Name _____ *Formatted For Two-Sided Printing*

Staple This Side

GPU Programming
LSU EE 4702-1
Final Examination

Thursday, 11 December 2025 7:30-9:30 CST

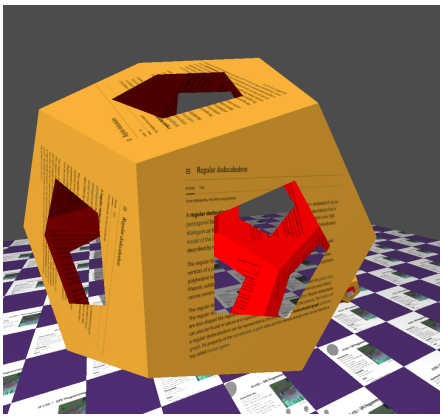
Problem 1	_____	(20 pts)
Problem 2	_____	(10 pts)
Problem 3	_____	(20 pts)
Problem 4	_____	(15 pts)
Problem 5	_____	(15 pts)
Problem 6	_____	(20 pts)
Exam Total	_____	(100 pts)

Alias _____

Good Luck!

Staple This Side

Problem 1: [20 pts] The screenshot to the right shows our dodecahedron with a pentagon-shaped hole in each face. On the facing page is a geometry shader (from the Homework 5 solution) that draws a dodecahedron face, with some code omitted. The `i` loop iterates over *pie slice* triangles. A triangle strip is used to iterate *each* of these triangles, and so the geometry shader emits $3 \times 5 = 15$ vertices.

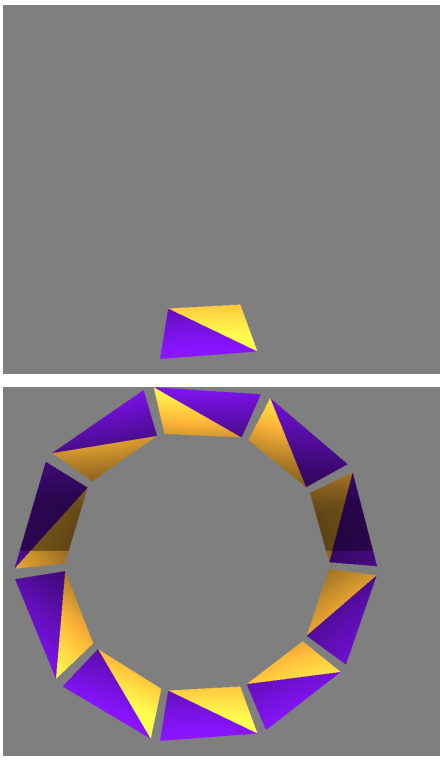


- [ ] Modify the geometry shader so that it draws the face with a pentagon-shaped hole.
- [ ] The size of the pentagon hole should be about half that of the face.
- [ ] Draw the face with a **single triangle strip for the whole face**, not one triangle strip per triangle.
- [ ] Update `max_vertices` to reflect the new number of vertices.
- [ ] Do not emit extra triangles, including zero-area triangles and overlapping triangles.
- [ ] Be sure that `EndPrimitive` is called at appropriate time(s).

Staple This Side

```
layout ( points ) in;
layout ( triangle_strip, max_vertices = 15 ) out;

void gs_main_fe_prob_1() {
  int dod_idx = gl_PrimitiveIDIn / 12,  face_idx = gl_PrimitiveIDIn % 12;
  vec4 color = buf_hw05_color[dod_idx];
  mat4 eye_from_local = eye_from_object * buf_hw05_matrix[dod_idx];
  vec4 dod_ctr_e = eye_from_local * vec4(0,0,0,1);

  // Retrieve pentagon vertex coordinates and transform them to eye space.
  vec4 pent_pts_e[5];
  for ( int i=0; i<5; i++ )
    pent_pts_e[i] = eye_from_local * buf_dod_coors_l[face_idx * 5+i];

  // Compute the pentagon normal and use as a local z axis.
  vec3 az = normalize( cross( vec3( pent_pts_e[2] - pent_pts_e[1] ),
                              vec3( pent_pts_e[0] - pent_pts_e[1] ) ) );

  // Compute the pentagon center.
  float edge_len = distance( pent_pts_e[1], pent_pts_e[2] );
  float ri_e = edge_len * 1.11351636441;
  vec4 pent_ctr_e = dod_ctr_e + vec4(ri_e * az,0);




  for ( int i=0; i<5; i++ )
    {
      vec4 pts[3] = { pent_ctr_e, pent_pts_e[i], pent_pts_e[(i+1)%5] };




      for ( int j=0; j<3; j++ )
        {
          Out.vertex_e = pts[j];
          gl_Position = clip_from_eye * Out.vertex_e;
          Out.color = color;
          Out.normal_e = az;
          Out.tcoor = m2 * pts[j];
          EmitVertex();
        }
      EndPrimitive();




    }



}
```

