This document contains assignments given in LSU GPU Programming over many semesters. It was automatically generated and so some solutions (and even some assignments) are possibly missing. At the top of each page of each assignment is a link to the original assignment. Those who want to print an assignment might follow that link. All assignments and public solutions are available at `https://www.ece.lsu.edu/gpup/prev.html`.

Contents

1 Fall 2025 **4**
- 1.1 mt.pdf . . . 5
- 1.2 fe.pdf . . . 12

2 Fall 2024 **25**
- 2.1 mt.pdf . . . 26
- 2.2 fe.pdf . . . 35

3 Fall 2023 **46**
- 3.1 mt.pdf . . . 47
- 3.2 fe.pdf . . . 54

4 Fall 2022 **65**
- 4.1 mt.pdf . . . 66
- 4.2 fe.pdf . . . 74

5 Fall 2021 **84**
- 5.1 mt.pdf . . . 85
- 5.2 fe.pdf . . . 98

6 Fall 2020 **108**
- 6.1 mt.pdf . . . 109
- 6.2 fe.pdf . . . 118

7 Fall 2019 **127**
- 7.1 mt.pdf . . . 128
- 7.2 fe.pdf . . . 137

8 Fall 2018 **149**
- 8.1 mt.pdf . . . 150
- 8.2 fe.pdf . . . 161

9 Fall 2017 **171**
- 9.1 mt.pdf . . . 172
- 9.2 fe.pdf . . . 179

10 Fall 2016 **188**
10.1 mt.pdf . . . 189
10.2 fe.pdf . . . 196

11 Fall 2015 **204**
11.1 mt.pdf . . . 205
11.2 fe.pdf . . . 213

12 Fall 2014 **222**
12.1 mt.pdf . . . 223
12.2 fe.pdf . . . 230

13 Fall 2013 **240**
13.1 mt.pdf . . . 241
13.2 fe.pdf . . . 250

14 Fall 2012 **258**
14.1 mt.pdf . . . 259
14.2 fe.pdf . . . 265

15 Fall 2011 **272**
15.1 mt.pdf . . . 273
15.2 fe.pdf . . . 279

16 Fall 2009 **287**
16.1 mt.pdf . . . 288
16.2 fe.pdf . . . 293

17 Fall 2025 Solutions **302**
17.1 mt sol.pdf . . . 303

18 Fall 2024 Solutions **311**
18.1 mt sol.pdf . . . 312
18.2 fe sol.pdf . . . 321

19 Fall 2023 Solutions **332**
19.1 mt sol.pdf . . . 333
19.2 fe sol.pdf . . . 341

20 Fall 2022 Solutions **352**
20.1 mt sol.pdf . . . 353
20.2 fe sol.pdf . . . 362

21 Fall 2021 Solutions **374**
21.1 mt sol.pdf . . . 375
21.2 fe sol.pdf . . . 389

22 Fall 2020 Solutions **401**
22.1 mt sol.pdf 402
22.2 fe sol.pdf 414

23 Fall 2019 Solutions **425**
23.1 mt sol.pdf 426

24 Fall 2018 Solutions **437**
24.1 mt sol.pdf 438
24.2 fe sol.pdf 457

25 Fall 2017 Solutions **467**
25.1 mt sol.pdf 468
25.2 fe sol.pdf 478

26 Fall 2016 Solutions **489**
26.1 mt sol.pdf 490
26.2 fe sol.pdf 499

27 Fall 2015 Solutions **509**
27.1 mt sol.pdf 510
27.2 fe sol.pdf 519

28 Fall 2014 Solutions **531**
28.1 mt sol.pdf 532
28.2 fe sol.pdf 540

29 Fall 2013 Solutions **552**
29.1 mt sol.pdf 553
29.2 fe sol.pdf 564

30 Fall 2012 Solutions **576**
30.1 mt sol.pdf 577
30.2 fe sol.pdf 585

31 Fall 2011 Solutions **593**
31.1 mt sol.pdf 594

32 Fall 2009 Solutions **601**
32.1 mt sol.pdf 602

1 Fall 2025

Name _____ *Formatted For Two-Sided Printing*

Staple This Side

GPU Programming
EE 4702-1
Midterm Examination

Friday, 31 October 2025, 9:30-10:20 CDT

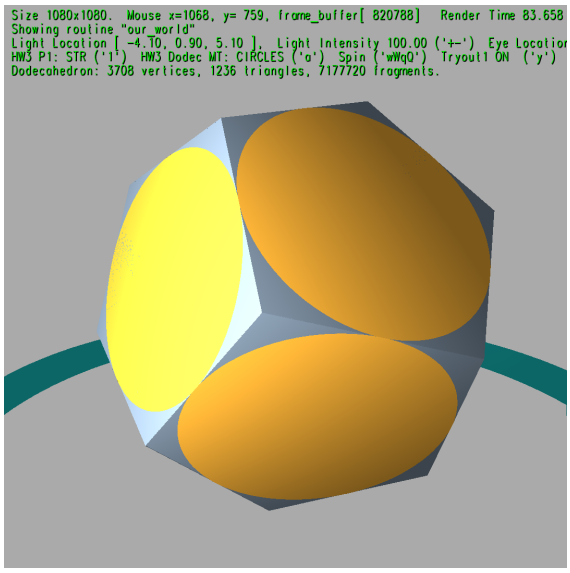
Problem 1	_____	(35 pts)
Problem 2	_____	(15 pts)
Problem 3	_____	(15 pts)
Problem 4	_____	(20 pts)
Problem 5	_____	(15 pts)
Exam Total	_____	(100 pts)

Alias _____

Good Luck!

Staple This Side

Problem 1: [35 pts] Appearing to the right is a dodecahedron with gold circles just above its faces. The code below, based on the Homework 3 solution, draws the dodecahedron but the circle code is incomplete. Complete it.



- ☐ Several quantities are computed for you, such as the pentagon center and the dodecahedron center.
- ☐ Approximate the circle as an `n_sides` sided polygon.
- ☐ Use `ri` for the radius of the circle.
- ☐ The circle plane (circle local `ax`,`ay`) should be parallel to the pentagon plane.
- ☐ Place the circle center on the line from the dodecahedron center, `dod_ctr`, to the pentagon center, `pen_ctr`, a very small distance away from pentagon. See `vcc`.
- ☐ Just emit circle coordinates, **don't emit colors**.

```
case MT_Circles: {
  pCoor dod_ctr = cyl_center + pVect(0,1.5,0);          // Dodecahedron Center
  float ri = len_edge / ( 2 * tanf(numbers::pi/5) );    // Circle Radius
  int n_sides = 100;                                    // Number of "circle" sides
  float delta_theta = 2 * numbers::pi / n_sides;

  for ( size_t i=0; i<coors.size(); i+=5 ) {
    for (int j=1; j<4; j++) coors_os<<coors[i]<<coors[i+j]<<coors[i+j+1]; // Emit pentagon.
    pVect vi(0,0,0); for ( int j=1; j<5; j++ ) vi += pVect(coors[i],coors[i+j]);

    // Dodecahedron Face (Pentagon) Coords in coors[i] to coors[i+4];
    pCoor pen_ctr = coors[i] + 0.2 * vi;                // Pentagon Center
    pVect vcc( dod_ctr, pen_ctr );                      // Should be useful.




    for ( int j =    ; j  <   n_sides;  j++ ) { //  ☐ Finish this line.
      float theta = j * delta_theta;




      coors_os <<
  }}
  gc.topology_set(Topology_Individual);
```


Staple This Side
Staple This Side

Staple This Side

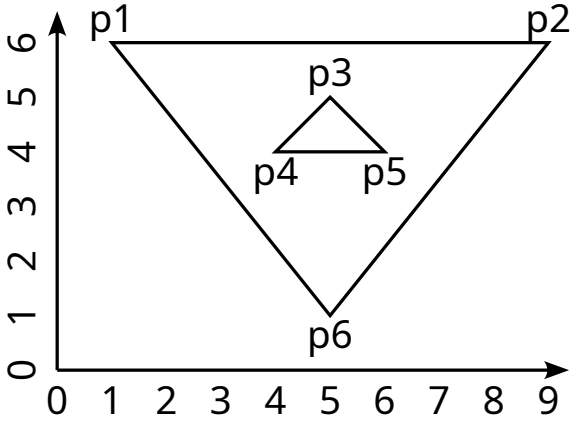
Problem 2: [15 pts] Below are two unsuccessful attempts to render a triangle-in-a-triangle shape using the `Our_3D` demo code from class, part (c) is your shot at success.

(*a*) The code below inserts vertices for an **individual triangle topology**.

☐ On the illustration to the right show the triangles that **actually are** rendered. ☐ Label each triangle with a letter (a, b, c, ...).

```
// Pipeline set up for individual triangles.
// (Vulkan: PrimitiveTopology::eTriangleList).

coors_os << p1 << p2 << p3 << p4 << p5 << p6;
```

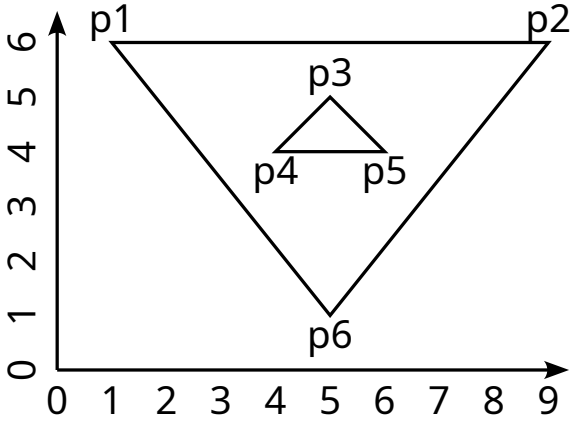


(*b*) The code below inserts vertices for a **triangle strip topology**.

☐ On the illustration to the right show the triangles that **actually are** rendered. ☐ Label each triangle with a letter (a, b, c, ...).

```
// Pipeline set up for a triangles strip.
// (Vulkan: PrimitiveTopology::eTriangleStrip).

coors_os << p1 << p2 << p3 << p4 << p5 << p6;
```



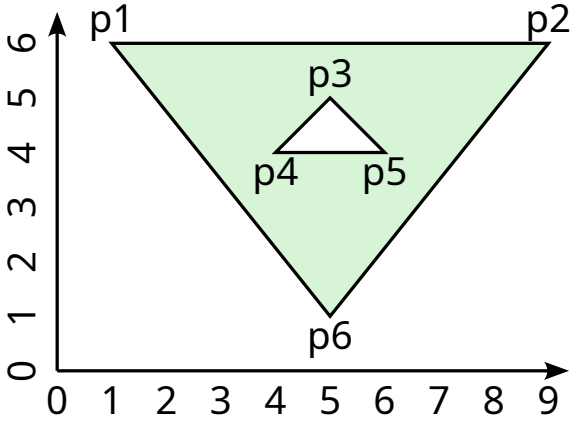
(*c*) The code below is incomplete.

☐ Complete the code below so that green shaded area shown is rendered correctly, ☐ and without overlapping triangles, for a pipeline set for a triangle strip topology.

☐ Coordinates `p1` through `p6` are already defined correctly, don't assign values to them.

```
// Pipeline set up for a triangles strip.
// (PrimitiveTopology::eTriangleStrip).

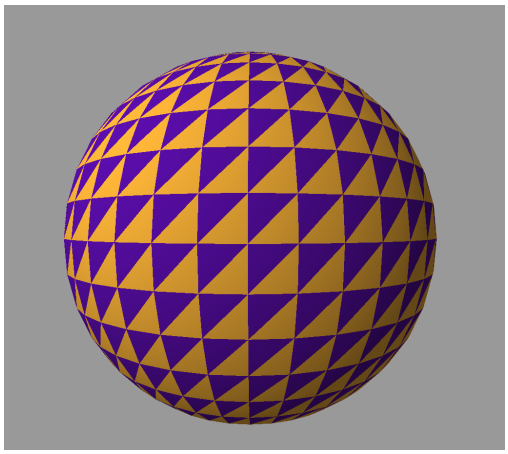
coors_os <<
```



Staple This Side

Problem 3: [15 pts] Answer the following questions about spheres rendered using rasterization.

(*a*) The sphere shown to the right consists of $T = 1600$ triangles. (Not all are visible.)



Original Sphere

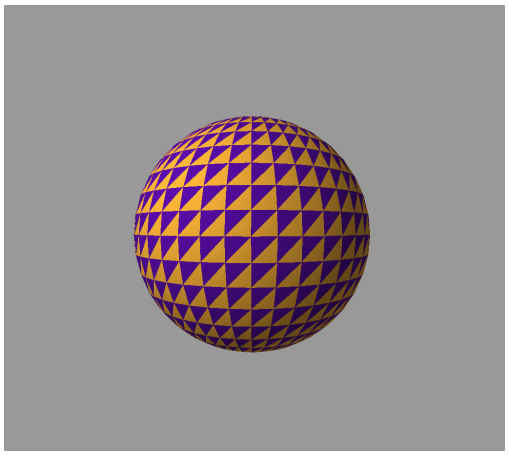
☐ How many vertices would be needed to render the sphere using a individual triangles (a triangle list)? ☐ Answer in terms of T.

☐ How many vertices would be needed to render the sphere using a triangle strip? ☐ Answer in terms of T.

To render the original sphere using a triangle list about $F = 500,000$ fragments were generated.

☐ About how many fragments would be generated if the sphere were rendered using a triangle strip? ☐ Answer in terms of F.

(*b*) Suppose for the original sphere above the eye were at distance d from the nearest point on the surface of the sphere. Let V_o denote the number of vertices used to render the original sphere using triangle strips and F_o denote the number of fragments generated rendering the original sphere. In the otherwise identical *far sphere* to the right the eye is at distance $2d$ from the nearest point. Both the original and far spheres have T triangles. *Note: In the original exam F_o was not mentioned, and it was not clear whether V should denote the number of vertices in the near or far sphere.*



Far Sphere

☐ How many vertices in the far sphere? ☐ Answer in terms of T, V_o, F_o, and d.

☐ **Approximately** how many fragments in the far sphere? (There is not enough information to answer this question exactly, don't try.) ☐ Answer in terms of T, V_o, F_o, and d.

Staple This Side

Staple This Side

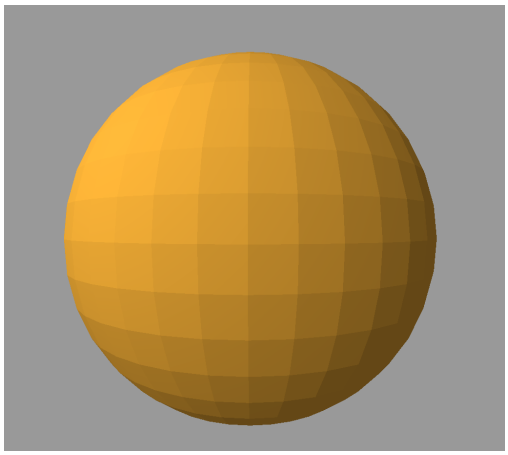
Staple This Side

(*c*) The **only difference** between the two spheres below on the right is in the normal associated with each vertex. The normals for the smooth sphere were chosen to make it look more like a sphere.

☐ Complete the code below so that it inserts normals that will result in the faceted sphere.

```
// Center of sphere at sp_ctr_os;

// Coordinates of a triangle approximating
// the surface of a sphere.
//
coors_os << p1 << p2 << p3;
```



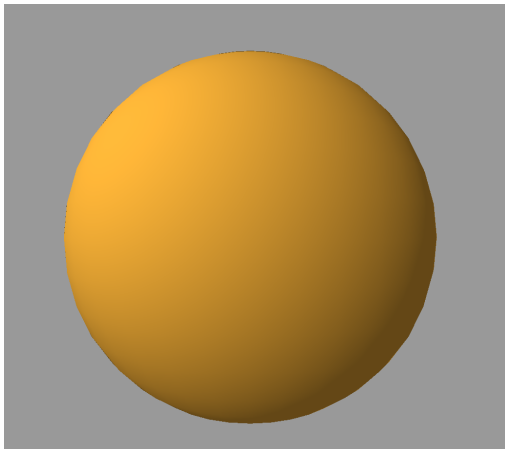
Faceted Sphere

```
normals_os <<
```

☐ Complete the code below so that it inserts normals that will result in the smooth sphere.

```
// Center of sphere at sp_ctr_os;

// Coordinates of a triangle approximating
// the surface of a sphere.
//
coors_os << p1 << p2 << p3;
```



Smooth Sphere

```
normals_os <<
```

Staple This Side

Problem 4: [20 pts] Answer each question below.

(*a*) Answer each question about the amount of computation needed for common operations. All vectors have three components and all coordinates have four components.

☐ How many multiplies are needed to compute a dot product such as `dot(v1,v2)`?

☐ How many multiplies are needed to compute a cross product such as `cross(v1,v2)`?

☐ How many multiplies are needed to perform a coordinate transformation `m * p1` , where `m` is a transformation matrix and `p1` is a coordinate?

☐ How many multiplies are needed to compute a matrix product `m1 * m2` where `m1` and `m2` are transformation matrices?

(*b*) Given coordinate P and vector v, we can easily compute $Q = P + v$. Show a transformation matrix, $\mathbf{M}$, that achieves the same result by computing a matrix/homogeneous coordinate product, $Q = \mathbf{M}P$. (For example, with $v = (v_x, v_y, v_z) = (1, 2, 3)$ and $P = (10, 20, 30, 1)$ we would compute $Q = (11, 22, 33, 1)$).

☐ Show the transformation matrix:

☐ Describe a situation in which it would make more sense to compute $Q = P + v$ rather than $Q = \mathbf{M}P$.

☐ Describe a situation in which it would make more sense to use a translation matrix rather than just adding a vector. *Hint: we do it all the time in our code.*

Staple This Side

Staple This Side

Staple This Side

Problem 5: [15 pts] Answer the questions below.

(*a*) The code below inserts the coordinates and colors for a cylinder:

```
for ( int i=0; i<=n_segs; i++ ) {
    // Some code omitted.
    coors_os << c + cyl_axis << c;
    colors   << color_cyan   << color_cyan;
  }
```

☐ What would the cylinder look like if the frame buffer were written with the color `color_cyan`?

☐ What should the frame buffer be written with, if not exactly `color_cyan`?

(*b*) The `Our_3D` C++ class in the course CPU-only code is modeled very loosely on a Vulkan implementation, and `World` is modeled on something like an application (such as a game) a developer would write.

☐ Who writes Vulkan and OpenGL implementations? ☐ Why do most of them give away the implementations for free?

(*c*) Rasterization and ray tracing in `Our_3D` differ in what their outer loops iterate over.

☐ What kind of an object does the outer loop of `Our_3D` rasterization code iterate over?

☐ What kind of an object does the outer loop of `Our_3D` ray tracing code iterate over?

Staple This Side

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

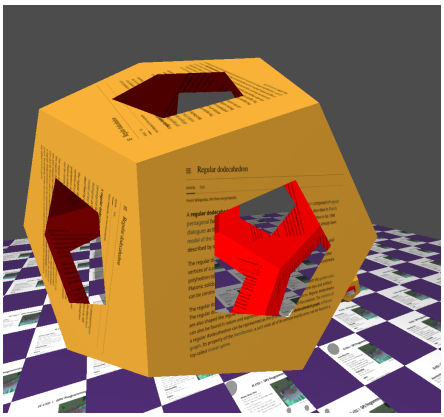
Problem 1	_____	(20 pts)
Problem 2	_____	(10 pts)
Problem 3	_____	(20 pts)
Problem 4	_____	(15 pts)
Problem 5	_____	(15 pts)
Problem 6	_____	(20 pts)
Exam Total	_____	(100 pts)

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

Staple This Side

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();




    }

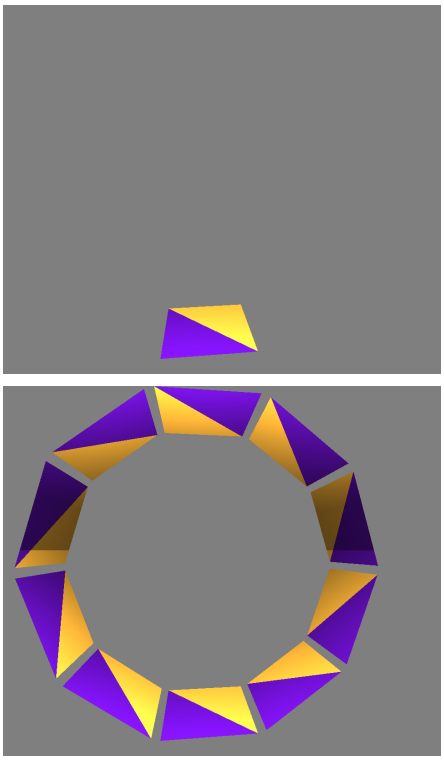


}
```

Staple This Side

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.

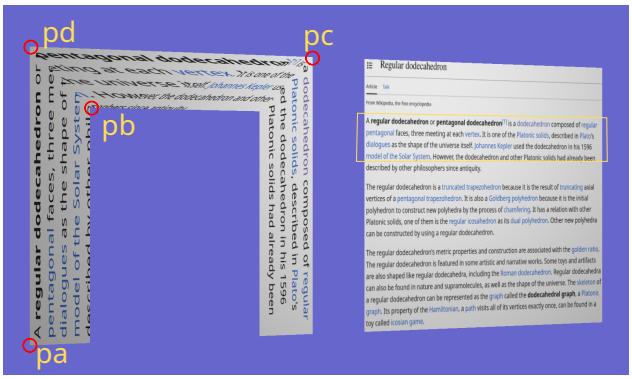
for ( int i=0; i<n_squares; i++ )
  {
    // Don't modify code in this loop.
    bset_twisted << p0 << p1 << p2;
    bset_twisted << n << n << n;
    bset_twisted << color_purple << color_purple << color_purple;

    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

Staple This Side

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 <<
```

Staple This Side

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

Staple This Side

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 );
```

Staple This Side

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

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

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

Staple This Side

2 Fall 2024

Name _____ *Formatted For Two-Sided Printing*

Staple This Side

GPU Programming
EE 4702-1
Midterm Exam

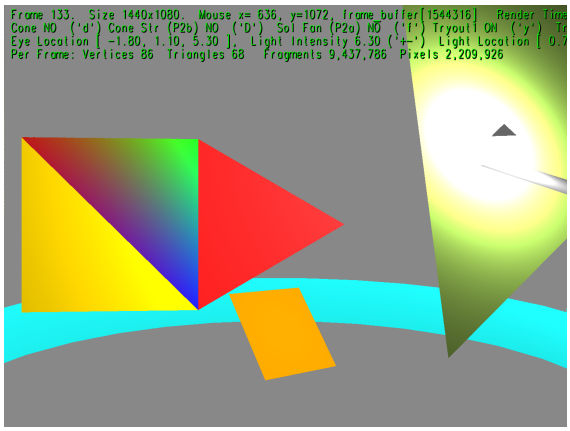
25 October 2024, 9:30-10:20 CDT

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

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;
```

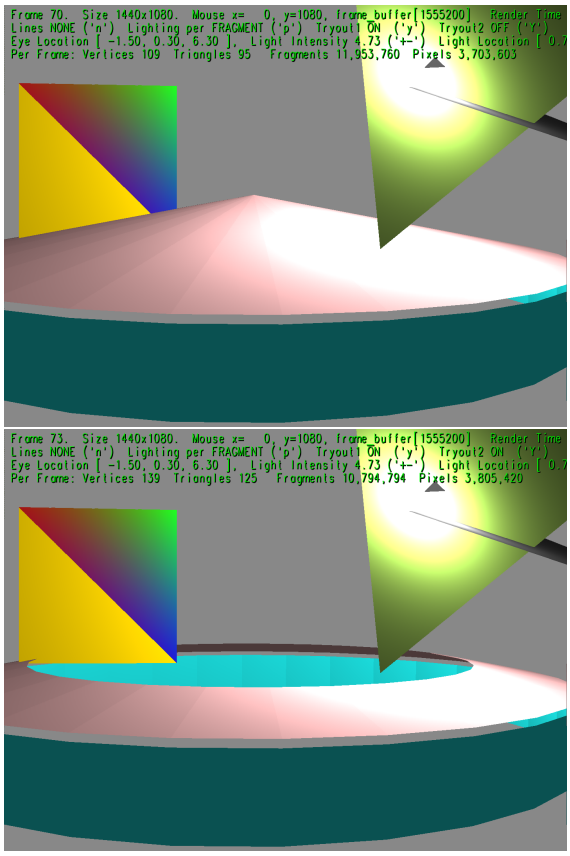
☐ Add code to the loop body below so that it inserts triangle normals into container `norms_os`.

```
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

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

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
Staple This Side

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

Name _____ *Formatted For Two-Sided Printing*

Staple This Side

GPU Programming
LSU EE 4702-1
Final Examination

Thursday, 12 December 2024 7:30-9:30 CST

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

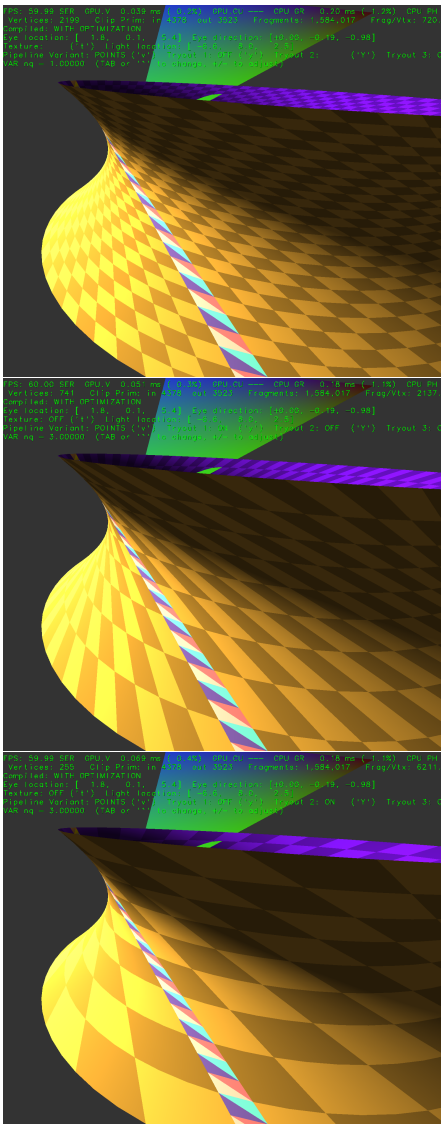
Alias _____

Good Luck!

Staple This Side

Staple This Side

Problem 1: [20 pts] The screenshots to the right show our hyperboloid rendered using code based on the Homework 5 solution. The top screenshot is from code nearly the same as the solution (other than the checkered appearance). Each invocation of the geometry shader (shown on the facing page) draws one *quad*. The lower-left of a quad, is at level `level0` and side `side0`, the upper right at level `level0+1` and side `side0+1` using variable names from the code. So the width of the quad is 1 and the height is 1. In the middle screenshot the height is 3 (level `level0` to `level0+3`), and width is 1, while in the lower screenshot both the width and height are 3. The 3×3 quad consists of $2 \times 3 \times 3 = 18$ triangles, it is **not** two big triangles.



On the facing page is the geometry shader code used by the points grouping (topology). Modify the shader code to draw quads of width `nq` and height `nq`, where `nq` is a variable. The CPU will record the draw with $\lceil n_s/n_q \rceil \lceil n_l/n_q \rceil$ vertices, where n_q is `nq`, n_l is `n_levels`, and n_s is `n_sides`. The unmodified code works correctly for $n_q = 1$.

- ☐ Correctly set `max_vertices`. It is okay to show an expression in terms of `nq`.
- ☐ Correctly set `level0` and `side0` in the shader so that they specify the lower-left of the quad that the shader will draw.
- ☐ Modify the code so that it draws a quad of width and height `nq`.
  - ☐ Don't just emit two big triangles.
- ☐ Emit fewer than $6n_q^2$ vertices.
- ☐ Do not emit extra triangles, including zero-area triangles and triangle vertices that are not on the hyperboloid surface.
- ☐ Don't assume that n_q is a factor of either n_s or n_l. That is, don't assume $\lceil \frac{n_l}{n_q} \rceil n_q = n_l$ nor $\lceil \frac{n_s}{n_q} \rceil n_q = n_s$.
- ☐ Be sure that `EndPrimitive` is called at appropriate times.

Staple This Side

Staple This Side

```
layout ( triangle_strip, max_vertices = 4 ) out;   // ☐ Modify max_vertices.

void gs_points() {


  int level0 = gl_PrimitiveIDIn % uc.n_levels;


  int side0 = gl_PrimitiveIDIn / uc.n_levels;


  for ( int dl = 0; dl < 2; dl++ )
    {


      int level = level0 + dl;


      for ( int ds = 0; ds < 2; ds ++ )
        {
          int side = side0 + ds;


          int idx = level + side * uc.s_levels;


          // Hint: Should not need to change code from here to EmitVertex.
          // Retrieve the coordinate and normal.
          vec4 p = hyperb_coords[ idx ];
          vec4 n = hyperb_norms[ idx ];

          // Convert coordinate spaces.
          gl_Position = ut.clip_from_object * p;
          Out.vertex_e = ut.eye_from_object * p;
          Out.normal_e = normalize( mat3(ut.eye_from_object) * n.xyz );
          // Don't worry about color index and texture coordinates.
          EmitVertex(); // Hint: Should not need to change this line.
        }


    }

  EndPrimitive();
}
```

Staple This Side

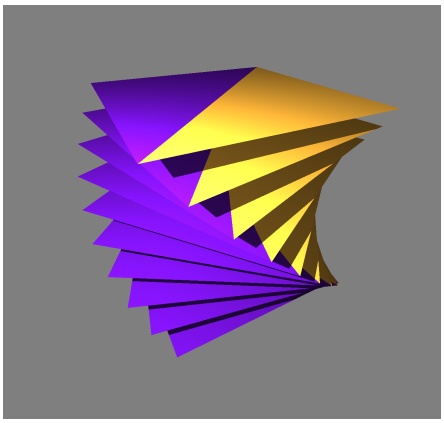
Problem 2: [15 pts] The code on the facing page emits the square shown in the upper screenshot in the first iteration of the loop. The coordinates `p0`, `p1`, `p2`, `p3` form a square, constructed using two triangles. After writing the buffer set the code applies transformation matrix `m` to these coordinates, 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.

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.

Let l be the length of a side of the square, and let q be the value of `n_squares`. Modify the code so that `m` rotates the square $\frac{\pi}{2(q-1)}$ radians around its center using the square's normal for the axis of rotation, and so that it moves the square along its normal by a distance of $\frac{l}{q-1}$, assume $q > 1$. See the lower screenshot. 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.
```

```
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: [15 pts] On the facing page is the CPU code from Homework 5 showing a buffer set being prepared for an individual triangle grouping and a triangle strip grouping, followed by the geometry shader. The geometry shader (at the bottom of the page) uses color index 0 to detect the start of a triangle strip. For the questions below answer per frame.

☐ For the individual triangle grouping compute the number of bytes sent ☐ per vertex. ☐ State any assumptions about the size of data types. ☐ That's per vertex, not per draw.

☐ For the individual triangle grouping how many times will the vertex shader be invoked per draw (frame)?

☐ For the individual triangle (triangle list) grouping how many times will the geometry shader be invoked per draw (frame)? ☐ How many triangles will be sent to the rasterizer per draw (frame)?

☐ For the triangle strip grouping how many times will the vertex shader be invoked per draw (frame)? ☐ Pay attention to the loop bounds.

☐ For the triangle strip grouping how many times will the geometry shader be invoked per draw (frame)? ☐ Don't forget color index 0. ☐ How many triangles will be sent to the rasterizer per draw (frame)?

☐ How much better is the code using the triangle strip grouping than the code using individual triangles at reducing the number of times the same vertex is sent to the rendering pipeline? ☐ Is there room for further reductions in this redundancy? ☐ Explain.

Staple This Side

Staple This Side

Staple This Side

```
    switch ( pipeline_variant ) {  // CPU Code

    case PV_Individ:
      for ( int side = 0; side < n_sides; side++ )
        for ( int lev = 0; lev < n_levels; lev++ ) {
            pCoor p00 = coords[ idxf( lev,   side ) ],  p01 = coords[ idxf( lev,   side+1 ) ];
            pCoor p10 = coords[ idxf( lev+1, side ) ],  p11 = coords[ idxf( lev+1, side+1 ) ];
            pNorm n00 = norms[ idxf( lev,   side ) ],   n01 = norms[ idxf( lev,   side+1 ) ];
            pNorm n10 = norms[ idxf( lev+1, side ) ],   n11 = norms[ idxf( lev+1, side+1 ) ];

            bset_hyperb << p00 << p01 << p10;
            bset_hyperb << pTCoor(0,0) << pTCoor(0,1) << pTCoor(1,0);
            bset_hyperb << n00 << n01 << n10;

            bset_hyperb << p01 << p11 << p10;
            bset_hyperb << pTCoor(0,1) << pTCoor(1,1) << pTCoor(1,0);
            bset_hyperb << n01 << n11 << n10;

            int c1 = side==0 ? ( lev%2 ? 5 : 2 ) : 1,  c2 = side==0 ? ( lev%2 ? 3 : 4 ) : 1;
            bset_hyperb << c1 << c1 << c1 << c2 << c2 << c2;
          }
      break;

    case PV_Strip:
      for ( int side = 0; side < n_sides; side++ )
        for ( int lev = 0; lev <= n_levels; lev++ ) {
            pCoor p00 = coords[ idxf( lev, side ) ],  p01 = coords[ idxf( lev, side + 1 ) ];
            pNorm n00 = norms[ idxf( lev, side ) ],  n01 = norms[ idxf( lev, side + 1 ) ];

            bset_hyperb << p00 << p01;
            bset_hyperb << pTCoor(0,lev%2) << pTCoor(1,lev%2);
            bset_hyperb << n00 << n01;

            int c1 = lev == 0 ? 0 : side == 0 ? ( (lev+1)%2 ? 5 : 2 ) : 1;
            int c2 = lev == 0 ? 0 : side == 0 ? ( (lev+1)%2 ? 3 : 4 ) : 1;
            bset_hyperb << c1 << c2;
          }
      break;


void gs_main_clean() {   // Geometry Shader
  int i_last = 2 - gl_PrimitiveIDIn % 2;
  if ( In[i_last].color_idx == 0 ) return;
  for ( int i=0; i<3; i++ ) {
      gl_Position = In[i].vertex_c;
      Out.color_idx = In[i_last].color_idx;
      Out.vertex_e = In[i].vertex_e;
      Out.normal_e = In[i].normal_e;
      Out.tcoor = In[i].tcoor;
      EmitVertex();
    }
  EndPrimitive();  }
```

Staple This Side

Problem 4: [20 pts] Appearing on the facing page is an excerpt from the ray generation shader used in class, followed by the miss shader.

(*a*) The ray generation shader does not write the frame buffer if the ray does not intersect geometry. (The frame buffer is initialized with a background color before ray tracing starts.) Modify the code so that if a ray does not intersect geometry the frame buffer is written with `color_tan` (thus making initialization unnecessary). There are several ways to do this, do it in the most efficient way possible.

☐ Write the frame buffer with `color_tan` if a ray misses ☐ efficiently. (Efficiently means with as little extra code as possible, or even with less code than is here now.)

(*b*) The code sets `tmin` to `1`, which is correct, but it sets `tmax=10000`, which works but probably does not match the far plane from `ut.object_from_clip`.

☐ Explain what `tmin` and `tmax` do.

☐ Explain why increasing `tmax` would increase the execution time of real ray tracing code ☐ and why it would not have much of an impact on the course CPU-only ray tracing code. Assume that both codes render the same scene and do so correctly and that the application programmer is satisfied with the results (other than execution time).

☐ Set `tmax` to the correct value based on `ut.object_from_clip`.

Hint: Pay attention to how `pixel_c` *and* `pixel_g` *are computed and think about clip space. It may be helpful to use function* `length(MYVECTOR)` *to find the length of a vector.*

Staple This Side

Staple This Side

Staple This Side

```
void main() {
  // Code above omitted for brevity.

  // Prepare clip-space coordinate of pixel on near plane ..
  vec4 pixel_c = vec4( pixel_c2d.x, pixel_c2d.y,  0,  1 );
  // .. and convert to global space.
  vec4 pixel_g = homogenize( ut.object_from_clip * pixel_c );




  // Compute global- (object-) space coordinates of eye.
  //
  vec4 eye_e = vec4(0,0,0,1);
  vec4 eye_g = ut.object_from_eye * eye_e;




  // Compute vector from eye to pixel.
  //
  vec3 eye_to_pixel_g = pixel_g.xyz - eye_g.xyz;
  //
  // This is the ray to be cast by this shader.




  float tmax = 10000.0;  // This does not match far plane.
  float tmin = 1;

  rp_color = vec4(0);

  traceNV
   ( topLevelAS,      // Acceleration Structure: Holds optimized geometry info.
     gl_RayFlagsOpaqueNV, 0xff, // Types of objects to consider (or ignore).
     0, 0, 0,         // Specify which sets of shaders to call.
     eye_g.xyz,       // Ray Origin
     tmin,
     eye_to_pixel_g,  // Ray Direction.
     tmax,
     0                // Payload Location.
     );

  if ( rp_color.a == 0 ) return;

  imageStore(fb_image, ivec2(gl_LaunchIDNV.xy), vec4(rp_color.rgb, 0.0));
}
#endif

#ifdef _RT_MISS_EYE_GEO_
layout ( location = 0 ) rayPayloadInNV vec4 rp_color;
void main(){}
#endif
```

Staple This Side

Problem 5: [10 pts] A retroreflective coating reflects light back towards its source. They are often used on highway signs to reflect the light from a car's headlights back towards the car.

The fragment shader below sets `retro` to `true` if the material property is retroreflective. Note that `vertex_e` and `uni_light.position` are in eye space.

☐ Modify the fragment shader so that when `retro` is true the lighted color accounts for the retroreflective coating taking into account the location of the ☐ light, ☐ vertex, and ☐ eye. It is okay to assume the coating reflects light *exactly* to its source, even though that would not help a driver.

```
void fs_retro() {
  vec4 texel = texture(tex_unit_0,tcoor);
  bool retro = uc.retroreflective[color_idx];

  // Vector from fragment to light.  Note: light position is in eye space.
  vec3 vec_vl = uni_light.position.xyz - vertex_e.xyz;

  // Distance squared.
  float dist_sq = dot( vec_vl, vec_vl );

  // False if the light illuminates the side we can't see.
  bool lit_side = dot( normal_e, vec_vl )>0 == dot( normal_e, -vertex_e.xyz )>0;

  // Amount of light reaching the fragment.
  float attenuation = lit_side ? abs( dot( normal_e, vec_vl ) / dist_sq ) : 0;

  // Material color.
  vec4 mat_color = gl_FrontFacing ? uc.front[color_idx] : uc.back[color_idx];

  // Compute lighted color.
  frag_color = texel * mat_color * uni_light.color * attenuation;
}
```


Staple This Side
Staple This Side

Staple This Side

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

(*a*) Answer the following questions about colors.

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

Consider three options: providing a color with each vertex (as an attribute), such as `color_gold`, providing a color index (as an attribute) as done in some homework assignments this semester, or just providing a color in a uniform variable.

☐ Describe a situation in which a vertex color attribute is the best of the three options. ☐ Explain the advantage.

☐ Describe a situation in which a color index attribute is the best of the three options. ☐ Explain the advantage.

☐ Describe a situation in which a color in a uniform variable is the best of the three options. ☐ Explain the advantage.

(*b*) Answer the following questions on homogeneous coordinates.

☐ Provide an example of a homogeneous coordinate ☐ and its homogenized value.

☐ Describe something that can be done with a homogeneous coordinate needed for rendering that can't be done with ordinary coordinates.

Staple This Side

3 Fall 2023

Name _____ *Formatted For Two-Sided Printing*

Staple This Side

GPU Programming
EE 4702-1
Midterm Exam

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

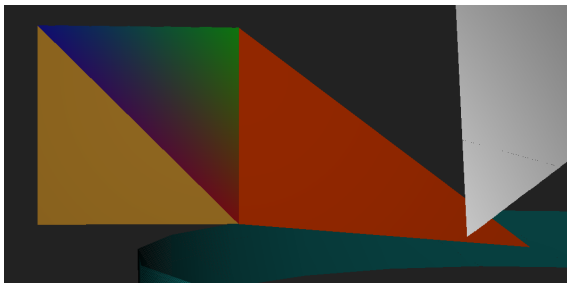
Problem 1	_____	(30 pts)
Problem 2	_____	(15 pts)
Problem 3	_____	(25 pts)
Problem 4	_____	(30 pts)
Exam Total	_____	(100 pts)

Alias _____

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.

(*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.

☐ 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.

/// 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.


coors_os << pCoor(-4,0,-3) << pCoor(-4,2,-3) << pCoor(-2,0,-3);
colors << color_lsu_gold << color_lsu_gold << color_lsu_gold;


coors_os << pCoor(-4,2,-3) << pCoor(-2,2,-3) << pCoor(-2,0,-3);
colors << color_blue << color_green << color_red;


coors_os << pCoor(-2,2,-3) << pCoor(-2,0,-3) << center + sz;
colors << color_orange_red << color_orange_red << color_orange_red;


// 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();
```


Staple This Side
Staple This Side

Staple This Side

/// 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+0]; 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;

          // Blend color of three vertices together.
          pColor color = b0*c0 + b1*c1 + b2*c2;

          // Find approximate object-space coordinate of fragment.
          pCoor of = b0*o0 + b1*o1 + b2*o2;

          // 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;

          // Write the frame (color) buffer with the lighted color
          frame_buffer[ idx ] = lighted_color.int_rgb();
          frame_buffer.n_px_frame++; // Count of number of written pixels.
        }}
```

Staple This Side

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.
```

```
pMatrix m =                                          ; //  ☐  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

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:

☐ Min number of depth buffer reads:

☐ Max number of depth buffer writes:

☐ Min number of depth buffer writes:

☐ Max number of color buffer reads:

☐ Min number of color buffer reads:

☐ Max number of color buffer writes:

☐ Min number of color buffer writes:

☐ Briefly explain difference between maximum and minimum cases.

(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:

☐ Min number of color buffer reads:

☐ Max number of color buffer writes:

☐ Min number of color buffer writes:

(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.

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.)

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

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

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

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

  return t;
}
```

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

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

vec2 qa = p.xy;          // ☐ qa =

vec2 qb = p.xx;          // ☐ qb =

vec2 qc = p.ba;          // ☐ qc =
```

(*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?

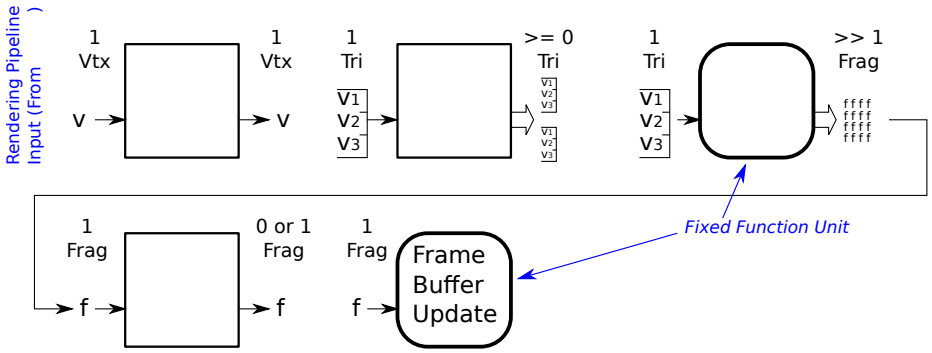
☐ The frame buffer should be written with one of these two items. How could one tell if the wrong one of these two were written?

Staple This Side

Staple This Side

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.

☐ 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*?

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

(*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.

☐ 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.

Staple This Side

Name _____ *Formatted For Two-Sided Printing*

Staple This Side

GPU Programming
LSU EE 4702-1
Final Examination

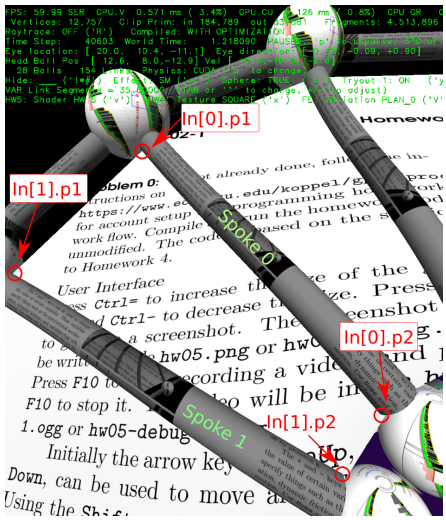
Thursday, 7 December 2023 7:30-9:30 CST

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

Alias _____

Good Luck!

Staple This Side



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 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.

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

```
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();

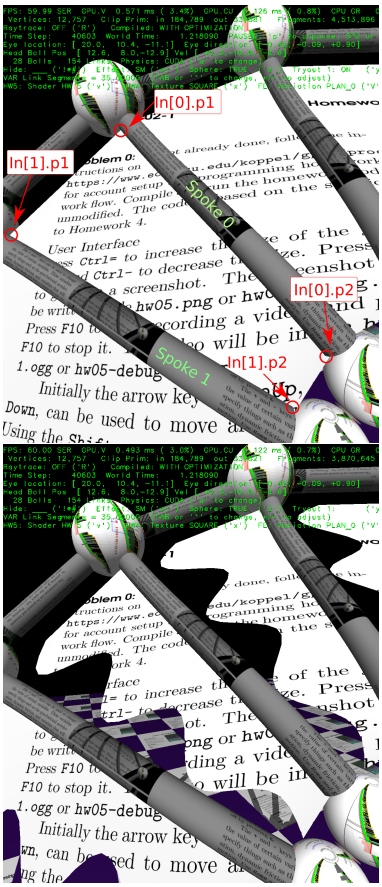
    // Emit the vertex along spoke 1  ( In[1] ).
    gl_Position = ut.clip_from_object * pos1t;
    Out.vertex_e = ut.eye_from_object * pos1t;
    Out.tex_coor = mix( In[1].tc1, In[1].tc2, t );
    EmitVertex();

    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();
}
```


Staple This Side
Staple This Side

Staple This Side

(*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.

