}$ and $\begin{bmatrix} 4 \\ 5 \\ 6 \end{bmatrix}$.

☒ $\begin{bmatrix} 1 \\ 2 \\ 3 \end{bmatrix} \cdot \begin{bmatrix} 4 \\ 5 \\ 6 \end{bmatrix} =$

The dot product is $4 + 10 + 18 = 32$.

(b) Compute a vector orthogonal to $\begin{bmatrix} 1 \\ 2 \\ 3 \end{bmatrix}$.

☒ A vector orthogonal to $\begin{bmatrix} 1 \\ 2 \\ 3 \end{bmatrix}$ is:

An orthogonal vector is $\begin{bmatrix} 0 \\ 3 \\ -2 \end{bmatrix}$.

(c) Write a parametric equation of a line, $S(t)$, that passes through points P_1 and P_2 and such that $S(0) = P_1$ and $S(1) = P_2$. Also for this line find an expression $t(x_1)$ which gives the value of t for which the x component of the point is x_1 .

☒ $S(t) =$

$S(t) = P_1 + t(P_2 - P_1).$

☒ $t(x_1) =$

$t(x_1) = \frac{x_1 - P_{0x}}{P_{1x} - P_{0x}}$, where P_{0x} is the x component of coordinate P_0 .

This page intentionally left blank.
I had many more questions, but it's only a 2-hour exam.
The reason that this page is blank, BTW, is to provide some privacy for the answer to the last question.