Problem 2: [10 pts] The code on the facing page emits the square shown in the upper screenshot in the first iteration of the loop. The coordinates `p0`, `p1`, `p2`, `p3` form a square, constructed using two triangles. After writing the buffer set the code applies transformation matrix `m` to these coordinates, preparing them for the next iteration. For the upper screenshot that matrix is identity, so there are `n_squares` squares in exactly the same place. In the lower screenshot the squares are rotated to form a ring.



The code writes vector `n` into the buffer set to use as the vertices' normal.

☐ Add code to compute `n` as many times as needed, but not more.

The code computes a value `h`. The *ring center* is a point located `h` units from the center of the square in the direction of the square normal. The *ring axis* is parallel to the vector from `p0` to `p2` and passes through the ring center. (The ring axis **does not** to through `p0` or `p2`.)

Modify the code to compute `m` so that it rotates the square around the ring axis. The rotation angle, `delta_theta`, is provided.

Matrix constructor examples are provided for your convenience.

☐ Compute `m` so that the square positions are computed as described.

☐ **Do not** compute `m` inside the `i` loop.

☐ Don't assume that `p0`, `p1`, `p2`, `p3` are in any particular orientation or location. All that's known is that they form a square.

Staple This Side

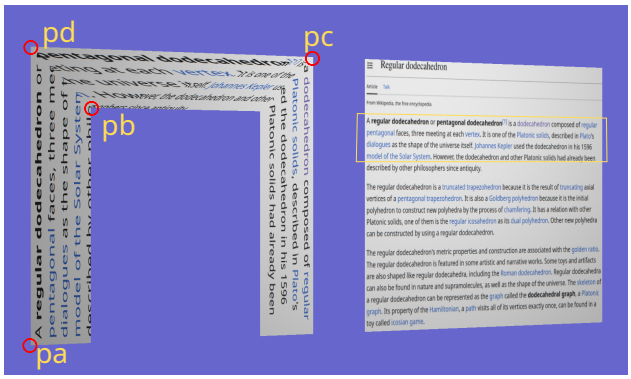
Staple This Side

Staple This Side

```
// Example Matrix Declarations -- Provided For Reference -- Can Abbreviate
pVect axis, t_amt;
float angle_radians, scale_factor;
pMatrix_Rotation mat_r( axis, angle_radians );
pMatrix_Translate mat_t( t_amt );
pMatrix_Scale mat_s( scale_factor );

//  ☐  Use p0, p1, p2, p3, and n_squares to compute m and n.

float delta_theta = 2 * numbers::pi / n_squares;
float h = 1.1 * pre_s / ( 2 * tan( numbers::pi / n_squares ) );









pNorm n =                                   ; //  ☐  Compute n.









pMatrix m =                                 ; //  ☐  Compute m.

for ( int i=0; i<n_squares; i++ )
  {
    // Don't modify code in this loop.
    bset_twisted << p0 << p1 << p2;
    bset_twisted << n << n << n;
    bset_twisted << color_purple << color_purple << color_purple;

    bset_twisted << p2 << p1 << p3;
    bset_twisted << n << n << n;
    bset_twisted << color_gold << color_gold << color_gold;

    p0 = m * p0;
    p1 = m * p1;
    p2 = m * p2;
    p3 = m * p3;
  }
```

Staple This Side

Problem 3: [20 pts] On the right is an arch-like figure and a square. The facing page shows code that will draw something, though it's not the arch. The location of the four coordinates, `pa`, `pb`, `pc`, and `pd`, are shown in the screenshot. They are all on the same plane (but don't assume a particular plane).



(*a*) Sketch the figure that will be drawn by the code at the top of the facing page.

☐ Sketch what will be drawn by code at top of next page.

(*b*) Insert points into `bset_plain` so that the arch is rendered, including the texture. The square on the right shows the entire texture, the yellow rectangle on the right shows the part of the texture appearing on the arch.