```
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);









      //  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();
    }
  EndPrimitive();
}
```

Staple This Side

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?

☐ 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?

☐ 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?

☐ 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.

☐ 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.

Staple This Side

Staple This Side

Staple This Side

```
/// Homework 4 CPU Code: Input topology is a triangle strip. Vertex attribute: pCoor
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;
    pCoor 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];
    pCoor 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;
        pCoor 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.
```

Staple This Side

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

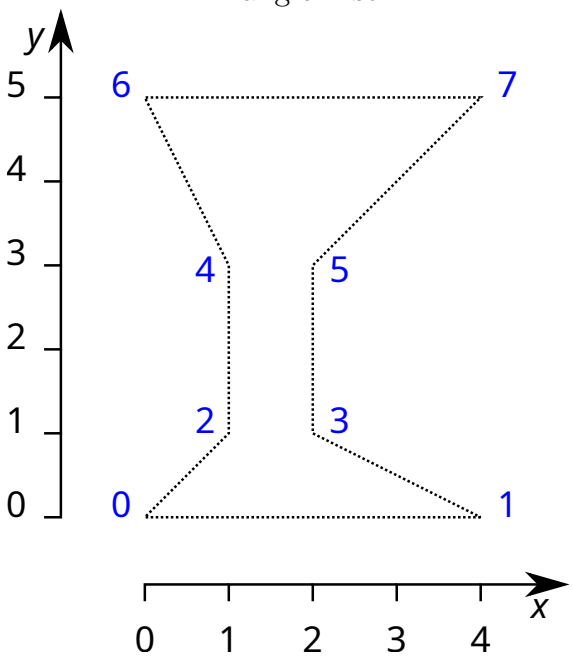
<< pCoor(4,5)  // Vtx 7
<< pCoor(0,5)  // Vtx 6
<< pCoor(1,3); // Vtx 4
```

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

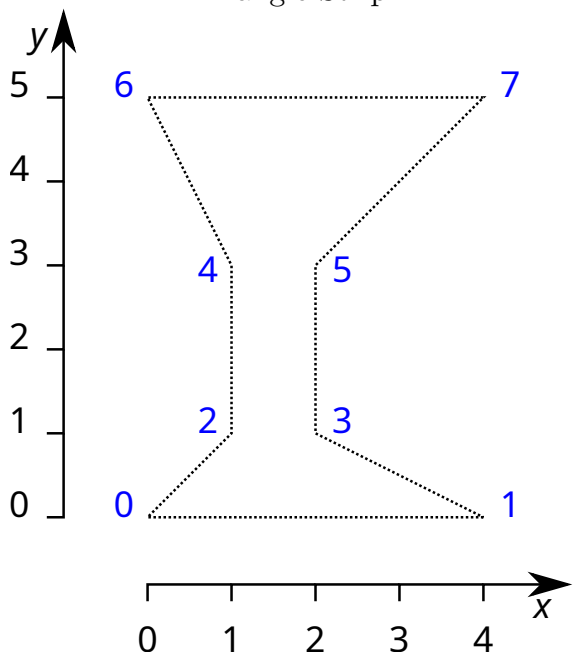
☐ On the diagram on the left sketch the triangles that will be rendered by the code below when the pipeline is set up for a triangle list.

☐ 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.

Triangle List



Triangle Strip



Staple This Side
Staple This Side

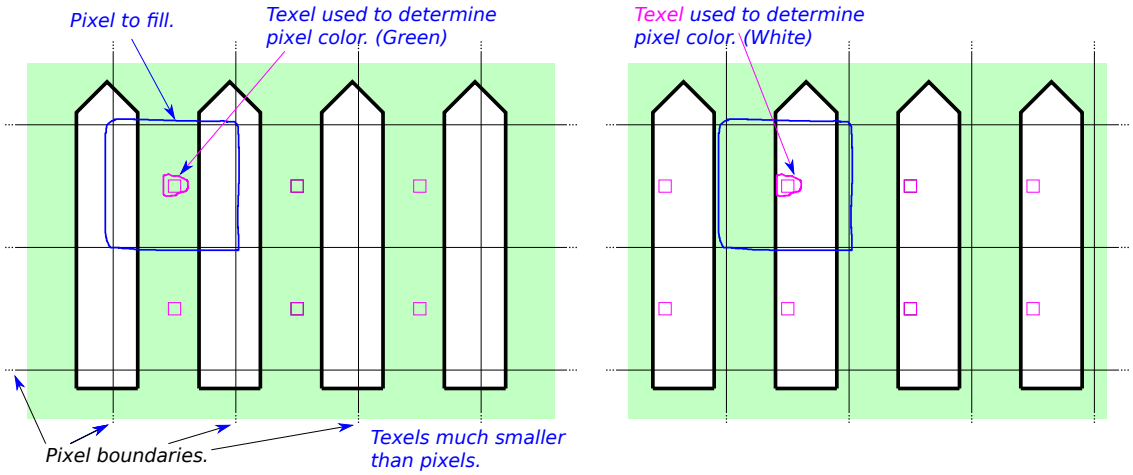
Staple This Side

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:

(*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 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?

Staple This Side

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;
      pCoor 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.

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

(*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;
     pCoor wf = b0*w0 + b1*w1 + b2*w2;  // Window-space coordinate of fragment.

     float bc0 = 1/(f.w0ow2z*(1-b0)/b0+1),   bc1 = 1/(f.w1ow2z*(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.

☐ 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`.

Staple This Side
Staple This Side

Staple This Side

(*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.

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.

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

Staple This Side

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

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

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

(*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:

(*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.

☐ $S(t) =$

☐ $t(x_1) =$

Staple This Side

Staple This Side

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.

Staple This Side

Staple This Side

4 Fall 2022

Name _____

GPU Programming
EE 4702-1
Midterm Exam

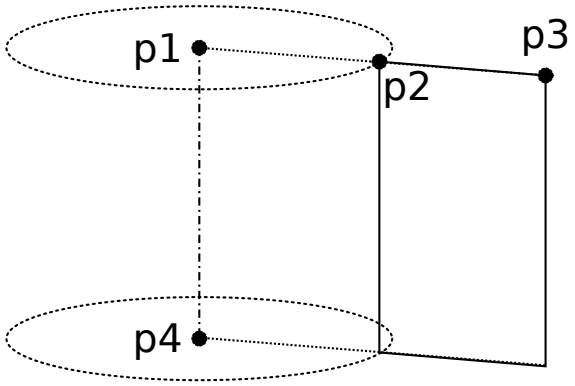
21 October 2022, 9:30-10:20 CDT

Problem 1	_____	(30 pts)
Problem 2	_____	(15 pts)
Problem 3	_____	(15 pts)
Problem 4	_____	(40 pts)
Exam Total	_____	(100 pts)

Alias _____

Good Luck!

Problem 1: [30 pts] Appearing below is code based on the solution to Homework 2, in which a paddle wheel was to be rendered with the shape and positioning determined by given points p1, p2, p3, and p4. The code has been modified to work with the course Vulkan library and the shaders written for Homework 3.



(*a*) The last lines insert normals into the buffer set. These normals are obviously wrong. Modify the code to compute the correct normal.

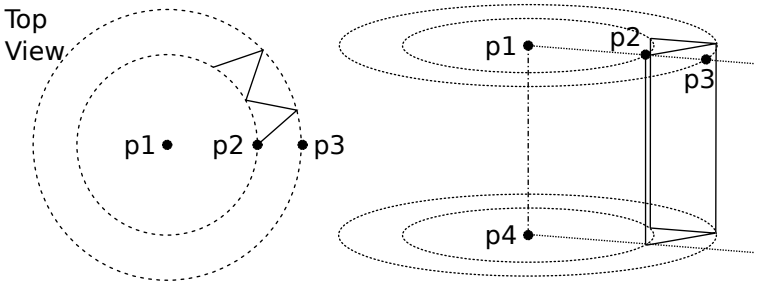
☐ Modify code to compute correct normal.

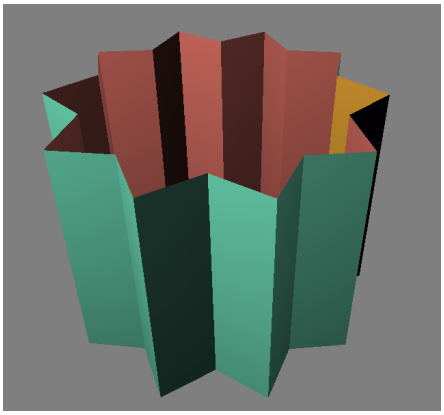
```
pVect vx(p1,p2), vz(p1,p4);
pNorm ax(vx), ay( cross(vz,ax) );
pVect vy = ay * ax.magnitude;
float r2or1 = pNorm(p1,p3).magnitude / ax.magnitude;
float delta_theta = 2 * M_PI / n_pieces;

for ( int i=0; i<n_pieces; i++ ) {
    float theta = i * delta_theta;
    pVect v = vx * cosf(theta) + vy * sinf(theta);
    pVect v2 = v * r2or1;
    bset_mta <<  p1 + v   <<  p1 + v2  <<  p4 + v;
    bset_mta <<  p1 + v2  <<  p4 + v2  <<  p4 + v;

    pNorm n = pVect(1,2,3);  // Obviously incorrect normal.

    // Insert normals into buffer set.
    bset_mta << n << n << n;
    bset_mta << n << n << n;
}
```





(*b*) Appearing below again is the code from the Homework 2 solution. This time the goal is to render a (crude) gear positioned and oriented using the same four points, `p1`-`p4`. The points define two pairs of circles as illustrated above. The center illustration shows how one tooth (gear part) is constructed. There should be `n_pieces` teeth, positioned so they touch and go around the gear as shown on the screenshot to the right. Modify the code to draw the gear.

☐ Complete code so that it draws the figure. ☐ Avoid computationally wasteful code.

☐ Write coordinates and normals. Don't write colors.

```
pVect vx(p1,p2), vz(p1,p4);
pNorm ax(vx), ay( cross(vz,ax) );
pVect vy = ay * ax.magnitude;
float r2or1 = pNorm(p1,p3).magnitude / ax.magnitude;
float delta_theta = 2 * M_PI / n_pieces;


for ( int i=0; i<n_pieces; i++ ) {
    float theta = i * delta_theta;
    pVect v = vx * cosf(theta) + vy * sinf(theta);
    pVect v2 = v * r2or1;









    bset_mta <<




    bset_mta <<
}
```

Problem 2: [15 pts] Coordinates `w0`, `w1`, and `w2` are the window-space coordinates of a triangle, call it `w`, ready to be rasterized. As pointed out in class, a triangle can be rasterized more efficiently if an edge is parallel to the x axis. Split triangle `w` into two triangles, `a` and `b`, such that `a` and `b` both have an edge parallel to the x axis and of course `a` and `b` cover the same points as `w`. To make things easier, assume that `w0.y > w1.y > w2.y`. Vectors between `w`'s vertices have been computed for convenience.

☐ Assign `a0` through `b2` so that `a` and `b` each have an edge parallel to the x axis and ☐ `a` and `b` together form `w`.

```
pCoor w0 = coors_w[i++];   // Assume w0.y > w1.y > w2.y
pCoor w1 = coors_w[i++];
pCoor w2 = coors_w[i++];

pVect v02(w0,w2);
pVect v01(w0,w1);
pVect v12(w1,w2);
```

```
// Triangle a
//
pCoor a0 =

pCoor a1 =

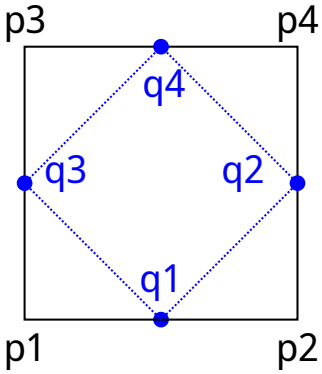
pCoor a2 =

// Triangle b
//
pCoor b0 =

pCoor b1 =

pCoor b2 =
```

Problem 3: [15 pts] The points `p1` through `p4` and points `q1` through `q4` in the diagram to the right each form a square, and both squares are in the same plane. Compute transformation matrix `m` (see below) that will transform points `p1`-`p4` to points `q1`-`q4`. The code at the bottom of the page shows how the matrix will be used. Some transformations are shown for reference.



☐ Compute transformation `m` that rotates points as described.

```
pCoor p1 = get_point(),  p2 = get_point();
pCoor p3 = get_point(),  p4 = get_point();

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

```
pMatrix m =                                                    ;


pCoor q1 = m * p1;
pCoor q2 = m * p2;
pCoor q3 = m * p3;
pCoor q4 = m * p4;
```

Problem 4: [40 pts] Answer each question below.

(*a*) The vertex shader code below is getting the clip-from-object transformation matrix from a uniform variable. It is possible to supply this matrix as a vertex shader input. What would be the disadvantage of doing so?

```
// Conventional Approach
layout (location = LOC_IN_POS) in vec4 in_vertex_o;
void vs_main() {
  // Transform the vertex object-space coordinate to clip space.
  gl_Position = ut.clip_from_object * in_vertex_o;


// Alternative Approach
layout (location = LOC_IN_POS) in vec4 in_vertex_o;
layout (location = LOC_IN_MAT) in mat4 in_clip_from_object;
void vs_main() {
  // Transform the vertex object-space coordinate to clip space.
  gl_Position = in_clip_from_object * in_vertex_o;
```

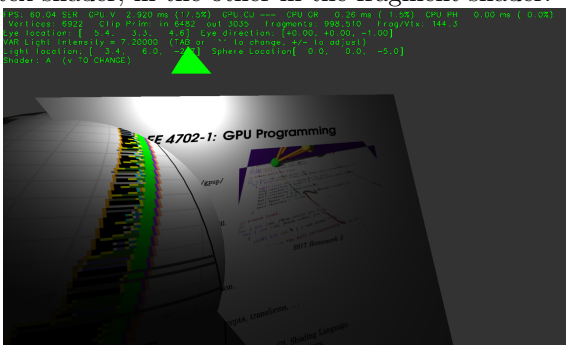
☐ Disadvantage of sending matrix as shader input.

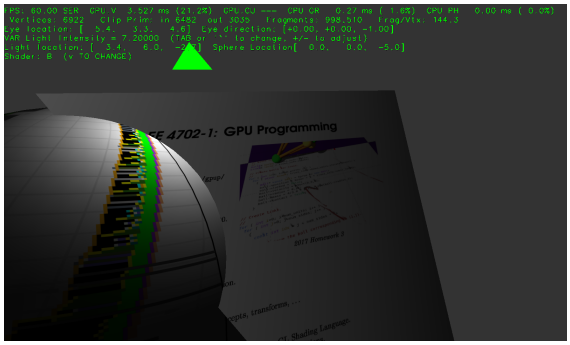
(*b*) In Homework 3 a set of colors was placed in a uniform variable. Would it be feasible to put vertex coordinates in a uniform variable also for typical rendering?

☐ Is putting vertex coordinates in a uniform variable ◯ *feasible* or ◯ *not feasible* ?

☐ Explain.

(*c*) The two screenshots below show the same scene rendered with two different sets of shaders. The scene consists of one big triangle and a sphere (consisting of many triangles). In one image lighting is done in the vertex shader, in the other in the fragment shader.





☐ In which image is lighting being performed in the fragment shader? ◯ *Shader A (left)* or ◯ *Shader B (right)* . ☐ Explain.

☐ Performing lighting in the vertex shader ◯ *looks better* ◯ *uses less computation* . ☐ Explain.

☐ Performing lighting in the fragment shader ◯ *looks better* ◯ *uses less computation* . ☐ Explain.

(*d*) Suppose the big triangle in the images above were split into 64 triangles.

☐ How many additional vertices would there be when rendered using a triangle list.

☐ How many additional fragments would there be?

☐ How would that affect the two images above?

(*e*) One thing our CPU-only ray tracing code did not do was check for shadows. But it would be easy to do so. Suppose ray tracing code included a routine `cast_ray(ray_origin,ray_direction)` that returned the triangle closest to the ray origin. Using that routine, how can one check for shadows?

☐ With ray tracing can check for shadows by:

(*f*) In our CPU-only rasterization routines we had our own rasterization code:

```
for ( float b0=0; b0<=1; b0 += db0 )
  for ( float b1=0; b1<=1-b0; b1 += db1 ) {
      pCoor c = w2 + b0 * v20 + b1 * v21;  // Window-space coordinate.
      if ( uint(c.x) >= win_width || uint(c.y) >= win_height ) continue;
      demo_frame_buffer[ c.x + int(c.y) * win_width ] = color;
    }
```

The routine above was kept simple for teaching purposes, but we could have tuned it for the CPU or GPU the code was to run on. However in both Vulkan and OpenGL rasterization is a fixed-function stage, meaning we can't write our own specially tuned routine. Doesn't that hurt—not help—performance?

☐ Why is rasterization fixed-function? ☐ How does preventing us from tuning help performance?

Name _____

GPU Programming
LSU EE 4702-1
Final Examination

Wednesday, 7 December 2022 15:00-17:00 CST

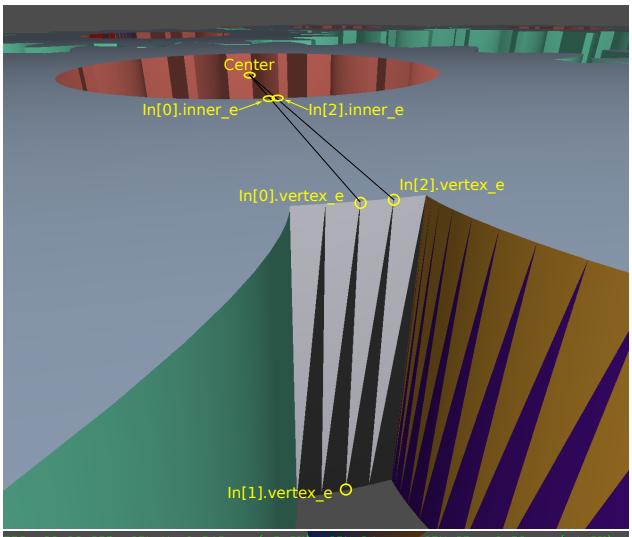
Problem 1	_____	(20 pts)
Problem 2	_____	(20 pts)
Problem 3	_____	(15 pts)
Problem 4	_____	(30 pts)
Problem 5	_____	(15 pts)
Exam Total	_____	(100 pts)

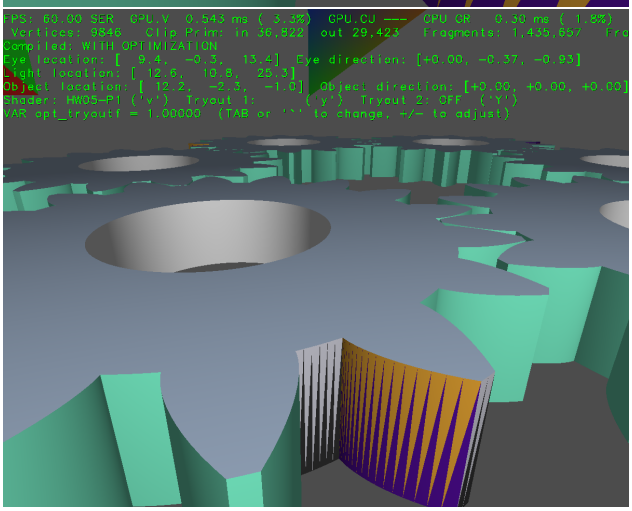
Alias _____

Good Luck!

Problem 1: [20 pts] Appearing below is the geometry shader based on Homework 5, Problem 1, in which gears were rendered.

(*a*) The screenshot to the upper right shows how coordinates provided to the geometry shader, such as `In[0].inner_e`, relate to the gear. In the screenshot to the lower right the gear hole has walls, shown in white. Add code to the shader so that the walls are rendered. (The code rendering the sides of the gear [shown in gray] is not shown, and should be ignored for this problem.) *Hint: Don't write too much code. A loop is needed, but a correct solution does not need code before the loop.*





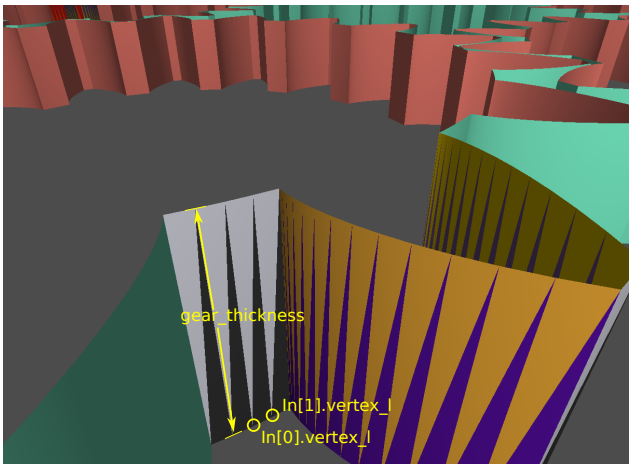
- ☐ Add code to render the walls.
- ☐ Be sure to correctly compute the normal, including for points on the lands (such as the purple/gold section).
- ☐ Use color index 7 for the walls.
- ☐ Make sure that the front of the triangle is visible.

```
layout ( triangles ) in;
layout ( triangle_strip, max_vertices=8 ) out;
layout (location = 0) in Data_VG {
  vec4 vertex_c, vertex_e, inner_c, inner_e;
  vec3 normal_e;
  int color_idx;
} In[];

void gs_main_prob1() {  /// Geometry Shader
  for ( int i=0; i<3; i++ ) {    // Emit edge of gear (lands and faces).
      gl_Position = In[i].vertex_c;
      vertex_e = In[i].vertex_e;
      normal_e = In[i].normal_e;
      color_idx = In[i].color_idx;
      EmitVertex();
    }
  EndPrimitive();

  // Add code for hole walls here.
```

(*b*) The geometry shader below correctly renders the gear (not the sides) and was written for a geometry shader input set to triangles (three vertices each). Modify the geometry shader so that it works when the geometry shader input is set to lines (two vertices each). The vertices provided are labeled in the screenshot to the right. These are vertices at $z = 0$ in local space. The (The output of the geometry shader remains a triangle strip.)



Input `vertex_l` is a local space coordinate along the bottom of the gear. The code correctly transforms these to eye and clip space. Modify the code so that it computes the coordinates along the top of the gear (`In[0] and In[2]` in the previous problem), and then emits triangles for the edge (but not the sides).

☐ Modify code so that it works for line strip input, and uses `gear_thickness` to compute the upper gear coordinates.

☐ Hint: Will need to update `max_vertices`. ☐ Hint: Not much more code is needed.

```
layout ( lines ) in; // Changed from triangles to lines.
layout ( triangle_strip, max_vertices = 3 ) out;  // [ ] Update as needed.
void gs_main_prob2() {   /// Geometry Shader
  int inst_idx = In[0].inst_idx;
  mat4 g_from_l = gears_xform[inst_idx] * rot;
  mat3 e_from_l = mat3(ut.eye_from_object) * mat3(g_from_l);
  vec4 avertex_e[3], vertex_c[3];
  vec3 anormal_e[3];
  float gt = gear_thickness; // Gear height in z axis in local space.

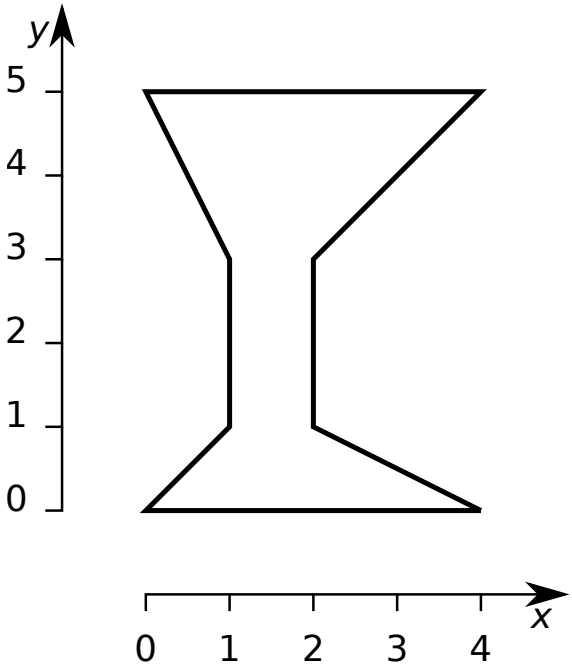
  for ( int i=0; i<3; i++ )
   {
      vec4 vertex_o = g_from_l * In[i].vertex_l;
      // Compute clip- and eye-space coordinates for vertex and normal.
      vertex_c[i] = ut.clip_from_object * vertex_o;
      avertex_e[i] = ut.eye_from_object * vertex_o;
      anormal_e[i] = normalize( e_from_l * In[i].normal_l );
    }



  for ( int i=0; i<3; i++ ) {
      gl_Position = vertex_c[i];
      vertex_e = avertex_e[i];
      normal_e = anormal_e[i];
      color_idx = In[i].color_idx;
      EmitVertex(); }
  EndPrimitive();
}
```

Problem 2: [20 pts] Answer the questions below about rendering an hourglass figure.

(*a*) Add coordinates to the buffer set below so that it renders the illustrated figure as a triangle strip. The z coordinate of all points is zero. Note that `pC(x,y)` can be used as an abbreviation for `pCoor(x,y,0)`. The first coordinate, `pC(0,0)`, is already inserted.



☐ Insert coordinates so that the buffer set renders the figure ☐ as a triangle strip.

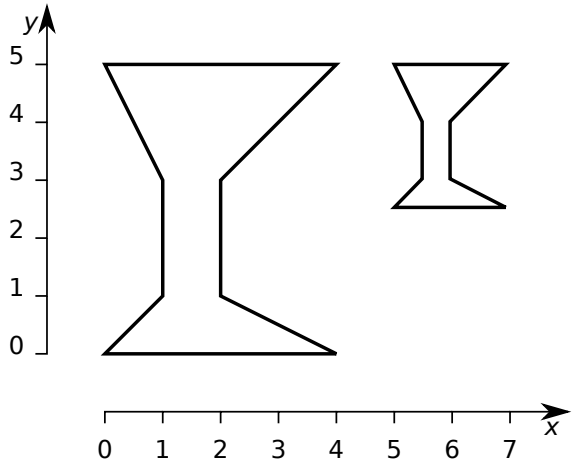
☐ Don't overlap triangles.

☐ Just insert coordinates, don't insert normal, colors, or anything else.

```
bset_hg.reset(pipe_hg);
auto pC = [&](float x, float y)
  { return pCoor(x,y,0); };

bset_hg << pC(0,0) <<
```

(*b*) The figure to the right shows the hourglass from the previous problem in its original position (which is described by `bset_hg`), and a second time in a new position (which is to be described by `bset_hg2`). Buffer set `bset_hg` contains the coordinates for the hourglass in its original position. The code below inserts coordinates into `bset_hg2` using coordinates from `bset_hg` transformed using `m`. Complete the code so that `m` is assigned an appropriate transformation matrix. For reference, several standard transformation matrices are provided below.



☐ Compute `m` to transform larger hourglass to smaller hourglass.

☐ Note that there is a change in position and in size. ☐ Pay attention to the order in which transformations are applied.

```
// The declarations below are for reference.
float angle, scale_factor;
pVect vec;
pMatrix_Rotation mat_r(axis,angle);
pMatrix_Translate mat_t(vec);
pMatrix_Scale mat_s(scale_factor);
// The declarations above are for reference.

// Note: bset_hg already has coordinates of hourglass.
```

```
pMatrix m = ;

for ( pCoor c: bset_hg.pos ) bset_hg2 << m * c;
```

Problem 3: [15 pts] Answer the following questions about clip and eye space.

(*a*) Draw a diagram illustrating the view volume. The diagram should include the eye, the near plane, and the far plane.

☐ Diagram showing eye, view volume, near plane, and far plane.

(*b*) Answer the following questions.

☐ The near plane distance was supposed to be set to 4.3. It is accidentally changed to 8.6. How does this affect the scene?

Note: The question in the original exam was: "What would happen if the near plane were too far from the eye?"

☐ What is the practical disadvantage of making the far plane too far from the eye?

(*c*) In a scene the user's eye is at location `eloc` (type `pCoor`) and the center of the user's monitor is at location `mloc` (type `pCoor`). The user is looking straight at the monitor, so it will appear as a rectangle, not as a trapezoid. The width of the user's monitor is `mwid` (type float). All of these dimensions are in global space. (For this problem consider object space equivalent to global space.)

Given this information plus needed assumptions compute the `eye_from_object` and `clip_from_eye` transformations. Some declarations have been provided for reference.

☐ Compute the `eye_from_object` transform. (OpenGL name: `gl_ModelViewMatrix`.)

☐ Compute the `clip_from_eye` transform. (OpenGL name: `gl_ProjectionMatrix`.)

☐ Make sure that `near` in the frustum transform is set correctly. ☐ Make sure that `height` in the frustum is set correctly, so that squares remain squares.

```
const int win_width = vh.s_extent.width;    // Frame buffer window width
const int win_height = vh.s_extent.height;
const float aspect = float(win_width) / win_height;

const float mwid = app.global_space_monitor_width;
const pCoor mloc = app.global_space_monitor_location;
const pCoor eloc = app.global_space_eye_location;

// Matrix examples below are for reference.
pMatrix_Rows m_r3( ax, ay, az );
// Rotate so that m_r3 * ax -> pVect(1,0,0), m_r3 * az -> pVect(0,0,1),...

pMatrix_Rotation m_r1( vec1, vec2 );
// Rotate so that m_r1 * vec1 = vec2;

pMatrix_Translate m_tlate(vec);
// Translate so that m_tlate * P -> P + vec;

pMatrix_Frustum m_frust( width, height, near, far );
// Width and height of "monitor" (small part of frustum).
// Near plane distance, far plane distance.

// Matrix examples above are for reference.
```

```
pMatrix eye_from_object = ;

pMatrix clip_from_eye = ;
```

Problem 4: [30 pts] Answer the following questions about rendering pipelines and shaders.

(*a*) Answer the following questions about shaders used for rasterization under Vulkan (or OpenGL).

☐ To render just one triangle in the whole scene, how many times would a vertex shader *typically* be called?

☐ Describe how it is possible to render T triangles with $T + 2$ invocations of a vertex shader.

☐ Describe how it is possible to render T triangles with T invocations of a vertex shader.

☐ To render one triangle in the whole scene, how many times would a geometry shader typically be invoked?

☐ A geometry shader is invoked G times. Why is it not possible to determine how many triangles are rendered?

☐ What does one need to know about a triangle to determine how many times a fragment shader is invoked to render the triangle?

(*b*) Answer the following questions about shaders in a Vulkan (NVidia or Khronos extension) ray tracing "pipeline".

☐ What does one need to know to determine how many times is the ray generation shader invoked? ☐ Provide an example.

☐ The ray generation shader casts a ray. In what direction does it cast the ray?

☐ The ray generation shader used in class set its ray payload, a four-element vector, to zero. What does it do with the payload after the ray is cast?

(*c*) Consider a scene that can be rendered using ray tracing or using rasterization. This was the case for many classroom examples. Consider a scene without shadows and without reflections.

☐ For the same scene, which is invoked more often, the fragment shader or the nearest hit shader? ☐ Explain.

Problem 5: [15 pts] Answer each question below.

(*a*) Consider Vulkan buffers bound to a pipeline as uniform buffers or as storage buffers.

☐ Both a uniform buffer and a storage buffer can hold arrays. ☐ Which is better for larger arrays?

☐ Both uniform buffers and storage buffers can be used to store a single (meaning just one) object, such as a transformation matrix or small structure. ☐ Which is better for single objects, and why?

(*b*) Textures are read using a built-in function in the fragment shader, such as `vec4 texel = texture(tex_unit_0,tex_coor);`.

☐ What are the range of values for `tex_coor`? How does that relate to the texture?

☐ How does the `texture` builtin function determine which MIPMAP level to use when retrieving texels?

5 Fall 2021

Name _____

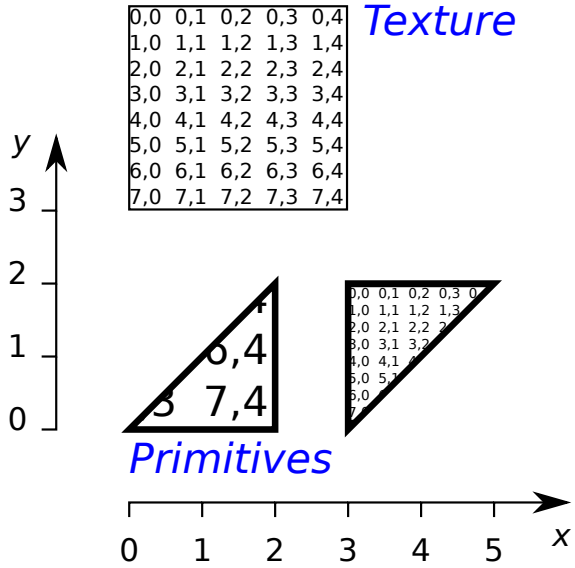
GPU Programming
EE 4702-1
Take-Home Pre-Final Examination

Due: 4 December 2021 at 23:59 CST

Problem 1	_____	(10 pts)
Problem 2	_____	(10 pts)
Problem 3	_____	(30 pts)
Problem 4	_____	(20 pts)
Problem 5	_____	(30 pts)
Exam Total	_____	(100 pts)

Good Luck!

Problem 1: [10 pts] Appearing to the right are two primitives (triangles) and a texture image. Further below is code that creates a pipeline for the two triangles and populates a buffer set with coordinates. Using convenience function `tC` insert texture coordinates so that the texture image appears as shown on the triangles.



☐ Insert texture coordinates into `bset_triss` so that the texture renders as shown.

☐ Just insert texture coordinates. Use the `tC` function to avoid writing out `pTCoor(foo,bar)`.

```
void World::render_tris(vk::CommandBuffer& cb) {
  if ( !pipe_tris )
    pipe_tris .init(vh.qs)
      .color_uniform_set( color_white * 0.9, color_red )
      .use_uni_light( uni_light )
      .use_texture( sampler, texid_tris )
      .topology_set( vk::PrimitiveTopology::eTriangleList )
      .create();

  auto pC = [&](float x, float y) { return pCoor(x,y,0); };
  auto tC = [&](float x, float y) { return pTCoor(x,y); };

  bset_triss.reset( pipe_tris );

  bset_triss
    << pC(0,0)

    << pC(2,0)

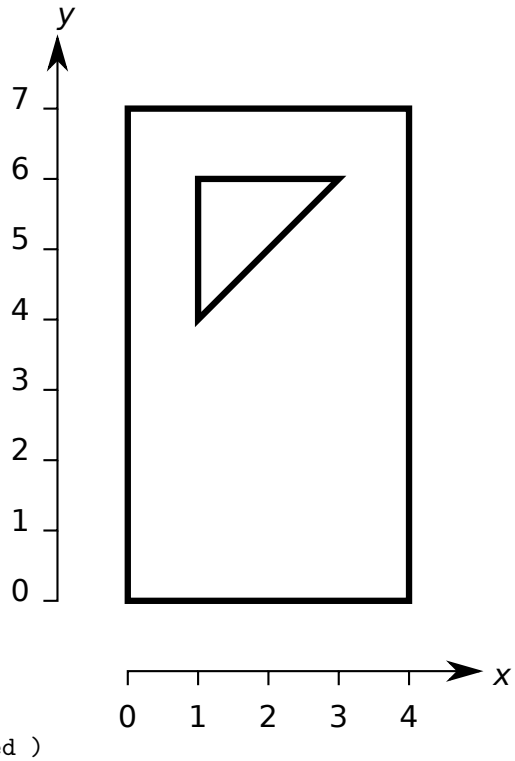
    << pC(2,2)


    << pC(3,0)

    << pC(5,2)

    << pC(3,2)
    ;
```

Problem 2: [10 pts] Appearing below is an illustration of a rectangle with a triangular hole, call it a card. Also in the illustration are x and y axes in a local coordinate space. For all points $z = 0$.



(a) Appearing below is code that prepares a pipeline and is to populate a buffer set for rendering the card. The first coordinate, $\begin{bmatrix} 0 \\ 7 \\ 0 \end{bmatrix}$, is inserted using the convenience function `pC`. Insert the remaining coordinates needed to render the card. Note that the pipeline expects vertices in triangle strip order.

- ☐ Describe the shape using one—just one—triangle strip.
- ☐ Use the coordinates given in the diagram. Omit the z coordinate.
- ☐ For this part just insert coordinates.
- ☐ No part of the hole should be covered by a triangle.

```
void World::render_card(vk::CommandBuffer& cb)
{
  if ( !pipe_card )
    pipe_card .init(vh.qs)
      .color_uniform_set( color_white * 0.9, color_red )
      .use_uni_light( uni_light ) .use_texture( sampler, texid_card )
      .topology_set( vk::PrimitiveTopology::eTriangleStrip )
      .create();

  auto pC = [&](float x, float y) { return pCoor(x,y,0); };

  bset_cards.reset( pipe_card );

  bset_cards
    << pC(0,7) <<
```

Problem 3: [30 pts] The code below renders a pointy p (from last year's pre-final and final exams) above each ball. The array `pts` holds the coordinates needed to draw a pointy p in its local coordinate space. (x and y only, all z values are zero). The `i` loop computes a transformation matrix, `p_xform`, that maps points from the pointy p local space to a position above ball `i`. The matrix is put into a storage buffer and the local-space coordinates are written to the buffer set. The z component of the coordinate is set to i, not zero. The vertex shader will use that value of i to retrieve the transformation matrix.

```
void World::render_p2(vk::CommandBuffer& cb) {
  if ( !pipe_p2 )
    pipe_p2.init(vh.qs)
      .storage_bind("BIND_PXFORM").uniform_bind( buf_uni_common, "BIND_UNI_COMMON" )
      .color_uniform_set( color_white * 0.9, color_red ).use_uni_light( uni_light )
      .shader_inputs_info_set<pCoor>()
      .shader_code_set
      ("mt-shdr-p.cc", "vs_main_p2(); ", "gs_main_p2();", "fs_main();")
      .topology_set( vk::PrimitiveTopology::eTriangleStrip )
      .create();

  // Coordinates needed to draw a pointy p using a triangle strip.
  vector<vec2> pts = { {0,0},{1,0},  {0,7},{1,6},  {4,5},{3,5},  {1,3},{1,4} };

  const int n_balls = balls.size();
  const bool n_changed = n_balls != n_balls_seen;
  n_balls_seen = n_balls;

  buf_p_xform.clear();    bset_p2.reset( pipe_p2 );

  for ( int i=0; i<n_balls; i++ )
    {
      Ball* const ball = balls[i];
      pMatrix p_xform = pMatrix_Translate(ball->position) * ball->omatrix
        * pMatrix_Translate(pVect(0,ball->radius * 1.5,0))
        * pMatrix_Rotation(pVect(0,1,0),M_PI)
        * pMatrix_Scale( ball->radius / 3.5 );

      buf_p_xform << p_xform;

      // Emit the coordinates for a pointy p for ball i.
      // The value of i is put in the z coordinate so that the
      // shader can retrieve the transformation matrix.
      for ( auto p: pts ) bset_p2 << pCoor(p.x, p.y, i);
    }

  buf_p_xform.to_dev();  // Send transformation matrices to the GPU.
  bset_p2.to_dev();      // Send shader inputs to the GPU.

  pipe_p2.storage_bind( *buf_p_xform, "BIND_PXFORM" );

  transform.use_global_for( pipe_p2 );
  pipe_p2.record_draw(cb,bset_p2);
}
```

Problem 3, continued: The vertex and geometry shaders appear below.

```
// Interface block for vertex shader output / geometry shader input.
layout ( location = 0 ) out Data_to_GS
{
  vec3 normal_e;
  vec4 vertex_e;
  vec4 vertex_c;
};

void vs_main_p2() {
  vec4 vertex_o = vec4(in_vertex_o.xy,0,1);
  mat4 xform = p_xform[int(in_vertex_o.z)];
  vec3 normal_o = vec3(0,0,1);
  vec3 normal_g = mat3(xform) * normal_o;
  vec4 vertex_g = xform * vertex_o;

  vertex_c = gl_ModelViewProjectionMatrix * vertex_g;
  vertex_e = gl_ModelViewMatrix * vertex_g;
  normal_e = normalize(gl_NormalMatrix * normal_g );
}

layout ( location = 0 ) in Data_to_GS
{
  vec3 normal_e;
  vec4 vertex_e;
  vec4 vertex_c;

} In[];

layout ( triangles ) in;
layout ( triangle_strip, max_vertices = 3 ) out;

void gs_main_p2() {
  for ( int i=0; i<3; i++ )
    {
      gl_Position = In[i].vertex_c;
      normal_e = In[i].normal_e;
      vertex_e = In[i].vertex_e;
      EmitVertex();
    }
  EndPrimitive();
}
```

(*a*) Compute the amount of data sent from the CPU to the GPU for each call to `render_p2`. Don't include things that are done just once, such as the initialization of `pipe_p2`. Include the storage buffer and the buffer set. Base the size on the data inserted into the buffers by the routine. All data is inserted using the `<<` operator. In your answer let n denote the value of `n_balls` and let p denote the number of coordinates in `pts`. (We know $p = 8$ but use p anyway.)

☐ Amount of data, in bytes, sent for a typical call to `render_p2` in terms of n and c is:

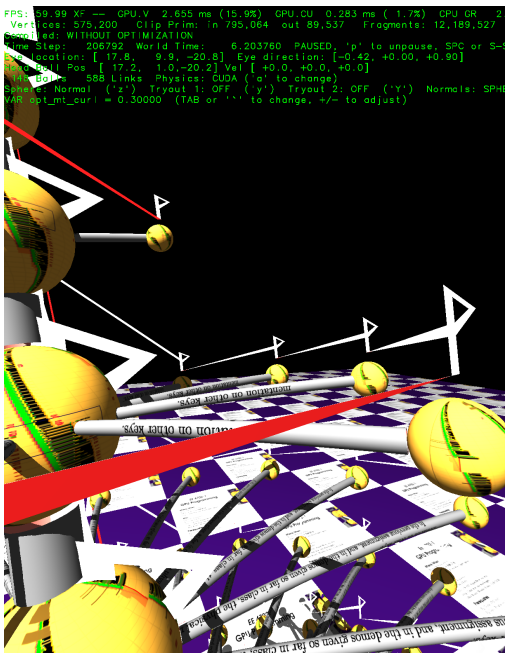
(*b*) Assume that the value of `n_balls` does not change very often, but that the values of `ball->position` do change each time `render_p2` is called. Based on this observation, modify the code to reduce the amount of data sent from the CPU to the GPU. The variable `n_changed` should be useful, it is true when `render_p2` is called the first time and when the number of balls has changed.

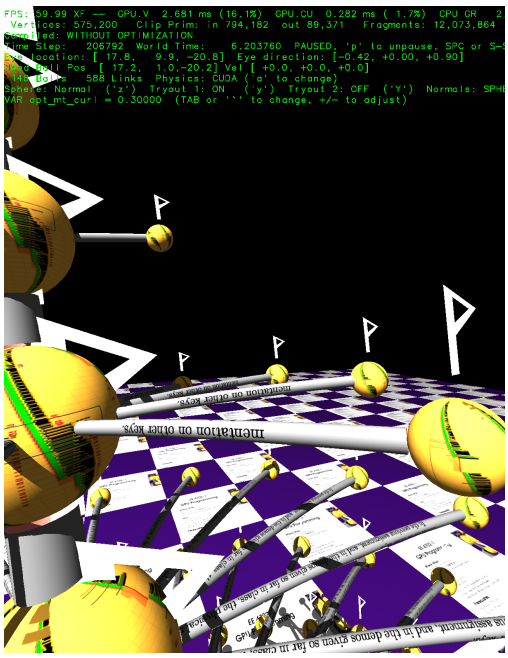
Show your solution on the code several pages back.

☐ Modify code to reduce CPU to GPU transfer when number of balls does not change.

☐ By what fraction has the amount of data been reduced?

(*c*) Because all the pointy p's are rendered using one triangle strip they are connected. See the screenshot on the lower left. Modify the shader code so that they are not connected, as in the screenshot on the right. *Hint: Use the value of* `i` *placed into the z component of the coordinate.*





☐ Modify shaders so that pointy p's are not connected.

Problem 4: [20 pts] Appearing below is the code at the end of the default closest-hit shader, which is part of the course demo of the ray tracing extension to Vulkan and OGSL. (The code is in file `dev-vulkan/rt-shdr-main.cc`, and is in the `main` routine following `#ifdef _RT_CLOSEST_HIT_`.) This code casts a ray at the light and dims the color if the closest-hit vertex is shaded.

```
  // At this point rp_color is set to the lighted color, assuming no shadow.
  vec4 light_g = ut.mvi * ul.cgl_LightSource[0].position;
  vec3 vtx_to_light_g = light_g.xyz - vertex_g.xyz;
  float tmin = 0.001 / length(vtx_to_light_g),   tmax = 1.0;

  rp_shadowed = true; // The miss shader will set this to false, if invoked.

  traceNV
   ( topLevelAS,
     gl_RayFlagsTerminateOnFirstHitNV
     | gl_RayFlagsOpaqueNV | gl_RayFlagsSkipClosestHitShaderNV,
     0xfe, // Don't include lights in intersection test.
     0,0,  // Specify which kind of hit shader to use.
     1,    // Specify which miss shader to use.
     vertex_g.xyz,               // Ray Origin,
     tmin, vtx_to_light_g, tmax,  // Ray Vector, and Distance Range (tmin,tmax)
     2 );   // Ray payload location.

  rp_color = rp_shadowed ? 0.3f * c : c;
}
```

(*a*) Answer the following questions about, `tmin` and `tmax`, which are the minimum and maximum positions on the ray for ray traversal.

☐ Note that `tmax` is set to `1`. What would happen if `tmax` were set to `0.5`? ☐ Your answer should include a description of the flaw in rendered image due to this change.

☐ Care is being taken to set `tmin` to a value that's small, but not too small. What might go wrong if `tmin` were set to zero, or would certainly go wrong if `tmin=-.001`? ☐ Your answer should include a description of the flaw in rendered image due to this change.

(*b*) The code above initializes `rp_shadowed = true` and uses a miss shader to set `rp_shadowed = false`. An alternative approach would be to initialize `rp_shadowed = false` and use a closest-hit shader to set `rp_shadowed = true`. This can easily be made to work, but it would have lower performance. Why?

☐ Why might the alternative approach, using a closest-hit shader, run more slowly than the code above, which uses a miss shader?

(*c*) In an instance using an intersection shader, the acceleration structure is prepared using axis-aligned bounding boxes (AABBs). For example, for our sphere shader (in file `rt-shdr-sphere.cc`) the AABB for a sphere of radius r is a cube with each edge of length $2r$, a perfect fit. Describe the impact on execution time and correctness if the AABBs were made too large, say, twice as large as they needed to be.

☐ Impact on execution time if AABBs were too large. ☐ Explain.

☐ Impact on correctness if AABBs were too large. ☐ Explain.

Problem 5: [30 pts] Answer each question below.

(*a*) The two shaders below do the same thing, though slightly differently.

```
void vs_plan_a() {
  vertex_e = gl_ModelViewMatrix * in_vertex;
  gl_Position = gl_ProjectionMatrix * vertex_e;
}

void vs_plan_b() {
  vertex_e = gl_ModelViewMatrix * in_vertex;
  gl_Position = gl_ModelViewProjectionMatrix * in_vertex;
}
```

The shader code is provided a modelview matrix and a projection matrix at the beginning of the pipeline execution. In both `vs_plan_a` and `vs_plan_b` there are two matrix/vector multiplies, which each require $4^2 = 16$ multiply/add operations. But `vs_plan_b` uses `gl_ModelViewProjectionMatrix`, which is the product of the modelview and projection matrices. The product of these two matrices is computed using $4^3 = 64$ multiply/add operations. That brings the total to $16 + 16 + 64 = 96$ operations, much more than 32 for `vs_plan_a`. Therefore, `vs_plan_a` is more computationally efficient than `vs_plan_b`.

☐ Describe the flaw with this `vs_plan_a`-is-better-than-`vs_plan_b` argument.

☐ Describe a case when the argument is correct, but explain why this case does not reflect typical use.

(*b*) Answer the following questions about view volumes.

☐ What is a view volume?

☐ It is easy to determine whether a vertex is in the view volume by using its coordinate in ◯ *object space*, ◯ *eye space*, or ◯ *clip space* (check one).

☐ Let $P = \begin{bmatrix} x \\ y \\ z \\ w \end{bmatrix}$ be a coordinate in the coordinate space chosen for the previous answer. Show a mathematical expression or write code that tests whether P is in the view volume.

☐ It is easy to determine whether some triangles are in the view volume. ☐ Provide an example of such a triangle and ☐ explain why.

☐ Provide an example of a triangle for which it is not so easy to determine if it is in the view volume. ☐ Illustrate with a diagram.

(*c*) Describe how suitable an OpenGL Shading Language uniform variable is for each of the following purposes.

☐ Explain whether this a suitable use for a uniform variable: To hold the lighted color computed by a vertex shader.

☐ Explain whether this a suitable use for a uniform variable: To hold the location of a light source.

(*d*) Vertex coordinates are usually three dimensional but texture coordinates are usually two dimensional. Why? (Ignore the w component in your answer.)

☐ Texture coordinates have two, not three, dimensions because:

(*e*) A homogeneous coordinate consists of four components, compared to just three for ordinary Cartesian coordinates. Homogeneous coordinates increase the amount of work needed for a matrix/vector multiply from 9 to 16 multiplications. Transformations are realized by multiplying a transformation matrix by a coordinate.

☐ Describe a transformation that cannot be done without homogeneous coordinates.

☐ Describe a transformation that can be done using ordinary Cartesian coordinates.

Name _____

GPU Programming
LSU EE 4702-1
Take-Home Final Examination

Wednesday 8 December to Friday 10 December 2021 16:30 CST

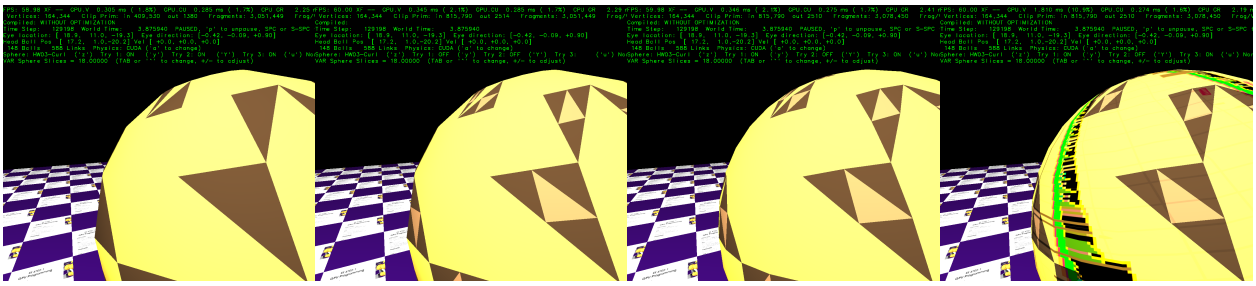
Work on this exam alone. Regular class resources, such as notes, papers, solutions, documentation, and code, can be used to find solutions. In addition outside Vulkan and OpenGL references and tutorials, other programming resources, and mathematical references can be consulted. Do not try to seek out references that specifically answer any question here. **Do not discuss this exam with classmates or anyone else**, except questions or concerns about problems should be directed to Dr. Koppelman.

Warning: Unlike homework assignments collaboration is not allowed on exams. Suspected copying will be reported to the dean of students. The kind of copying on a homework assignment that would result in a comment like "See ee4702xx for grading comments" will be reported if it occurs on an exam.

Problem 1	_____	(30 pts)
Problem 2	_____	(30 pts)
Problem 3	_____	(40 pts)
Exam Total	_____	(100 pts)

Good Luck!

Problem 1: [30 pts] An issue faced when approximating a sphere with triangles is the number of triangles to use. If too few are used the sphere looks like a polyhedron, especially around the edges. Using too many can slow things down. One possible solution is to split the triangles in the geometry shader—if necessary. The lower-left image shows a sphere that does not have enough triangles. Some triangles are highlighted (in brown) to make them apparent. Looking along the edge (limb) of the sphere one can see straight lines, which is not what we want to see. The second image shows what happens when each triangle is split into four triangles. (Remember that only some are highlighted in brown shades.) But, looking along the edge we see that this splitting hasn't helped. The reason is that the old triangle and the four new ones all fall in the same plane. The third image shows the way the triangles should be split. The three new vertices are positioned to be on the sphere surface, rather than the plane of the original triangle. The last shows the same sphere as the third, but with a texture applied.



(*a*) Modify the shader code on the next page so that it splits a triangle into three triangles as described above. The vertices of all triangles must be on the surface of the sphere. The code is provided with some routines that should be helpful. Routine `emit(i)`, to be called in the geometry shader, emits vertex `i`. It is used in the geometry shader main routine, `gs_main()`. Routine `emit(i,j)` emits one vertex, but the vertex attributes are interpolated from geometry shader inputs `i` and `j`. So, `emit(0,1)` will emit a vertex halfway between vertex 0 and 1. That vertex is **not** on the sphere surface.

Solve this problem by first calling `emit` so that the four triangles are emitted, though on the plane of the original triangle. Try to do so using no more than 8 vertices.

Then modify `emit(i,j)` so that the interpolated vertex lies on the sphere surface. Though it is possible to do so using only information currently provided to the geometry shader, it is okay if additional information is passed from the vertex shader, such as the sphere center.

☐ Modify `gs_main` so that it emits four triangles as illustrated.

☐ Modify `emit(i,j)` so the interpolated vertex lies on the sphere surface.

☐ If needed, pass additional information from the vertex shader to the geometry shader.

☐ Set `max_vertices` to the number of vertices emitted.

```
void vs_main_instances_sphere()   // Vertex Shader
{
  vec4 pos_rad = sphere_pos_rad[gl_InstanceIndex];
  vec4 center_o = vec4(pos_rad.xyz,1);
  float sphere_radius = pos_rad.w;
  mat4 rot = sphere_rot[gl_InstanceIndex];
  vec4 normr = rot * in_vertex_o;
  vec3 normal_o = normr.xyz;
  normal_e = normalize(gl_NormalMatrix * normal_o );
  vec4 center_e = gl_ModelViewMatrix * center_o;
  vertex_e = center_e + vec4(normal_e * sphere_radius,0);
  tex_coor = in_tex_coor;
  color = sphere_color[gl_InstanceIndex];
}

layout ( location = 0 ) in Data_to_GS {
  vec3 normal_e;
  vec4 vertex_e, color;
  vec2 tex_coor;
} In[];

layout ( triangles ) in;
layout ( triangle_strip, max_vertices = 3 ) out;

void emit(int i0, int i1)
{
  float f = i0 == i1 ? 0 : 0.5;

  normal_e = mix(In[i0].normal_e,In[i1].normal_e,f);




  vertex_e = mix(In[i0].vertex_e,In[i1].vertex_e,f);
  color = mix(In[i0].color,In[i1].color,f);
  tex_coor = mix(In[i0].tex_coor,In[i1].tex_coor,f);
  gl_Position = gl_ProjectionMatrix * vertex_e;
  EmitVertex();
}

void emit(int i) {emit(i,i);}

void gs_main()   // Geometry Shader
{
  emit(0); emit(1); emit(2);
  EndPrimitive();



}
```

(*b*) The code in the previous part always splits a triangle. But there is no point splitting a triangle if it is already small. Consider a triangle small if an edge spans less than 20 pixels. Also, it is triangles that are near the edge that should be split. Those in the center of our view look good even when relatively large. So it would make sense only to split triangles near the edge.

Modify the geometry shader below so that it assigns `split_tri` to true if the triangle should be split based on the size (in pixels) and whether it is close to the edge.

To help determining the size the window width in height in pixels are provided. What remains is determining the length of an edge, measured in pixels. That can be done using a particular coordinate space.

To determine if something is on the edge use the vertex normal and other information. (It is not necessary to use the exact method of the sphere shader.)

☐ Set `is_large` to true iff an edge is larger than 20 pixels.

☐ Set `near_edge` to true if the triangle is close to the edge of the sphere as seen by the viewer.

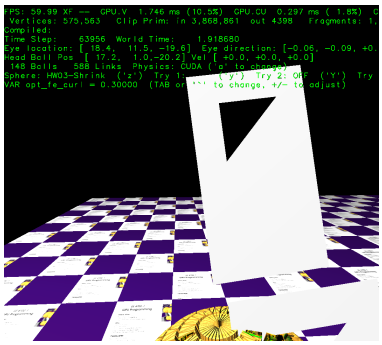
```
void gs_main_b()
{
  vec3 normal0_e = In[0].normal_e;
  vec3 vertex0_e = In[0].vertex_e;
  int w_px = com.win_width;
  int h_px = com.win_height;
```

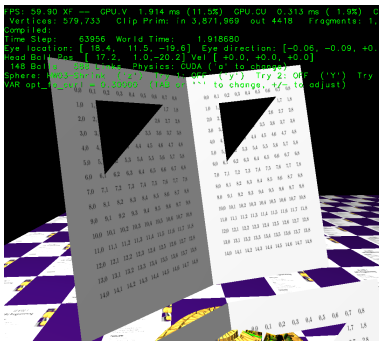
```
  bool is_large = ;

  bool near_edge = ;

  bool split_tri = is_large && near_edge;
  // Assume remainder of geometry shader code is here.
}
```

Problem 2: [30 pts] The code below, once this problem is solved, will show the cards from the pre-final rendered in pairs as shown in the screenshot on the right. Each card shows the entire texture image (once per card). The screenshot on the left shows the original card, and without a texture applied.





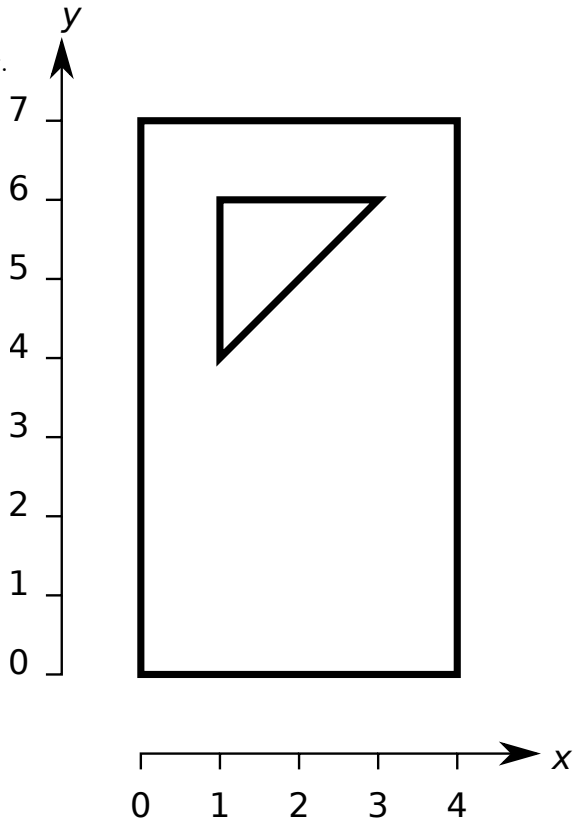
(*a*) First, get the texture right. The code below inserts texture coordinates into the buffer set for the card. Using the diagram show how the texture will appear based on this code. (It won't be blank.) The sampler has been set to repeat the texture.

☐ Using diagram above show texture mapped by code below.

```
World::render_card(vk::CommandBuffer& cb)
{
  auto pC = [&](float x, float y)
   { return pCoor(x,y,0); };
  bset_card.reset( pipe_card );
  bset_card
    << pC(0,7) << pC(1,6) << pC(4,7) << pC(3,6)
    << pC(4,0) << pC(1,4) << pC(0,0) << pC(1,6)
    << pC(0,7);

  for ( pCoor c: bset_cards.pos.vals )
    {
      float x = c.x,  y = c.y;
      bset_cards << pTCoor( x, y );
    }

  bset_cards.to_dev();
```



(*b*) Modify the code below so that it shows the entire texture image on the card, as illustrated on the previous page.

☐ Modify code to show texture image as shown on right image.

```
World::render_card(vk::CommandBuffer& cb)
{
  auto pC = [&](float x, float y) { return pCoor(x,y,0); };
  bset_card.reset( pipe_card );
  bset_card
    << pC(0,7) << pC(1,6) << pC(4,7) << pC(3,6)
    << pC(4,0) << pC(1,4) << pC(0,0) << pC(1,6) << pC(0,7);

  for ( pCoor c: bset_cards.pos.vals )
    {
      // This loop iterates through the coordinates inserted above.
      // That is, on the first iteration c=pCoor(0,7,0), on the second
      // iteration c=pCoor(1,6,0), etc.

      float x = c.x,  y = c.y;




      bset_cards << pTCoor( x, y );
    }

  bset_cards.to_dev();
```

(*c*) Modify the code below so that it shows the second card placed to the left of each original card and rotate 30 degrees. The right edge of the new card should touch the left edge of the old card. Do this by inserting a new transformation matrix into `buf_card_xform`. The code below starts off a new transform, `t2`. It needs to be finished and inserted into the buffer.

☐ Add a transform so that there is a new card to the left of each existing one, as described above.

```
buf_card_xform.clear();
for ( auto& ball: balls ) {
    pMatrix ctform =
      pMatrix_Translate(ball->position)
      * ball->omatrix
      * pMatrix_Translate(pVect(0,ball->radius * 1.5,0))
      * pMatrix_Rotation(pVect(0,1,0),M_PI)
      * pMatrix_Scale( ball->radius / 3.5 );

    buf_card_xform << ctform;

    pMatrix t2 = ctform * pMatrix_Scale(1);









  }

buf_card_xform.to_dev();
pipe_card.storage_bind( buf_card_xform, "BIND_XFORM" );
transform.use_global_for(pipe_card);
pipe_card.record_draw_instanced(cb,bset_card,buf_card_xform.size());
```

Problem 3: [40 pts] Answer each question below.

(*a*) A piece of information that we often compute and send down the rendering pipeline is the eye-space normal, often called `normal_e`. What is that used for, and how would the image be affected if it were set to always point up.

☐ In the pipelines used in class the eye-space normal is used for:

☐ How might an image, say a sphere, appear if `normal_e` always pointed up.

(*b*) In ray tracing, the ray generation shader casts a ray from the eye to each pixel. For each ray the frame buffer is written for the geometry item closest to the eye using the color provided (typically) by a closest-hit shader. In rasterization rendering (the material before ray tracing) how do we insure that the item closest to the eye is written to the frame buffer?

☐ Rasterization rendering ensures the frame buffer is written with an item closest to the eye by:

(*c*) In class we prepared a buffer set holding the vertices of a sphere, and used that for rendering multiple spheres in a scene. We did that for the entire sphere. (This problem is about the tessellated sphere, not the true sphere shader.) That sounds wasteful because the whole sphere is never visible. Why not just store vertices for half of a sphere, say the front half ($-z$). Explain the flaw in the following argument:

One cannot save space by storing vertices for half of a sphere because the eye can be moved to any position around the sphere, and so the eye could move to the part in which the vertices were omitted.

☐ The flaw in the argument above is:

(*d*) How does the shape of the view volume in a rasterization rendering compare to the shape of the view volume using the following ray-generation shader:

```
void main() {
  const vec2 pixel_p = vec2(gl_LaunchIDNV.xy) + vec2(0.5);
  vec2 inUV = pixel_p/vec2(gl_LaunchSizeNV.xy);
  inUV.y = 1 - inUV.y;
  vec2 pixel_c = inUV * 2.0 - 1.0;

  vec4 eye_g = ut.mvi * vec4(0,0,0,1);
  vec4 pixel_e = ut.pi * vec4(pixel_c, 1, 1) ;
  vec4 eye_to_pixel_g = ut.mvi * vec4(normalize(pixel_e.xyz), 0) ;

  uint rayFlags = gl_RayFlagsOpaqueNV, cullMask = 0xff;
  float tmin = 0.001, tmax = 10000.0;

  traceNV
   ( topLevelAS, rayFlags, cullMask,
     0 /* intersect/hit sbtRecordOffset*/,
     0 /* intersect/hit sbtRecordStride*/,
     0 /*missIndex*/,
     eye_g.xyz, tmin, eye_to_pixel_g.xyz, tmax,
     0 /*payload location */);

  imageStore(fb_image, ivec2(gl_LaunchIDNV.xy), vec4(rp_color, 0.0));
}
```

☐ How do the shapes compare. Be specific, use a sketch.

(*e*) CPU code is compiled ahead of time. In a production setting, it is compiled by the developers and shipped in binary form to the customer. But shader code is typically compiled when the program is run. Why?

☐ Why is it important for shader code to be compiled at run time on the user's system while its okay for CPU code to be compiled in advance by the developer?

(*f*) Our true sphere shader enabled us to render a perfect sphere. For each sphere one vertex was sent down the pipeline. The geometry shader emitted a rectangle framing the sphere, and the fragment shader computed which point on the sphere is projected on to that fragment. A true cube shader could be designed along the same lines. Would it improve things over the conventional method of rendering cubes? (The conventional method is 12 triangles, perhaps using 6 triangle strips.)

☐ Would a true cube shader make a better-looking cube than the conventional rendering methods? ☐ Explain.

☐ Would a true cube shader reduce computation over the conventional rendering method? ☐ Explain.

(*g*) A triangle strip is rendered with different colors, perhaps our card:

```
bset_card.reset( pipe_card );
bset_card
  << pC(0,7) << pC(1,6) << pC(4,7) << pC(3,6)
  << pC(4,0) << pC(1,4) << pC(0,0) << pC(1,6) << pC(0,7);

bset_card
  << color_red << color_white << color_green << color_orange
  << color_blue << color_lsu_spirit_purple << color_lsu_spirit_gold
  << color_turquoise << color_gray;
```

The interface block for the fragment shader input uses the `flat` interpolation qualifier for color:

```
layout ( location = 0 ) out Data_to_FS {
  vec3 normal_e;
  vec4 vertex_e;
  vec2 tex_coor;
  flat vec4 color;
};
```

☐ How would the appearance differ if the `flat` qualifier were omitted? Illustrate with a portion, not the whole thing.

6 Fall 2020

Name _____

GPU Programming
EE 4702-1
Take-Home Pre-Final Examination

Due: 5 December 2020 at 23:59 CST

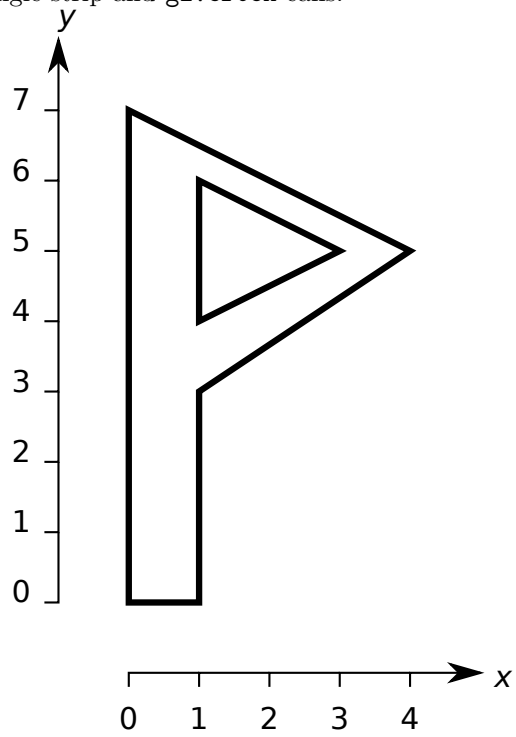
Work on this exam alone. Regular class resources, such as notes, papers, solutions, documentation, and code, can be used to find solutions. In addition outside OpenGL references and tutorials, other programming resources, and mathematical references can be consulted. Do not try to seek out references that specifically answer any question here. **Do not discuss this exam with classmates or anyone else**, except questions or concerns about problems should be directed to Dr. Koppelman.

Warning: Unlike homework assignments collaboration is not allowed on exams. Suspected copying will be reported to the dean of students. The kind of copying on a homework assignment that would result in a comment like "See ee4702xx for grading comments" will be reported if it occurs on an exam. Please do not take advantage of pandemic-forced test conditions to cheat!

Problem 1	_____	(20 pts)
Problem 2	_____	(20 pts)
Problem 3	_____	(30 pts)
Problem 4	_____	(30 pts)
Exam Total	_____	(100 pts)

Good Luck!

Problem 1: [20 pts] Appearing below is an illustration of a pointy p and an incomplete routine. Complete the routine so that it emits a green pointy p using a single triangle strip and `glVertex` calls.



☐ Add code to start and stop a triangle-strip rendering pass.

☐ Emit the shape using one—just one—triangle strip. Use `glV` as an abbreviation for `glVertex3f`.

☐ Use the coordinates given in the diagram. Omit the z coordinate.

☐ Set a correct normal and color.

```
void render_p()
{



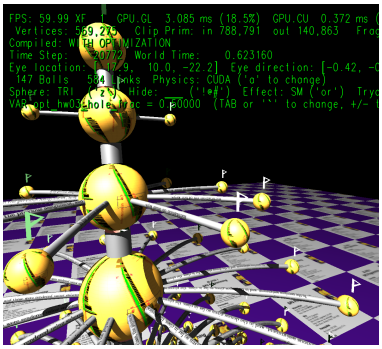

}
```

Problem 2: [20 pts] Consider `render_p` from the previous problem.

(*a*) Compute the amount of data sent from the CPU to the GPU each time `render_p` is called. If you did not solve the previous problem then assume that `render_p` calls `glVertex3f` eight times.

☐ Amount of data due to execution of `render_p`. ☐ Show units and state assumptions.

(*b*) The code below calls `render_p` for each ball. Before calling `render_p` the modelview matrix is modified so that the pointy p is rendered above the sphere. Suppose that the updated modelview matrix is sent to the GPU at the beginning of a rendering pass.



```
void World::render_prob1() {
  glMatrixMode(GL_MODELVIEW);
  for ( Ball *ball: balls ) {
      glPushMatrix();
      pNorm b_to_eye(ball->position,eye_location);
      pVect d_p = ball->omatrix * pVect(0,1,0);
      pCoor p_pos = ball->position + 1.5 * ball->radius * d_p;
      pVect az = b_to_eye;
      pNorm ax = cross(b_to_eye,pVect(0,-1,0));
      pVect ay = cross(az,ax);
      float s = 0.2 * ball->radius;
      pVect vx = s * ax, vy = s * ay, vz = s * az;
      pMatrix xform(vx,vy,vz,p_pos); // Concatenate vectors into a matrix.
      glMultTransposeMatrixf(xform);
      render_p();
      glPopMatrix();
    }}
```

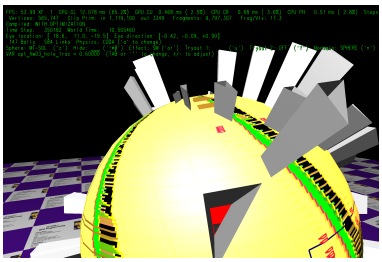
Consider an alternative implementation where the vertex shader computes and uses the updated modelview matrix rather than having it computed by the routine above.

☐ How much data (in bytes) would this vertex shader invocation need to read in order compute the new matrix?

☐ Explain why the total amount of computation using this vertex shader would be larger.

☐ Explain why, even if the total computation were larger, execution time might still be faster.

Problem 3: [30 pts] The shader code in routine `gs_monolith` below is based on the solution to Homework 3, in which the sphere was to be rendered with triangular holes.



(*a*) Modify the code so that there is a prism-shaped gray monolith over each hole. Render only the sides of the prisms, not the ends. A viewer looking down the axis of a prism should be able to see through the prism and into the sphere, perhaps to check whether it's full of stars. See the screenshot to the right. (Sorry, no stars.)

☐ Emit the prism faces. (The coordinates of one end of the prism are the same as the coordinates of the hole.)

☐ Determine the normal of each prism face, and emit the normal. ☐ All fragments of a face must have the same normal.

☐ Correctly set `max_vertices`. Don't set it larger than needed.

(*b*) The problem above could be solved using one triangle strip to render the prism, or using three triangle strips (one for each face). For the one-triangle-strip solution there would be a problem with the normals. Explain what that problem is, and how it might be fixed by modifying the fragment shader. (Not shown.)

☐ Problem with normals with a one-triangle-strip prism.

☐ Key ideas (but not complete code) to fix normals by modifying fragment shader.

```
layout ( triangle_strip, max_vertices = 8 ) out;
void gs_monolith() {
  vec4 ctr_ce = AVG(vertex_e);       // Coord at triangle center.
  vec3 ctr_ne = AVG(normal_e);       // Normal at triangle center.
  vec2 ctr_tx = AVG(gl_TexCoord[0]); // Texture coord at triangle center.

  const float f = tryoutf.y; // Relative hole size.
  color = In[0].color;

  // Render the triangle-with-a-hole using a triangle strip that wraps
  // around the hole in the triangle.
  for ( int ii=0; ii<=3; ii++ ) {
      int i = ii % 3;  // Values of i:  0, 1, 2, 0.  (0 appears twice.)
      vec3 n_e = In[i].normal_e;

      // Prepare and emit inner vertex by blending outer vertex and center.
      normal_e =       f * n_e                  + (1-f) * ctr_ne;
      vertex_e =       f * In[i].vertex_e       + (1-f) * ctr_ce;
      gl_TexCoord[0] = f * In[i].gl_TexCoord[0] + (1-f) * ctr_tx;
      gl_Position = gl_ProjectionMatrix * vertex_e;
      EmitVertex();

      // Prepare and emit vertex of original triangle.
      normal_e = n_e;
      vertex_e = In[i].vertex_e;
      gl_TexCoord[0] = In[i].gl_TexCoord[0];
      gl_Position = In[i].gl_Position;
      EmitVertex();
    }
  EndPrimitive();

  color = vec4(0.5,0.5,0.5,1); // Change color to gray.
  // SOLUTION GOES HERE.









}
```

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

(*a*) A program uses OpenGL and includes shader code. That program runs fine on a system using a GPU. Is it possible in principle to run that same code on a system without a GPU? The system still has a display which can show graphics. Explain why or why not.

☐ Can the program run on the system without a GPU?

☐ Explain why it can run or why it cannot ever run.

(*b*) In class we described how vertices could be assigned different kind of colors (material properties) such as diffuse and specular using OpenGL calls such as `glMaterial`. From these a lighted color would be computed, accounting for the material properties and also light properties. Do current GPUs have special hardware to compute this lighted color? Explain.

☐ Do current GPUs (not just Ampere) have hardware to compute lighted colors? ☐ Explain.

☐ Why was it necessary for OpenGL to have calls such as `glMaterial`, `glLight`, and so on?

(*c*) Appearing below are three variations on a geometry shader. They are part of a rendering pipeline that also includes a fragment shader. The shader labeled `Original` is written clearly. A bored programmer came up with the `Stunt A` and `Stunt B` versions. Suppose that all variables are of type `vec4`, and that the names of the input variables accurately describe their contents. Explain whether each of the Stunt variations can be made to work by modifying the fragment shader.

```
void gs_typical() {  // Original
  for ( int i=0; i<3; i++ ) {
      gl_FrontColor = In[i].color;
      gl_BackColor = In[i].color;
      gl_Position = In[i].gl_Position;
      gl_TexCoord[0] = In[i].gl_TexCoord[0];
      EmitVertex();
    }
  EndPrimitive();
}
```

```
void gs_typical() { // Stunt A
  for ( int i=0; i<3; i++ ) {
      gl_FrontColor = In[i].color;
      gl_BackColor = In[i].gl_TexCoord[0];
      gl_Position = In[i].gl_Position;
      gl_TexCoord[0] = In[i].color;
      EmitVertex();
    }
  EndPrimitive();
}
```

```
void gs_typical() {  // Stunt B
  for ( int i=0; i<3; i++ ) {
      gl_FrontColor = In[i].color;
      gl_BackColor = In[i].gl_TexCoord[0];
      gl_Position = In[i].color;
      gl_TexCoord[0] = In[i].gl_Position;
      EmitVertex();
    }
  EndPrimitive();
}
```

☐ `Stunt A` ◯ *can be made to work* or ◯ *cannot be made to work* .

☐ Explain why or why not.

☐ `Stunt B` ◯ *can be made to work* or ◯ *cannot be made to work* .

☐ Explain why or why not.

(*d*) The code fragment below, taken from the classroom demo-7 code, updates the buffer object when `gpu_buffer_stale` is `true`. Variable `gpu_buffer_stale` is set to `true` initially, and again only when something about the sphere changes (for example, the number of triangles used to approximate it). The line commented `DISASTER` was added for this problem. Explain what will go wrong. *Hint: The problem occurs when* `gpu_buffer_stale` *is frequently true. Note: This problem also appeared on the 2019 final exam.*

```
if ( gpu_buffer_stale ) {
    gpu_buffer_stale = false;

    // Generate buffer id (name), if necessary.
    if ( !gpu_buffer ) glGenBuffers(1,&gpu_buffer); // ORIGINAL
    glGenBuffers(1,&gpu_buffer);                    // DISASTER

    // Tell GL that subsequent array pointers refer to this buffer.
    glBindBuffer(GL_ARRAY_BUFFER, gpu_buffer);

    // Copy data into buffer.
    glBufferData
      (GL_ARRAY_BUFFER,           // Kind of buffer object.
       coords_size*sizeof(pCoor), // Amount of data (bytes) to copy.
       sphere_coords.data(),      // Pointer to data to copy.
       GL_STATIC_DRAW);           // Hint about who, when, how accessed.

    // Tell GL that subsequent array pointers refer to host storage.
    glBindBuffer(GL_ARRAY_BUFFER, 0);
}
```

☐ Explain the disaster that the `DISASTER` line causes.

(*e*) A vertex shader appears below, it is based on code from `links-shdr.cc` in the links demo and is used to render spheres using an instanced rendering pass. If there are 20 spheres in the scene and each sphere has 300 vertices, the vertex shader is called $20 \times 300 = 6000$ times. Label the variables in the routine below as follows:

☐ Label each OpenGL defined uniform variable with a U.

☐ Label each OpenGL defined input variable with an I.

☐ Label each OpenGL defined output variable with an O.

```
void vs_main_instances_sphere() {
  vec4 pos_rad = sphere_pos_rad[gl_InstanceID];
  float rad = pos_rad.w;
  mat4 rot = transpose(sphere_rot[gl_InstanceID]);
  vec4 normr = rot * gl_Vertex;
  normal_o = normr.xyz;
  vertex_o = vec4( pos_rad.xyz + rad * normal_o, 1 );

  gl_Position = gl_ModelViewProjectionMatrix * vertex_o;
  gl_TexCoord[0] = gl_MultiTexCoord0;
  color = sphere_color[gl_InstanceID];
}
```

(*f*) The storage type of `sphere_pos_rad` in `vs_main_instances_sphere` from the previous subproblem could be a shader input, a uniform variable, or a buffer object. Assume that all could be made to work correctly under some circumstances.

☐ Would it be ◯ *good* , ◯ *doable* , or ◯ *bad* for `sphere_pos_rad` to be a shader input? ☐ Explain. If bad, how bad.

☐ Would it be ◯ *good* , ◯ *doable* , or ◯ *bad* for `sphere_pos_rad` to be a uniform variable? ☐ Explain. If bad, how bad.

☐ Would it be ◯ *good* , ◯ *doable* , or ◯ *bad* for `sphere_pos_rad` to be a buffer object? ☐ Explain. If bad, how bad.

Name _____

GPU Programming
LSU EE 4702-1
Solve-Home Final Examination

Wednesday 9 December to Friday 11 December 2020 16:30 CST

Work on this exam alone. Regular class resources, such as notes, papers, solutions, documentation, and code, can be used to find solutions. In addition outside OpenGL references and tutorials, other programming resources, and mathematical references can be consulted. Do not try to seek out references that specifically answer any question here. **Do not discuss this exam with classmates or anyone else**, except questions or concerns about problems should be directed to Dr. Koppelman.

Warning: Unlike homework assignments collaboration is not allowed on exams. Suspected copying will be reported to the dean of students. The kind of copying on a homework assignment that would result in a comment like "See ee4702xx for grading comments" will be reported if it occurs on an exam. Please do not take advantage of pandemic-forced test conditions to cheat!

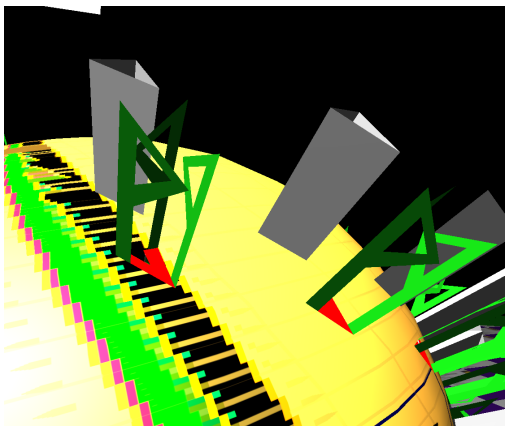
Problem 1	_____	(30 pts)
Problem 2	_____	(15 pts)
Problem 3	_____	(15 pts)
Problem 4	_____	(40 pts)
Exam Total	_____	(100 pts)

Good Luck!

Problem 1: [30 pts] The geometry shader below is based on the solution to the pre-final exam Problem 3. The geometry shader renders a prism centered on the triangle with a hole cut in it. The prism is also called a monolith in a playful reference to a recent news story. Notice that there is a `vec2` array, `pts`, declared in the shader. This array has the coordinates of the points on the pointy p from Pre-Final Exam Problem 1. The points are in the pointy p's local coordinate space and are in the right order to render the p as a triangle strip.

Modify the shader so that p's are rendered instead of monolith faces. Each p must be on the same plane as the monolith face and must be the same size. That is, the p can't extend outside where a face would be, the bottom, top and left edge must touch the respective edges of the face. The pointy part must touch or be near the right edge.

See the screenshot to the right. In the screenshot some holes have monoliths, and some have p's. In your solution just render p's.



☐ Modify the shader to render the p's where the monolith faces would go.

```
void gs_mono_p() {
  vec4 ctr_ce = AVG(vertex_e);       // Coord at triangle center.
  vec3 ctr_ne = AVG(normal_e);       // Normal at triangle center.

  // Render the triangle-with-a-hole using a triangle strip that wraps around hole.
  for ( int ii=0; ii<=3; ii++ ) {
      int i = ii % 3;
      normal_e =       f * In[i].normal_e       + (1-f) * ctr_ne;
      vertex_e =       f * In[i].vertex_e       + (1-f) * ctr_ce;
      gl_Position = gl_ProjectionMatrix * vertex_e;
      EmitVertex();     // Emit inner triangle vertex.
      normal_e = In[i].normal_e;
      vertex_e = In[i].vertex_e;
      gl_Position = In[i].gl_Position;
      EmitVertex();     // Emit original triangle vertex.
    }
  EndPrimitive();

  float size = length(In[0].vertex_e-In[1].vertex_e); // Height of monolith.
  color = vec4( 0.05, 0.5, 0.05, 1 );

  // Compute and save the coordinates of the top of the monolith, and surface normals.
  vec3 vtx_e[3], snorm_e[3];
  for ( int i=0; i<3; i++ ) {
      snorm_e[i] = size * normalize( f * In[i].normal_e + (1-f) * ctr_ne );
      vtx_e[i] = f * In[i].vertex_e.xyz + (1-f) * ctr_ce.xyz; }

  // Local x and y coordinates of the points on the pointy p.
  vec2 pts[] = { {0,0},{1,0},  {0,7},{1,6},  {4,5},{3,5},  {1,3},{1,4} };

  for ( int i=0; i<3; i++ ) {  // Iterate over faces of monolith.
      int ni = ( i + 1 ) % 3;  // Next i, possibly wrapped around.

      // Compute the normal of a face of the monolith.
      normal_e = normalize( cross( vtx_e[ni] - vtx_e[i], snorm_e[i].xyz ) );

      // Use a triangle strip to emit one face of the monolith.
      vec3 va[4] = { vtx_e[i], vtx_e[ni], vtx_e[i] + snorm_e[i], vtx_e[ni] + snorm_e[ni] };
      for ( int j=0; j<4; j++ ) {
          vertex_e.xyz = va[j];
          gl_Position = gl_ProjectionMatrix * vertex_e;
          EmitVertex();
        }
      EndPrimitive();
    }
}
```

Problem 2: [15 pts] The vertex and geometry shaders below are used in rendering pipeline T, in which the input primitives are individual triangles, and rendering pipeline S, in which the input primitive is a triangle strip. In both cases the shaders work correctly, but there might be differences in performance.

(a) Modify the vertex and geometry shader to reduce the amount of data sent from the vertex shader to the geometry shader. Do so by moving some of the work performed by the vertex shader to the geometry shader. If necessary, declare new input and output variables. For this part the shaders will be run on pipeline S. Do not make changes that result in additional computation unless those changes also reduce vertex-to-geometry shader data transfer. Of course, your changes should not change what the shaders do.

☐ Modify shaders to reduce vertex-to-geometry shader data transfer.

☐ Don't make a change the has no impact on data transfer but does increase the amount of computation.