☐ Compute the additional coordinates needed to render the arch. ☐ All coordinates must be computed in terms of `pa`, `pb`, `pc`, and `pd`.

☐ Insert coordinates to draw the arch, remember to ☐ use a triangle strip topology.

☐ Insert texture coordinates so that the texture appears as shown. ☐ The yellow box on the right shows the portion of the texture that should appear on the arch.

Staple This Side

```
// Part (a)
// bset_plain is set to a Triangle Strip topology.

// Draws something. It's not the arch.
bset_plain << pa << pb << pc << pd;
bset_plain << pTCoor(0,0) << pTCoor(0,0) << pTCoor(0,0)<< pTCoor(0,0);



// Part (b)
// bset_plain is set to a Triangle Strip topology.

// Draw the arch. Use pT as an abbreviation for pTCoor.

bset_plain <<
```

Problem 4: [15 pts] On the facing page are excerpts from the Homework 5 solution.

(*a*) For the questions below consider the `HW5_Triangles` code, in which each face of the dodecahedron is rendered with three triangles.

☐ For the `HW5_Triangles` code how much data is sent to the GPU per vertex? (Per vertex shader invocation.)

☐ For the `HW5_Triangles` code how much data is sent to the GPU per dodecahedron?

(*b*) For the questions below consider the `HW5_Inst_Prob_2` code, in which the dodecahedron is rendered using an instanced draw. Consider data from both the buffer set and storage buffers.

☐ For the `HW5_Inst_Prob_2` code how much data is sent to the GPU per dodecahedron for the **first** frame. Let n denote the number of dodecahedra rendered per frame.

☐ For the `HW5_Inst_Prob_2` code how much data is sent to the GPU per dodecahedron for the second frame.
☐ Explain why this amount of data is different than for the first frame.

(*c*) The two code versions above render the same dodecahedron. (Ignore stellation.)

☐ For which version is the fragment shader invoked more often? ◯ *For* `HW5_Triangles` ◯ *for* `HW5_Inst_Prob_2`, or ◯ *it's a tie, they are invoked about the same number of times.* ☐ Explain.

(*d*) In an instanced draw there are n instances and v vertices (number of vertices in the buffer set).

☐ In terms of n and v how many times is the vertex shader invoked?

Staple This Side

```
case SO_HW5_Triangles:
  for ( Ball* b: balls ) {                                  // n iterations
      pMatrix m;                          // Code computing m omitted for brevity.
      for ( size_t i=0; i<buf_dod_coors_l.size(); i+=5 )  // Twelve iterations.
        // Render the pentagon by emitting three triangles.
        for ( int j=1; j<4; j++ )                           // Three iterations.
          bset_doda
            << buf_dod_coors_l[ i ]
            << buf_dod_coors_l[ i + j ]
            << buf_dod_coors_l[ i + j + 1 ];

      for ( int i=0; i<3*3*12; i++ ) bset_doda << b->color << m;
    }

  bset_doda.to_dev();
  pipe_doda.record_draw( cb, bset_doda );
  break;


case SO_HW5_Inst_Prob_2: {

    if ( bset_doda_inst_prob_2.size() == 0 ) {
        for ( int i=0; i<buf_dod_coors_l.size(); i += 5 )  // Twelve iterations.
          {
            pMatrix m;                  // Code computing m omitted for brevity.
            vector<pCoor> coors;
            for ( int j=0; j<5; j++ )
              coors << pent_ctr_l
                    << buf_dod_coors_l[ i + j ]
                    << buf_dod_coors_l[ i + ( j + 1 ) % 5 ];

            for ( pCoor c: coors )                          // Fifteen iterations.
              bset_doda_inst_prob_2 << c << (m * c).tcoor();
          }
        bset_doda_inst_prob_2.to_dev();
      }

    buf_hw05_color.clear(); buf_hw05_matrix.clear();
    for ( Ball* b: balls )  {                               // n iterations.
        pMatrix m;                 // Code computing m omitted for brevity.
        buf_hw05_color << b->color;
        buf_hw05_matrix << m;
      }

    buf_hw05_color.to_dev(); buf_hw05_matrix.to_dev();
    const int n_instances = buf_hw05_matrix.size();
    pipe_doda_inst_prob_2.record_draw_instanced( cb, bset_doda_inst_prob_2, n_instances );
```

Problem 5: [15 pts] Answer the following ray tracing questions.