```
void vs_main_basic() {  // The Vertex Shader
  gl_Position = gl_ModelViewProjectionMatrix * gl_Vertex;
  vertex_e = gl_ModelViewMatrix * gl_Vertex;
  normal_e = normalize(gl_NormalMatrix * gl_Normal);
  gl_BackColor = gl_FrontColor = gl_Color;
  tex_coord = gl_MultiTexCoord0.xy;
}
```

```
void gs_main_basic() {  // The Geometry Shader
  const bool type_a = glPrimitiveIDIn & 1;
  vec4 color_adjust = type_a ? vec4(0.5,0.5,0.5,1) : vec4(1);

  for ( int i=0; i<3; i++ )
    {
      gl_FrontColor = gl_FrontColorIn[i] * color_adjust;
      gl_BackColor =  gl_BackColorIn[i] * color_adjust;
      gl_Position = gl_PositionIn[i];
      tex_coord = In[i].tex_coord;
      normal_e = In[i].normal_e;
      vertex_e = In[i].vertex_e;
      EmitVertex();
    }
  EndPrimitive();
}
```

(b) The requirement to *not make changes that result in additional computation unless those changes also reduce vertex-to-geometry shader data transfer* is much easier to comply with when the shaders are run on pipeline T. Explain why and include an example of such a change. The change should not change what the shaders do and should be useful. (That is, move something.)

☐ Why is it easier to avoid additional computation when the shaders are used in pipeline T than it is when the same shaders are used in pipeline S? ☐ Note: Don't compare the absolute performance of S to T, compare how much the change impacts computation on each pipeline.

☐ Provide an example.

Problem 3: [15 pts] In Homework 3 we experimented with two ways to render a triangle with a hole in it. In the one-triangle method we relied on the fragment shader to render the hole. The geometry shader just emitted one triangle with little work. In the many-triangle method we emitted several triangles, forming a triangle-with-a-hole shape.

Based on performance measurements, we found that the one-triangle method was faster. That must mean that the extra work done by the geometry shader in the many-triangle method had more of an impact than the extra work done by the fragment shader in the one-triangle method.

Let t_{g1} denote the time used by one invocation of the geometry shader for the one-triangle method, let t_{gm} denote the time used by one invocation of the fragment shader for the many-triangles method. Let t_{f1} and t_{fm} denote the times for one invocation of the respective fragment shaders.

Let n denote the number of spheres rendered, and let g denote the number of triangles in one sphere tessellation. Finally, let f denote the fraction of the triangle covered by the hole.

(*a*) Based on these, find an expression for the time used by the geometry shaders for each method.

☐ Time for geometry shader using one-triangle method:

☐ Time for geometry shader using many-triangle method:

(*b*) Find an expression for the time used by the fragment shaders. Use n_{f1} for the total number of fragment shader invocations in the one-triangle method. (But use it for both expressions.)

☐ Time for fragment shader using one-triangle method:

☐ Time for fragment shader using many-triangle method:

(*c*) What does n_{f1} depend on? How can n_{f1} be made larger or smaller when viewing a scene without changing the primitives sent into the rendering pipeline. That is, one can't send fewer spheres or more spheres into the rendering pipeline.

☐ Quantity n_{f1} depends on:

☐ It can be changed when viewing a scene by:

Problem 4: [40 pts] Answer each question below.

(*a*) The two shaders below do the same thing, though slightly differently.

```
void vs_plan_a() {
  vertex_e = gl_ModelViewMatrix * gl_Vertex;
  gl_Position = gl_ProjectionMatrix * vertex_e;
}

void vs_plan_b() {
  vertex_e = gl_ModelViewMatrix * gl_Vertex;
  gl_Position = gl_ModelViewProjectionMatrix * gl_Vertex;
}
```

OpenGL is provided a modelview matrix and a projection matrix at the beginning of a rendering pass. In both `vs_plan_a` and `vs_plan_b` there are two matrix/vector multiplies, which require $4^2 = 16$ multiply/add operations each. But `vs_plan_b` uses `gl_ModelViewProjectionMatrix`, which is the product of the modelview and projection matrices. The product of these two matrices is computed using $4^3 = 64$ multiply/add operations. That brings the total to $16 + 16 + 64 = 96$ operations, much more than 32 for `vs_plan_a`, right?

☐ Describe the flaw with this argument.

☐ Describe a case when the argument is correct, but explain why this case does not reflect typical use.

(*b*) Answer the following questions about view volumes.

☐ What is a view volume?

☐ It is easy to determine whether a vertex is in the view volume by using its coordinate in ◯ *object space*, ◯ *eye space*, or ◯ *clip space* (check one).

☐ Given the coordinate in that space, how can one tell whether it is inside or outside the view volume?

☐ It is easy to determine whether some triangles are in the view volume. ☐ Provide an example of such a triangle and ☐ explain why.

☐ Provide an example of a triangle for which it is not so easy to determine if it is in the view volume. ☐ Illustrate with a diagram.

(*c*) Describe how suitable an OpenGL uniform variable is for the following:

☐ Explain whether this a suitable use for a uniform variable: To hold the lighted color computed by a vertex shader.

☐ Explain whether this a suitable use for a uniform variable: To hold the location of a light source.

(*d*) Vertex coordinates are usually three dimensional but texture coordinates are usually two dimensional. Why? (Ignore the w component in your answer.)

☐ Texture coordinates have two, not three, dimensions because:

(*e*) In many of our sphere examples we put the coordinates into a buffer object.

☐ What are the advantages of a buffer object over using individual `glVertex` calls to feed a rendering pipeline?

☐ What are the advantages of a buffer object over using a client (CPU) array to feed a rendering pipeline?

(*f*) A homogeneous coordinate consists of four components, compared to just three for ordinary Cartesian coordinates. Homogeneous coordinates increase the amount of work needed for a matrix/vector multiply from 9 to 16 multiplications. Transformations are realized by multiplying a transformation matrix by a coordinate.

☐ Describe a transformation that cannot be done without homogeneous coordinates.

☐ Describe a transformation that can be done using ordinary Cartesian coordinates.

7 Fall 2019

Name _____

GPU Programming
EE 4702-1
Take-Home Pre-Final Examination

Due: 5 December 2019 at 16:30

Work on this exam alone. Regular class resources, such as notes, papers, solutions, documentation, and code, can be used to find solutions. In addition outside OpenGL references and tutorials, other programming resources, and mathematical references can be consulted. Do not try to seek out references that specifically answer any question here. **Do not discuss this exam with classmates or anyone else**, except questions or concerns about problems should be directed to Dr. Koppelman.

Problem 1	_____	(30 pts)
Problem 2	_____	(25 pts)
Problem 3	_____	(5 pts)
Problem 4	_____	(25 pts)
Problem 5	_____	(15 pts)
Exam Total	_____	(100 pts)

Alias _____

Good Luck!

Problem 1: [30 pts] The `gs_points` shader in the solution to Homework 3, Problem 2 renders protrusions, which are blocks attached to the ring. The shader always emits five faces: the top (which we called the diamond), and the four sides. (Since it is attached to the ring there is no need to emit a bottom face.) Normally only the outside of the block is visible so the user can see at most three faces. That means at least three faces that can't be seen are emitted nevertheless. That can't be good. In this problem shader code will be modified to avoid those faces, and in the next problem the implications of this change will be analyzed.

The code on the next page is based on (but not identical to) the Homework 3 Problem 2 solution. The intention is that this be solved by hand, though solving by editing code is allowed. The code from the next page will be provided on request or may already be available.

(*a*) In the code below (meaning the next page) there is a 4-element Boolean array `visible` that is declared but not used. Add code that sets `visible[i]` to `true` is side `i` is visible to the user. By no coincidence the declaration of `visible` is just after code that sets `face_normal_e[i]` to the eye-space normal of side `i`. *Hint: Remember that in eye space the eye is at the origin (coordinate* $(0, 0, 0)$*). Also take advantage of the fact that eye-space coordinates and normals are available. Another Hint: A correct solution to this problem consists of a loop with a single statement,* `visible[i]=.x.`*, with* `.x.` *replaced by [am I giving away too much] a dot product.*

☐ Modify code so `visible[i]` set to `true` iff face `i` is visible.

(*b*) Modify the code below so that only the visible faces are emitted. This includes the diamond on top and the four sides of the protrusion. Feel free to use the `visible` array from the previous subpart (even is that subpart has not been solved). *Hint: Only one line is needed for the diamond on top. The changes needed to the code emitting the sides are more extensive.*

☐ Modify code so that diamond is not emitted if it is not visible.

☐ Modify code so that only visible sides are emitted.

Code is on the next page.

```
void gs_points() {
  /// Note: (Code setting ctop, etc. not shown.)

  // Prepare an array of object-space coordinates of vertices
  // describing the protrusion, and use that to initialize an array of
  // eye-space coordinates.
  vec3 pts_o[8] =
    { ctop,       cr,       cbot,       cl,   ctop+nc*dr, cr+nr*dr, cbot+nc*dr, cl+nr*dr };
  vec4 pts_e[8];
  for ( int i=0; i<8; i++ ) pts_e[i] = gl_ModelViewMatrix * vec4(pts_o[i],1);
  int ord[4] = { 1, 0, 2, 3 }; // Order in which to emit vertices.

  // Emit the diamond-shaped protrusion top.
  color = color_diamond;
  normal_e = gl_NormalMatrix * nl;


  for ( int i=0; i<4; i++ ) {
      vertex_e = pts_e[ord[i]];  // Retrieve vertices in the correct order.
      gl_Position = gl_ProjectionMatrix * vertex_e;
      EmitVertex();  }
  EndPrimitive();

  // Compute the eye-space normal of each protrusion face.
  vec3 face_normal_e[4];
  for ( int i0=0; i0<4; i0++ )
    face_normal_e[i0] = cross( pts_e[(i0+1) & 0x3].xyz - pts_e[i0].xyz,
                               pts_e[i0+4].xyz - pts_e[i0].xyz );

  bool visible[4];




  color = color_edge; // Color for sides.
  for ( int i=0; i<5; i++ )
    {
      int i0 = i & 0x3;          // The current edge.



      vertex_e = pts_e[i0+4];  gl_Position = gl_ProjectionMatrix * vertex_e;
      EmitVertex();
      vertex_e = pts_e[i0];   gl_Position = gl_ProjectionMatrix * vertex_e;
      EmitVertex();
      normal_e = face_normal_e[i0]; // This normal will be used for the next two vertices.
    }
 EndPrimitive();
}
```

Problem 2: [25 pts] Assume that the previous problem has been solved correctly and so only those sides of the protrusions facing the user are emitted. Below, roughly estimate the benefit of this improvement when rendering protrusions by considering the amount of work done by the vertex, geometry, and fragment shaders. Make the estimate using the original (Homework 3 solution) vertex and fragment shaders, and the geometry shader obtained by solving the previous problem. When estimating, assume that the number of emitted faces per protrusion drops from 5 to 2.5. (The actual number of faces emitted for a particular protrusion with a particular eye location and orientation can range from zero [for example, if the viewer is in the center of the ring looking at some point on the ring] to three.)

When answering the problems below consider the change in the amount of work done by each shader (the change in the amount of work using the Homework 3 solution code to the amount of work using the geometry shader from the previous problem) invocation, and the change in the number of times the shader needs to be invoked.

(*a*) For this part assume a typical view of the ring. (Don't assume that no part of the ring is visible, or other special cases.)

☐ Compared with the Homework 3 solution code, the total work for the vertex shader stage after the change above is: ◯ *about half the amount of work,* ◯ *no change in the amount of work,* ◯ *roughly the same amount of work,* ◯ *double the amount of work.*

☐ Explain. Consider ☐ work per shader and ☐ number of shader invocations.

☐ Compared with the Homework 3 solution code, the total work for the geometry shader stage after the change above is: ◯ *about half the amount of work,* ◯ *no change in the amount of work,* ◯ *roughly the same amount of work,* ◯ *double the amount of work.*

☐ Explain. Consider ☐ work per shader and ☐ number of shader invocations.

☐ Compared with the Homework 3 solution code, the total work for the fragment shader stage after the change above is: ◯ *about half the amount of work,* ◯ *no change in the amount of work,* ◯ *roughly the same amount of work,* ◯ *double the amount of work.*

☐ Explain. Consider ☐ work per shader and ☐ number of shader invocations.

(*b*) Suppose again that the ring is visible, and suppose that each invocation of the new geometry shader requires 10% more work than the geometry shader from the Homework 3 solution. (The extra time is needed to detect invisible faces.) What sort of view would make that extra work increase the total time needed for a rendering pass, and would make the extra work offset by a large reduction elsewhere?

☐ Characteristics of a view that would increase rendering time even with invisible faces omitted?

☐ Characteristics of a view that would sharply reduce rendering time with invisible faces omitted?

Problem 3: [5 pts] Appearing below is a correct solution to Homework 3 Problem 1, followed by a possible alternative solution. Indicate whether the alternative solution will render the strip as intended (the same way as the first solution) and whether there is any problem with it.

```
void gs_strip_plus() {  // Homework 3 Solution
  if ( In[2].normal_e == vec3(0,0,0) ) return;
  for ( int i=0; i<3; i++ )
    {
      normal_e = In[i].normal_e;
      vertex_e = In[i].vertex_e;
      color = In[i].color;
      gl_Position = In[i].gl_Position;
      EmitVertex();
    }
  EndPrimitive();
}
```

```
void gs_strip_plus() {  // A possible alternative solution.
  for ( int i=0; i<3; i++ )
    {
      if ( In[2].normal_e == vec3(0,0,0) ) return;
      normal_e = In[i].normal_e;
      vertex_e = In[i].vertex_e;
      color = In[i].color;
      gl_Position = In[i].gl_Position;
      EmitVertex();
    }
  EndPrimitive();
}
```

☐ Will the alternative solution render the strip correctly?

☐ Explain the problem with the alternative solution.

Problem 4: [25 pts] Answer each question below.

(*a*) Any uniform variable that a vertex shader uses can be replaced by a vertex shader input. Why is that not a good idea? Explain the impact on performance.

☐ Uniform variables should not be replaced by shader inputs because:

☐ How might one estimate the impact on performance for such a replacement?

(*b*) In some cases a vertex shader input can be replaced by a uniform variable. Describe a situation in which a vertex shader input can be replaced by a uniform, and in which a shader input cannot be replaced by a uniform. An example can be found in the shaders for Homework 3 and 4.

☐ Situation where vertex shader input can be replaced by a uniform. ☐ Explain.

☐ Situation where vertex shader input **can not** be replaced by a uniform. ☐ Explain.

(*c*) The geometry shader must write clip-space coordinates to output variable `gl_Position`. Anything else the geometry shader needs to write, such as a surface normal, can be written to variables of the programmer's choosing. Why does OGSL require clip-space coordinates in that way?

☐ Why must the geometry shader write clip-space coordinates?

☐ Why must it write them to a variable with a specific name, `gl_Position`?

(*d*) The true sphere shader presented in class rendered a sphere perfectly. Why would there be no advantage in a true cube shader?

☐ No advantage to a true cube shader because:

(*e*) The shadow volume code and our lighting code took advantage of built in variables describing how user code set lighting state such as
`gl_LightSource[i].position`, which provides the position of light `i`. Light position and other lighting variables are deprecated, meaning they may be removed in future versions of OpenGL.

☐ Why are they deprecated?

☐ Why were they there in the first place?

Problem 5: [15 pts] Many of our fragment shaders make library function calls such as `texture(tex_unit_0,TexCoord)` to obtain a filtered texel located at coordinate `TexCoord`. Coordinate `TexCoord` is something the shader code provided. Consider an alternative method of obtaining a texel in which the user prepares an array of texels bound to a buffer object named `my_texture`. The fragment shader might obtain the texel using code such as `my_texel[twid*TexCoord.x + twid*tht*TexCoord.y]`, where `twid` is the width of the texture image and `tht` is the height of the image.

(*a*) Explain why the code fragment `my_texel[twid*TexCoord.x + twid*tht*TexCoord.y]` might return a texel, but not a filtered texel. In particular, what would need to be done for it to be filtered, and why could it not provide the benefit of mipmap levels. Remember that `my_texel` is an ordinary array of RGBA values from a texture image (called texels). Each element is a `vec4`, the `vec4` members are `.r`, `.g`, `.b`, and `.a`.

☐ `my_texel[twid*TexCoord.x + twid*tht*TexCoord.y]` could not return a filtered texel because:

☐ Roughly, how could code accessing the texel array `my_texel` return a filtered texel?

☐ Explain why accessing the single texel array could not provide the benefit of multiple mipmap levels.

(*b*) The library function `texture` is part of the OpenGL shading language. Unlike `gl_LightSource`, `texture` is not deprecated, meaning that it won't be removed from OpenGL anytime soon. Why is it that lighting-related features are deprecated while texture accesses and filtering are not deprecated?

☐ The `texture` function and most other texture-related features are not deprecated because:

Name _____

GPU Programming
LSU EE 4702-1
Take-Home Final Examination

Sunday 8 December to Wednesday 11 December 2019 16:30 CST

Work on this exam alone. Regular class resources, such as notes, papers, solutions, documentation, and code, can be used to find solutions. In addition outside OpenGL references and tutorials, other programming resources, and mathematical references can be consulted. Do not try to seek out references that specifically answer any question here. **Do not discuss this exam with classmates or anyone else**, except questions or concerns about problems should be directed to Dr. Koppelman.

Problem 1	_____	(25 pts)
Problem 2	_____	(20 pts)
Problem 3	_____	(20 pts)
Problem 4	_____	(10 pts)
Problem 5	_____	(25 pts)
Exam Total	_____	(100 pts)

Alias _____

Good Luck!

Problem 1: [25 pts] Appearing below is code based on the solution to Homework 4, in which the Strip-Plus shader is modified so that the start of a triangle strip is indicated by setting the w component of the first two vertices to zero.

```
glBegin(GL_TRIANGLE_STRIP);

// Render the diamond top.
glColor3fv(color_tan);
for ( auto& e: hw04.bump_info ) {   // Note: n iterations.
    pCoor vtop(e.ctop), vl(e.cl);
    vtop.w = vl.w = 0;
    glVertex4fv(vtop);    glVertex4fv(vl);
    glVertex4fv(e.cr);    glVertex4fv(e.cbot);  }

// Render the sides.
glColor3fv(color_olive_drab);
for ( auto& e: hw04.bump_info ) {   // Note: n iterations.

    pCoor pts[2][4] =                    // Construct an array of coordinates.
      {  { e.ctop, e.cr, e.cbot, e.cl }, // Original Diamond
           // Points "below" diamond on cylinder surface.
         { e.ctop+e.nc*dr, e.cr+e.nr*dr, e.cbot+e.nc*dr, e.cl+e.nr*dr } };

    // Render Block's Sides
    for ( int i=0; i<5; i++ ) {
        const int i0 = i & 0x3;
        pCoor v1 = pts[1][i0], v2 = pts[0][i0];
        if ( !i ) v1.w = v2.w = 0;
        glVertex4fv( v1 ); glVertex4fv( v2 );  }
  }
glEnd();
```

(a) Appearing above is the CPU code performing a rendering pass for the protrusions. Compute the amount of data sent from the CPU to the GPU for one rendering pass. The amount of data should be in units of bytes and in terms of n, the number of protrusions. (The `hw04.bump_info` container holds n elements.)

☐ Amount of data per pass in units of bytes in terms of n:

(b) For a rendering pass using the code above rendering n protrusions, what is the number of vertex shader and geometry shader invocations.

☐ Number of vertex shader invocation in terms of n:

☐ Number of geometry shader invocation in terms of n:

(*c*) Appearing below is the interface block for the input to the fragment shaders used by both the Strip-Plus and points shaders (Homework 3 and 4). Notice that both `normal_e` and `color` have interpolation qualifier `flat`. For one of the two removing the `flat` qualifier would hurt performance but would not change the appearance of the protrusions. For one of the two removing the `flat` qualifier would hurt performance and change the appearance of the protrusions. Identify which and explain.

```
in Data_to_FS
{
  flat vec3 normal_e;
  vec4 vertex_e;
  flat vec4 color;
};
```

☐ Removing `flat` (from either or both) hurts performance because:

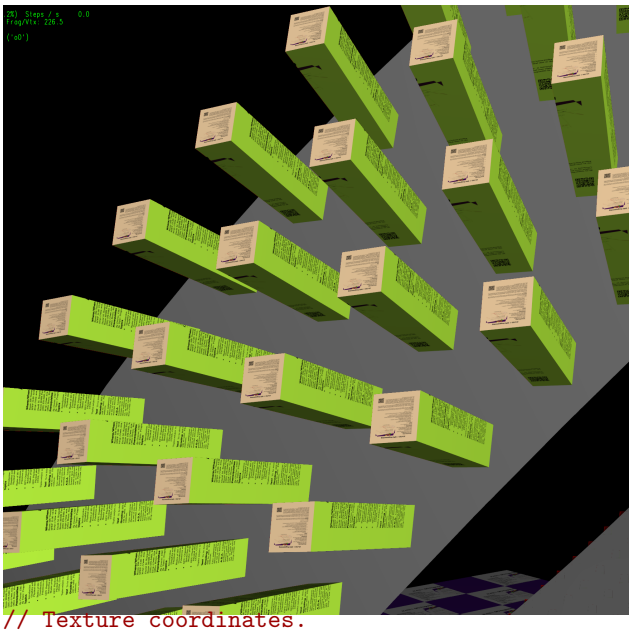
☐ Removing `flat` from `normal_e` ◯ *will* or ◯ *will not* change appearance.

☐ Explain.

☐ Removing `flat` from `color` ◯ *will* or ◯ *will not* change appearance.

☐ Explain.

Problem 2: [20 pts] The screenshot to the right shows textures applied to the protrusions rendered using a modified version of the Homework 4 solution. Appearing below is the points geometry shader with code to apply the texture to the diamond (top), but lacking code to apply the texture to the sides.



(*a*) Add code so that the texture is applied to the sides as shown. In particular, each side of the protrusion should show exactly $\frac{1}{4}$ of the texture. Note that `texcoord` is a `vec2` type, and `texcoord.x` and `texcoord.y` are the x and y coordinates of the texture.

```
// Emit the diamond-shaped top.
color = color_diamond;
normal_e = gl_NormalMatrix * nl;

int ord[4] = { 1, 0, 2, 3 }; // Vertex order.
vec2 tc[4] = { {0,0}, {1,0}, {1,1}, {0,1} }; // Texture coordinates.
for ( int i=0; i<4; i++ )
  {
    texcoord = tc[ord[i]];     // Set texture coordinates.
    vertex_e = pts_e[ord[i]];  // Retrieve vertices in the correct order.
    gl_Position = gl_ProjectionMatrix * vertex_e;
    EmitVertex();
  }
EndPrimitive();

// Emit the sides.
color = color_edge;
for ( int i=0; i<5; i++ ) {
    int i0 = i & 0x3;          // The current edge.
    int i1 = ( i + 1 ) & 0x3;  // The next edge. (Used to compute the normal.)



    vertex_e = pts_e[i0+4];
    gl_Position = gl_ProjectionMatrix * vertex_e;
    EmitVertex();



    vertex_e = pts_e[i0];
    gl_Position = gl_ProjectionMatrix * vertex_e;
    EmitVertex();

    // This normal will be used for the next two vertices.
    normal_e = cross( pts_e[i1].xyz   - pts_e[i0].xyz, pts_e[i0+4].xyz - pts_e[i0].xyz );
  }
EndPrimitive();
```

☐ Add code to apply texture to sides as described above.

(*b*) Notice that the image on the diamonds appears upside down. Modify the code below so that the image on the diamond always appears right side up in the global coordinate space (but is still aligned with the diamond so it will be crooked). This means that the texture coordinates assigned on one side of the ring will be different than the other. For example, in the part of the ring shown in the screen shot the images are upside down. But on the other side of the ring those images would be right-side up.

```
// Emit the diamond-shaped protrusion top.
color = color_diamond;
normal_e = gl_NormalMatrix * nl;

int ord[4] = { 1, 0, 2, 3 }; // Order in which to emit vertices.
vec2 tc[4] = { {0,0}, {1,0}, {1,1}, {0,1} }; // Texture coordinates.
```

```
for ( int i=0; i<4; i++ )
  {
    texcoord = tc[ord[i]];     // Set texture coordinates.
    vertex_e = pts_e[ord[i]];  // Retrieve vertices in the correct order.
    gl_Position = gl_ProjectionMatrix * vertex_e;
    EmitVertex();
  }
EndPrimitive();
```

☐ Modify code above so that texture on each diamond is rightside up.

Problem 3: [20 pts] Appearing below are kernels based on the `demo-cuda-02-basics.cu` classroom demonstration file along with a routine that launches the kernels, `launch_kernels`.

```
void launch_kernels() {
  const int thd_per_block = 512;
  const int number_of_blocks = 16;

  app.array_size = 1 << 20;
  cudaMemcpyToSymbol( d_app, &app, sizeof(app), 0, cudaMemcpyHostToDevice );

  kmain_simple <<< number_of_blocks, thd_per_block >>> ();
  kmain_efficient <<< number_of_blocks, thd_per_block >>> ();
  kmain_tuned <<< number_of_blocks, thd_per_block >>> ();
}

__global__ void kmain_simple() {
  const int tid = threadIdx.x + blockIdx.x * blockDim.x;
  const int num_threads = blockDim.x * gridDim.x;

  const int elt_per_thread = ( d_app.array_size + num_threads - 1 ) / num_threads;
  const int start = elt_per_thread * tid;  // Bad: Non-consecutive access.
  const int stop = start + elt_per_thread;

  for ( int h=start; h<stop; h++ )
    {
      int idx = h;
      float4 p = d_app.d_in[idx];  // Bad: Non-consecutive access.
      float sos = p.x * p.x + p.y * p.y + p.z * p.z + p.w * p.w;
      d_app.d_out[idx] = sos;
    }
}

__global__ void kmain_efficient() {
  const int tid = threadIdx.x + blockIdx.x * blockDim.x;
  const int num_threads = blockDim.x * gridDim.x;

  for ( int h=tid; h<d_app.array_size; h += num_threads )
    {
      const int idx = h;
      float4 p = d_app.d_in[idx];  // Good: Consecutive access.
      float sos = p.x * p.x + p.y * p.y + p.z * p.z + p.w * p.w;
      d_app.d_out[idx] = sos;
    }
}
```

```
__global__ void kmain_tuned() {
  const int tid = threadIdx.x + blockIdx.x * blockDim.x;
  const int num_threads = blockDim.x * gridDim.x;
  constexpr int strip_len = 4;
  // Data "strip" is 32 threads wide and strip_len threads long.

  const int wp_sz = 32;         // Warp size.
  const int wp = tid / wp_sz;   // This thd's warp number within kernel. (0-)
  const int ln = tid % wp_sz;   // This thd's lane number within warp.  (0-31)
  const int start = wp * wp_sz * strip_len + ln;

  for ( int h=start; h<d_app.array_size; h += strip_len * num_threads )
    {
      float soses[strip_len];
      for ( int i=0; i<strip_len; i++ )
        {
          int idx = h + i * wp_sz;
          float4 p = d_app.d_in[ idx ];
          soses[i] = p.x * p.x + p.y * p.y + p.z * p.z + p.w * p.w;
        }
      for ( int i=0; i<strip_len; i++ )
        d_app.d_out[ h + i * wp_sz ] = soses[i];
    }
}
```

(*a*) The tables below are to be filled with the value of variable `idx` from the three kernels, a total of eight values per table. Each column shows a different thread (two threads from each of two blocks), and each row shows a different loop iteration. Fill the tables based on the code above. *Hint: The upper left entry of each table will be zero.*

☐ Show the values of `idx` for the `kmain_simple` kernel.

```
blockIdx.x:     0        0        1        1
threadIdx.x:    0        1        0        1
-------------------------------------------

1st h iter


2nd h iter
```

☐ Show the values of `idx` for the `kmain_efficient` kernel.

```
blockIdx.x:     0        0        1        1
threadIdx.x:    0        1        0        1
-------------------------------------------

1st h iter


2nd h iter
```

☐ Show the values of `idx` for the `kmain_tuned` kernel. ☐ Note that the rows are for the first `i` loop, **not the `h` loop**.

```
blockIdx.x:     0        0        1        1
threadIdx.x:    0        1        0        1
-------------------------------------------

1st i iter


2nd i iter
```

(*b*) The routine `launch_kernels` always launches a configuration with 16 blocks.

Suppose a GPU had s SMs (also called MPs in class).

☐ Explain why launching s blocks is a good idea.

☐ Which is worse, launching ◯ $s-1$ *blocks* or ◯ $s+1$ *blocks* ? ☐ Explain.

Problem 4: [10 pts] The code fragments below are based on the `time_step_intersect_1` kernel from classroom demo-cuda-04. In the first fragment global memory (`helix_position`) is copied to shared memory (`pos_cache`) before use. In the second fragment global memory is accessed directly.

```
/// First Fragment – With Shared Memory
__shared__ float4 pos_cache[1024];
for ( int b_idx = b_idx_start; b_idx < hi.phys_helix_segments; b_idx += thd_per_a )
  {
    __syncthreads(); // Line A
    if ( threadIdx.x < thd_per_a )
      pos_cache[threadIdx.x] =
        helix_position[ b_idx - b_idx_start + threadIdx.x ];
    __syncthreads(); // Line B

    float4 b_position = pos_cache[b_idx_start];

    pVect ab = mv(a_position,b_position);
    // [snip]
```

```
/// Second Fragment – Without Shared Memory
for ( int b_idx = b_idx_start; b_idx < hi.phys_helix_segments; b_idx += thd_per_a )
  {
    float4 b_position = helix_position[b_idx];

    pVect ab = mv(a_position,b_position);
    // [snip]
```

(*a*) Why are the `syncthreads` needed in the first fragment. In particular:

☐ What would happen if the `syncthreads` on `Line B` were removed?

☐ What would happen if the `syncthreads` on `Line A` were removed?

(*b*) Since both fragments access `helix_position`, why is the fragment using shared memory potentially better?

☐ Ignoring request size, reason that first (shared) fragment might be faster than second?

☐ Accounting for request size, reason that first (shared) fragment might be faster than second?

Problem 5: [25 pts] Answer each question below.

(*a*) The code fragment below, taken from the classroom demo-7 code, updates the buffer object when `gpu_buffer_stale` is `true`. Variable `gpu_buffer_stale` is set to `true` initially, and again only when something about the sphere changes (for example, the number of triangles used to approximate it). The line commented `DISASTER` was added for this problem. Explain what will go wrong. *Hint: The problem occurs when* `gpu_buffer_stale` *is frequently true.*

```
if ( gpu_buffer_stale ) {
    gpu_buffer_stale = false;

    // Generate buffer id (name), if necessary.
    if ( !gpu_buffer ) glGenBuffers(1,&gpu_buffer); // ORIGINAL
    glGenBuffers(1,&gpu_buffer);                    // DISASTER

    // Tell GL that subsequent array pointers refer to this buffer.
    glBindBuffer(GL_ARRAY_BUFFER, gpu_buffer);

    // Copy data into buffer.
    glBufferData
      (GL_ARRAY_BUFFER,           // Kind of buffer object.
       coords_size*sizeof(pCoor), // Amount of data (bytes) to copy.
       sphere_coords.data(),      // Pointer to data to copy.
       GL_STATIC_DRAW);           // Hint about who, when, how accessed.

    // Tell GL that subsequent array pointers refer to host storage.
    glBindBuffer(GL_ARRAY_BUFFER, 0);
  }
```

☐ Explain the disaster that the `DISASTER` line causes.

(*b*) Explain why the true sphere shader has higher performance than the tessellated sphere shader when there are lots of small (in window space) spheres.

☐ True sphere faster than tessellated for small spheres because:

(*c*) When rendering shadow volumes lighting calculations and texture lookups are not done, both of which can be time consuming. Then what is it about rendering shadow volumes that makes it potentially take more time than rendering the objects that cast the shadows?

☐ Rendering shadow volumes takes longer than objects because:

(*d*) How can a surface normal be used by the lighting routine to compute the lighted color? What are the advantages and disadvantages of doing the calculation in the fragment shader?

☐ How is a surface normal used to compute lighted color?

☐ Advantage of computing lighted color in fragment shader over doing the calculation in the vertex shader.

☐ Disadvantage of computing lighted color in fragment shader.

(*e*) Which rendering pipeline stages would be affected by a switch from an individual triangle (`GL_TRIANGLES`) rendering pass to a triangle strip rendering pass.

☐ The affected stages are (check all that apply): ◯ *vertex* , ◯ *geometry* , ◯ *fragment* . ☐ Explain.

8 Fall 2018

Name _____

GPU Programming
EE 4702-1
Take-Home Pre-Final Examination

Due: 30 November 2018 at 16:30

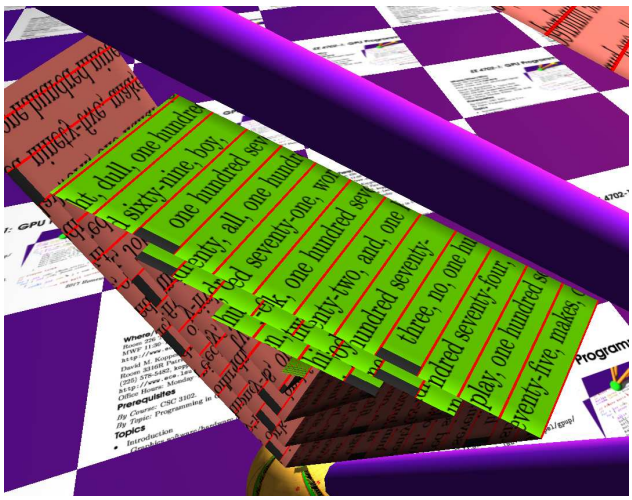
Work on this exam alone. Regular class resources, such as notes, papers, solutions, documentation, and code, can be used to find solutions. In addition outside OpenGL references and tutorials, other programming resources, and mathematical references can be consulted. Do not try to seek out references that specifically answer any question here. **Do not discuss this exam with classmates or anyone else**, except questions or concerns about problems should be directed to Dr. Koppelman.

Problem 1	_____	(33 pts)
Problem 2	_____	(34 pts)
Problem 3	_____	(33 pts)
Exam Total	_____	(100 pts)

Alias _____

Good Luck!

Problem 1: [33 pts] Appearing on the next page is the fragment shader code taken from the solution to Homework 3, in which a texture is applied to the triangular spiral in such a way that it looks like the texture image was cut into strips and glued together.



(*a*) Modify the shader code so that the text is rotated by 90°, as shown in the screenshot above. The text aspect ratio must be consistent along segments. As with the homework, the text must be readable without skipped or repeated sections. **Do not** assume or make any changes to the host code. In particular, the fragment shader should work with the same primitives as emitted for the Homework 3 code.

(*b*) Modify the shader code so that red lines are drawn separating the lines of text.

It is okay to try out your solution on the Homework 3 package, but put the solution in this exam. That is, the solution **will not** be collected by the TA-bot.

See the Homework 3 solution for detailed comments explaining the shader code.

Use next page for solution.

☐ Modify shader so text rotated 90°.

☐ Draw red lines between groups of text.

☐ Text aspect ratio and size must be consistent on all segments and ☐ there must be no gaps or skipped sections.

```
void fs_main_hw03() {
  float line_num = floor(tex_coord.x);
  vec2 tc = vec2( tex_coord.x, ( line_num + tex_coord.y ) * tex_ht );

  vec4 color =
   gl_FrontFacing ? gl_FrontMaterial.diffuse : gl_BackMaterial.diffuse;

  // No need to modify code below.
  vec4 texel = texture(tex_unit_0,tc);
  vec3 nne = normalize(normal_e);
  float edge_dist = tex_coord.y;
  vec3 bnorm = tryout.x ? nne : edge_dist > 0.9
    ? normalize(mix(nne,spiral_normal,2*(edge_dist-0.9))) : edge_dist < 0.1
    ? normalize(mix(-spiral_normal,nne,0.8+2*edge_dist)) : nne;
  vec4 lighted_color =
    generic_lighting( vertex_e, color, bnorm, gl_FrontFacing );
  gl_FragColor = texel * lighted_color;
  gl_FragDepth = gl_FragCoord.z;
}
```

Problem 2: [34 pts] Recall that the true sphere code covered in class emits a square (two triangles) for each sphere to be rendered. The square is chosen so that it perfectly frames the sphere as seen by the viewer. The square is perfect in the sense that if it were made any smaller parts of the sphere would not be visible. But it is still wasteful because the sphere is not visible in the corners of the square and so those fragment shader invocations are wasted.

(*a*) Modify the true-sphere geometry shader below so that it emits a $2s$-sided polygon rather than a square. Show the code for computing the polygon coordinates, **do not** use the assumed functions from the next problem. Note that when $s = 2$ the code should emit a bounding square, which is what the original code does.

Use next page for solution.

☐ Modify code to emit $2s$-sided bounding polygon. ☐ Take advantage of the fact that $2s$ is an even number.

```
const int s = 8;     // Don't assume that s is always 8!!
layout ( points ) in;
layout ( triangle_strip, max_vertices = 4 ) out;

void gs_main() {
  vertex_id = In[0].vertex_id;
  mat4 mvp = trans_proj * gl_ModelViewMatrix;
  vec4 pos_rad = sphere_pos_rad[vertex_id];
  vec3 ctr_o = pos_rad.xyz;
  float r = pos_rad.w;
  vec3 e_o = vec3(gl_ModelViewMatrixInverse*vec4(0,0,0,1));  // Eye location in object space.

  vec3 ec_o = ctr_o - e_o;  // Eye to Center (of sphere).
  float ec_len = length(ec_o);
  vec3 ne = ec_o / ec_len;

  // Find vectors orthogonal to ec_o.
  vec3 atr_o = abs(ec_o);
  int min_idx = atr_o.x < atr_o.y ? ( atr_o.x < atr_o.z ? 0 : 2 )
    : ( atr_o.y < atr_o.z ? 1 : 2 );
  vec3 nx_raw = min_idx == 0 ? vec3(0,-ec_o.z,ec_o.y)
    : min_idx == 1 ? vec3(-ec_o.z,0,ec_o.x) : vec3(-ec_o.y,ec_o.x,0);
  vec3 nx = normalize(nx_raw),  ny = cross(ne,nx);

  // Compute center of the limb, lmb_o. (The limb is the sphere outline.)
  float sin_theta = r / ec_len;
  float a = r * sin_theta;
  vec3 lmb_o = ctr_o - a * ne;

  // Compute axes to draw the limb.
  float b = r * sqrt( 1 - sin_theta * sin_theta );
  vec3 ax = b * nx,  ay = b * ny;

  // Emit a bounding square for the limb.  --- Solution should start here.
  for ( int i = -1; i < 2; i += 2 ) for ( int j = -1; j < 2; j += 2 ) {
        vertex_o = lmb_o + ax * i + ay * j;
        gl_Position = mvp * vec4(vertex_o,1);
        EmitVertex();  }
  EndPrimitive();




}
```

(*b*) Let t_f denote the execution time of 1 invocation of the true-sphere fragment shader when the sphere covers the fragment, and let t_n denote the execution time of 1 invocation when the sphere does not cover the fragment. Assume that $t_f > t_n$. Let $t_{g0} + st_{g1}$ denote the execution time of one invocation of the true-sphere geometry shader that emits a $2s$-sided polygon.

In general, emitting a polygon with more sides reduces the work performed by the fragment shader but increases the work done by the geometry shader, improving overall performance. But there will be a point at which increasing s *reduces* performance.

Using the symbols defined above **plus other information**, find an expression for the time needed to render a sphere. The other information is needed to compute a sphere rendering time using the timings above. Of course, no credit will be given if that other information *is* the execution time. Also determine the value of s that will minimize the total time used by the geometry and fragment shaders for rendering a sphere. When computing the s that minimizes time it is okay to apply simplifications based on s being large. (Otherwise a closed-form solution can't easily be found.) *Hint: Yes, the solution requires calculus.*

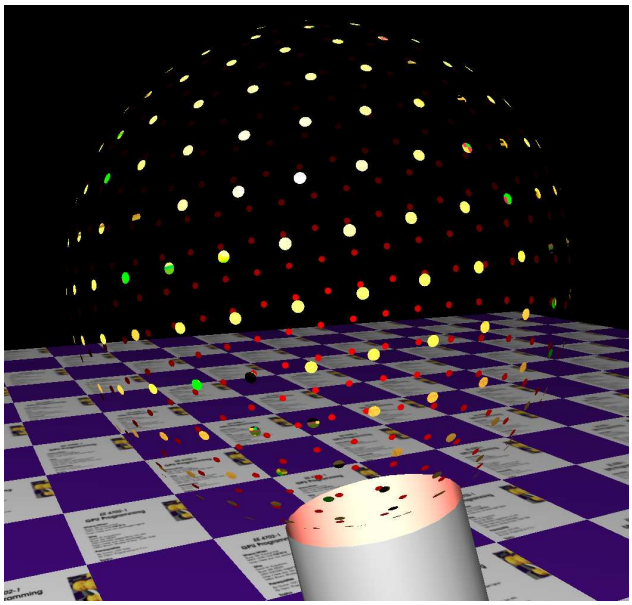
☐ Indicate the other information that is needed.

☐ Show an expression for the time needed to render a sphere using a $2s$-sided bounding polygon in terms of t_f, etc., and the other information.

☐ Find an expression for s that will minimize time in terms of t_f, etc., and the other information.

Problem 3: [33 pts] The screenshot below is based on the code from Homework 4. The code is in *lenses* mode, meaning that the sphere is visible only at certain points. Unlike Homework 4, the lenses are much further apart from each other, and therefore much less of the sphere is visible. If the code in the Homework 4 solution were used then most invocations of the fragment shader would be wasted because they end up discarding the fragment.

There would be much less waste if a separate primitive were emitted for each lens, rather than one primitive for the entire sphere.



Suppose that each sphere has the same number of lenses, call that number h, and refer to the lenses as lens 0, lens 1, etc. Assume that a library of functions is available for computing positions for each lens. Among these is a function `lens_get_ctr_o(i,sphere_info)` which returns the object-space coordinates of the center of lens `i`. Variable `sphere_info` holds the value of the sphere center, radius, and orientation. Assume that `sphere_info` has already been prepared. You may assume that related functions exist such as `lens_get_ctr_e(i,sphere_info)` for eye-space coordinates. Also convenient for this problem is function `lens_get_bb_o(i,j,n,sphere_info)` which returns the object-space coordinates of point `j` of an `n`-sided bounding polygon for lens `i`. The bounding polygon is like the bounding square used in the true-sphere code: it tightly fits around the lens as seen by the viewer (or equivalently, when projected on to the frame buffer). The points wrap around the lens in a clockwise direction with increasing `j`.

Show changes needed to the geometry and fragment shaders on the following pages to render the lenses efficiently for cases in which h is relatively small, say on the order of 10.

Problem 3, continued:

(*a*) Appearing below is a shortened version of the geometry shader from the Homework 4 solution. The comments have been removed, but feel free to look at comments in the posted solution. Modify the geometry shader below so that it emits a primitive for each lens and provides information needed by the fragment shader (see next problem). Use the functions described above to find coordinates of bounding polygons for the lenses.

☐ Modify so that it emits a primitive for each lens.

☐ If necessary, change layout and output declarations.

☐ Provide information needed by fragment shader (see next subproblem).

```
out Data_to_FS // Data to fragment shader
{
  flat int vertex_id;
  vec3 vertex_o;


};

const int h = 8; // Number of holes. Don't assume it is always 8.
layout ( points ) in;
layout ( triangle_strip, max_vertices = 4 ) out;

void gs_main() {
  vertex_id = In[0].vertex_id;
  vec4 pos_rad = sphere_pos_rad[vertex_id];
  vec3 ctr_o = pos_rad.xyz;
  float r = pos_rad.w;
  Sphere_Info sphere_info = make_sphere_info(pos_rad,sphere_rot[vertex_id]);

  // Eye location in object space.
  vec3 e_o = vec3(gl_ModelViewMatrixInverse * vec4(0,0,0,1));

  // Vectors from eye to sphere center.
  vec3 ec_o = ctr_o - e_o;  // Eye to Center (of sphere).
  float ec_len = length(ec_o);
  vec3 ne = ec_o / ec_len;

  // Vectors orthogonal to ec_o.
  vec3 atr_o = abs(ec_o);
  int min_idx = atr_o.x < atr_o.y ? ( atr_o.x < atr_o.z ? 0 : 2 )
    : ( atr_o.y < atr_o.z ? 1 : 2 );
  vec3 nx_raw = min_idx == 0 ? vec3(0,-ec_o.z,ec_o.y)
    : min_idx == 1 ? vec3(-ec_o.z,0,ec_o.x) : vec3(-ec_o.y,ec_o.x,0);
  vec3 nx = normalize(nx_raw);
  vec3 ny = cross(ne,nx);

  // Compute center of the limb, lmb_o, the circle formed by the most
  // distant visible parts of the sphere surface.
```

```
    float sin_theta = r / ec_len;
    float a = r * sin_theta;
    vec3 lmb_o = ctr_o - a * ne;

    // Compute axes to draw the limb.
    float b = r * sqrt( 1 - sin_theta * sin_theta );
    vec3 ax = b * nx, ay = b * ny;
    mat4 mvp = trans_proj * gl_ModelViewMatrix;

    // Emit a bounding square for the limb.
    for ( int i = -1; i < 2; i += 2 ) for ( int j = -1; j < 2; j += 2 ) {
          vertex_o = lmb_o + ax * i + ay * j;
          gl_Position = mvp * vec4(vertex_o,1);
          EmitVertex();
        }
    EndPrimitive();

}
```

(*b*) Appearing below is the fragment shader from the Homework 4 solution. The comments have been removed for brevity, feel free to look at the full Homework 4 solution. Show the changes needed to the fragment shader. The shader should be substantially simpler.

☐ Changes needed, based on geometry shader from previous part.

☐ The fragment shader should render lenses only, not full spheres.

☐ Simplify based on the fact that geometry shader is providing lens information.

```
void fs_main()
{
  vec4 pos_rad = sphere_pos_rad[vertex_id];
  vec3 ctr_o = pos_rad.xyz;
  float rsq = pos_rad.w * pos_rad.w;
  vec3 e_o = gl_ModelViewMatrixInverse[3].xyz;
  vec3 ef = vertex_o - e_o;   // Eye to fragment.
  vec3 ce = e_o-ctr_o;
  float qfa = dot(ef,ef), qfb = 2 * dot(ef,ce), qfc = dot(ce,ce) - rsq;
  float discr = qfb*qfb - 4 * qfa * qfc;

  if ( discr < 0 ) discard;

  const float pi = 3.14159265359;
  const float two_pi = 2 * pi;
  const bool holes = opt_holes == OHO_Holes;    // If true sphere has holes
  const bool lenses = opt_holes == OHO_Lenses;
  const bool solid = !holes && !lenses;         // If true show complete sphere.

  vec4 sur_o;  // Global surface cordinates (front or back) of sphere.
  vec3 normal_o, sur_l;
  bool front;          // If true, cordinates are of front of sphere.
  float theta, eta;    // Local angular cordinates of sphere.
  bool found_hole = false;

  for ( float dir = -1;  dir <= 1;  dir+=2 ) {
      front = dir == -1;
      float t = ( -qfb + dir * sqrt( discr ) ) / ( 2 * qfa );

      sur_o = vec4(e_o + t * ef, 1);
      normal_o = normalize(sur_o.xyz - ctr_o);

      if ( !front ) normal_o = -normal_o;

      sur_l = normalize(mat3(sphere_rot[vertex_id]) * normal_o);
      theta = atan(sur_l.x,sur_l.z);
      eta = acos(sur_l.y);

      if ( solid ) break;

      const float hole_frac = 0.2;
      const float radians_per_hole_eq = two_pi / opt_n_holes_eqt;
```

```
      const float hole_radius = 0.5 * hole_frac * radians_per_hole_eq;

      float eta_hole = round( eta / radians_per_hole_eq ) * radians_per_hole_eq;
      float r = sin(eta_hole);
      const float n_holes = floor( two_pi * r / radians_per_hole_eq );
      if ( n_holes < -1 )
        {
          if ( holes ) break;
          if ( dir == 1 ) discard;
        }

      const float radians_per_hole = two_pi / n_holes;
      float theta_hole = round( theta / radians_per_hole ) * radians_per_hole;
      vec3 hole_dir_l =
        vec3( r * sin(theta_hole), cos(eta_hole), r * cos(theta_hole) );

      float dist = distance(hole_dir_l,sur_l);
      found_hole = dist < hole_radius;
      if ( lenses && found_hole  holes && !found_hole ) break;
      if ( !front ) discard;
    }

  vec3 normal_ee = normalize(gl_NormalMatrix * normal_o);
  vec4 sur_ee = gl_ModelViewMatrix * sur_o;

  // Compute clip-space depth. Take care so that compiler avoids full
  // matrix / vector multiplication.
  vec4 sur_e = gl_ModelViewMatrix * sur_o;
  vec4 sur_c = trans_proj * vec4(0,0,2*sur_e.z,1);
  gl_FragDepth = sur_c.z / sur_c.w;

  vec2 tcoord = vec2( ( 1.5 * pi + theta ) / two_pi, eta / pi );
  vec4 texel = front ? texture(tex_unit_0,tcoord) : vec4(1);
  vec4 color2 = front ? sphere_color[vertex_id] : vec4(0.3,0,0,1);
  gl_FragColor = texel * generic_lighting(sur_ee, color2, normal_ee);
}
```

Name _____

GPU Programming
LSU EE 4702-1
Final Examination

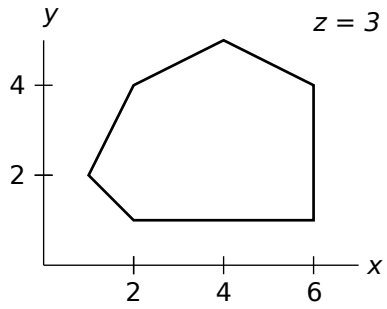
Tuesday, 4 December 2018 12:30–14:30 CST

Problem 1	_____	(15 pts)
Problem 2	_____	(30 pts)
Problem 3	_____	(15 pts)
Problem 4	_____	(40 pts)
Exam Total	_____	(100 pts)

Alias _____

Good Luck!

Problem 1: [15 pts] Appearing below is a geometric figure.



(*a*) Complete the individual-triangle rendering pass below so that it renders the figure with all triangles facing in the positive z direction and without overlapping triangles. Use the provided abbreviation `glV`. *Note: The part about facing the $+z$ direction was not in the original exam.*

☐ Complete code to render shape using individual triangles.

```
#define glV glVertex3f
 glBegin(GL_TRIANGLES);




 glEnd();
```

(*b*) Complete the triangle-strip rendering pass below so that it renders the figure.

☐ Complete code to render shape using a triangle strip.

```
#define glV glVertex3f
 glBegin(GL_TRIANGLE_STRIP);




 glEnd();
```

Problem 2: [30 pts] Appearing below is shortened host and shader code based on Homework 3, in which text was drawn on the triangular spiral. One drawback of this code is that it uses old-fashioned, deprecated, inefficient `glVertex` calls. On the following pages are routines that will implement a more efficient version of this code in which data such as `ctr` are placed in buffer objects and a rendering pass is performed *for the entire chain*, not just one triangular spiral.

```
for ( int i=2; i<chain_length; i++ ) {  // Host Code
    pCoor p0 = balls[i-2].position, p1 = balls[i-1].position, p2 = balls[i].position;
    pCoor ctr = (p0+p1+p2)/3; // Compute location of triangle center.
    // --- CODE REMOVED FOR BREVITY ---
    pCoor pprev = ctr;
    float delta_a = 0.6 / opt_n_segs;

    if ( opt_shader == SO_HW03 ) {
        glBegin(GL_TRIANGLE_STRIP);  // Render spiral using 1 triangle strip.

        for ( int j=0; j<opt_n_segs; j++ ) {
            pVect v = va[j%3];                // Vector from center to fold.
            pCoor p = ctr + j * delta_a * v;  // Coordinate of fold.
            pNorm n = cross( p - pprev, vz ); // Normal of segment.
            float tex_x = total_len_compute(j,delta_a,distv) * tex_scale;

            glNormal3fv(n);
            glTexCoord2f(tex_x,0);  glVertex3fv(p + vz);
            glTexCoord2f(tex_x,1);  glVertex3fv(p - vz);
            pprev = p;
          }
        glEnd();
      } }
```

```
void vs_main_hw03() { // Vertex Shader Code
  gl_Position = gl_ModelViewProjectionMatrix * gl_Vertex;
  vertex_e = gl_ModelViewMatrix * gl_Vertex;
  normal_e = normalize(gl_NormalMatrix  * gl_Normal);
  tex_coord = gl_MultiTexCoord0.xy;   }
```

(*a*) But first let c denote the value of `chain_length` and n denote the value of `opt_n_segs`. Determine the amount of data, in bytes, sent from the CPU to the GPU for the code shown above. (Do not consider uniforms and other hidden code.)

☐ Amount of data sent from CPU to GPU in terms of c and n, in bytes.

(*b*) In the vertex shader above, label variables with the appropriate letter as requested below.

☐ Label uniform variables with a **U**, ☐ shader input variables with an **I**, and ☐ fixed-function shader outputs with an **O**.

Problem 2, continued: Appearing below is the improved triangular spiral host code and shaders. The host code prepares four buffer objects and then starts an instanced rendering pass with line strips as the input primitive. The rendering pass uses a vertex shader and a geometry shader to complete the primitives. (The fragment shader is not a part of this problem.) The only inputs to the vertex shader are the vertex and instance IDs, which should be used to retrieve or compute information about the spiral segments.

```
// Host Code -- DO NOT MAKE OR ASSUME MODIFICATIONS TO THIS CODE.
      for ( int i=2; i<chain_length; i++ ) {
          // Put ball structures and coordinates into convenient variables.
          pCoor p0 = balls[i-2].position, p1 = balls[i-1].position, p2 = balls[i].position;
          pCoor ctr = (p0+p1+p2)/3;   ctr_a.push_back(ctr);

          // -- CODE PREPARING va_a, vz_a, and distv_a NOT SHOWN --
        }
      glUniform1i(5, opt_n_segs);
      TO_BUFFER_OBJECT(ctr_a,1);       TO_BUFFER_OBJECT(va_a,2);
      TO_BUFFER_OBJECT(distv_a,3);     TO_BUFFER_OBJECT(vz_a,4);
      glBindBuffer(GL_ARRAY_BUFFER,0);

      glDrawArraysInstanced(GL_LINE_STRIP, 0, opt_n_segs, ctr_a.size() );
```

(*c*) Modify the vertex and geometry shaders on the following pages to efficiently render the triangle spirals, make any needed changes to the interface blocks, but **do not modify** or assume modifications to the host code (above). For your convenience each shader contains a copy of the host code. Cross out or modify that code as needed. Use the handy abbreviations at the top of the page.

☐ Update the interface blocks as needed.

```
#ifdef _VERTEX_SHADER_
out Data {  // Out of Vertex Shader to Geometry Shader
 int ins_id, vtx_id;
 //  ☐   Add any needed declarations here.




};
#endif
#ifdef _GEOMETRY_SHADER_
in Data {  // In to Geometry Shader from Vertex Shader
 int ins_id, vtx_id;
 //  ☐   Add any needed declarations here.




} In[2];

// Out of Geometry Shader to Fragment Shader -- No changes needed
out Data {   vec3 normal_e;   vec4 vertex_e;   vec2 tex_coord;  };
#endif
```

Problem 2, continued: Vertex shader code on this page.

☐ Cross out unneeded code. ☐ Cross out data type for shader outputs. ☐ Avoid redundant computation.

☐ Update the interface blocks as needed.

```
const mat4 mv = gl_ModelViewMatrix, mvp = gl_ModelViewProjectionMatrix;
const mat3 nm = gl_NormalMatrix;

void vs_main_lines() {  // Vertex Shader Main Routine
  int ins_id = gl_InstanceID, vtx_id = gl_VertexID;

  int j = gl_VertexID;
  float delta_a = 0.6 / opt_n_segs;
  vec4 distv = distv_a[ins_id];
  vec3 vz = vz_a[ins_id].xyz;
  float tex_scale = 0.2;









  vec3 v = va_a[ins_id][j%3].xyz;
  vec3 p = ctr_a[ins_id].xyz + j * delta_a * v;
  vec3 pprev = vec3(0,0,0); // PLACEHOLDER. Won't work.
  vec3 n = cross( p - pprev, vz );
  float tex_x = total_len_compute(j,delta_a,distv) * tex_scale;









}
```

Problem 2, continued: Geometry shader code on this page.

☐ Cross out unneeded code. ☐ Cross out data type for shader outputs. ☐ Avoid redundant computation.

☐ Update the interface blocks as needed.

```
layout ( lines ) in;
layout ( triangle_strip, max_vertices = 4 ) out;

const mat4 mv = gl_ModelViewMatrix, mvp = gl_ModelViewProjectionMatrix;
const mat3 nm = gl_NormalMatrix;

void gs_main_lines() {  // Geometry Shader Main Routine
  int ins_id = gl_InstanceID, vtx_id = gl_VertexID; // PLACEHOLDER. Won't work.

  int j = 0;  // PLACEHOLDER, Won't work.
  float delta_a = 0.6 / opt_n_segs;
  vec4 distv = distv_a[ins_id];
  vec3 vz = vz_a[ins_id].xyz;
  float tex_scale = 0.2;









  vec3 v = va_a[ins_id][j%3].xyz;
  vec3 p = ctr_a[ins_id].xyz + j * delta_a * v;
  vec3 pprev = vec3(0,0,0); // PLACEHOLDER. Won't work.
  vec3 n = cross( p - pprev, vz );
  float tex_x = total_len_compute(j,delta_a,distv) * tex_scale;









}
```

Problem 3: [15 pts] Answer each CUDA question below.

(*a*) Both CUDA kernels below do the same thing, but one will execute much less efficiently. Explain why in terms of the minimum request size.

```
__global__ void kmain_simple(float4 *d_in, float *d_out) {
  const int tid = threadIdx.x + blockIdx.x * blockDim.x;
  const int elt_per_thread = ( d_app.array_size + d_app.num_threads - 1 ) / d_app.num_threads;
  const int start = elt_per_thread * tid;
  const int stop = start + elt_per_thread;

  for ( int h=start; h<stop; h++ ) {
      float4 p = d_in[h];
      float sos = p.x * p.x + p.y * p.y + p.z * p.z + p.w * p.w;
      d_out[h] = sos;
    }
}

__global__ void kmain_efficient(float4 *d_in, float *d_out) {
  const int tid = threadIdx.x + blockIdx.x * blockDim.x;
  for ( int h=tid; h<d_app.array_size; h += d_app.num_threads ) {
      float4 p = d_in[h];
      float sos = p.x * p.x + p.y * p.y + p.z * p.z + p.w * p.w;
      d_out[h] = sos;
    }
}
```

☐ Problem with inefficient kernel due to request size.

☐ What is the maximum request size that will avoid this inefficiency? ☐ Explain.

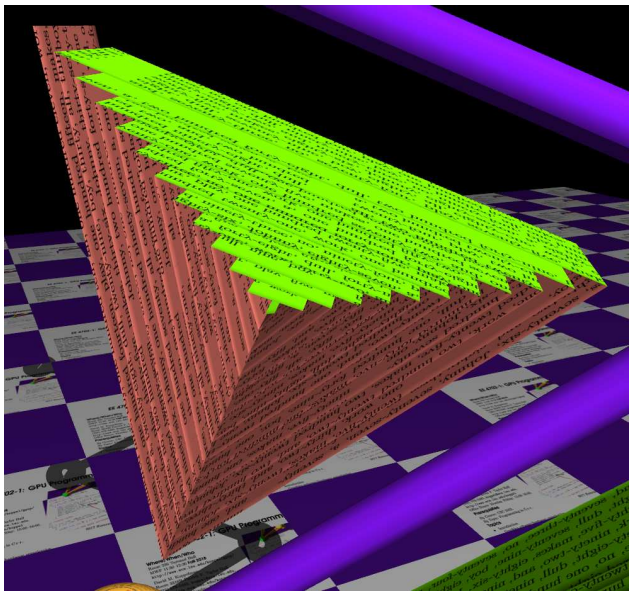
(*b*) A CUDA kernel is to run on a GPU with 8 SMs (MPs). Configuration A consists of 4 blocks of 32 threads each. Configuration B consists of 8 blocks of 16 threads each. Neither is very good. Explain how each one underutilizes the hardware on current NVIDIA GPUs.

☐ Configuration A underutilizes hardware by . . .

☐ Configuration B underutilizes hardware by . . .

Problem 4: [40 pts] Answer each question below.

(*a*) The screenshot below is the Homework 3 triangular spiral with 37 segments per spiral. Imagine a spiral with even more segments, say 1000.



With a large number of segments there will usually be a large computational load on both the vertex and fragment shaders. In one of these shaders the computational load can be considered wasted, depending on the eye location, **even when the spiral is visible**. In which shader is computation wasted, and why.

☐ Which shader wastes computation? ☐ Why?

Sketch two views in which the spiral is visible. In one the computational load is high and mostly wasted. In the other the computational load is lower and not wasted.

☐ View with lower load and little waste. ☐ View with high load and waste.

(*b*) The OpenGL call `glColor` is used to specify a color, say purple `glColor3f(1,0,1);`. In typical use is that the color that will be written to the frame buffer? Explain.

☐ Are the arguments to `glColor` the exact color to be written to the frame buffer? ☐ Explain.

(*c*) Describe the difference between the `flat`, `noperspective`, and `smooth` interpolation qualifiers.

☐ The `flat` qualifier's feature, ☐ the `noperspective` qualifier's, feature, ☐ the `smooth` qualifier's, feature.

(*d*) Consider a rendering pass using triangles as the primitive. OpenGL (compatibily profile) allows one to specify a normal for each triangle vertex, but as we all know a triangle, geometrically, has just one normal. Why would one specify different normals for each vertex? Explain how such normals are chosen.

☐ Why might one choose different normals for each vertex of a triangle?

☐ Give an example and describe how those normals are chosen.

(*e*) Describe what the inputs to the rasterization stage are, what the rasterization stage does, and what its outputs are.

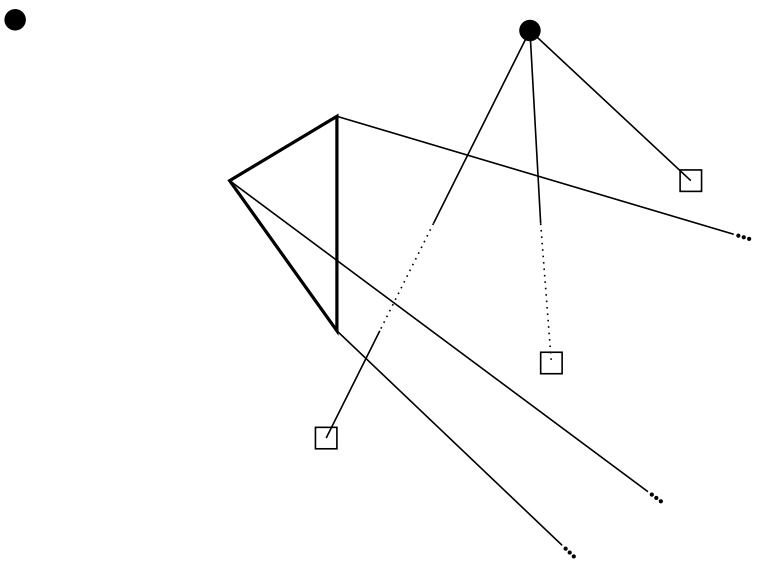
☐ Rasterization stage input, ☐ rasterization stage job, ☐ rasterization stage output:

(*f*) The true-sphere shader used in class rendered spheres perfectly. Would it make sense to use a similar approach to write a true-cube shader that can perfectly render a cube?

☐ Does a true cube shader make sense? ☐ Explain.

(*g*) The unlabeled diagram below shows how shadow volumes can be used to render shadows. On the diagram show the location of the eye and light source, fragment(s) found to be in the shade, and fragment(s) found to be illuminated.

☐ Show: ☐ eye, ☐ light source, ☐ shaded fragment(s), ☐ illuminated fragment(s).



9 Fall 2017

Name _____

GPU Programming
EE 4702-1
Midterm Examination

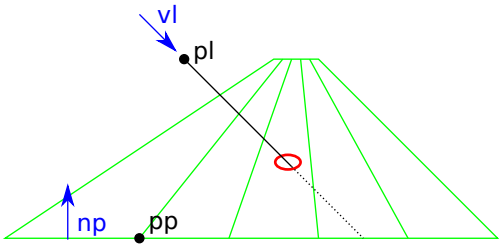
Wednesday, 18 October 2017 11:30–12:20 CDT

Problem 1	_____	(30 pts)
Problem 2	_____	(20 pts)
Problem 3	_____	(25 pts)
Problem 4	_____	(25 pts)
Exam Total	_____	(100 pts)

Alias _____

Good Luck!

Problem 1: [30 pts] The illustration to the right shows a ring surrounding the point at which a line crosses a plane. When completed the code fragment below will render the ring. The line is defined by parametric equation $P(t) = \texttt{pl} + t \times \texttt{vl}$. Point `pp` is on the plane and vector `np` is normal to the plane. The ring is circular and is on the plane.



(*a*) Modify the code below so that `t` is assigned the value for which `pc` is on the plane.

☐ Compute t such that $P(t)$ is on the plane.

(*b*) Modify the code below to render the ring. Use `r1` for the inner radius and `r2` for the outer radius. Render it using `slices` sets of triangles. Use variable `pc` even if you haven't yet solved the previous part.

☐ Include: ☐ Code to set up the rendering pass, ☐ to set the color (red), ☐ to set the correct normal, ☐ and to compute and set vertex coordinates.

```
World::render_ring( pCoor pl,  pVect vl,     // Line definition.
                    pCoor pp,  pNorm np,     // Plane definition.
                    float r1,  float r2 ){  // Ring radii.


  float t =                          ; // Fill In

  pCoor pc = pl + t * vl;









  const int slices = 100;
  const float delta_theta = 2 * M_PI / slices;
  for ( int i=0; i<=slices; i++ ) {
      const float theta = i * delta_theta;









    }
}
```

Problem 2: [20 pts] Recall that in Homework 2 a sphere was rendered using a spiral slice. The coordinates of the slice were placed in a buffer object and a sphere was rendered by performing multiple rendering passes each using the same buffer object but a different transformation. Let c denote the number of coordinates in the buffer object, and s denote the number of slices per sphere.

(a) What is the size of the buffer object, in bytes? *Hint: This is an easy question.*

☐ Size of buffer object, in bytes. ☐ State any assumptions made.

(b) Appearing below is the code rendering a sphere, adapted from the Homework 2 solution. Based on this code, estimate the amount of data send from the CPU to the GPU to render the sphere, **not counting the buffer object**.

```
pMatrix_Rotation rot_xz( pVect(0,1,0), delta_theta ),  rot_yz( pVect(1,0,0), M_PI );
// Code to render one sphere.
glTranslatefv( ball->position );
glScale1f( ball->radius );
glMultTransposeMatrixf( pMatrix_Rotation(ball->orientation) );

for ( int j=0; j<2; j++ ) {
     for ( int i=0; i<s/2; i++ ) {
        glColor3fv( i & 1 ? color_lsu_spirit_purple : ball->color );
        glDrawArrays(GL_TRIANGLE_STRIP,0,c);
        glMultTransposeMatrixf(rot_xz);
      }
    glMultTransposeMatrixf(rot_yz);
  }
```

☐ In terms of s, amount of data sent to GPU to render sphere.

(c) Buffer object BO2 holds coordinates to render an entire sphere, not just a slice, $s \times c$ coordinates total. Buffer object BO1 holds the slice discussed above. Suppose that initially all data is on the CPU. Let n be the number of sphere to be rendered. In terms of s and c, what is the smallest value of n for which using BO2 will require less data than BO1?

☐ Break-even value for n:

Problem 3: [25 pts] The diagram below shows a texture on the left-hand side and a scene to be rendered on the right. The scene consists of three triangles with the texture applied. The text in the texture, written by Mathy Vanhoef, is from the description of a widespread vulnerability in WiFi WPA2 implementations. See `https://www.krackattacks.com`, and please, Fall 2017 people, update your OS and wireless access point firmware. *Grading Note: The description of where the text came from was not in the original exam. The KRACK vulnerability became public two days before the midterm exam.*

Texture

We discovered serious weaknesses

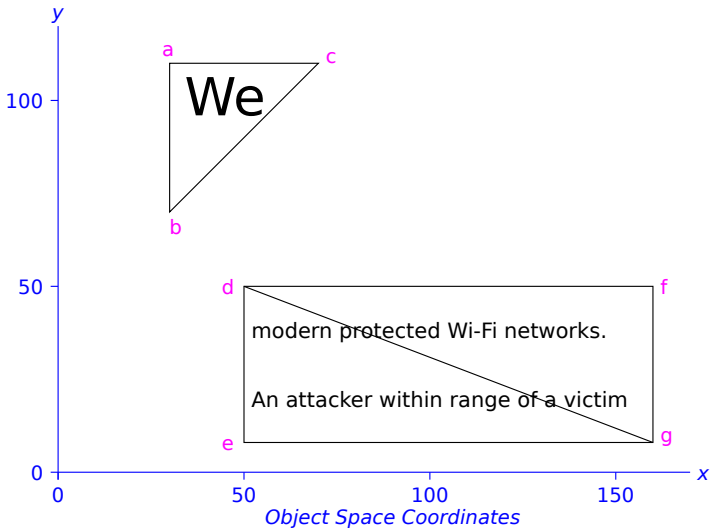
in WPA2, a protocol that secures all

modern protected Wi-Fi networks.

An attacker within range of a victim

can exploit these weaknesses using

key reinstallation attacks (KRACKs).



(*a*) Complete the code below so that it renders the triangles with the texture applied as shown. The color and normal have already been set, just specify vertex and texture coordinates. Use abbreviation `glV` for `glVertex2f` and `glT` for `glTexCoord2f`. Only specify x and y components for the vertex coordinates.

☐ Specify ☐ texture and ☐ vertex coordinates so that the scene is rendered as shown.

```
glBegin(GL_TRIANGLES);
glNormal3f(0,0,1);
glColor3f(0.5,0.5,0.5);

glV(30,110);   // Vertex a









glEnd();
```

Problem 3, continued:

Texture

We discovered serious weaknesses

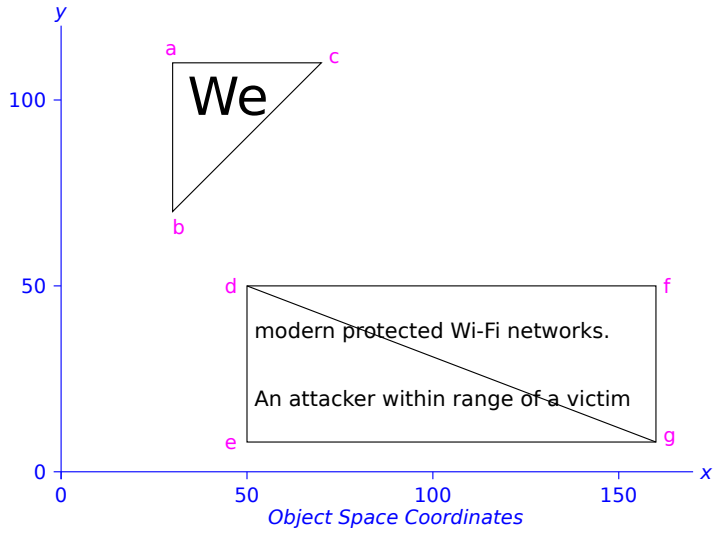
in WPA2, a protocol that secures all

modern protected Wi-Fi networks.

An attacker within range of a victim

can exploit these weaknesses using

key reinstallation attacks (KRACKs).



(*b*) Suppose that defg in the diagram above is four pixels across and two pixels down. Suppose that texturing were set up to use only a single mipmap level and nearest filtering. Explain how defg could appear all white in that case.

☐ At 4×2 pixels defg all white because:

(*c*) Explain how the 4×2 rectangle would appear with linear filtering and mipmap levels. Explain the key points of linear texture filtering.

☐ Appearance with mipap levels and linear filtering.

☐ Key points of filtering.

Problem 4: [25 pts] Answer each question below.

(*a*) Show a transformation matrix M that will translate a homogeneous coordinate P to $P + \begin{bmatrix} 7 \\ 6 \\ 5 \end{bmatrix}$. (That is, find M such that $MP = P + \begin{bmatrix} 7 \\ 6 \\ 5 \end{bmatrix}$.) Only show nonzero array elements.

☐ $M =$

(*b*) Describe the "from" and "to" coordinate spaces for the following OpenGL matrices:

☐ The ModelView matrix maps a coordinate from ☐ _____ space to ☐ _____ space.

☐ The Projection matrix maps a coordinate from ☐ _____ space to ☐ _____ space.

(*c*) Show two drawings, each consisting of four triangles. The one should be well-suited for rendering using a triangle strip, for the other a triangle strip should make no difference.

☐ Show a 4-triangle example well-suited to triangle strips and one in which a ☐ triangle strip doesn't help at all.

(*d*) Suppose that buffer object o30 has 30 sphere coordinates and buffer object o1k has 1000 sphere coordinates. In both buffer objects the sphere coordinates are well chosen. They will be used to render a sphere which is fully visible. *Note: In the original exam the sizes were 1000 and 10,000.*

☐ Describe a situation in which the sphere would look better using o1k than using o30 and ☐ describe a situation in which the sphere would look the same with o1k as with o30. ☐ Explain.

Suppose that it takes much more time to render a sphere using o1k than with o30. The vertex, geometry, and fragment shader stages are running custom written code which could use some tuning.

☐ Which shaders should be tuned to potentially fix the performance problem? ☐ Which shaders be ignored? ☐ Explain.

Name _____

GPU Programming
EE 4702-1
Final Examination

Tuesday, 5 December 2017 12:30–14:30 CST

Problem 1	_____	(15 pts)
Problem 2	_____	(20 pts)
Problem 3	_____	(30 pts)
Problem 4	_____	(35 pts)
Exam Total	_____	(100 pts)

Alias _____

Good Luck!

Problem 1: [15 pts] Appearing below is code based on Homework 5 (the tile cloud). First is code that launches the rendering passes, followed by the shaders. The shaders below were written to work with the rendering pass launched by `case 1`. In `case 3`, which is incomplete, the rendering pass input primitives are lines instead of triangles. Complete the `case 3` code and modify the shaders to work with your completed `case 3` code.

☐ Complete the `case 3` code. Use abbreviations such as `glV` (for `glVertex4f`), `glC`, and `glN`.

```
case 1: {  pShader_Use use(s_hw05_tiles_1);
           glBegin(GL_TRIANGLES);
           for ( Tile* tile: tiles ) {  glColor4fv(tile->color);    glVertex4fv(tile->pt_00);
                                        glColor4fv(tile->tact_pos); glVertex4fv(tile->ax);
                                                                    glVertex4fv(tile->ay); }
           glEnd();  } break;
case 3: {  pShader_Use use(s_hw05_tiles_3);
           glBegin(GL_LINES);   //  <--- DON'T FORGET, LINES
           for ( Tile* tile: tiles ) {



           }
           glEnd();  } break;
```

☐ Modify shader code (below) to work with `case 3` (above). Look at ☐ type of primitive (above), ☐ interface blocks, ☐ shader routines, and ☐ layout declarations.

```
out Data_to_GS { vec4 vertex_o;    vec4 color;       };

void vs_main_tiles_3() {  // Vertex shader routine.                // CHANGE/ADD SOMETHING
  vertex_o = gl_Vertex;
  color = gl_Color;
}

in Data_to_GS { vec4 vertex_o;      vec4 color;        } In[3];     // CHANGE/ADD SOMETHING

layout ( triangles ) in;                                           // CHANGE/ADD ONE OF
layout ( triangle_strip, max_vertices = 4 ) out;                   // THE LAYOUT DECLARATIONS

void gs_main_tiles_3() {   // Geometry shader routine.              // CHANGE/ADD SEVERAL THINGS

  vec4 pt_00 = In[0].vertex_o;

  vec4 ax_o = vec4(In[1].vertex_o.xyz,0);

  vec4 ay_o = vec4(In[2].vertex_o.xyz,0);

  vec4 tact_pos = In[1].color;

  color = In[0].color;

  vec4 vtx_o[4];
  vtx_o[0] = pt_00;          vtx_o[1] = pt_00 + ax_o;
  vtx_o[2] = pt_00 + ay_o;   vtx_o[3] = pt_00 + ay_o + ax_o;
  // The code below does not need to be changed and so isn't shown.
```

Problem 2: [20 pts] Appearing below is the `render_tiles` routine from Homework 5.

(*a*) Estimate the amount of data sent from the CPU to the GPU for the rendering pass started by the code below. Use the following symbol: n, the number of tiles.

```
case 1: {
    pShader_Use use(s_hw05_tiles_1);
    glUniform2i(1, opt_tryout1, opt_tryout2);
    glUniform1i(2, light_state_get());
    glUniform1f(3, world_time);

    glBegin(GL_TRIANGLES);
    for ( Tile* tile: tiles ) {
        glColor4fv(tile->color);
        glVertex4fv(tile->pt_00);
        glColor4fv(tile->tact_pos);
        glVertex4fv(tile->ax);
        glVertex4fv(tile->ay);
      }
    glEnd();
  }
  break;
```

☐ Amount of data, in bytes:

(*b*) The vertex shader below is used with the `case 1` code above. In terms of n, how much data is read by this vertex shader for a rendering pass?

```
void vs_main_tiles_1() {
  vertex_o = gl_Vertex;
  color = gl_Color;
}
```

☐ Total data read by shader above for a rendering pass:

☐ Explain why the amount of data read by the vertex shader might be different than the amount of data sent from CPU to GPU when executing the `case 1` code.

Problem 2, continued: The rendering pass below provides the same data to the shaders as the one in the previous part, but there are significant differences.

```
#define TO_BO(name,num,update)                                                  \
  glBindBuffer(GL_ARRAY_BUFFER,bos_tiles[num]);                                 \
  if ( update ) glBufferData                                                    \
    (GL_ARRAY_BUFFER, name.size()*sizeof(name[0]), name.data(), GL_STREAM_DRAW); \
  glBindBufferBase(GL_SHADER_STORAGE_BUFFER,num,bos_tiles[num]);

      pShader_Use use(s_hw05_tiles_2);
      glUniform2i(1, opt_tryout1, opt_tryout2);
      glUniform1i(2, light_state_get());
      glUniform1f(3, world_time);

      TO_BO(pt_00,     1, pt_00_data_stale);
      TO_BO(ax,        2, axes_data_stale);
      TO_BO(ay,        3, axes_data_stale);
      TO_BO(color,     4, color_data_stale);
      TO_BO(tact_pos,  5, tact_data_stale);
      pt_00_data_stale = axes_data_stale = color_data_stale = tact_data_stale = false;

      glDrawArrays(GL_POINTS,0,tiles.size());
```

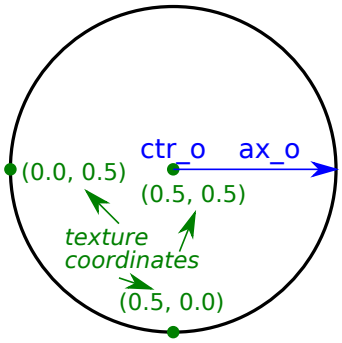
(*c*) Explain why the code above would be more efficient when rendering the very first frame and why it might be more efficient later in the execution, depending on how the tiles are used.

☐ More efficient for the very first frame because:

☐ More efficient later in execution because . . .

☐ This efficiency can be assumed by use of variable such as . . . because . . .

Problem 3: [30 pts] The code in this problem is similar to the Homework 5 tile code except that it will be used to render discs.



(*a*) Complete the geometry shader below so that it emits primitives for a disc (a filled-in circle) writing the geometry shader outputs shown. The provided shader code reads the shader disc center, `ctr_o`, disc normal, `nz_o`, and a vector to a point on the circumference, `ax_o` (see diagram), all in object space. The shader output includes `tcoord`, a texture coordinate. Assign this consistent with the texture coordinates shown on the diagram. Assume that the shader output primitive, a triangle fan, will work correctly even though it is not included in OpenGL Shading Language 4.5. A triangle fan makes the solution simpler. *Note: This triangle fan discussion was intentionally omitted from the original exam.*

☐ Emit primitives for the disc, make sure it's filled in.

☐ Don't forget to: ☐ Set `max_vertices`, ☐ set `normal_e`, ☐ set `tcoord`, and of course ☐ set `gl_Position`.

```
out Data_to_FS { flat vec3 normal_e;  vec3 vertex_e;  vec2 tcoord;};

layout ( points ) in;
layout ( triangle_fan, max_vertices =       ) out;  // <--- FILL IN

void gs_main_disc() {
  int vertex_id = In[0].vertex_id;
  vec4 ctr_o = ctrs[vertex_id];
  vec3 ax_o = axs[vertex_id].xyz;
  vec3 nz_o = nzs[vertex_id].xyz;

  const int slices = 10;
  const float pi = 3.1415926536;
  const float delta_theta = 2 * pi / slices;
// Abbrevs: glmvp, gl_ModelViewProjectionMatrix; glmv, gl_ModelViewMatrix; gln, gl_NormalMatrix




  for ( int i=0; i<slices; i++ )
    {
      float theta = i * delta_theta;
      float costh = cos(theta);
      float sinth = sin(theta);









    }
}
```

Problem 3, continued:

(*b*) Appearing below is the fragment shader for the code above. If variable `nevermind` were `true` the fragment shader would not write a fragment, but it's set to `false` in the code. (The `discard` keyword returns from the fragment shader without writing a fragment.)

The primitives emitted by the geometry shader (if solved correctly) will render a 10-sided polygon, which is not exactly a disc (circle). Modify the code so that it emits a perfect disc based on the largest circle that can fit inside the polygon. (For partial credit, a circle of radius 0.4 in texture coordinate units.) Use texture coordinates to determine whether a fragment is in the circle.

☐ Assign `radius` the correct value in terms of `slices`.

☐ Set `nevermind` so that a fragment is discarded if it's outside a radius-`radius` circle based on `tcoord`.

```
void fs_main_disc() {
  const int slices = 10;
  const float pi = 3.1415926535;
  const float delta_theta = 2 * pi / slices;



  float radius = 0.4;                          // CHANGE TO CORRECT VALUE.



  const bool nevermind = false;               // CHANGE FOR PERFECT CIRCLE



  if ( nevermind ) discard; // Don't write fragment, exit the shader.

  vec4 texel = texture(tex_unit_0,tcoord);   // NO NEED TO CHANGE THIS CODE.
  vec4 color = colors[vertex_id];
  gl_FragColor = texel * generic_lighting(vertex_e, color, normal_e);
  gl_FragDepth = gl_FragCoord.z;
}
```

(*c*) The inputs to fragment shader `fs_main_disc` appear below. Explain the implications of moving the `flat` qualifier as described below.

```
in Data_to_FS  {  flat vec3 normal_e;   flat int vertex_id;
                  vec3 vertex_e;        vec2 tcoord;              };
```

☐ Explain impact on ☐ correctness and ☐ efficiency if `flat` were removed from `normal_e` and `vertex_id`.

☐ Explain impact on ☐ correctness and ☐ efficiency if `flat` were added to `vertex_e` and `tcoord`.

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

(*a*) Explain how the depth (z-buffer) test is used. Provide an example in which sorting primitives by eye distance makes the depth test unnecessary, and another example in which even with sorting a depth test is necessary for proper rendering.

☐ Explain how depth test used.

☐ Example where sorting makes depth test unnecessary. *Hint: Example can have two primitives.*

☐ Example where depth test necessary even with sorting. *Hint: Example can have three primitives.*

(*b*) Appearing below are sample uses of two procedures related to the stencil buffer. Explain what each one does in general (not necessarily in the example).

```
glStencilFunc(GL_EQUAL,4,-1);
glStencilOp(GL_REPLACE,GL_KEEP,GL_KEEP);
```

☐ The `glStencilFunc` procedure is used to ….

☐ The `glStencilOp` procedure is used to ….

(*c*) In general, why doesn't it make sense to access a texture in a vertex shader?

☐ Bad idea to access a texture in a vertex shader because ...

(*d*) How is eye space defined? Describe the transformations that need to be applied to map from object space to eye space. Illustrate your answer using a sample scene.

☐ Defining features of eye space are ....

☐ In the following scene to transform from object to eye space the modelview matrix is used, which ...

(*e*) An NVIDIA GPU has 10 SMs. Consider a kernel which evenly divides work among its threads, the usual assumption made in class. Further assume that there is a large amount of work. Let $t(G)$ denote the execution time when the kernel is launched with G blocks. In all cases the block size is 1024 threads.

Let $a = t(10)$, the time when launched with 10 blocks on the 10-SM GPU.

☐ Find an expression for $t(5)$ in terms of a.

☐ Find an expression for $t(15)$ in terms of a.

☐ Find an expression for $t(20)$ in terms of a.

(*f*) Draw a sketch showing the shadow volume corresponding to a triangle. Include the triangle, the light source, some object in the shadow and some object seen through the shadow.

☐ Show: ☐ the light, ☐ triangle, ☐ shadow volume for the triangle, ☐ shaded object, ☐ object seen through shade.

10 Fall 2016

Name _____

GPU Programming
EE 4702-1
Midterm Examination

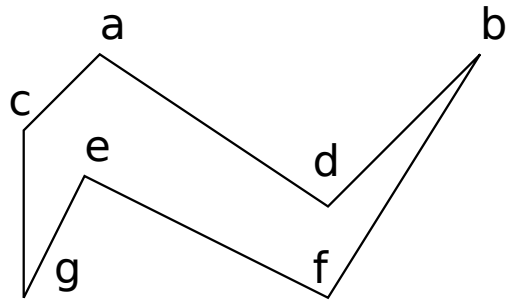
Friday, 28 October 2016 14:30–15:20 CDT

Problem 1	_____	(20 pts)
Problem 2	_____	(30 pts)
Problem 3	_____	(15 pts)
Problem 4	_____	(35 pts)
Exam Total	_____	(100 pts)

Alias _____

Good Luck!

Problem 1: [20 pts] Appearing below is a figure with labeled vertices. In the code below `pa`, `pb`, ... are initialized to the coordinates of the corresponding vertices. Two arrays are declared, `coords_wrong` and `coords`.




```
void World::render_p1(pCoor p1, pCoor p2, pCoor p3) {
  pCoor pa(2,4,0);
  pCoor pb(7,4,0);
  pCoor pc(1,3,0);
  pCoor pd(5,2,0);
  pCoor pe(1.8,2.4,0);
  pCoor pf(5,0.8,0);
  pCoor pg(1,0.8,0);

  pCoor coords_wrong[] = { pa, pb, pc, pd, pe, pf, pg };

  pCoor coords[] = {                                    } ; //  Fill In

  glMatrixMode(GL_MODELVIEW);
  glPushMatrix();
```

```
  pMatrix m =                                           ; //  Fill In

  glMultTransposeMatrixf(m);


  glBegin(GL_TRIANGLE_STRIP);
  for ( int i=0; i<coords.size(); i++ ) glVertex3fv( coords[i] );
  glEnd();    glPopMatrix();  }
```

(*a*) Initialize array `coords` so that the figure is rendered properly.

☐ Put `pa` ... `pg` into the `coords` initialization above in the correct order.

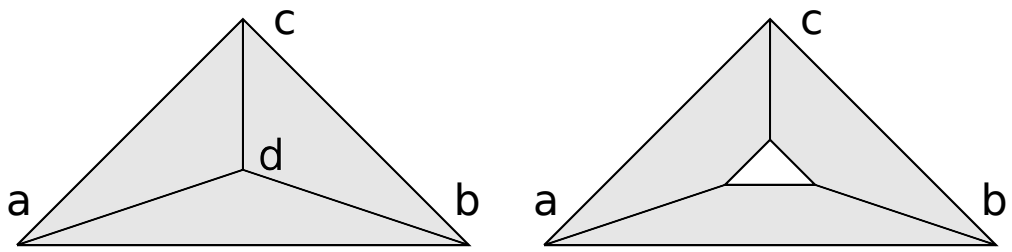
(*b*) Complete the code for the modelview matrix so that the whole figure is moved such that vertex `g` is at coordinate `p1`. (Note: the problem below also solves this problem.)

☐ Assign the correct value for `m` in the code above and make any other necessary changes.

(*c*) Set the modelview matrix so that the figure is rendered such that vertex `g` is at coordinate `p1`, vertex `c` is at `p2`, and all points are in the plane formed by `p1`, `p2`, and `p3`. This will move and rotate the figure, but won't change its shape. The matrix constructors `pMatrix_Rows rot1(v1,v2,v3)` and `pMatrix_Cols rot2(v1,v2,v3)` may come in handy. *Note: In the original version of the exam* `b` *was to be moved to* `p3`.

☐ Assign `m` to move the entire figure so that `g` is at `p1`, `c` is at `p2`, and `b` is at `p3`.

Problem 2: [30 pts] Illustrated below are two possible outputs of a geometry shader processing input primitive *abc*. The geometry shader code shown below, when completed, renders the figure on the left (this page's problem) or right (next page's problem).