(*a*) The same scene (including the same eye) is rendered using either rasterization or ray tracing. (We do it all the time.) When using rasterization the fragment shader is invoked for fragment F_{600} of triangle T_6 and F_{700} of T_7, both for pixel $(66, 77)$. With ray tracing in a cast from the eye to pixel $(66, 77)$ the closest hit shader is invoked for F_{600} of T_6 but not for F_{700}. *Note: the portion about pixel* $(66, 77)$ *was omitted from the original exam.*

☐ Normally, why wouldn't the closest hit shader have been invoked for F_{700}?

☐ Describe a scene in which the fragment shader under rasterization is invoked many more times than the closest hit shader for ray tracing. ☐ Remember that the same scene is used for both.

(*b*) Explain how a closest hit shader can determine whether its location is illuminated by a light or the light is blocked by something.

☐ How can the shader determine if its in a shadow? ☐ Explain the role of `traceNV`.

(*c*) The simplified fragment shader below shows a round hole in the pentagon rather than a yellow dot. It works correctly.

```
void fs_main_exam() {
  vec4 our_color = gl_FrontFacing ? In.color : vec4(1,0,0,1);
  if ( opt_hw05_dot && length(In.tcoor) < 0.5 ) { discard; return; }
  vec4 texel = texture(tex_unit_0,tc);
  vec4 lighted_color = generic_lighting( In.vertex_e, our_color, In.normal_e );
  out_frag_color = lighted_color * texel;
}
```

The closest-hit shader below tries to do the same for ray tracing. That is, rather than changing the color to gold it just returns. (If it didn't return there it would show the gold dot.)

Staple This Side

Staple This Side

```
void main_dod() {
  const uint ii = gl_InstanceCustomIndexNV;
  RT_Uni_Per_Instance upi = uni_per_instance_a[ii];
  //  ..Code skipped..
  vec2 ctr_rel = m1 * vertex_g;
  bool gold_circ = length(ctr_rel) < 0.5;
  vec4 color = gold_circ    ? vec4(1,1,0,1) : front_facing ? color_front   : color_back;
  if ( gold_circ ) return;  // Won't result in a hole.
```

☐ If the shader returns what will be shown where the hole is supposed to be?

(*d*) The code below is an excerpt from the closest hit shader. It is trying to use `traceNV` to implement the hole.

```
bool gold_circ = length(ctr_rel) < 0.5;
vec4 color = gold_circ    ? vec4(1,1,0,1) : front_facing ? color_front   : color_back;
if ( gold_circ ) {
    // Do something.
    traceNV
      ( topLevelAS, gl_RayFlagsOpaqueNV,
        0xff, 0, 0, 0, // sbtRecordOffset, sbtRecordStride, missIndex
        ray_origin,
        tmin,
        ray_direction,
        tmax, 0 /*payload location*/);
    // Do something.
    return;
  }
```

☐ How can `traceNV` be used so that the hole looks like a hole (though not perfectly)? ☐ Explain where the ray is cast and what to do with the ray payload. ☐ There is no need to show working code.

☐ Explain why an intersection shader could be used to implement the hole correctly. ☐ Explain why shadows would work with the intersection shader but not the `traceNV` call for the gold dot situation.

Staple This Side

Problem 6: [20 pts] Answer each question below.

(*a*) Provide the number of multiplications needed for each of the operations below.

☐ How many multiplies are needed to compute a dot product of two 3-element vectors?

☐ How many multiplies are needed to compute a cross product of two 3-element vectors?

☐ How many multiplies are needed to compute the product of a 4×4 matrix and a homogeneous coordinate?

(*b*) Consider two vectors, v_1 and v_2.

☐ Show how to compute the angle between v_1 and v_2 using a dot product. (There is no need to show the dot product operations.)

☐ Show how to compute the angle between v_1 and v_2 using a cross product. (There is no need to show the dot product operations.)

Staple This Side

(*c*) Show how the Vulkan rendering pipeline is organized. Show where the fragment shader, frame buffer update, geometry shader, rasterizer, and vertex shader are.

☐ Sketch the pipeline including all the stages above. ☐ Be sure that the stages are in the correct order.

(*d*) Textures have MIPMAP levels.

☐ Explain how the MIPMAP levels are created from a texture image.

Staple This Side