```
layout (   triangle_strip,   max_vertices = 3   ) out;          // ☐ Fix.

void gs_main_1() {
  vec3 pt_sum = vec3(0);    vec3 n_sum = vec3(0);
  for ( int i=0; i<3; i++ ) { pt_sum += In[i].vertex_e.xyz; n_sum += In[i].normal_e; }
  vec4 center_e = vec4( pt_sum/3, 1 );
  vec3 center_norm_e = n_sum/3;




  for ( int i=0; i<=3; i++ ) {
      int idx = i%3;
      vertex_e = center_e;   normal_e = center_norm_e;
      gl_Position = In[idx].gl_Position;
      EmitVertex();

      normal_e = In[idx].normal_e;   vertex_e = In[idx].vertex_e;
      gl_Position = In[idx].gl_Position;
      EmitVertex(); }
  EndPrimitive(); }
```

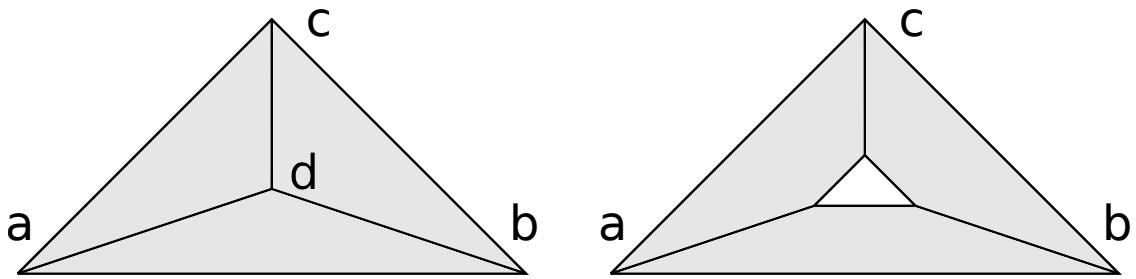
(*a*) Fix the layout declaration and modify the code so that `gl_Position` is assigned a correct value for the center point (d in the diagram). Assume that `center_e` is already correct. *Hint: Don't just look at the figure for the layout declaration fix.*.

☐ Fix the layout declaration. ☐ Assign `gl_Position` correctly for the center point.

(*b*) Suppose that the goal is to split *abc* into three triangles that each have $\frac{1}{3}$ the fragments that *abc* would have had. Does the code above (assuming `gl_Position` is set correctly) achieve that goal? If not, briefly explain how to modify the code so that there will be three triangles with $\frac{1}{3}$ the vertices each.

☐ Are fragments equally split? If no, explain how to fix.

Problem 2, continued:




```
layout (   triangle_strip,   max_vertices = 3   ) out;          // ☐   Fix.

void gs_main_1() {
  vec3 pt_sum = vec3(0);    vec3 n_sum = vec3(0);
  for ( int i=0; i<3; i++ ) { pt_sum += In[i].vertex_e.xyz; n_sum += In[i].normal_e; }
  vec4 center_e = vec4( pt_sum/3, 1 );
  vec3 center_norm_e = n_sum/3;









  for ( int i=0; i<=3; i++ ) {
      int idx = i%3;









      vertex_e = center_e;  normal_e = center_norm_e;
      gl_Position = In[idx].gl_Position;
      EmitVertex();

      normal_e = In[idx].normal_e;   vertex_e = In[idx].vertex_e;
      gl_Position = In[idx].gl_Position;
      EmitVertex(); }
  EndPrimitive(); }
```

(*c*) Modify the geometry shader so that it splits the triangle into the shape on the right, which is like the original triangle with a triangle-shaped hole in the center. The rendered primitives should only cover the area shown in gray. The exact area of the hole is not important, as long as it's not zero.

☐ Modify shader to render hole as shown in the shape on the right.

Problem 3: [15 pts] The incomplete code below is supposed to test whether pairs of balls *interpenetrate*, meaning that there is a volume of space that both balls occupy, and if so apply a separation force. The separation force works like an ideal spring pushing apart the balls. *Note: in the original exam the phrase defining interpenetration read "they occupy the same space".*

```
for ( int i=0; i<chain_length; i++ ) for ( int j=i+1; j<chain_length; j++ )
  {
    Ball* const ball_i = &balls[i];
    Ball* const ball_j = &balls[j];
    pCoor pos_i = ball_i->position;
    pCoor pos_j = ball_j->position;
    float rad_i = ball_i->radius;
    float rad_j = ball_j->radius;









    bool interpenetrating =                                    ; //  Fill in.

    if ( ! interpenetrating ) continue;









    ball_i->force +=                                           ; //  Fill in.

    ball_j->force +=                                           ; //  Fill in.
  }
```

(*a*) Write code to determine whether the two balls are interpenetrating, and assign the result to `interpenetrating`.

☐ Set `interpenetrating` to true if balls are intersecting.

(*b*) Write code to add on a separation forces to the balls. Use `sep_spring_constant` for the spring constant.

☐ Write code to add the separation force to balls' `force` members. ☐ Don't forget that force is a vector quantity.

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

(*a*) Convert homogeneous coordinate $\begin{bmatrix} 1 \\ 2 \\ 3 \\ 4 \end{bmatrix}$ into an ordinary Cartesian coordinate.

☐ Cartesian coordinate:

(*b*) Explain why `gl_Vertex`, the object-space vertex coordinate vertex shader input, is only in the compatibility profile (meaning that it's not really needed) but `gl_Position`, the output of the vertex and geometry shader stages is in the core profile (meaning that it is really needed).

☐ Input `gl_Vertex` **is not** really needed because:

☐ Output `gl_Position` **is** really needed because:

(*c*) The modelview matrix is predefined as a uniform variable. What would be the disadvantage of making it a vertex shader input?

☐ Disadvantage of making modelview a vertex shader input.

(*d*) The code below updates a buffer object whenever its old contents becomes outdated. Two lines have been accidentally commented out. What are the consequences of each.

```
//      if ( gpu_buffer_stale )  // LINE A, ACCIDENTALLY Commented out.
   {
//          if ( !gpu_buffer )   // LINE B, ACCIDENTALLY Commented out.
              glGenBuffers(1,&gpu_buffer);
            glBindBuffer(GL_ARRAY_BUFFER, gpu_buffer);
            glBufferData
              (GL_ARRAY_BUFFER, coords_size*sizeof(float),
               coords, GL_STATIC_DRAW);

            glBindBuffer(GL_ARRAY_BUFFER, 0);
            gpu_buffer_stale = false;
          }
```

☐ Consequences of LINE A commented out, but LINE B not commented out, in particular on ☐ performance and ☐ stability.

☐ Consequences of LINE B commented out, but LINE A not commented out, in particular on ☐ performance and ☐ stability..

(*e*) What are the typical operations performed in a fragment shader? What must the fragment shader do?

☐ Typical operations for fragment shader.

☐ Mandatory operation for fragment shader.

Name _____

GPU Programming
EE 4702-1
Final Examination

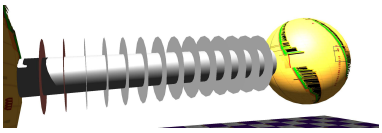
Monday, 5 December 2016 17:30–19:30 CST

Problem 1	_____	(20 pts)
Problem 2	_____	(20 pts)
Problem 3	_____	(15 pts)
Problem 4	_____	(20 pts)
Problem 5	_____	(25 pts)
Exam Total	_____	(100 pts)

Alias _____

Good Luck!

Problem 1: [20 pts] The geometry shader code below is based on the solution to Homework 6 Problem 2 in which a curved link is rendered by having the vertex shader compute points and vectors related to a curve and by having the geometry shader, whose input primitive type is a line strip, emitting a cylinder between the locations described by its two input vertices. Input `In[].t` is the position along the curve, with $0 \leq t \leq 1$.



Add code to the shader so that it draws rings around the cylinder. The inner radius should be the same as the cylinder, the outer radius should be twice the cylinder radius. Do not emit a ring at the link endpoints (the parts that touch the spheres). See the illustration to the upper right.

☐ Complete the code so that it draws the rings. ☐ Can use suggested and other reasonable abbreviations.

☐ Set `normal_e` properly don't forget to set ☐ `vertex_e` and ☐ `gl_Position`.

☐ The code should be reasonably efficient. ☐ Don't overlap primitives.

```
void gs_main_2() {
  const float rad = tex_rad[In[1].iid].z,  sides = sides_rad.x;
  for ( int j=0; j<=sides; j++ ) {
      const float theta = j * ( 2 * M_PI / sides );
      vec3 vect0 = cos(theta) * In[0].norm_e + sin(theta) * In[0].binorm_e;
      vec3 vect1 = cos(theta) * In[1].norm_e + sin(theta) * In[1].binorm_e;

      normal_e = vect1;    vertex_e = In[1].ctr_e + vec4( rad * vect1, 0);
      gl_Position = gl_ProjectionMatrix * vertex_e;  // SUGGESTED ABBREV: glP = glPM * v_e;
      EmitVertex();
      normal_e = vect0;    vertex_e = In[0].ctr_e + vec4(rad * vect0,0);
      gl_Position = gl_ProjectionMatrix * vertex_e;
      EmitVertex();   }
  EndPrimitive();
  float t0 = In[0].t,  t1 = In[1].t;   // Use if needed.




  for ( int j=0; j<=sides; j++ ) {
      const float theta = j * ( 2 * M_PI / sides );
      vec3 v0 = cos(theta) * In[0].norm_e + sin(theta) * In[0].binorm_e; // Use if needed.
      vec3 v1 = cos(theta) * In[1].norm_e + sin(theta) * In[1].binorm_e; // Use if needed.
      vec4 c0 = In[0].ctr_e,  c1 = In[1].ctr_e;   // Use if needed.









    }


}
```

Problem 2: [20 pts] Appearing below is a typical vertex shader.

```
in vec2 gl_MultiTexCoord0;  // Declare size.
void vs_main() {
  gl_Position = gl_ModelViewProjectionMatrix * gl_Vertex;
  vertex_e = gl_ModelViewMatrix * gl_Vertex;
  normal_e = normalize(gl_NormalMatrix * gl_Normal);
  gl_TexCoord[0] = gl_MultiTexCoord0;
  color = gl_Color;
}
```

(*a*) Suppose that the shader above was used in a rendering pass of v vertices. No buffer objects were used and all shader inputs were sourced from client arrays. Compute the amount of data sent from the CPU to the GPU for the rendering pass.

☐ CPU to GPU data for a rendering pass of v vertices counting shader inputs is:

☐ CPU to GPU data for a rendering pass of v vertices counting uniform variables is:

Problem 2, continued: The vertex shader below is used for an instanced draw of spheres. The vertex shader inputs are sourced from buffer objects and other buffer objects are accessed directly using the instance ID.

```
layout ( binding = 1 ) buffer sr { mat4 sphere_transform[]; };
layout ( binding = 3 ) buffer sc { vec4 sphere_color[]; };
void vs_main_instances_sphere() {
  mat4 transform = sphere_transform[gl_InstanceID];
  vec3 ctr = transform[3].xyz;         // Extract coordinate of sphere center.
  float radius = transform[3][3];      // Extract sphere radius.
  mat3 rot = mat3(transform);          // Extract rotation matrix.

  vec3 normal_o = rot * gl_Vertex.xyz;
  vec4 vertex_o = vec4( ctr + radius * normal_o, 1 );

  color = sphere_color[gl_InstanceID];
  gl_Position = gl_ModelViewProjectionMatrix * vertex_o;
  vertex_e = gl_ModelViewMatrix * vertex_o;
  normal_e = normalize(gl_NormalMatrix * normal_o );
  gl_TexCoord[0] = gl_MultiTexCoord0;  }
```

(*b*) In the code above label the vertex shader inputs and uniform variables.

☐ Label vertex shader inputs with a `I` and uniform variables with a `U`.

(*c*) Compute the amount of data sent from the CPU to the GPU for an instanced draw rendering pass in which n instances (spheres) are rendered, each consisting of v vertices. Assume that initially all buffer objects are on the CPU. Note that `mat4` is a 4×4 matrix of floats.

☐ Amount of CPU to GPU data for n instances of v vertices, accounting for ☐ uniforms, ☐ buffer objects, ☐ and anything else that really needs to be moved.

Problem 3: [15 pts] Consider our well written CUDA example (slightly simplified):

```
__global__ void kmain_efficient() {
  const int tid = threadIdx.x + blockIdx.x * blockDim.x;
  const int n_threads = blockDim.x * gridDim.x;

  // Note: There's no need to understand this code. Just remember
  // that each thread does about the same amount of work.

  for ( int h=tid; h<d_app.array_size; h += n_threads ) {
      float4 p = d_app.d_in[h];
      d_app.d_out[h] = dot(p,p);
    }
}
```

Let n denote the value of `array_size`, and let S denote the number of streaming multiprocessors (abbreviated SMs and sometimes MPs). Assume that n is large so that there's plenty of work to do for each thread.

(a) Determine a launch configuration in terms of n and S that uses the minimum number of blocks needed to maximize warp (or thread) occupancy on the SMs. Do this for a device with a block size limit of 1024 threads, an SM limit of 2048 threads, and an SM limit of 16 blocks. *Hint: There's no need to use both n and S.* Use B for the block size and G for the number of blocks (grid size).

☐ Block size $B =$

☐ Explain.

☐ Grid size (number of blocks) $G =$

☐ Explain.

(b) Let B and G be a solution to the problem above (meaning a good choice for B and G for S SMs) and let t_o be the execution time in a kernel launch of that configuration. Estimate the execution time for the configurations below. *Hint: Remember that G was chosen to maximize the number of warps per SM.*

☐ Estimate the execution time for a launch of G blocks of size $B - 1$. ☐ Explain.

☐ Estimate the execution time for a launch of $G - 1$ blocks of size B. ☐ Explain.

☐ Estimate the execution time for a launch of $G + 1$ blocks of size B. ☐ Explain.

Problem 4: [20 pts] Consider the CUDA kernel below. Let n denote the value of `array_size`, B the block size, and G the number of blocks.

```
__global__ void prob_x(float2 *d_in, float *d_out) {
  const int tid = threadIdx.x + blockIdx.x * blockDim.x;
  const int n_threads = blockDim.x * gridDim.x;

  for ( int h=tid; h<d_app.array_size-1; h += n_threads ) {
      float2 p = d_in[h];
      float2 q = d_in[h+1];
      d_out[h] = dot(p,q);
    }
}
```

(a) Compute how much data is read from global memory during the execution of the kernel. Note that each element `d_in[i]` is accessed by two threads.

☐ Amount of data read:

☐ Explain.

(b) The kernel below includes a declaration for shared memory intended to fix the inefficiency in the kernel above. Complete the kernel.

☐ Use shared memory to avoid waste.

```
__global__ void prob_x2(float2 *d_in, float *d_out) {
  const int tid = threadIdx.x + blockIdx.x * blockDim.x;
  const int n_threads = blockDim.x * gridDim.x;

  __shared__ float2 sm[1025];

  for ( int h=tid; h<d_app.array_size; h += n_threads ) {
      float2 p = d_in[h];









      float2 q =
      d_out[h] = dot(p,q);
    }
}
```

Problem 5: [25 pts] Answer each question below.

(*a*) Show a diagram illustrating the relationship between vertices and primitives in a triangle strip. Number both the vertices and the triangles in the correct order. Show at least three triangles.

☐ Diagram with number vertices and triangles illustrating a triangle strip.

(*b*) Describe what is tested in the stencil test and what the stencil test might be used for.

☐ A stencil test checks:

☐ Typical use for a stencil test:

(*c*) Interpolation qualifiers, `flat`, `noperspective`, and `smooth`, are used for the inputs of which rendering pipeline stage? What do they do?

☐ They appear at the input to:

☐ They affect the qualified variable by:

(*d*) Between which coordinate spaces does the OpenGL projection matrix transform.

☐ The projection matrix transforms coordinates ☐ from ☐ to

(*e*) What is the difference between a material property (including attributes emitted with `glColor3f`) and a lighted color?

☐ Difference between material property and lighted color?

11 Fall 2015

Name _____

GPU Programming
EE 4702-1
Midterm Examination

Monday, 26 October 2015 13:30–14:20 CDT

Problem 1	_____	(21 pts)
Problem 2	_____	(25 pts)
Problem 3	_____	(12 pts)
Problem 4	_____	(18 pts)
Problem 5	_____	(12 pts)
Problem 6	_____	(12 pts)
Exam Total	_____	(100 pts)

Alias _____

Good Luck!

Problem 1: [21 pts] In the code below three coordinates are defined, `ph` (P_h, head), `pt` (P_t, tail), and `ps` (P_s, surface).

(a) Complete the code below so that it renders a cylinder with axis $\overrightarrow{P_h P_t}$, with P_h at the top (flat part) of the cylinder, P_t at the bottom (the other flat part), and with P_s on the surface. Render only the curved parts, not the flat parts. The cylinder surface should be approximated by `slices` flat sides.

☐ Show code before the loop, such as for `ax` and `ay`.

☐ Show code in the loop that emits vertex ☐ coordinates and ☐ normals.

```
pCoor ph( 1.5, 0, -3.2 );
pCoor pt( 0,   5, -5 );
pCoor ps( 9,   6, -9 );
const int slices = 100;
const float delta_theta = 2 * M_PI / slices;
pNorm axis = pt - ph;
// Some of the solution goes here.




pVect ax =                                                    ; // FILL IN

pVect ay =                                                    ; // FILL IN




glBegin(GL_TRIANGLE_STRIP);
glColor3ub( 256, 89, 16 );
for ( int i=0; i<=slices; i++ )
  {
    const float theta = i * delta_theta;
    pVect n = cos(theta) * ax + sin(theta) * ay;
    // Some of the solution goes here.



    pCoor p1 =                                                ; // FILL IN
    glVertex3fv(p1);



    pCoor p2 =                                                ; // FILL IN
    glVertex3fv(p2);




  }
glEnd();
```

Problem 2: [25 pts] The code below, taken from the solution to Homework 2, renders the surface of a truss. Recall that the truss might be a part of a spinning gyroscope.

```
glEnableClientState(GL_VERTEX_ARRAY);
glVertexPointer(3, GL_FLOAT, sizeof(pCoor), coord_buffer);
glEnableClientState(GL_NORMAL_ARRAY);
glNormalPointer(GL_FLOAT, 0, norm_buffer);
glColor3fv(strip->color);
glDrawArrays(GL_TRIANGLE_STRIP, 0, elts);
glDisableClientState(GL_NORMAL_ARRAY);
glDisableClientState(GL_VERTEX_ARRAY);
```

(*a*) How much data is sent from the CPU to the GPU to render a frame based on the code above. Your answer should be in terms of the variables above.

☐ The amount of data sent per frame is ... ☐ Indicate unit.

(*b*) How could someone who is not familiar with Homework 2 be reasonably sure that the code above uses client arrays and not buffer objects? What would be different *in the parts shown* if buffer objects were used?

☐ We would guess code doesn't use buffer objects because ....

☐ Show changes to parts shown if buffer objects were used.

(*c*) Explain why using buffer objects in the code above would not make a big difference, whereas using buffer objects to render the balls (used to show nodes in the truss) in the same simulation does make a big difference?

☐ Buffer objects would not make a big difference for truss surface because ...

☐ Buffer objects do make a big difference for balls (part of truss) because ...

Problem 2, continued: The Homework 2 code from the previous page is repeated below.

```
glEnableClientState(GL_VERTEX_ARRAY);
glVertexPointer(3, GL_FLOAT, sizeof(pCoor), coord_buffer);
glEnableClientState(GL_NORMAL_ARRAY);
glNormalPointer(GL_FLOAT, 0, norm_buffer);
glColor3fv(strip->color);
glDrawArrays(GL_TRIANGLE_STRIP, 0, elts);
glDisableClientState(GL_NORMAL_ARRAY);
glDisableClientState(GL_VERTEX_ARRAY);
```

(*d*) Re-write the code so that it does not use client arrays, but instead uses calls like `glVertex`. *Hint: Only a few lines of code needed.*

☐ Re-write code so that calls like `glVertex` is used.

Problem 3: [12 pts] The code fragment below is from a demo-10 vertex shader.

```
void vs_main_basic() {
  gl_Position = gl_ModelViewProjectionMatrix * gl_Vertex;
  // Other code removed.
}
```

(*a*) Identity the variable that uses the storage types given below.

☐ Which variable is a vertex shader input?

☐ Which variable is a uniform variable?

☐ Which variable is a vertex shader output?

(*b*) Two of these variables appear in the OpenGL shading language compatibility profile but not in the undeprecated OpenGL SL standard, meaning that they aren't truly needed but are being kept so as not to break old software. Identify the variable that *is* truly needed and explain why.

☐ The undeprecated, truly needed variable is:

☐ It is truly needed because ...

Problem 4: [18 pts] Answer each question below.

(*a*) The interface block below is used as a fragment shader input. Among other things the fragment shader does lighting calculations. Explain what the `flat` qualifier does and why it is needed.

```
in Data_GF
{
  vec3 var_normal_e;
  vec4 var_vertex_e;
  flat vec4 color;
};
```

☐ The `flat` qualifier ...

☐ The `flat` qualifier helps this particular shader program by ...

(*b*) Suppose the depth (z) test was turned off. How would that affect the rendered image?

☐ Rendered image with depth test turned of would appear ...

(*c*) Describe what textures might be used for in a graphics program.

☐ An example of what a texture might be used for is ...

Problem 5: [12 pts] Answer the following questions.

(*a*) The code below renders two triangles. Add the minimum amount of code to make the first one blue and the second one orange (RGB:1,.35,.06). *Hint: This is an easy problem, don't overthink it. Note: The shade of orange is from the New York Mets logo.*

☐ Add minimum amount of code so that the first triangle is blue and the second is orange.

```
glBegin(GL_TRIANGLES);

glVertex3f( 10,  20,  0 );
glVertex3f( 15,  20,  0 );
glVertex3f( 10,   0,  0 );

glVertex3f( 30,  20,  0 );
glVertex3f( 30,   0,  0 );
glVertex3f( 25,   0,  0 );

glend();
```

(*b*) The code below renders two triangles. Add texture coordinates so that when the two triangles are put together they show the entire texture. The texture has already been set up, just provide the coordinates.

☐ Add OpenGL calls to provide texture coordinates.

```
glBegin(GL_TRIANGLES);

glVertex3f( 10,  20,  0 );
glVertex3f( 15,  20,  0 );
glVertex3f( 10,   0,  0 );

glVertex3f( 30,  20,  0 );
glVertex3f( 30,   0,  0 );
glVertex3f( 25,   0,  0 );

glend();
```

Problem 6: [12 pts] Clipping is performed between the output of the geometry shader and the input to the rasterizer.

(*a*) Provide a simple example of a triangle that can be completely clipped. The example should include the triangle's vertex coordinates in clip space.

☐ Easily completely clipable triangle coordinates:

(*b*) Provide an example of a primitive for which clipping would be difficult or tedious.

☐ Illustration (not coordinates) of a triangle that is tedious to clip.

☐ Briefly explain what makes it tedious.

(*c*) Suppose due to some error the clipping stage doesn't clip anything. Explain why there would be no difference in the appearance of the rendered image.

☐ Image no different if clip stage doesn't clip because ...

(*d*) Again, suppose due to some error the clipping stage doesn't clip anything. Explain why performance would suffer.

☐ Performance suffers because ...

Name _____

GPU Programming
EE 4702-1
Final Examination

Friday, 11 December 2015 15:00–17:00 CST

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

Alias _____

Good Luck!

Problem 1: [20 pts] The code below, taken from Homework 4, performs the rendering pass used to draw the balls. Let n denote the number of balls and let s denote the number of vertices for each ball. It does so using an instanced draw in which a set of s vertices is sent into the rendering pipeline n times, for a total of sn vertices. Procedure `glDrawArraysInstanced(type,0,s,n)` sends a set of s vertices for rendering n times.

```
// CPU code for rendering pass.

  glBindBufferBase(GL_SHADER_STORAGE_BUFFER,1,balls_pos_rad_bo);
  glBindBufferBase(GL_SHADER_STORAGE_BUFFER,2,balls_color_bo);
  glBindBuffer(GL_ARRAY_BUFFER,sphere.points_bo);
  glVertexPointer(3,GL_FLOAT,0,0);
  glEnableClientState(GL_VERTEX_ARRAY);
  glDrawArraysInstanced(GL_TRIANGLE_STRIP, 0, num_vertices, num_balls );

// Vertex Shader Code for rendering pass.

void vs_main_instances() {
  vec4 center_o_rad = balls_pos_rad[gl_InstanceID];
  float rad = center_o_rad.w;
  vec3 to_surface = rad * gl_Vertex.xyz;
  vec4 position_o = vec4(center_o_rad.xyz + to_surface, 1f);

  gl_Position = gl_ModelViewProjectionMatrix * position_o;
  vertex_e = gl_ModelViewMatrix * position_o;
  normal_e = gl_NormalMatrix * gl_Vertex.xyz;
}
```

(*a*) In the shader code above indicate which variables are uniforms and which are vertex shader inputs. *Hint: This question can be answered without understanding what an instanced draw is.*

☐ Put a `U` next to all uniform variables.

☐ Put an `I` next to all vertex shader inputs.

(*b*) For each rendering pass indicate how much data must be sent from the CPU to the GPU due to this vertex shader. Assume that **all uniform variables** need to be **re-sent each rendering pass**. But, make an intelligent determination on whether vertex shader inputs and other data needs to be re-sent.

☐ Amount of CPU to GPU data for uniform variables per pass in terms of s and n.

☐ Amount of CPU to GPU data for vertex shader inputs per pass in terms of s and n.

☐ Amount of CPU to GPU data for other data used by vertex shader per pass in terms of s and n.

Problem 1, continued:

(*c*) Appearing below is the vertex shader used for spheres in an ordinary rendering pass, followed by the vertex shader for the instanced rendering. Both render the same balls, but using different CPU code.

```
void vs_main() {                // Vertex Shader for Ordinary Rendering
  gl_Position = gl_ModelViewProjectionMatrix * gl_Vertex;
  vertex_e = gl_ModelViewMatrix * gl_Vertex;
  normal_e = gl_NormalMatrix * gl_Vertex.xyz;
}

void vs_main_instances() {      // Vertex Shader for Instanced Rendering
  vec4 center_o_rad = balls_pos_rad[gl_InstanceID];
  float rad = center_o_rad.w;
  vec3 to_surface = rad * gl_Vertex.xyz;
  vec4 position_o = vec4(center_o_rad.xyz + to_surface, 1f);

  gl_Position = gl_ModelViewProjectionMatrix * position_o;
  vertex_e = gl_ModelViewMatrix * position_o;
  normal_e = gl_NormalMatrix * gl_Vertex.xyz;
}
```

Because it accesses `balls_pos_rad`, `vs_main_instances` *looks like* it's accessing more data than `vs_main`, but in reality less data is being sent to render all the balls.

☐ What data was sent from CPU to GPU in the original code that substituted for `balls_pos_rad`?

☐ How did this data compare in size to `balls_pos_rad`?

(*d*) Explain why the instanced draw gives higher performance than ordinary rendering.

☐ Higher performance because ...

Problem 2: [15 pts] CUDA kernel A launched with 10 blocks of 1024 threads on a GPU with 10 streaming multiprocessors (abbreviated SMs or MPs) takes 27 s to run. Consider several scenarios in which the kernel is launched on a GPU with 9 SMs. The only difference between the GPUs is the number of SMs. In both GPUs the maximum number of threads per SM and per block is 1024.

(a) Suppose that kernel A is launched again with 10 blocks but this time on the GPU with 9 SMs.

☐ Run time for A in a 10-block launch on the 9 SM GPU? ☐ Explain.

(b) How long would A take to run on the 9-SM GPU if launched with 9 blocks of 1024 threads, but doing the same amount of work as the 10-block launch. Kernel A was written by a skilled programmer.

☐ Run time for A in a 9-block launch on the 9 SM GPU? ☐ Explain.

(c) CUDA kernel C launched with 10 blocks of **32 threads** takes 72 s on the 10-SM GPU. How long would the 10-block launch take on the 9-SM GPU?

☐ Run time for C in a 10-block launch on the 9 SM GPU?

☐ Explain how the low thread count makes part c (this question) different than part a.

Problem 3: [15 pts] The CUDA kernel below is based on a classroom example. Notice that the computation of `tid` has been changed. Indicate the change in execution time (relative to the commented out code) and whether the new code is correct (produces the same answer as the commented out code). *Note: The original exam contained a second part.*

(*a*) Consider the simple kernel:

```
__global__ void cuda_thread_start_simple() {
  // const int tid = threadIdx.x + blockIdx.x * blockDim.x;
  const int tid = threadIdx.x;                  // Changed code. Original line above.

  const int elt_per_thread = array_size / num_threads;
  const int start = elt_per_thread * tid;
  const int stop = start + elt_per_thread;

  for ( int h=start; h<stop; h++ ) {
      float4 p = d_v_in[h];
      d_m_out[h] = dot( p, p ); }}
```

☐ Due to change, execution time is ◯ *longer* ◯ *about the same or* ◯ *shorter* .

☐ Due to change, results are ◯ *incorrect* or ◯ *still correct.*

☐ Important: Explain by showing how array elements are assigned to threads.

Problem 4: [20 pts] Answer the following questions about CUDA.

(*a*) In an execution of the code below an SM reads 32 times more data than it needs to due to the way the code is written.

```
__global__ void some_prob(char *in_data, int *out_data) {
  const int tid = threadIdx.x + blockIdx.x * blockDim.x;
  const int width = 128;
  const int start_id = tid * width;
  int sum = 0;
  for ( int i=0; i<width; i++ ) sum += in_data[start_id+i];
  out_data[tid] = sum;
}
```

☐ Code reads 32× more data than it needs because ...

Suppose that the type of `in_data` were changed from `char*` to `int*`. Instead of reading 32× more data it would only be reading x× more data than needed.

☐ What is the value of x?

(*b*) The code sample below is based on the shared memory classroom demo. *Note: The second* `syncthreads` *did not appear in the original exam.*

```
__shared__ int sum;

if ( threadIdx.x == 0 )   sum = 0;
__syncthreads();
if ( threadIdx.x == 40 )  sum += 40;
if ( threadIdx.x == 70 )  sum += 70;
if ( threadIdx.x == 200 ) sum += 200;
__syncthreads();
out_data[tid] = sum;
```

☐ Show at least four different possible values for `sum` that the code above can write to `out_data`.

☐ Explain.

(*c*) Consider the use of `__syncthreads()` in the CUDA code below and in general.

☐ In general, what does `__syncthreads` do?

☐ What might go wrong if `__syncthreads` were removed from the code below?

```
__shared__ int our_data[1025];
our_data[threadIdx.x] = my_element;
__syncthreads();
output_data[tid] = my_element + our_data[threadIdx.x + 1];
```

☐ What's wrong with the use of `__syncthreads` below?

```
if ( threadIdx.x != 20 ) __syncthreads();
```

Problem 5: [20 pts] The geometry shader below, from Homework 4, passes a triangle unchanged.

```
layout ( triangles ) in;
layout ( triangle_strip, max_vertices = 3 ) out;

void gs_main_simple() {



  vec3 tnorm =                                  ; // FILL IN.



  for ( int i=0; i<3; i++ )
    {
      normal_e = In[i].normal_e;
      vertex_e = In[i].vertex_e;
      color = In[i].color;
      gl_Position = In[i].gl_Position;
      EmitVertex();
    }









  EndPrimitive();
}
```

(*a*) Add code so that `tnorm` is assigned the triangle's eye-space geometric normal.

☐ Set `tnorm` to eye-space geometric normal of triangle.

(*b*) Modify the shader so that it emits a second triangle of the same shape but displaced one unit in the direction of `tnorm` (see above). (For `normal_e` see next part.) The new triangle should be blue. The following library functions are available: `cross`, `dot`, `normalize`, and `length`. A library of colors **is not** available.

☐ Add code to emit second triangle.

☐ Be sure to assign ☐ `vertex_e`, ☐ `color` to blue, and (important) ☐ `gl_Position`.

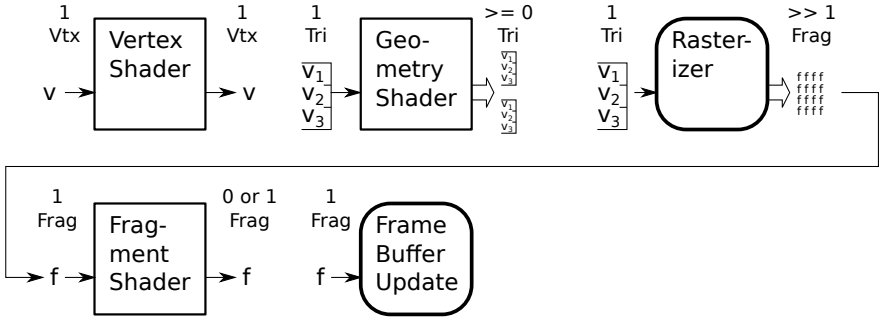
☐ Modify the `layout` declarations, if necessary.

(*c*) What about `normal_e` for the new triangle? Suppose we are rendering a sphere. Compare the appearance of the new triangle with `normal_e` set to `tnorm` to the appearance when `normal_e` is set to `In[i].normal_e`.

☐ Describe difference in appearance for `normal_e = tnorm` versus `normal_e = In[i].normal_e`.

Problem 6: [10 pts] Answer the following questions.

(*a*) The depth test (also called the z test) in an important part of the rendering pipeline.



☐ Where in the rendering pipeline is the depth test performed?

☐ Why would it make no sense to perform the depth test in the vertex shader?

(*b*) Switching from a triangle strip to individual triangles increases the amount of work performed by the vertex shader by a factor of 3, but does not change the amount of work performed by the geometry and fragment shaders.

☐ Why does the vertex shader do 3× more work?

☐ Why do the geometry and fragment shaders do the same amount of work?

12 Fall 2014

Name _____

GPU Programming
EE 4702-1
Midterm Examination

Wednesday, 12 November 2014 9:30–10:20 CST

Problem 1	_____	(18 pts)
Problem 2	_____	(12 pts)
Problem 3	_____	(24 pts)
Problem 4	_____	(12 pts)
Problem 5	_____	(14 pts)
Problem 6	_____	(20 pts)
Exam Total	_____	(100 pts)

Alias _____

Good Luck!

Problem 1: [18 pts] Appearing below is a simplified version of the vertex shader `vs_main` from Homework 4. One way to reduce the amount of computation it must perform is by pre-computing values and making them available as `vs_main` inputs. For example, rather than compute `radial_idx` each time, it can instead be pre-computed and stored in the `w` component of `gl_Vertex` or it can be stored in a user-defined vertex shader input named `radial_idx`.

```
void vs_main() {
  int bidx = gl_Vertex.x;               int ti = gl_Vertex.y;

  int radial_idx = bidx * opt_segments + ti;
  float theta = (1.0/opt_segments) * radial_idx * omega;

  vec3 pos1 = balls_pos[bidx-1].xyz;    vec3 pos2 = balls_pos[bidx].xyz;
  vec3 v12 = pos2.xyz - pos1.xyz;
  vec3 p = pos1 + float(ti) / opt_segments * v12;

  vec3 va = spiral_radius * normalize( vec3(0,v12.z,-v12.y) );
  vec3 vb = spiral_radius * normalize( cross(v12,va) );

  vec3 radial = va * cos(theta) + vb * sin(theta);
  vec3 p_outer = p + radial;            vec3 p_inner = p + 0.5 * radial;
```

Consider two candidates for new vertex shader inputs: `va/vb` (count these as one candidate) and `theta`.

(*a*) Estimate the amount by which each candidate would reduce the amount of computation to be performed **by the vertex shader**.

☐ Computation reduction by making `va/vb` an input:

☐ Computation reduction by making `theta` an input:

(*b*) One of the candidates would result in a large increase in CPU to GPU data transfer. Which one is it, and why?

☐ Candidate causing a big increase in CPU to GPU data transfer:

☐ Reason for **big** increase:

(*c*) In the code above there is a much better candidate for a new vertex shader input other than `va/vb` or `theta`. It would reduce computation by a significant amount and would not have a big impact on data transfer. *Hint: it's not a variable, but an expression, or several expressions.*

☐ The much better candidate for an input attribute is:

Problem 2: [12 pts] Appearing below is the fragment shader from the solution to Homework 4. Recall that this shader does not render the spiral surface in places where the texture is dark, giving the appearance of holes.

(*a*) Modify the shader code so that the texture itself is only applied to the front side of the spiral, and so that the holes still appear on **both** sides. *Hint: Can be done with a line or two of code.*

☐ Modify so holes are on both sides, but texture just on front.

```
void fs_main() {
  vec4 color = is_edge ? gl_FrontMaterial.ambient :
       gl_FrontFacing ? gl_FrontMaterial.diffuse : gl_BackMaterial.diffuse;

  vec4 texel = is_edge ? vec4(1,1,1,1) : texture(tex_unit_0,gl_TexCoord[0].xy);

  bool hole = texel.r + texel.g + texel.b < 0.05;
  if ( hole ) discard;

  gl_FragColor = texel * generic_lighting(vertex_e, color, normalize(normal_e));

  gl_FragDepth = gl_FragCoord.z;
}
```

(*b*) The fragment shader code below, based on Homework 4, is supposed to use different colors for the front, back, and edge of the spiral. The code below has an error and the commented-out line shows the correct code. How will this error change the appearance of the spiral?

☐ With this error spiral will appear...

```
void fs_main() {
  vec4 our_color = is_edge ? gl_FrontMaterial.ambient :
            gl_FrontFacing ? gl_FrontMaterial.diffuse : gl_BackMaterial.diffuse;

  vec4 texel = is_edge ? vec4(1,1,1,1) : texture(tex_unit_0,gl_TexCoord[0].xy);
  bool hole = texel.r + texel.g + texel.b < 0.05;
  if ( hole ) discard;

  //  gl_FragColor = texel * generic_lighting(vertex_e, our_color, normalize(normal_e));
  gl_FragColor = texel * our_color;

  gl_FragDepth = gl_FragCoord.z;
}
```

Problem 3: [24 pts] Answer each question below.

(*a*) An object at point P_1 is moving at velocity v_1 just before collision with a wall with normal $\hat{w}$. The collision is completely elastic, meaning the object's speed (the magnitude of velocity) will not change. Write an expression for the velocity after the collision?

☐ Expression showing velocity after collision:

(*b*) An object at P_1 is connected to an object at P_2 by an ideal spring having a relaxed distance of d and a spring constant h. Write an expression for the force on P_1 due to this spring.

☐ Expression giving force on P_1 due to spring:

(*c*) Point P_1 in 3D space has homogeneous coordinate $(3, 4, 6, 0.5)$. What is the corresponding Cartesian (ordinary every day) coordinate?

☐ Cartesian coordinate is:

(*d*) Homogeneous coordinates make it possible to express certain transformations using matrix multiplication that would not be possible with ordinary Cartesian coordinates. Name one such transformation.

☐ Transformation that's **not** possible using matrix multiplication with Cartesian coordinates:

Problem 4: [12 pts] Answer each question below.

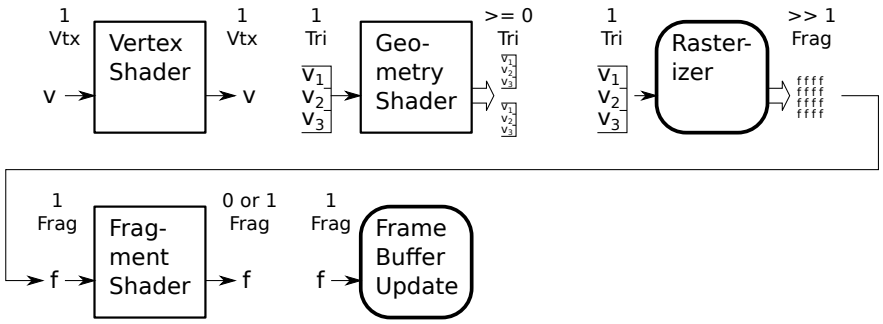
(*a*) Our user's eye is at location (x, y, z) in our coordinate system. How do we provide this information to OpenGL (using the compatibility profile)? Be specific.

☐ User's location given to OpenGL by ...

(*b*) The series of transformations performed by OpenGL (and shaders) maps 3D coordinates in our coordinate space to *window space* which are the 2D window coordinates corresponding to pixel locations. This transformation preserves coordinates' z component. Why is the z component needed, and how is it used?

☐ The z component is used for ...

Problem 5: [14 pts] The diagram below shows a simplified form of the OpenGL rendering pipeline.



(*a*) The diagram shows 16 f's at the output of the rasterizer stage, but that's not necessarily the actual number.

☐ What does the actual number of f's depend on?

☐ How could one increase the number of f's without changing the input, $(V_1V_2V_3)$.

(*b*) The boxes with rounded corners are part of what's called the *fixed functionality*. What does that mean, and why are those stages part of it.

☐ Fixed functionality means . . .

☐ The Rasterizer and Frame Buffer Update are fixed functionality because . . .

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

(*a*) The code below specifies vertices using a triangle strip. Modify the code so that it renders the same triangles using individual triangles.

☐ Modify code to use individual triangles ☐ without changing what's rendered.

```
glBegin(GL_TRIANGLE_STRIP);

for ( int i=0   ;    i<num_v   ;    i++    )
  {


    glVertex3fv( coord[i] );


  }
glEnd();
```

(*b*) There is an important difference between specifying a color using `glColor` and `glMaterial`. One of them allows each vertex to have its own color, the other allows different properties to be set, such as ambient, diffuse, and specular but all vertices in a rendering pass must share these colors. *Note: This question applies only to the compatibility profile.*

☐ The command that allows each vertex to have its own color is:

☐ Why can't each vertex have its own set of color properties using either of these commands?

(*c*) Describe how the `GL_LINEAR_MIPMAP_LINEAR` minification method computes a filtered texel. Indicate how many texels are read from texture images, which images are read, and how they are combined.

☐ Number of texture images read:

☐ Number of texels read:

☐ How they are combined:

Name _____

GPU Programming
EE 4702-1
Final Examination

Tuesday, 9 December 2014 7:30–9:30 CST

Problem 1	_____	(20 pts)
Problem 2	_____	(15 pts)
Problem 3	_____	(20 pts)
Problem 4	_____	(20 pts)
Problem 5	_____	(25 pts)
Exam Total	_____	(100 pts)

Alias _____

Good Luck!

Problem 1: [20 pts] Part of the vertex shader from Homework 6 (which is similar to the vertex shader in earlier assignments) appears below. For the questions below let c denote the chain length (the number of balls), let s denote the value of `opt_segments` (the number of iterations of the `t` loop in the CPU code), and let $n = cs$ denote the number of "vertices" sent to the vertex shader.

```
in ivec2 gl_Vertex; // Vertex Shader Inputs

out Data_to_GS {    // Interface block for vertex shader output / geometry shader input.
  vec4 vertex_e[2][2];  // Vertex coordinates in eye space.
  vec4 position[2][2];  // Vertex coordinates in clip space.
  vec2 texCoord[2];
  vec3 normal_e[2];
  vec3 radial_e;  // Normal for edge primitives.
};

void vs_main_lines() {
  const int bidx = gl_Vertex.x;
  const int ti = gl_Vertex.y;
  const int radial_idx = bidx * opt_segments + ti;
  const float delta_t = 1.0 / opt_segments;
  const float t = float(ti) * delta_t;
  const float theta = delta_t * radial_idx * omega;
  vec3 pos1 = balls_pos[bidx-1].xyz;                  // Buffer object access.
  vec3 pos2 = balls_pos[bidx].xyz;                    // Buffer object access.
  vec3 v12 = pos2.xyz - pos1.xyz;
  // ...
```

(a) For an entire rendering pass (execution of `glDrawArrays`) how much data is sent from the CPU to the GPU for this vertex shader? The answer should be in bytes and in terms of c, s, and n.

☐ Amount of data from CPU to GPU:

(b) For an entire rendering pass (execution of `glDrawArrays`) how much data is output by the vertex shader for the geometry shader? The answer should be in bytes and in terms of c, s, and n.

☐ Amount of data output from vertex shader:

(c) For an entire rendering pass (execution of `glDrawArrays`) how much data is read by the geometry shader (not shown) per frame? It's important to remember that the input to the vertex shader is line strips and the input to the geometry shader is lines. The answer should be in bytes and in terms of c, s, and n.

☐ Amount of data input to geometry shader:

Problem 1, continued: Recall that `balls_pos` is an array of ball positions, something that changes each frame. Note that the vertex shader is reading the positions from a buffer object.

It would be possible, though wasteful, to use a vertex shader input for the ball positions.

(*d*) Show the vertex shader input declarations that can be used for ball positions. *Hint: This part is easy, look at the code on the previous page.*

☐ Declarations:

(*e*) Compute the amount of data sent from the CPU to the GPU needed when the ball positions are read as vertex shader inputs. The answer should be in bytes and in terms of c, s, and n.

☐ Amount of data for ball positions as vertex shader inputs:

Problem 2: [15 pts] The fragment shader used in many class demonstrations and homework assignments performs lighting calculations. This yields the best results, but is computationally expensive. The historic location for computing lighting is the vertex shader.

(*a*) With the help of a diagram describe how fragment-shader lighting computation yields a more realistic image than vertex-shader lighting computation.

☐ Fragment shader lighting more realistic because:

(*b*) How many times more work does it take to compute lighting in the fragment shader? **State any assumptions** used for your answer.

☐ Using the fragment shader for lighting requires $x\times$ more computation assuming:

(*c*) Doing lighting calculations in the fragment shader does not always make a big difference in appearance. Consider a hybrid scheme in which a fragment-shader input determines whether the fragment shader should do the lighting calculations or just use the `color` input. Which stage is best suited to determine if the fragment shader should do lighting calculations?

☐ Shader stage best suited to determine where to compute lighting?

☐ Explain.

Problem 3: [20 pts] The CUDA kernel below computes an operation on all pairs of elements of array `d_pos`, the number of array elements is `chain_length`. Let B denote the block size of the kernel launch. The grid size (number of blocks) is (and must be) `chain_length`.

```
__global__ void force_pairs()
{
  const int a_idx = blockIdx.x;
  const float4 pos_a = d_pos[a_idx];                  // Global Access A

  float3 force = make_float3(0,0,0);

  for ( int b_idx = threadIdx.x;  b_idx < chain_length;  b_idx += blockDim.x)
    {
      const float4 pos_b = d_pos[b_idx];
      force += force_compute(pos_a,pos_b);
    }

  d_vel[a_idx] += delta_t * d_balls[a_idx].mass_inv * force; // Error here.
}
```

(*a*) Compute the amount of data read for the entire kernel due to the line Global Access A. The size of a `float4` is 16 bytes and the minimum request size is 32 bytes. Compute this number in terms of B (block size) and c (`chain_length`). *Note: In the original exam n was used for chain length.*

☐ Amount of data read due to Access A.

(*b*) Modify the code so that it uses shared memory to reduce the amount of data read due to Global Access A.

☐ Modify to use shared memory to reduce data read for A.

Problem 3, continued: The CUDA code below is the same as the previous page. The line marked "Error here" has two problems.

```
__global__ void force_pairs()
{
  const int a_idx = blockIdx.x;
  const float4 pos_a = d_pos[a_idx];                    // Global Access A

  float3 force = make_float3(0,0,0);

  for ( int b_idx = threadIdx.x;  b_idx < chain_length;  b_idx += blockDim.x)
    {
      const float4 pos_b = d_pos[b_idx];
      force += force_compute(pos_a,pos_b);
    }

  d_vel[a_idx] += delta_t * d_balls[a_idx].mass_inv * force; // Error here.
}
```

(*c*) Due to one problem the wrong value will be written into `d_vel`. Explain why.

☐ Wrong value written to `d_vel` because ....

(*d*) The line marked "Error here" is also inefficient. Explain why and fix the problem. *Hint: Use shared memory.*

☐ Reason for inefficiency.

☐ Fix with shared memory.

Problem 4: Answer each question below.

(*a*) [5 pts] A CUDA application launches a kernel with a grid size of 20 blocks. Describe a situation in which that is inefficient and explain why.

☐ Twenty blocks is inefficient when:

☐ Because:

(*b*) [5 pts] A CUDA application launches a kernel with a block size of 19 threads. Why is that inefficient?

☐ Nineteen threads is inefficient because:

Problem 4, continued:

(*c*) [5 pts] The loop body in the CUDA code below does one of five kinds of operations. Which is the biggest concern for the code below, grid size, block size, or warp size?

```
const int tid = threadIdx.x + blockIdx.x * blockDim.x;
for ( int i = tid; i<size; i += num_threads ) {
  int k = kind[i];
  float4 d = data[i];
  switch ( k ) {
    case 0: operation_x(i,d); break;
    case 1: operation_y(i,d); break;
    case 2: operation_z(i,d); break;
    case 3: operation_a(i,d); break;
    case 4: operation_b(i,d); break;
  }
}
```

☐ Biggest concern for code is grid, block, or warp. (Circle one.)

☐ Explain:

(*d*) [5 pts] For many classroom CUDA examples a structure is used to hold application related variables, such as the one below. For the GPU, this had been declared in the `__constant__` address space, but it could have been declared in the global (device) address space (shown commented out below). What's the disadvantage of the global space for this?

```
struct App {
  int num_threads, array_size;
  Vector *v_in;
  Vector *d_v_in;
};

App app;  // In host address space.
__constant__ App d_app;  // In device constant address space.
// __device__ App d_app;  // In device global address space.
```

☐ Disadvantage of global address space for the `App` structure?

Problem 5: Answer each question below.

(*a*) [5 pts] The vertex shader below uses the deprecated output `gl_FrontColor` to hold a lighted color. Those setting the OpenGL standard would prefer that you define your own vertex shader output to hold the lighted color. On the other hand, `gl_Position` is not deprecated and it must be written (if not by the vertex shader, then by the geometry shader). Why?

```
void vs_main() {
  gl_Position = gl_ModelViewProjectionMatrix * gl_Vertex;

  vec4 vertex_e = gl_ModelViewMatrix * gl_Vertex;
  vec3 normal_e = normalize(gl_NormalMatrix * gl_Normal);
  gl_FrontColor = generic_lighting(vertex_e,gl_Color,normal_e);
}
```

☐ Cannot substitute `gl_Position` with a user-defined output because:

(*b*) [5 pts] It's possible to define user-defined outputs for the vertex and geometry shaders, but not for the fragment or compute shaders.

☐ Why can't there be user-defined fragment shader outputs?

☐ Why can't there be user-defined compute shader outputs?

(*c*) [5 pts] In the OpenGL shader code below `b` is assigned the value of `a`, but with the elements reversed. It does so the hard way. Do it the easy way (using what we called swizzle operations).

```
vec4 a, b;
b.x = a.w;  b.y = a.z;  b.z = a.y;  b.w = a.x;
```

☐ Assign `b` the easy way.

(*d*) [5 pts] The fragment shader input's `smooth` interpolation qualifier (which is the default for floating-point values) indicates that the qualified variable (`color` in the example below) should be linearly interpolated based on object-space coordinates. Linear interpolation is easy to do, so why is `smooth` interpolation expensive?

```
in Data_GF {
  smooth vec4 color;
}
```

☐ `smooth` interpolation is expensive because . . .

(*e*) [5 pts] The statement below is correct but does not provide the full justification for GPUs. Complete the statement.

> Executing 3D graphics is computationally intensive and so many computers have GPUs to take this computation load off the CPU.

☐ To justify GPUs the statement should go on to say . . ..

13 Fall 2013

Name _____

EE 4702

Take-Home Pre-Final Questions

Due: 9 December 2013

Please work on this exam alone. You may use references for OpenGL, CUDA, and similar topics.

Problem 1	_____	(20 pts)
Problem 2	_____	(30 pts)
Problem 3	_____	(15 pts)
Problem 4	_____	(20 pts)
Problem 5	_____	(15 pts)
Exam Total	_____	(100 pts)

Alias _____

Good Luck!

Problem 1: [20 pts]The code below is the `data_unpack` routine from the Homework 3 solution. Recall that the code in Homework 3 requires that the number of threads must be no smaller than the number of balls (the value of `chain_length`).

Re-write the routine so that it works correctly even if the number of balls is greater than the number of threads.

☐ Re-write so number of threads can be fewer than `chain_length`.

☐ Compute the number of threads using CUDA-provided variables such as `blockDim`.

☐ Array `dc.d_pos` must be accessed efficiently.

```
__global__ void data_unpack()
{




  const int bi = threadIdx.x + blockIdx.x * blockDim.x;
  if ( bi >= dc.chain_length ) return;




  Ball* const ball = &dc.d_balls[bi];
  ball->position = dc.d_pos[bi];




}
```

Problem 2: [30 pts]Recall that the code for Homework 3 simulated a chain of balls (or string of beads). In this problem we'll extend that code by giving the balls electric charge, either positive or negative. The balls are sealed in a special coating that retains the charge.

The CUDA kernel below applies the force due to these charges. Like the penetration routine it must consider nearly all possible pairs. The code is written to optionally use shared memory when accessing the "b" object.

```
__global__ void time_step_gpu_sol_part_m(float delta_t) {
  const int tid = threadIdx.x + blockIdx.x * blockDim.x;
  const int threads_per_object = num_threads / dc.chain_length;

  const int a_idx = tid % dc.chain_length;
  const int b_idx_start = tid / dc.chain_length;

  __shared__ float4 c_pos[1024];

  float4 pos_a = dc.d_pos[a_idx];                                    // Global Access
  float3 force = make_float3(0,0,0);

  for ( int j=b_idx_start; j < dc.chain_length; j += threads_per_object )
    {
      if ( use_cache )
        {
          __syncthreads();
          if ( a_idx == 0 ) c_pos[b_idx_start] = dc.d_pos[j];        // Global Access
          __syncthreads();
        }

      if ( a_idx != j )
        {
          float4 pos_b = use_cache ? c_pos[b_idx_start] : dc.d_pos[j];  // Global Access
          pNorm a_to_b = mn(pos_a,pos_b);
          const bool repel = ( a_idx & 1 ) == ( j & 1 );
          force += ( repel ? -0.15f : 0.15f ) / a_to_b.mag_sq * a_to_b;
        }
    }

  const float3 delta_v = delta_t * dc.d_balls[a_idx].mass_inv * force;
  atomicAdd(&dc.d_vel[a_idx].x,delta_v.x);
  atomicAdd(&dc.d_vel[a_idx].y,delta_v.y);
  atomicAdd(&dc.d_vel[a_idx].z,delta_v.z); }
```

Questions are on next page.

Problem 2, continued:

(*a*) In this part assume that `use_cache` is false, and so shared memory won't be used. Compute the amount of data requested due to accesses to `d_pos` and compute how much of that data is actually used. Do this for the following configuration: `chain_length=256`, a block size of 1024 threads, and a grid size of 16 blocks. The code runs on an NVIDIA Kepler GPU which has memory request sizes of 32, 64, and 128 bytes. For your convenience, comments show where global memory can be accessed. (Array `c_pos` is not in global memory, and is anyway won't be needed until the next subproblem.)

☐ Total size of requests (in bytes) to `d_pos` considering all threads.

☐ Amount of data in those requests that was needed.

(*b*) Repeat the problem above for the case when `use_cache` is true. Remember that memory requests are not generated when reading or writing `c_pos` itself.

☐ Total size of requests (in bytes) to `d_pos` considering all threads.

☐ Amount of data in those requests that was needed.

(*c*) The `d_pos` accesses that are used to fill shared memory, `c_pos[b_idx_start] = dc.d_pos[j];`, are wasteful (though the waste is small in proportion to the total amount of data accessed). Explain why the accesses are wasteful and fix the problem.

☐ Fix problem.

☐ Why are these accesses wasteful?

(*d*) The amount of memory read when assigning `pos_a` can be reduced by using shared memory. Modify the code to do so.

☐ Use shared memory to reduce redundant global accesses to initialize `pos_a`.

Problem 3: [15 pts]Consider the following excerpt from the Cone code from the Homework 2 solution. (File `hw2-sol.cc` or visit `http://www.ece.lsu.edu/koppel/gpup/gpup/2013/hw2-sol.cc.html`).

```
void render_probb(pCoor base, float radius, pVect to_apex) {
    if ( opt_lod != bo_lod )  // If needed lod != stored lod.
      {
        bo_lod = opt_lod;

        /// Code for computing coordinates and normals omitted.


        if ( !buffer_obj_coords ) {
            glGenBuffers(1,&buffer_obj_coords);
            glGenBuffers(1,&buffer_obj_norms);  }

        glBindBuffer(GL_ARRAY_BUFFER, buffer_obj_coords);
        glBufferData(GL_ARRAY_BUFFER, coords.occ() * sizeof(float),
                     coords.get_storage(), GL_STATIC_DRAW);

        glBindBuffer(GL_ARRAY_BUFFER, buffer_obj_norms);
        glBufferData(GL_ARRAY_BUFFER, norms.occ() * sizeof(float),
                     norms.get_storage(), GL_STATIC_DRAW);
      }

    // Transform computation and modelview update omitted.

    glColor3fv(color);

    // Tell OpenGL to get coordinates and normals from buffer objects.
    //
    glBindBuffer(GL_ARRAY_BUFFER, buffer_obj_coords);
    glVertexPointer( 3, GL_FLOAT, 0, NULL);
    glEnableClientState(GL_VERTEX_ARRAY);

    glBindBuffer(GL_ARRAY_BUFFER, buffer_obj_norms);
    glNormalPointer(GL_FLOAT,0,NULL);
    glEnableClientState(GL_NORMAL_ARRAY);

    // Draw the cones. (There will be minor flaws since 1 strip used.)
    glDrawArrays(GL_QUAD_STRIP,0,num_coords);

    // Cleanup code omitted.
```

Questions are on next page.

Problem 3, continued:

(*a*) The code uses buffer objects for vertex coordinates and normals. Why doesn't it also use a buffer object for color?

☐ Why doesn't color need a buffer object?

(*b*) Modify the code so that it uses a buffer object for color, even if that's not a good idea.

☐ Modify code so color *does* use a buffer object for color.

(*c*) Suppose that each time the routine above was called `opt_lod` had a different value. Also suppose that `if ( !buffer_obj_coords )` were somehow changed to `if ( true )`.

☐ What would eventually happen?

Problem 4: [20 pts]Consider another excerpt from the Cone code from the Homework 2 solution. (File `hw2-sol.cc` starting at line 266. or visit `http://www.ece.lsu.edu/koppel/gpup/gpup/2013/hw2-sol.cc.html`).

```
// Compute transform moving and positioning code from local to global
// space.
//
const float to_height = to_apex.mag();
pVect from_apex(0,0,1);
pNorm rn(from_apex,to_apex);
const float rot_angle = pangle(from_apex,to_apex);
pMatrix_Translate trans_transl(base);
pMatrix_Rotation trans_rot(rn,rot_angle);
pMatrix_Scale trans_scale(radius);
trans_scale.rc(2,2) = to_height;
pMatrix xform = trans_transl * trans_rot * trans_scale;

// Specify our transformation to OpenGL.
//
glMatrixMode(GL_MODELVIEW);
glPushMatrix();
glMultTransposeMatrixf(xform.a);

if ( !dont_set_color ) glColor3fv(color);

glBindBuffer(GL_ARRAY_BUFFER, buffer_obj_coords);
glVertexPointer( 3, GL_FLOAT, 0, NULL);
glEnableClientState(GL_VERTEX_ARRAY);

glBindBuffer(GL_ARRAY_BUFFER, buffer_obj_norms);
glNormalPointer(GL_FLOAT,0,NULL);
glEnableClientState(GL_NORMAL_ARRAY);

glDrawArrays(GL_QUAD_STRIP,0,num_coords);
```

The code at the top of the excerpt computes a transform matrix that will position and scale the cone to the desired location. The transform is computed in terms of coordinate `base`, scalar `radius`, and vector `to_apex`. The command `glMultTransposeMatrixf` takes the transform matrix we computed and multiplies it by the existing modelview matrix. The resulting matrix is written as an OpenGL Shading Language uniform with name `gl_ModelViewMatrix`.

In this problem we'll consider computing the transform on the GPU instead. The GPU code will compute the transform in terms of `base`, `radius`, and `to_apex`.

Questions are on next page.

Problem 4, continued:

(*a*) Declare `base`, `radius`, and `to_apex` as uniforms as they would be in an OpenGL SL source file. *Hint: Use the demo-10 (Vertex and Geometry Shaders) code as an example, and look for* `wire_radius`. *The demo-10 code can be accessed from the repo or*
`http://www.ece.lsu.edu/koppel/gpup/code/gpup/demo-10-shader.cc.html` *(CPU code),*
`http://www.ece.lsu.edu/koppel/gpup/code/gpup/demo-10-shdr-simple.cc.html` *(simple shaders), and*
`http://www.ece.lsu.edu/koppel/gpup/code/gpup/demo-10-shdr-geo.cc.html` *(more elaborate shaders).*

(*b*) Write code to send the uniforms from the CPU to the GPU. *Hint: See the previous hint.*

(*c*) Suppose that `xform` is computed in a vertex shader making use of the enormous floating-point capability of the mighty GPU. Do you expect execution to be faster or slower than using the CPU to compute xform? Explain.

(*d*) Suppose that `xform` is computed in a geometry shader. Will execution be faster or slower than using the vertex shader when we are rendering quad strips? (That is, the systems we are comparing both use quad strips, one uses the vertex shader to compute the xform, the other uses the geometry shader.) Explain.

(*e*) Suppose that `xform` is computed in a geometry shader. Will execution be faster or slower than using the vertex shader when we are rendering individual quads? (That is, the systems we are comparing both use individual quads, one uses the vertex shader to compute the xform, the other uses the geometry shader.) Explain.

Problem 5: [15 pts]Answer each question below.

(*a*) Texture access is performed in the fragment shader. Suppose texture access was performed by the vertex shader and the texel values were interpolated, the same way other attributes such as color were.

☐ What impact would that have on appearance?

☐ What impact would that have on performance?

(*b*) Inputs to a fragment shader have interpolation qualifiers, they are `flat`, `noperspective`, and `smooth`. Why are they not needed for the inputs to the geometry shader?

☐ Interpolation qualifiers not needed for geometry shader inputs because:

(*c*) The OpenGL command `glMatrixMode(GL_MODELVIEW)`, used in setting the modelview matrix, is deprecated. (Meaning that its use is discouraged and that it may be removed from the language.) Why was the command once essential but now considered unnecessary?

☐ `glMatrixMode` now unnecessary because:

Name _____

GPU Programming
EE 4702-1
Final Examination

12 December 2013, 15:00–17:00 CST

Problem 1	_____	(25 pts)
Problem 2	_____	(25 pts)
Problem 3	_____	(15 pts)
Problem 4	_____	(10 pts)
Problem 5	_____	(15 pts)
Problem 6	_____	(10 pts)
Exam Total	_____	(100 pts)

Alias _____

Good Luck!

Problem 1: (25 pts) The code below computes forces between all pairs of a set of balls, similar to the code from the pre-final and Homework 3. Unlike those prior versions, the code below is written for the case where the number of balls (`chain_length`) is a multiple of the total number of threads.

Let c denote the number of balls, n denote the total number of threads (`num_threads`), and g denote the grid size (number of blocks). The size of a `float4` is 16 bytes. The code runs on an NVIDIA Kepler GPU, which supports requests of size 32, 64, and 128 bytes.

```
const int tid = threadIdx.x + blockIdx.x * blockDim.x;
const int num_threads = blockDim.x * gridDim.x;
const int a_idx_start = tid;
const int balls_per_thread = dc.chain_length / num_threads;
for ( int i=0; i<balls_per_thread; i++ )
  {
    const int a_idx = a_idx_start + i * num_threads;
    const float4 pos_a = dc.d_pos[a_idx];                  // Global Access A
    float3 force = make_float3(0,0,0);
    for ( int b_idx = 0;  b_idx < dc.chain_length;  b_idx++ )
      {
        const float4 pos_b = dc.d_pos[b_idx];              // Global Access B
        force += force_compute(pos_a,pos_b);
      }
    dc.d_vel[a_idx] += delta_t * dc.d_balls[a_idx].mass_inv * force;
  }
```

(a) In terms of n, c, and g compute the amount of data requested and used for the line marked Global Access A.

☐ Data requested by Global Access A. ☐ Data used by A.

(b) In terms of n, c, and g compute the amount of data requested and used for the line marked Global Access B.

☐ Data requested by Global Access B. ☐ Data used by B.

Problem 2: (25 pts) The code below is similar to the code from the previous problem except that shared memory is used to hold values needed for `pos_b`. The value of `csize` is not well chosen given the way the code is written. As before let c denote the number of balls, n denote the total number of threads, and g denote the grid size (number of blocks). The size of a `float4` is 16 bytes. The code runs on an NVIDIA Kepler GPU, which supports requests of size 32, 64, and 128 bytes.

```
int num_threads = blockDim.x * gridDim.x,  tid = threadIdx.x + blockIdx.x * blockDim.x,
int a_idx_start = tid,                     balls_per_thread = dc.chain_length / num_threads;
const int csize = 8;
__shared__ float4 c_pos[csize];
for ( int i=0; i<balls_per_thread; i++ ) {
    const int a_idx = a_idx_start + i * num_threads;
    const float4 pos_a = dc.d_pos[a_idx];                  // Global Access A
    float3 force = make_float3(0,0,0);
    for ( int b_idx = 0;  b_idx < dc.chain_length;  b_idx++ ) {
        const int c_idx = b_idx % csize;
        if ( c_idx == 0 ) {
            const int loc = threadIdx.x % csize;

            __syncthreads();

            c_pos[loc] = dc.d_pos[b_idx + loc];            // Global Access B

            __syncthreads();

          }
        const float4 pos_b = c_pos[c_idx];
        force += force_compute(pos_a,pos_b);
      }
    dc.d_vel[a_idx] += delta_t * dc.d_balls[a_idx].mass_inv * force;
  }
```

(*a*) In terms of n, c, and g compute the amount of data requested and used for the line marked Global Access B.

☐ Data requested by Global Access B. ☐ Data used by B.

(*b*) Choose `csize` to minimize memory access by Global Access B.

☐ Show new value of `csize`. ☐ Explain.

(*c*) Modify the code so that the amount of data accessed due to Global Access B matches the previous part when `csize` is set to the original value, 8. The change should be within the `if ( c_idx==0 )` block and it is only one line of code.

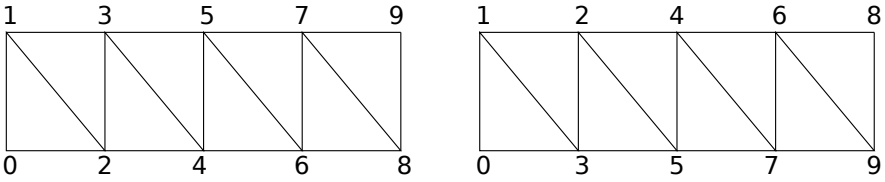
☐ Modify code to reduce waste when `csize=8`.

Problem 3: (15 pts) Answer the following questions about triangles.

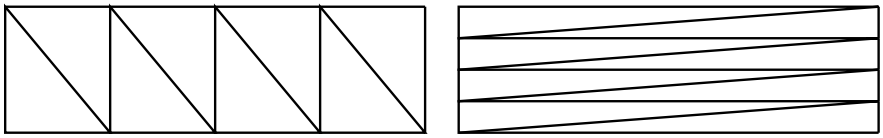
(*a*) The diagrams below show two possible numberings for vertices for rendering a rectangle as a triangle strip.

☐ Which one is correct?

☐ Show how the triangles would be rendered for the incorrect case.



(*b*) Several possible strategies are being considered to render a rectangle.



Suppose the left figure above were rendered as a triangle strip (with the vertices correctly numbered).

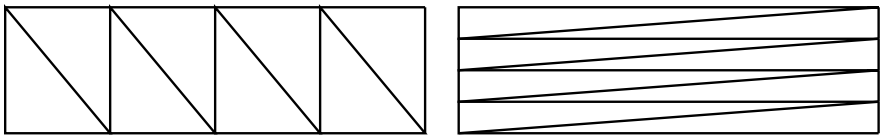
☐ How many times would the vertex shader be executed?

Suppose the left figure above was rendered as individual triangles.

☐ How many times would the vertex shader be executed?

Problem 3, continued:

(*c*) Consider the following two ways of tessellating a rectangle.



Suppose that lighting is computed in the vertex shader.

☐ Which tessellation above is better? ☐ Explain.

(*d*) Suppose that lighting were being computed in the fragment shader.

☐ Why would either tessellation above be wasteful?

Problem 4: (10 pts) In class we saw how we could obtain the mirror image of a vertex easily if the mirror were on the xz plane at $y = 0$. For such a mirror, the mirror image of a vertex at (x, y, z) would be at $(x, -y, z)$.

Appearing below are a vertex shader and geometry shader that do nothing special: The vertex shader computes the clip-space coordinates and lighting, and the geometry shader emits one triangle.

Modify the shader(s) so that for each original triangle two are emitted: one in the original position, and the other in the reflected position. (Don't worry about stenciling or anything like that.)

Hint: Don't forget that the mirror is in object-space (world-space) coordinates.

☐ Modify to emit reflected triangle, in addition to regular.

☐ Think about coordinate spaces.

```
#ifdef _VERTEX_SHADER_
void vs_main_basic() {
  gl_Position = gl_ModelViewProjectionMatrix * gl_Vertex;

  // Compute eye-space vertex coordinate and normal.
  // These are outputs of the vertex shader and inputs to the frag shader.
  vec4 var_vertex_e = gl_ModelViewMatrix * gl_Vertex;
  vec3 var_normal_e = normalize(gl_NormalMatrix * gl_Normal);
  vec4 lcolor = generic_lighting( var_vertex_e, gl_Color, var_normal_e);

  gl_BackColor = gl_FrontColor = lcolor;
}

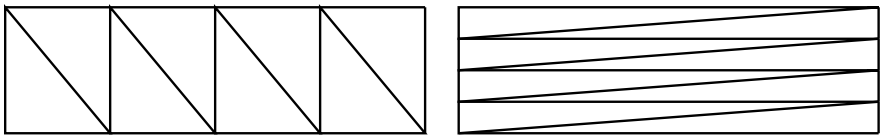
out Data  {
  int hidx;  // Not used.
};
#endif

#ifdef _GEOMETRY_SHADER_
in Data {
  int hidx;  // Not used.
} In[3];

void gs_main_helix() {
  // Pre-defined input: gl_PositionIn[] (array of vec4, size determined by prim)
  // Pre-defined output: gl_Position (a vec4, read when EmitVertex called).
  for ( int i=0; i<3; i++ )
    {
      gl_FrontColor = gl_FrontColorIn[i];
      gl_BackColor = gl_BackColorIn[i];
      gl_Position = gl_PositionIn[i];
      EmitVertex();
    }
  EndPrimitive();
}
#endif
```

Problem 5: (15 pts) Answer each question below.

(*a*) Suppose we have a buffer object for vertices corresponding to the object on the lower left.



☐ How can we use that buffer object to render an object that will look like the one on the right?

(*b*) Describe a situation in which using `glVertex` rather than vertex arrays (`glDraw`) is a good idea.

☐ Situation in which `glVertex` better:

(*c*) Describe a situation in which using vertex arrays but without buffer objects is a good idea.

☐ Situation in which vertex arrays without buffer objects better:

Problem 6: (10 pts) In class the topic of texture filtering was introduced with a description of what would happen if a texture image of a picket fence (white stripes in front of a green background) where rendered without an appropriate MIPMAP level and without filtering. Either the fence or the grass would disappear.

(*a*) Illustrate the problem. On your diagram show the fence, some pixels and some texels. It is important to choose the size of the pixels and texels correctly.

☐ Illustration of problem. ☐ Show pixels. ☐ Show texels.

(*b*) Re-draw the diagram showing how the choice of an appropriate MIPMAP level can avoid the problem of the fence disappearing.

☐ Illustration of how MIPMAP level avoids problem.

(*c*) Draw a diagram showing an example of how linear filtering can improve the appearance of the fence.

☐ Example of how linear filtering improves appearance.

14 Fall 2012

Name _____

EE 4702

Take-Home Pre-Final Questions

Tuesday, 4 December 2012

These are practice questions for the final exam.

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

Alias _____

Good Luck!

Problem 1: [20 pts]The advantage of an OpenGL triangle strip over individual triangles is that vertices shared by multiple triangles only need to be specified once.

(*a*) Why is the code below not a good example?

```
for ( int i=0; i<1000; i++ )
  {
    glBegin(GL_TRIANGLE_STRIP);
    glNormal3fv(normal[i]);  glVertex3fv(v0[i]);
    glNormal3fv(normal[i]);  glVertex3fv(v1[i]);
    glNormal3fv(normal[i]);  glVertex3fv(v2[i]);
    glEnd();
  }
```

(*b*) Explain the impact of the proper use of triangle strips on each of the items below. For one of the items below the answer should consider two cases, when a buffer object is being used and when it is not being used. Remember, for this part assume that triangle strips are being used properly.

- Total communication bandwidth.

- Total vertex shader execution time.

- Total fragment shader execution time.

(*c*) A triangle strip is well suited for situations in which triangles are approximating a smooth surface (such as the surface of a sphere). It is less well suited for surfaces that really are composed of triangles. Explain why. *Hint: Consider colors and normals.*

Problem 2: [20 pts]Assume that the lighted color of a triangle depends on conditions at its centroid (a point equidistant from its three vertices), including light and viewer location. Though computed for the center, this same lighted color would be used for all fragments.

(*a*) Which would be the most appropriate shader to compute this color: vertex, geometry, or fragment? Explain.

(*b*) Regardless of your answer above, explain how a vertex shader could be used to compute the centroid (a first step in computing the color). This will require some extra help from the CPU, but the burden of computation should be on the shader. (*Note: the challenge here is getting the vertex locations.*)

(*c*) Shown below are three possible declarations for the output of the shader that computes the triangle's lighted color. Each of these will produce the desired result. But one is clearly best, and one is clearly worst. Identify the best and worst to use and explain why for each.

```
flat out vec3 tri_color;
smooth out vec3 tri_color;
noperspective out vec3 tri_color;
```

Problem 3: [20 pts]Suppose an NVIDIA GPU of compute capability 2.0 has 8 multiprocessors. Each multiprocessor has 32 cores, but a single warp can use only 16 cores (it takes two different warps to keep all 32 cores in a multiprocessor busy).

The GPU is to be used to add two arrays element-wise. Suppose that the number of array elements is 2^{24}.

Let t_1 denote the amount of time it takes one thread (yes, just one) to perform the entire calculation on the GPU. The kernel code is shown below:

```
__device__ void prob(int array_size) {
  int tid = threadIdx.x + blockIdx.x * blockDim.x;
  for ( int i=tid; i<array_size; i += num_threads )
    result[i] = a[i] + b[i];
}
```

(*a*) Compute the amount of time it would take to perform the computation for each of the following launch configurations, and explain why they are the same or different (whichever the case applies):

☐ One block of 16 threads.

☐ Two blocks of 8 threads each.

☐ Explain why same or different.

(*b*) Estimate (an exact answer isn't possible) the amount of time it would take to perform the computation for each of the following launch configurations and explain why they are the same or different (whichever the case applies):

☐ One block of 1024 threads.

☐ Two blocks of 512 threads each.

☐ Explain why same or different.

(*c*) Explain how higher memory latency would increase the number of threads needed for good performance.

Problem 4: [20 pts]Consider the CUDA code below in which two alternatives are shown: putting a balls array in shared memory or in global memory. Assume the code runs on GPUs of compute capability 2.0 (though the answers could apply to 1.0 through 2.5).

(*a*) Explain why the Option 1 (shared memory) code below will run slowly. Fix the problem making as few changes as possible.

```
struct Ball {
  float3 position;   float3 velocity;
  float radius;      float mass;      };

__shared__ Ball balls[256]; // Option 1 - Shared memory.

__device__ void
update(float deltat)
{
  int start = threadIdx.x * balls_per_thread;
  int stop = start + balls_per_thread;
  for ( int i=start; i<stop; i++ )
    balls[i].position += deltat * balls[i].velocity;
}
```

(*b*) Explain why the Option 2 code below (using global memory) will run slowly. Fix the problems making as few changes as possible. (There are two major things to be fixed.)

```
struct Ball {
  float3 position;   float3 velocity;
  float radius;      float mass;      };

Ball* balls; // Option 2 - Global memory.

__device__ void
update(float deltat)
{
  int start = ( threadIdx.x + blockIdx.x * blockDim.x ) * balls_per_thread;
  int stop = start + balls_per_thread;
  for ( int i=start; i<stop; i++ )
    balls[i].position += deltat * balls[i].velocity;
}
```

Problem 5: [20 pts]Answer each question below.

(*a*) Why is the if/return needed in the CUDA code below?

```
__device__ void prob(int array_size) {
  int tid = threadIdx.x + blockIdx.x * blockDim.x;
  if ( tid >= array_size ) return;
  result[tid] = a[tid] + b[tid];
}
```

(*b*) The line of code below checks for a special case to avoid a square root. That might make sense for a CPU, but not for necessarily for CUDA code on a GPU. Describe a situation in which it makes sense for CUDA and a different situation when it makes no sense (meaning it would be faster to do the square root all the time). Assume that 50% of the time d is equal to 1.

```
if ( d == 1 ) s = 1; else s = sqrt(d);
```

(*c*) The loop below is innocent on a CPU, but on a GPU it can execute inefficiently. Identify the problem and fix it.

```
for ( int i=0; i<32; i++ )
  {
    if ( a[i] == t ) { do_stuff(i); break; }
  }
```

Name _____

EE 4702

Final Exam

Friday, 7 December 2012, 12:30-14:30 CST

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

Alias _____

Good Luck!

Problem 1: [15 pts]The two code fragments below render the same set of triangles.

```
// Code Fragment A

 glBegin(GL_TRIANGLES);
 for ( int i=0; i<1000; i++ )
   {
     glColor3fv(lsu_spirit_gold); glVertex3fv(v0[i]);
     glColor3fv(lsu_spirit_gold); glVertex3fv(v1[i]);
     glColor3fv(lsu_spirit_gold); glVertex3fv(v2[i]);
   }
 glEnd();

// Code Fragment B
 glColor3fv(lsu_spirit_gold);
 glBegin(GL_TRIANGLE_STRIP);
 for ( int i=0; i< XXX; i++ )
     glVertex3fv(v[i]);
 glEnd();
```

(*a*) Compute the amount of data sent from CPU to GPU for each case. Use an appropriate value of XXX.

Problem 2: [15 pts]Consider the declarations below which come from the vertex shader in the Homework 3 solution.

```
layout ( location = 2 ) uniform float wire_radius;
layout ( location = 3 ) uniform float theta;
layout ( location = 4 ) uniform vec2 texture_scale;
layout ( binding = 1 ) buffer Helix_Coord  { vec4  helix_coord[];  };
layout ( location = 1 ) in ivec2 helix_index;
out vec3 normal_e;
uniform sampler2D tex_unit_0;
```

(*a*) What do the location numbers indicate in the uniform declarations?

(*b*) How is a `uniform` typically used?

(*c*) Explain what the `in` and `out` qualifiers used for `helix_index` and `normal_e` indicate.

(*d*) Why can't the declarations above be for a geometry shader? (That is, something won't work.)

(*e*) Why can't the declarations above be for a fragment shader? (That is, something won't work.)

Problem 3: [20 pts]The Homework 5 code simulated a spring, bringing to life the helix we were working with most of the semester. A combination of CPU and GPU code was used to compute the wire surface coordinates. Excerpts of the code appear below.

Grading Note: This problem was based on code used in Homework 5, with which some familiarity was expected.

```
Physics Simulation in CUDA computes helix_position and orientation, read to CPU.
              Wire_Segment* const ws = &wire_segments[i];
              ws->position = helix_position[i];
              ws->orientation = helix_orientation[i];

Computation on CPU:
          helix_coords[i] = ws->position;
          pMatrix_Rotation c_rot(ws->orientation); // Mult: 3 * 9 = 27
          helix_u[i] = c_rot * pVect(0,0,1);
          helix_v[i] = c_rot * pVect(0,1,0);

Computation on Vertex Shader
  vec3 vtx = helix_coord[hidx.x].xyz;
  float pi = 3.14159265;
  float theta = hidx.y * 2 * pi / 20;
  vec3 u = helix_u[hidx.x].xyz;
  vec3 v = helix_v[hidx.x].xyz;
  vec3 normal = normalize( cos(theta) * u + sin(theta) * v );
  // Compute wire surface location by adding normal to helix coordinate.
  //
  vec4 vertex_o;
  vertex_o.xyz = helix_coord[hidx.x].xyz + wire_radius * normal;
  vertex_o.w = 1;
```

Computing the u and v vectors on the CPU requires about 18 multiplications. Consider the method used in the code above and two alternatives: computing these in CUDA or computing them in the vertex shader.

The values of `hidx.x` (position [index of wire segment] along the helix) vary from 0 to $S-1$ and the values of `hidx.y` (position around the cylinder of a wire segment) vary from 0 to $M-1$.

(*a*) Compute the amount of data moving between CPU and GPU (in both directions) for the code shown.

(*b*) Compute the amount of data that would be moved if u and v were computed in CUDA.

(*c*) Compute the amount of data that would be moved if u and v were computed in the vertex shader.

(*d*) Which option computes the same thing multiple times? Is there any benefit to this redundancy?

Problem 4: [15 pts]A goal of Homework 2 was to use `glDrawElements` in place of `glDrawArray`. For our task of rendering a helix, `glDrawElements` reduces the number of vertices that need to be sent from CPU to GPU by a factor of 2.

(*a*) Explain why half the number of vertices are needed.

(*b*) When using `glDrawElements` what needs to be sent from CPU to GPU that was not needed with `glDrawArray`? How large is it in comparison to the vertex arrays?

Problem 5: [15 pts]Answer the following questions abut coordinate spaces. For the transformations use $T_{\vec{v}}$ to indicate a translation transformation and $R_{\vec{u}\to\vec{v}}$ indicate a rotation transform that rotates $\vec{u}$ to $\vec{v}$.

(*a*) Let P_o be a coordinate in object space coordinates, let E be the location of the eye and $\vec{v}$ be a normal to the projection surface. Show how to transform P_o into an eye-space coordinate.

(*b*) Let P be a point in clip space. How can we determine if P is in the view volume?

(*c*) Suppose we are rendering to a window of size 1000×1000 pixels. Let $P_c = (0.5, 0.5, 0.5)$ be a coordinate in clip space.

☐ Provide an example of clip space coordinate $P_d \neq P_c$ that is likely to be in the same pixel as P_c.

☐ Provide an example of clip space coordinate that is in a pixel close to P_c.

Problem 6: [20 pts]Consider a multiprocessor in an NVIDIA CC 2.0 GPU, the type frequently discussed in class. Such a multiprocessor had 32 CUDA cores, two schedulers, and contexts for 48 warps. For all questions below assume a kernel is launched with one block per multiprocessor.

(*a*) Explain why a block size of one warp is guaranteed to leave half the cores unused.

(*b*) Explain why a block size of two warps will likely result in cores being idle most of the time.

For the next two parts consider Kernel A and Kernel B, below. Notice that in Kernel A I1 must wait for I0 and I2 must wait for I1. In Kernel B I3 must wait for I0, but I1 does not have to wait for I0. Inspect the code for additional dependencies. Suppose that this pattern of dependencies continues in each kernel. That is, in Kernel A each instruction depends on its immediate predecessor, while in Kernel B each depends on the third previous instruction.

```
# Kernel A
I0:  add  r1, r2, r3
I1:  mul  r4, r1, r5
I2:  sub  r6, r4, r7
I3:  or   r8, r6, r9
...

# Kernel B
I0:  add  r1, r2, r3
I1:  add  r10, r12, r13
I2:  add  r20, r22, r23
I3:  mul  r4, r1, r5
I4:  mul  r14, r11, r15
I5:  mul  r24, r21, r25
```

(*c*) Suppose that each kernel is launched with two warps per block (and one block per MP). Assume that the instruction latency is 24 cycles. Draw a diagram showing the time that each instruction executes, until the pattern is obvious.

(*d*) Compute the CUDA core utilization (the fraction of time a core will be used) for each kernel.

15 Fall 2011

Name _____

EE 4702

Take-Home Pre-Final Examination

Tuesday, 29 November 2011 to Friday, 2 December 2011

Work on this exam alone. Regular class resources, such as notes, papers, documentation, and code, can be used to find solutions. Do not try to seek out references that specifically answer any question here. Do not discuss this exam with classmates or anyone else. Any questions or concerns about problems should be directed to Dr. Koppelman.

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

Alias _____

Good Luck!

Problem 1: [20 pts]The advantage of an OpenGL triangle strip over individual triangles is that vertices shared by multiple triangles only need to be specified once.

(*a*) Why is the code below not a good example?

```
for ( int i=0; i<1000; i++ )
  {
    glBegin(GL_TRIANGLE_STRIP);
    glNormal3fv(normal[i]);  glVertex3fv(v0[i]);
    glNormal3fv(normal[i]);  glVertex3fv(v1[i]);
    glNormal3fv(normal[i]);  glVertex3fv(v2[i]);
    glEnd();
  }
```

(*b*) Explain the impact of the proper use of triangle strips on each of the items below. For one of the items below the answer should consider two cases, when a buffer object is being used and when it is not being used. Remember, for this part assume that triangle strips are being used properly.

- Total communication bandwidth.

- Total vertex shader execution time.

- Total fragment shader execution time.

(*c*) A triangle strip is well suited for situations in which triangles are approximating a smooth surface (such as the surface of a sphere). It is less well suited for surfaces that really are composed of triangles. Explain why. *Hint: Consider colors and normals.*

Problem 2: [20 pts]Assume that the lighted color of a triangle depends on conditions at its centroid (a point equidistant from its three vertices), including light and viewer location. Though computed for the centroid, this same lighted color would be used for all fragments.

(*a*) Which would be the most appropriate shader to compute this color: vertex, geometry, or fragment? Explain.

(*b*) Regardless of your answer above, explain how a vertex shader could be used to compute the centroid (a first step in computing the color). This will require some extra help from the CPU, but the burden of computation should be on the shader. (*Note: the challenge here is getting the vertex locations.*)

(*c*) Shown below are three possible declarations for the output of the shader that computes the triangle's lighted color. Each of these will produce the desired result. But one is clearly best, and one is clearly worst. Identify the best and worst to use and explain why for each.

```
flat out vec3 tri_color;
smooth out vec3 tri_color;
noperspective out vec3 tri_color;
```

Problem 3: [20 pts]Suppose an NVIDIA GPU of compute capability 2.0 has 8 multiprocessors. Each multiprocessor has 32 cores, but a single warp can use only 16 cores (it takes two different warps to keep all 32 cores in a multiprocessor busy).

The GPU is to be used to add two arrays element-wise. Suppose that the number of array elements is 2^{24}.

Let t_1 denote the amount of time it takes one thread (yes, just one) to perform the entire calculation on the GPU. The kernel code is shown below:

```
__device__ void prob(int array_size) {
  int tid = threadIdx.x + blockIdx.x * blockDim.x;
  for ( int i=tid; i<array_size; i += num_threads )
    result[tid] = a[tid] + b[tid];
}
```

(*a*) What would be the time to perform the calculation for a single block of just two threads (in terms of t_1)?

(*b*) Consider a launch of eight blocks of 64 threads each. Explain why reducing the number of blocks by one will leave some CUDA cores totally unused.

(*c*) Consider a launch of eight blocks of 64 threads each. Explain why reducing the block size by 17 or more will leave some CUDA cores totally unused.

(*d*) Explain why a launch of 16 blocks of 512 threads would take about the same amount of time to run as a launch of 32 blocks of 256 threads. *Note: Up to 8 blocks or 48 warps can run on a multiprocessor.*

(*e*) Explain how memory latency would increase the number of threads needed for good performance.

Problem 4: [20 pts]Consider the CUDA code below in which two alternatives are shown: putting a balls array in shared memory or in global memory. Assume the code runs on GPUs of compute capability 2.0 (though the answers could apply to 1.0 through 2.2).

(*a*) Explain why the Option 1 (shared memory) code below will run slowly. Fix the problem making as few changes as possible.

```
struct Ball {
  float3 position;   float3 velocity;
  float radius;      float mass;      };

__shared__ Ball balls[256]; // Option 1 - Shared memory.

__device__ void
update(float deltat)
{
  int start = threadIdx.x * balls_per_thread;
  int stop = start + balls_per_thread;
  for ( int i=start; i<stop; i++ )
    balls[i].position += deltat * balls[i].velocity;
}
```

(*b*) Explain why the Option 2 code below (using global memory) will run slowly. Fix the problems making as few changes as possible. (There are two major things to be fixed.)

```
struct Ball {
  float3 position;   float3 velocity;
  float radius;      float mass;      };

Ball* balls; // Option 2 - Global memory.

__device__ void
update(float deltat)
{
  int start = threadIdx.x * balls_per_thread;
  int stop = start + balls_per_thread;
  for ( int i=start; i<stop; i++ )
    balls[i].position += deltat * balls[i].velocity;
}
```

Problem 5: [20 pts]Answer each question below.

(*a*) Why is the if/return needed in the CUDA code below?

```
__device__ void prob(int array_size) {
  int tid = threadIdx.x + blockIdx.x * blockDim.x;
  if ( tid >= array_size ) return;
  result[tid] = a[tid] + b[tid];
}
```

(*b*) The line of code below checks for a special case to avoid a square root. That might make sense for a CPU, but not for necessarily for CUDA code on a GPU. Describe a situation in which it makes sense for CUDA and a different situation when it makes no sense (meaning it would be faster to do the square root all the time). Assume that 50% of the time d is equal to 1.

```
if ( d == 1 ) s = 1; else s = sqrt(d);
```

(*c*) The loop below is innocent on a CPU, but on a GPU it can execute inefficiently. Identify the problem and fix it.

```
for ( int i=0; i<32; i++ )
  {
    if ( a[i] == t ) { do_stuff(i); break; }
  }
```

Name _____

EE 4702

Take-Home Final Examination

Tuesday, 6 December 2011 to Monday, 11 December 2011

Work on this exam alone. Regular class resources, such as notes, papers, documentation, and code, can be used to find solutions. Do not try to seek out references that specifically answer any question here. Do not discuss this exam with classmates or anyone else. Any questions or concerns about problems should be directed to Dr. Koppelman.

Problem 1	_____	(30 pts)
Problem 2	_____	(20 pts)
Problem 3	_____	(50 pts)
Exam Total	_____	(100 pts)

Alias _____

Good Luck!

Problem 1: [30 pts]Consider the following schemes to reduce the resources needed to render a sphere. A buffer object will be used to hold the coordinates of triangles forming the entire sphere (similar to the method used in the classroom demos), organized as a single triangle strip. The coordinates are for a sphere of radius 1 with the center at the origin. Note that with this positioning the vertex coordinate (before transformation) is the same as the vertex normal. The modelview matrix (`gl_ModelViewMatrix`), which in this problem is assumed to be already set, will correctly position the sphere in eye space. (Note that `gl_NormalMatrix` (the transpose of the inverse of the modelview matrix) must be used to transform the vertex into a correct eye-space normal.) Unlike the classroom demos, there is no texture and so there is no need to orient (rotate) the sphere in any particular way.

(*a*) A shader will determine whether the item it is working on (vertex, triangle, or fragment, depending on the shader) is on the side of the sphere facing the user (and so potentially visible) or on the side away from the user (and so not visible). If it is on the side away from the user the shader will take some action to avoid rendering the object (if the shader allows that).

For each shader below:

- Write shader code that writes `true` to variable `visible` if the item is visible. If necessary, a shader can use information provided by an upstream shader. (*E.g.*, the fragment shader can be helped by the vertex or geometry shader). An attached sheet or file can be used for the shader code.
- Indicate whether the shader can stop the rendering of the item, and if so, show how.
- Estimate the amount of computation saved by stopping the rendering in the shader.

Afterward indicate the best shader(s) to use to minimize computation.

☐ Vertex Shader. ☐ Visibility Code ☐ Stop Rendering ☐ Saved Computation

☐ Geometry Shader. ☐ Visibility Code ☐ Stop Rendering ☐ Saved Computation

☐ Fragment Shader. ☐ Visibility Code ☐ Stop Rendering ☐ Saved Computation

☐ The best shader(s) to save computation time for invisible part of sphere. Explain.

Problem 1, continued: Continuing with the sphere problem.

One way to reduce the size of the buffer object is to have it store information on only half the sphere, say the north hemisphere. Given coordinate (x, y, z) in the north hemisphere one can trivially compute $(-x, -y, -z)$, the coordinate in the south hemisphere, if that coordinate were needed.

(*b*) Show shader code for rendering the sphere using the hemisphere buffer object. Show only the shaders that are needed and assume a routine `lighting(pos,norm)` is available that sets the correct lighting for a coordinate at `pos` with normal `norm`.

For this sub-problem **do not** rotate the sphere so that its north pole faces the user. That's a good idea, but it won't work when the buffer object stores only one eighth of a sphere.

As with the previous problem, the shader code should not emit triangles if they are not visible.

☐ Which is the most appropriate shader for this code?

☐ Show the shader code. (Or use attached sheet or file.)

Problem 1, continued: Still continuing with the sphere problem.

(*c*) Compare the performance of the approach used in part (a) to the approach used in part (b).

☐ Compare execution time for the two approaches, making reasonable assumptions.

Problem 2: [20 pts]The code below computes a cross product and stores the normalized result. Assume that the code runs on an NVIDIA GPU of compute capability 2.0.

```
__global__ void prob_cross_prod(int num_elements)
{
  int tid = threadIdx.x + blockIdx.x * blockDim.x;
  int num_threads = blockDim.x * gridDim.x;
  for ( int idx = tid;  idx < num_elements; idx += num_threads )
    // Line above: Four integer insn.
    {
      float3 a = arraya[idx];   // Vector load. (One add (for addr) + load insn)
      float3 b = arrayb[idx];   // Vector load. (One add (for addr) + load insn)
      float3 vec = cross(a,b);  // Six multiplications, three additions.
      float3 mag_sq = dot(a,b); // Three multiplications, two additions.
      float3 length_inv = 1.0f / sqrtf(mag_sq);  // See CUDA manual.
      float3 nvec = length_inv * vec; // Three multiplications.
      arrayc[idx] = nvec;       // Vector store. (One add (for addr)+store insn)
    }
}
```

(*a*) Assuming a very large number of threads, determine the number of clock cycles needed per element per multiprocessor. Base your answer on the information in the CUDA Programmers Guide Version section 5.4.1 (linked to `http://www.ece.lsu.edu/koppel/gpup/refs.html`) and on the comments in the code above. Be sure to figure out the instructions needed for the line assigning `length_inv`.

(For example, suppose it took only two floating-point adds for an the entire loop iteration (which computes one element). Since a multiprocessor, based on Table 5-1, can execute 32 FP adds per clock cycle, a multiprocessor requires just $\frac{2}{32}$ clock cycles per element.)

☐ Clock cycles per element assuming a large number of threads.

(*b*) Suppose a global memory load had a latency of 400 cycles. Based on your answer to the previous part, what would be the minimum number of threads needed to make full utilization of a multiprocessor. (*Hint: Compute the latency (beginning to end time) for one iteration of thread X, then determine how many threads can run in those times when thread X is waiting for memory.*

☐ Minimum number of threads.

Problem 3: [50 pts]The CUDA code below and on the next few pages operates on pairs of objects stored in shared memory, much like the collision code for the balls demo. Like the balls demo, each object might appear in about five or six pairs, and there is stable pattern of pairs. (Don't expect, say, `pair.x` to be numerically close to `pair.y`.)

The code in `planA` loads all objects from global memory. The code in `planB` first loads the objects into shared memory, then operates on them. The code in `planC` uses shared memory only to re-arrange the second of each object pair loaded. Read all parts of this problem before attempting to answer the questions below.

```
__global__ void planA(int num_pairs)
{
  int tid = threadIdx.x + blockIdx.x * blockDim.x;
  int num_threads = blockDim.x * gridDim.x;
  for ( int i=tid; i<num_pairs; i+=num_threads )
    {
      int2 pair = pairs[i];
      float3 pos_1 = pos_global[pair.x];
      float3 vel_1 = vel_global[pair.x];
      float3 pos_2 = pos_global[pair.y];
      float3 vel_2 = vel_global[pair.y];
      float3 f = resolve(pos_1,vel_1,pos_2,vel_2);
      pairs_f[i] = f;
    }}
```

(*a*) The performance of `planA` depends upon the array `pairs`. Suppose array `pairs` contains the correct pairs of objects to compare but nothing else was taken into account. Why might performance be poor? *Hint: This is due to what could be called the first rule of global memory access.*

☐ Reason for poor performance if `pairs` ill-chosen.

(*b*) For `planA`, what can be done to the `pairs` array to improve performance? That is, starting with an existing array `old_pairs` compute a new array of the same size, `pairs`, such that if element i of `old_pairs` is (a, b) (`old_pairs[i].x=a` and `old_pairs[i].y=b`) then there is some element j of `pairs` which includes (a, b) or (b, a). *Hint: The solution is simple but not 100% effective.*

☐ Explain how `pairs` should be constructed.

☐ Estimate the improvement in performance, be has specific as possible.

Problem 3, continued:

(*c*) By copying object data to shared memory the code in `planB` potentially solves two problems related to global memory access. The array `prefetch_idx` stores indices of objects to prefetch into shared memory, the array `pairsB` indicates which pairs of objects to `resolve`, but refers to them using shared memory indices (rather than global memory indices).

```
__device__ void planB(int num_pairs)
{
  int tid = threadIdx.x + blockIdx.x * blockDim.x;
  int num_threads = blockDim.x * gridDim.x;
  __shared__ float3 pos_shared[OBJ_PER_BLOCK], vel_shared[OBJ_PER_BLOCK];

  // Prefetch Loop
  for ( int j=threadIdx.x; j<OBJ_PER_BLOCK; j += blockDim.x )
    {
      int gidx = j + OBJ_PER_BLOCK * blockIdx.x;
      int bidx = prefetch_idx[gidx];
      pos_shared[j] = pos_global[bidx];
      vel_shared[j] = vel_global[bidx]; }

  // Resolve Loop
  for ( int i=tid; i<num_pairs; i += num_threads )
    {
      short2 pair = pairsB[i];
      float3 pos_1 = pos_shared[pair.x];
      float3 vel_1 = vel_shared[pair.x];
      float3 pos_2 = pos_shared[pair.y];
      float3 vel_2 = vel_shared[pair.y];
      float3 f = resolve(pos_1,vel_1,pos_2,vel_2);
      pairs_f[i] = f; }}
```

☐ Insert the minimum number of `syncthreads()` calls in the code above needed for correct operation.

☐ Given the use of `prefetch_idx`, how much more efficiently does the code in `planB` *Prefetch Loop* access global memory than the code in `planA`? Assume that `prefetch_idx` and `pairsB` are properly prepared (but the list of pairs is still uncontrollable). Is access ideal, or does it still depend on something.?

☐ When ordering `pairsB` one can emphasize reducing the total access to global memory or the efficiency of global memory access. Explain how each might be emphasized.

Problem 3, continued: The code below is for `planC`. It uses less shared memory than `planB`.

```
__global__ void planC(int num_pairs)
{
  int tid = threadIdx.x + blockIdx.x * blockDim.x;
  int num_threads = blockDim.x * gridDim.x;

  for ( int i=tid; i<num_pairs; i+=num_threads )
    {
      __shared__ float3 rearrange[BLOCK_SIZE];
      int3 pair = pairs[i];
      float3 pos_1 = pos_global[pair.x];
      float3 vel_1 = vel_global[pair.x];
      rearrange[pair.z] = pos_global[pair.y];
      float3 pos_2 = rearrange[threadIdx.x];
      rearrange[pair.z] = vel_global[pair.y];
      float3 vel_2 = rearrange[threadIdx.x];
      float3 f = resolve(pos_1,vel_1,pos_2,vel_2);
      pairs_f[i] = f;
    }}
```

☐ Insert the minimum number of `syncthreads()` calls so that the code above runs correctly.

☐ Compare the efficiency of global memory access in `planC` to the other two versions. Which is more efficient?

☐ Assume that, in `planB`'s favor, an object appears in multiple pairs (and so `planB`'s use of shared memory has potential benefit). By using less shared memory then `planB`, `planC` can potentially run faster than `planB`. Explain why, indicating a disadvantage `planB` has and how `planC` is faster despite the fact that `planB` caches objects.

16 Fall 2009

Name _____

GPU Programming
EE 4702-1
Midterm Examination

Friday, 6 November 2009, 9:40–10:30 CST

Problem 1	_____	(40 pts)
Problem 2	_____	(20 pts)
Problem 3	_____	(20 pts)
Problem 4	_____	(20 pts)
Exam Total	_____	(100 pts)

Alias _____

Good Luck!

Problem 1: [40 pts] Write OpenGL code to render a filled circle (a disc) of radius `r`, centered at the origin and with normal $(0, 1, 0)$.

- The distance between all vertices in a primitive should be approximately `vert_dist`.
- Use triangle strips. Multiple strips are okay, but there should be no easy way to make the strips longer.
- Use `glVertex3f` calls, don't try to construct arrays.
- Don't specify colors, normals, or other attributes.
- Assume transformations, lighting, etc, have all been set up. Start with `glBegin`.

☐ Code rendering a circle of radius `r`, center at origin.

☐ Vertex distance about `vert_dist`.

☐ Good use of strips.

☐ Code reasonably efficient.

Problem 2: [20 pts] Answer the normal questions below.

(*a*) Show an expression for the normal to triangle ABC, where A, B, and C are the vertex coordinates.

☐ Normal to ABC

(*b*) In the code sample below the pair of triangles is rendered using two different methods, identified as Method 1 and Method 2.

Describe the difference in appearance of the triangles rendered using Method 1 and Method 2 when diffuse lighting is used and `norm_ABC != norm_CBD`.

```
pVect norm_ABC = find_normal(A,B,C);    pVect norm_CBD = find_normal(C,B,D);

// Method 1
glBegin(GL_TRIANGLES);
glNormalfv(norm_ABC);
glVertex3fv(A);  glVertex3fv(B);  glVertex3fv(C);

glNormalfv(norm_CBD);
glVertex3fv(C);  glVertex3fv(B);  glVertex3fv(D);
glEnd();

// Method 2
pNorm norm_X = norm_ABC + normCBD; // Sum of two vectors normalized.
glBegin(GL_TRIANGLES);
glNormalfv(norm_ABC);  glVertex3fv(A);
glNormalfv(norm_X);    glVertex3fv(B);  glVertex3fv(C);

glVertex3fv(C);        glVertex3fv(B);
glNormalfv(norm_CBD);  glVertex3fv(D);
glEnd();
```

☐ Difference in appearance between Method 1 triangles and Method 2 triangles.

Problem 3: [20 pts] Consider the three methods of specifying vertices shown below.

```
switch ( opt_method ) {

case VM_Individual: {   /// Use Individual Vertices
    glBegin(GL_TRIANGLE_STRIP);
    for ( int i=0; i<coords_size; i+=3 ) {
        glNormal3f(coords[i],coords[i+1],coords[i+2]);
        glVertex3f(coords[i],coords[i+1],coords[i+2]);  }
    glEnd();
    break;  }

case VM_Array: {        /// Use Vertex Arrays
    glNormalPointer(GL_FLOAT,0,coords);
    glEnableClientState(GL_NORMAL_ARRAY);
    glVertexPointer(3,GL_FLOAT,3*sizeof(float),coords);
    glEnableClientState(GL_VERTEX_ARRAY);
    glDrawArrays(GL_TRIANGLE_STRIP,0,coords_size/3);
    glDisableClientState(GL_NORMAL_ARRAY);
    glDisableClientState(GL_VERTEX_ARRAY);
    break;  }

case VM_Buffer: {       /// Use Buffer Objects
    glBindBuffer(GL_ARRAY_BUFFER,gpu_buffer);
    glVertexPointer(3,GL_FLOAT,3*sizeof(float),NULL);
    glEnableClientState(GL_VERTEX_ARRAY);
    glNormalPointer(GL_FLOAT,0,NULL);
    glEnableClientState(GL_NORMAL_ARRAY);
    glDrawArrays(GL_TRIANGLE_STRIP,0,coords_size/3);
    glBindBuffer(GL_ARRAY_BUFFER,0);
    glDisableClientState(GL_NORMAL_ARRAY);
    glDisableClientState(GL_VERTEX_ARRAY);
  break;  } }
```

(*a*) Why is the individual vertex method slower than the others?

☐ Reason that individual vertex method slower than the others.

(*b*) When used the right way the method using buffer objects is much faster than the others.

☐ Why are buffer objects faster than vertex arrays, when used the right way?

☐ Describe a situation in which the buffer object and vertex array method would have about the same performance.

Problem 4: [20 pts] Answer each question below.

(*a*) OpenGL allows different material property colors for ambient, diffuse, emissive, and specular lighting. However only a few of these can be changed from vertex to vertex. Why?

☐ Why can't all material properties be changed each vertex?

(*b*) OpenGL lets you specify any transformation matrix for the projection, it doesn't have to be a frustum.

☐ Describe the appearance of a scene in which the projection matrix were identity. What parts of world space would be visible?

(*c*) Textures are provided or used to generate multiple MIPMAP levels. Explain what a MIPMAP level is and why it is necessary.

☐ What is a MIPMAP level?

☐ Why is it necessary?

Name _____

GPU Programming
EE 4702-1
Take-Home Final Examination

5 December - 16:00, 10 December 2009

Please do not discuss this exam with any other students. If you have any question E-mail me or stop by my office Wednesday or later. Do not attempt to find solutions to similar problems, but it's okay to otherwise seek out references beyond those used in class.

Please drop off solution at my office, slide it under the door if I'm not in. Solutions can also be E-mailed.

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

Alias _____

Good Luck!

Problem 1: (20 pts) In the balls simulation the balls cast shadows on the platform but they do not cast shadows on each other. The `vs_main_bshadow` routine below is a vertex shader that will show the shadows that one ball casts on another. It is called when rendering balls; if there are no shadows then it computes the lighting normally, otherwise it accounts for the lack of light from one or both light sources.

For each ball, the host code finds up to four other balls that might possibly shade it, the eye-space coordinates of these are passed to the vertex shader in `float4` variable `neighbor_ball_X`.

Routine `find_shadow(vertex_e,neighbor_ball_X,light)` determines whether ball `neighbor_ball_X` does indeed leave `vertex_e` in the shadow of light source `light`. That is, if it returns true the vertex is shaded from the light, otherwise it is illuminated.

```
void vs_main_bshadow() {
  vec4 vertex_e = gl_ModelViewMatrix * gl_Vertex;
  vec3 normal_e = normalize(gl_NormalMatrix * gl_Normal);
  float4 neighbor_ball[4] =
    { neighbor_ball_0, neighbor_ball_1, neighbor_ball_2, neighbor_ball_3 };
  bool shadow_light[2] = {false, false};

  for ( int light=0; light<2; light++ )
    for ( int n = 0; n < 4; n++ ) {
          bool in_shadow = find_shadow(vertex_e, neighbor_ball[n], light);
          shadow_light[light] |= in_shadow;
          // if ( in_shadow ) break;  // Bad idea.
        }

  generic_lighting(vertex_e,gl_Color,normal_e,shadow_light);
  gl_Position = gl_ModelViewProjectionMatrix * gl_Vertex;
}
```

(*a*) Modify `generic_lighting` to account for the shadows. *Hint: This is easy.*

```
void generic_lighting(vec4 vertex_e, vec4 color, vec3 normal_e, bool sh[2])
{
  vec4 new_color = vec4( color.rgb * gl_LightModel.ambient.rgb, color.a);
  for ( int i=0; i<2; i++ )
    {
      vec4 light_pos = gl_LightSource[i].position;
      vec3 v_vtx_light = light_pos.xyz - vertex_e.xyz;
      float phase_light = dot_pos(normal_e, normalize(v_vtx_light).xyz);
      vec3 ambient_light = gl_LightSource[i].ambient.rgb;
      vec3 diffuse_light = gl_LightSource[i].diffuse.rgb;
      float dist = length(v_vtx_light);
      float distsq = dist * dist;
      float atten_inv =
        gl_LightSource[i].constantAttenuation +
        gl_LightSource[i].linearAttenuation * dist +
        gl_LightSource[i].quadraticAttenuation * distsq;
      new_color.rgb += color.rgb *
        ( ambient_light + phase_light * diffuse_light ) / atten_inv;
    }
    gl_FrontColor = new_color;
}
```

Problem 1, continued:

(*b*) Indicate whether each storage type below can be used for `neighbor_ball_X`, and how well that would work.

☐ `uniform float4 neighbor_ball_0` Possible? Good?

☐ `attribute float4 neighbor_ball_0` Possible? Good?

☐ `varying float4 neighbor_ball_0` Possible? Good?

(*c*) Ordinary execution of the loop nest (the `for` loops around the call of `find_shadow`) would involve executing 13 branches. Improper use of branches can sometimes hurt shader code performance. Why does this not apply to the loop nest here?

(*d*) The `vs_main_bshadow` routine includes a commented-out loop exit, labeled "Bad idea." That loop exit would reduce execution time in CPU code, but in GPU code it would likely have no impact on performance. Why?

Problem 2: (20 pts) For this problem, use the code and description from the previous problem. (This is essentially a continuation of the previous problem.)

(*a*) Complete the `find_shadow` routine so that it returns true if `vertex_e` is blocked from light `light_num` by the ball at center `neighbor_e`.

```
bool find_shadow(vec4 vertex_e, vec4 neighbor_e, int light_num)
{
  float r = opt_ball_radius;
  // Solution goes here.
```

```
}
```

Problem 3: (20 pts) The CUDA routine below computes the vertex positions for a ball based on the ball's center position (a homogeneous coordinate) and orientation (a quaternion). It reads vertex positions for a unit-radius sphere centered on the origin, and transforms them using a transformation matrix that it computes. The sphere coordinates in `sphere` are pre-computed by the CPU and never change. The vertex coordinates in `vertices` are used by OpenGL code to render the sphere.

The host code will launch one block per ball, each thread in a block computes vertices for a different part of a ball. The block size determines how many vertices each thread computes. Assume that `sphere_size` is 1024 or even larger.

```
__device__ xf_ball_per_block()
{
  const int idx = blockIdx.x;  // This is the same for all threads in a block.
  if ( idx >= ball_count ) return;
  const int tid = threadIdx.x;

  float4 ball_position = balls_x.position[idx];
  float4 ball_orientation = balls_x.orientation[idx];

  __shared__ pMatrix transform_local_to_world;

  bool all_compute_matrix = false;
  if ( tid == 0 || all_compute_matrix )
    transform_local_to_world = compute_transf(ball_orientation, ball_position);

  __syncthreads();

  const int thr_per_block = blockDim.x;

  const int my_part = sphere_size / thr_per_block;
  const int start = tid * my_part;
  const int stop = start + my_part;

  for ( int i=start; i<stop; i++ )
    {
      float4 sphere_local = sphere[i];
      float4 sphere_world = transform_local_to_world * sphere_local;
      vertices[ idx * sphere_size + i ] = sphere_world;
    }
}
```

(*a*) Notice that if `all_compute_matrix` is false then only thread 0 (in each block) computes the transformation matrix. Explain why setting `all_compute_matrix` to true would only make a difference when the block size is larger than 32 threads (the warp size).

(*b*) Access to the global array `sphere` is not coherent, and so the GPU will make many more trips to memory than are necessary. Explain why the incoherence forces the GPU to make seemingly unnecessary trips to memory.

Problem 3, continued:

(*c*) Modify the code so access to `sphere` is coherent, but without modifying what the code does. *Hint: the code will have to access* `sphere` *in a different order.*

Problem 4: (20 pts) Answer each question below.

(*a*) A GPU has m multiprocessors, and a CUDA application needs to run a launch kernel with n threads. What range of block sizes will ensure that all multiprocessors are used?

(*b*) Suppose a GPU has two multiprocessor and a CUDA application needs to launch a kernel of just 30 threads. The threads do not share memory. Explain the difference in performance between launching one block of 30 threads and two blocks of 15 threads each.

(*c*) Access to CUDA memory takes hundreds of cycles and there is no cache that can reduce that number to only a few cycles. Why does a properly written CUDA program not loose performance despite the long wait for data?

(*d*) CUDA shared memory has no counterpart in the OpenGL shader language. Explain what shared memory might be used for.

Problem 5: (20 pts) Answer each question below.

(*a*) The variables below are available to OpenGL vertex shaders. Show how the value for each of these variables is set in host OpenGL code. That is, show a line or two of OpenGL code that provides a value for the variable. Don't show more than necessary.

☐ `gl_ModelViewMatrix`

☐ `gl_Vertex`

☐ `gl_LightSource[2].position`

(*b*) Suppose one wanted a strip of triangles, each of a different color, and the strip was to be rendered as a `GL_TRIANGLE_STRIP`.

☐ Explain why it's impossible to give each triangle its own color without using a geometry shader.

☐ Describe how one could give each triangle its own color with a geometry shader.

(*c*) Describe what the z-buffer and z test are for.

(*d*) The equation used to compute lighting is complex, as it must consider at least eight light sources, plus account for the four types of lighting. Why then would the amount of computation needed for texture processing be much larger, even when only one texture is used?

17 Fall 2025 Solutions

Name Solution

Formatted For Two-Sided Printing

Staple This Side

GPU Programming
EE 4702-1
Midterm Examination

Friday, 31 October 2025, 9:30-10:20 CDT

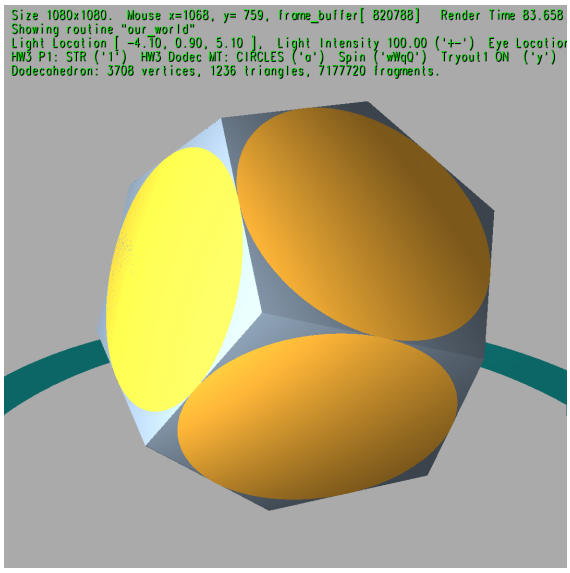
Problem 1		(35 pts)
Problem 2		(15 pts)
Problem 3		(15 pts)
Problem 4		(20 pts)
Problem 5		(15 pts)
Exam Total		(100 pts)

Alias This should be safe.

Good Luck!

Staple This Side

Problem 1: [35 pts] Appearing to the right is a dodecahedron with gold circles just above its faces. The code below, based on the Homework 3 solution, draws the dodecahedron but the circle code is incomplete. Complete it.



☑ Several quantities are computed for you, such as the pentagon center and the dodecahedron center.

☑ Approximate the circle as an `n_sides` sided polygon.

☑ Use `ri` for the radius of the circle.

☑ The circle plane (circle local `ax`,`ay`) should be parallel to the pentagon plane.

☑ Place the circle center on the line from the dodecahedron center, `dod_ctr`, to the pentagon center, `pen_ctr`, a very small distance away from pentagon. See `vcc`.

☑ Just emit circle coordinates, **don't emit colors**.

```
case MT_Circles: {
  pCoor dod_ctr = cyl_center + pVect(0,1.5,0);            // Dodecahedron Center
  float ri = len_edge / ( 2 * tanf(numbers::pi/5) );      // Circle Radius
  int n_sides = 100;                                      // Number of "circle" sides
  float delta_theta = 2 * numbers::pi / n_sides;

  for ( size_t i=0; i<coors.size(); i+=5 ) {
    for (int j=1; j<4; j++) coors_os<<coors[i]<<coors[i+j]<<coors[i+j+1]; // Emit pentagon.
    pVect vi(0,0,0); for ( int j=1; j<5; j++ ) vi += pVect(coors[i],coors[i+j]);

    // Dodecahedron Face (Pentagon) Coords in coors[i] to coors[i+4];
    pCoor pen_ctr = coors[i] + 0.2 * vi;                  // Pentagon Center
    pVect vcc( dod_ctr, pen_ctr );                        // Should be useful.

    pCoor cc = pen_ctr + .001 * vcc;  // SOLUTION -- Compute circle center ..
    pNorm ax( coors[i], coors[i+1] ), ay( cross(vcc,ax) );      // .. axes ..
    pVect vx = ri * ax, vy = ri * ay; // .. and scale axes to radius of circle.

    pCoor prev_coor( cc + vx );    // SOLUTION -- Coord used in consecutive iterations.
    for ( int j = 1; j  <= n_sides;  j++ ) { //  ☑  Finish this line.
      float theta = j * delta_theta;

      coors_os << cc << prev_coor;  // SOLUTION -- Insert first 2 coords of this triangle.
      prev_coor = cc + cosf(theta) * vx + sinf(theta) * vy;
      coors_os << prev_coor;        // Insert last coordinate of this triangle.
}}
gc.topology_set(Topology_Individual);
```


Staple This Side
Staple This Side

Staple This Side

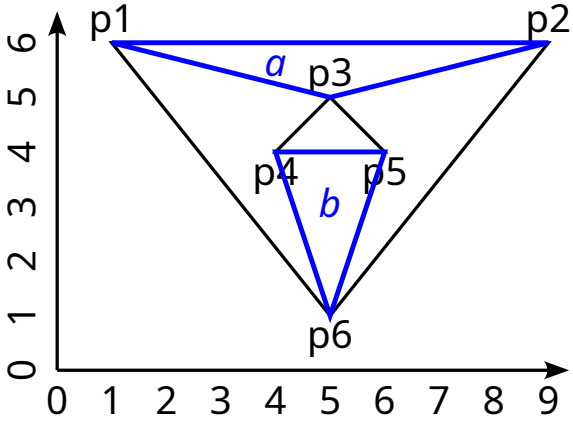
Problem 2: [15 pts] Below are two unsuccessful attempts to render a triangle-in-a-triangle shape using the `Our_3D` demo code from class, part (c) is your shot at success.

(*a*) The code below inserts vertices for an **individual triangle topology**.

☑ On the illustration to the right show the triangles that **actually are** rendered. ☑ Label each triangle with a letter (a, b, c, ...).

```
// Pipeline set up for individual triangles.
// (Vulkan: PrimitiveTopology::eTriangleList).

coors_os << p1 << p2 << p3 << p4 << p5 << p6;
```



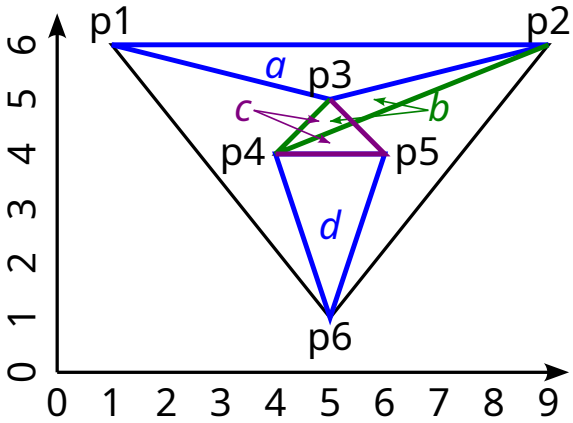
The solution appears above. The triangles rendered by the code are outlined in blue.

(*b*) The code below inserts vertices for a **triangle strip topology**.

☑ On the illustration to the right show the triangles that **actually are** rendered. ☑ Label each triangle with a letter (a, b, c, ...).

```
// Pipeline set up for a triangles strip.
// (Vulkan: PrimitiveTopology::eTriangleStrip).

coors_os << p1 << p2 << p3 << p4 << p5 << p6;
// Completed triangle-> a     b     c     d
```



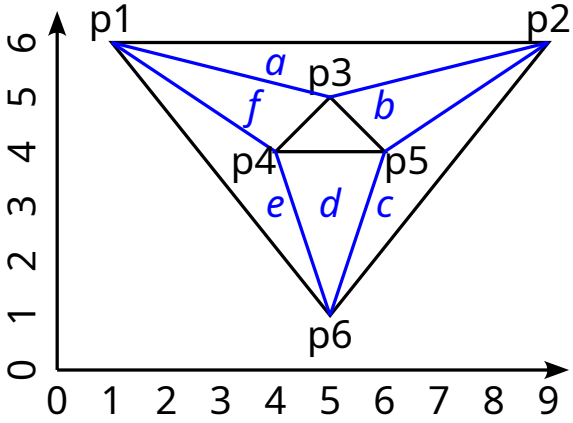
The solution appears above. The triangle letter in the comment shows the last (of three) vertices making that triangle. For example, triangle c consists of vertices `p3`, `p4`, and `p5`.

(*c*) The code below is incomplete.

☑ Complete the code below so that green shaded area shown is rendered correctly, ☑ and without overlapping triangles, for a pipeline set for a triangle strip topology.

☑ Coordinates `p1` through `p6` are already defined correctly, don't assign values to them.

```
// Pipeline set up for a triangles strip.
// (PrimitiveTopology::eTriangleStrip).
```

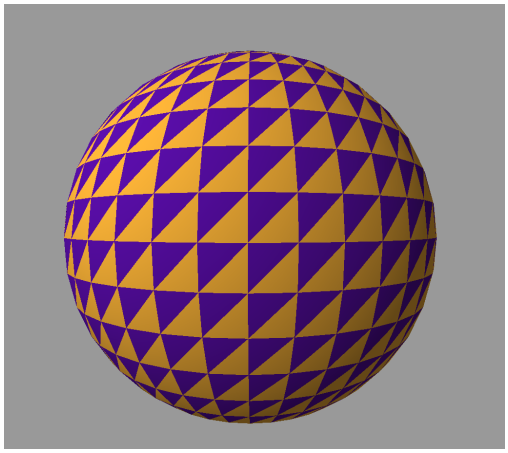



```
coors_os << p1 << p3 << p2 << p5 << p6 << p4 << p1 << p3;  // SOLUTION
//                      a     b     c     d     e     f    <-- Completed triangle.
```

Solution appears above. The triangle letter in the comment shows the last (of three) vertices making that triangle. For example, triangle c consists of vertices `p2`, `p5`, and `p6`.

Staple This Side

Problem 3: [15 pts] Answer the following questions about spheres rendered using rasterization.



Original Sphere

(*a*) The sphere shown to the right consists of $T = 1600$ triangles. (Not all are visible.)

☑ How many vertices would be needed to render the sphere using a individual triangles (a triangle list)? ☑ Answer in terms of T.

Three vertices per triangle, $V = 3T$.

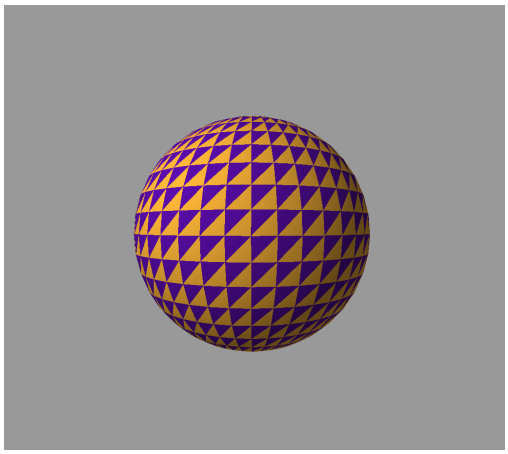
☑ How many vertices would be needed to render the sphere using a triangle strip? ☑ Answer in terms of T.

One vertex per triangle plus two at the beginning, $V = 2 + T$.

To render the original sphere using a triangle list about $F = 500,000$ fragments were generated.

☑ About how many fragments would be generated if the sphere were rendered using a triangle strip? ☑ Answer in terms of F.

Approximately F fragments. Whether it is exactly F fragments depends on how the rasterizer generates fragments along an edge shared by two triangles. For a triangle strip the rasterizer knows that an edge between adjacent triangles is shared and won't emit the same fragment for both triangles, just one or the other. For individual triangles it would be too costly to determine whether a triangle edge is shared with another, so the same fragment could be issued twice.



Far Sphere

(*b*) Suppose for the original sphere above the eye were at distance d from the nearest point on the surface of the sphere. Let V_o denote the number of vertices used to render the original sphere using triangle strips and F_o denote the number of fragments generated rendering the original sphere. In the otherwise identical *far sphere* to the right the eye is at distance $2d$ from the nearest point. Both the original and far spheres have T triangles. *Note: In the original exam F_o was not mentioned, and it was not clear whether V should denote the number of vertices in the near or far sphere.*

☑ How many vertices in the far sphere? ☑ Answer in terms of T, V_o, F_o, and d.

The number of vertices does not change with distance. Therefore there are V_o in the far sphere, the same as in the original sphere.

☑ **Approximately** how many fragments in the far sphere? (There is not enough information to answer this question exactly, don't try.) ☑ Answer in terms of T, V_o, F_o, and d.

Short Answer: The number of fragments is approximately $F_o/4$ because when the distance is doubled the approximate radius of the sphere in window (pixel) space will be half and so $\frac{1}{4}$ the number of pixels will be covered.

This is an approximation because when the distance from the closest part of the sphere changes from d to $2d$ the distance from the edge of the sphere we can see increases be *more than* d because the further we are, the closer the top edge of the sphere is to the "north pole" (assuming the closest point is on the "equator").

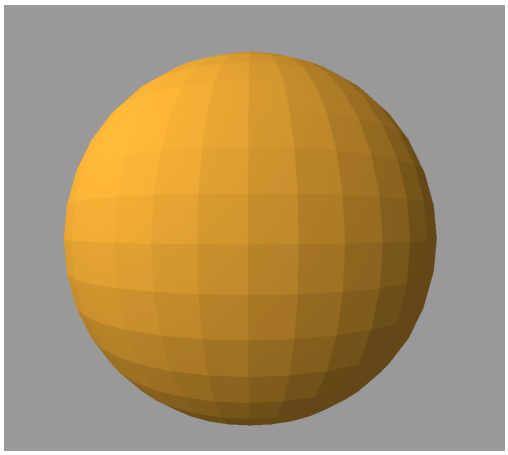
Staple This Side

Staple This Side

Staple This Side

(*c*) The **only difference** between the two spheres below on the right is in the normal associated with each vertex. The normals for the smooth sphere were chosen to make it look more like a sphere.

☑ Complete the code below so that it inserts normals that will result in the faceted sphere.



Faceted Sphere

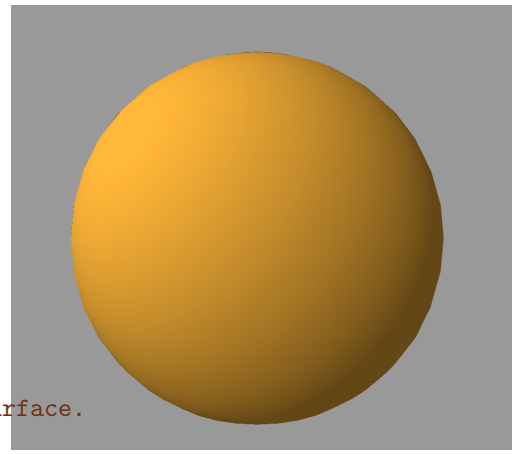
```
// Center of sphere at sp_ctr_os;

// Coordinates of a triangle approximating
// the surface of a sphere.
//
coors_os << p1 << p2 << p3;

// SOLUTION -- Compute triangle normal.
//
pNorm n = cross( pVect(p1,p2), pVect(p1,p3) );
normals_os << n << n << n;
```

The solution appears above. To give the faceted appearance insert the triangle normal into `normals_os`. That's what's done in the code above.

☑ Complete the code below so that it inserts normals that will result in the smooth sphere.



Smooth Sphere

```
// Center of sphere at sp_ctr_os;

// Coordinates of a triangle approximating
// the surface of a sphere.
//
coors_os << p1 << p2 << p3;

// SOLUTION -- Compute normal of points on sphere surface.
//
pCoor c = sp_ctr_os;
//
normals_os << pNorm( c, p1 ) << pNorm( c, p2 ) << pNorm( c, p3 );
```

To give the smooth appearance set the normal of each vertex to the normal of the sphere that the triangle is inserted to approximate.

Staple This Side

Problem 4: [20 pts] Answer each question below.

(*a*) Answer each question about the amount of computation needed for common operations. All vectors have three components and all coordinates have four components.

☑ How many multiplies are needed to compute a dot product such as `dot(v1,v2)`?

The dot product of three-component vectors u and v, $u \cdot v$ is computed $u_x v_x + u_y v_y + u_z v_z$. This consists of **3 multiplies**.

☑ How many multiplies are needed to compute a cross product such as `cross(v1,v2)`?

The cross product of three-component vectors u and v, $u \times v$ is computed $\begin{bmatrix} u_y v_z - u_z v_y \\ u_z v_x - u_x v_z \\ u_x v_y - u_y v_x \end{bmatrix}$. This consists of **5 multiplies**.

☑ How many multiplies are needed to perform a coordinate transformation `m * p1`, where `m` is a transformation matrix and `p1` is a coordinate?

Matrix `m` is a 4×4 matrix which consists of 16 elements. Each element is used once, so there are **16 multiplies**.

☑ How many multiplies are needed to compute a matrix product `m1 * m2` where `m1` and `m2` are transformation matrices?

Multiplying two 4×4 matrices is the same as multiplying one 4×4 matrix by a vector 4 times (each vector is a column of the second matrix). In total there are **64 multiplies**.

(*b*) Given coordinate P and vector v, we can easily compute $Q = P + v$. Show a transformation matrix, $\mathbf{M}$, that achieves the same result by computing a matrix/homogeneous coordinate product, $Q = \mathbf{M}P$. (For example, with $v = (v_x, v_y, v_z) = (1, 2, 3)$ and $P = (10, 20, 30, 1)$ we would compute $Q = (11, 22, 33, 1)$).

☑ Show the transformation matrix:

Start with an identity matrix and put the components of v in the first three rows of the fourth column of the matrix: $\mathbf{M}(v_x, v_y, v_z) = \begin{pmatrix} 1 & 0 & 0 & v_x \\ 0 & 1 & 0 & v_y \\ 0 & 0 & 1 & v_z \\ 0 & 0 & 0 & 1 \end{pmatrix}$

☑ Describe a situation in which it would make more sense to compute $Q = P + v$ rather than $Q = \mathbf{M}P$.

It would make sense to just add v if the only transformation we were doing was a translation or if we only had just a few coordinates to transform.

☑ Describe a situation in which it would make more sense to use a translation matrix rather than just adding a vector. *Hint: we do it all the time in our code.*

Short answer, sufficient for full credit: It would be a situation in which we have multiple transformations to apply and we would need to apply them to multiple coordinates.

Explanation, for the benefit of those studying: Let $\mathbf{M_1}, \mathbf{M_2}, \ldots \mathbf{M_n}$ be matrices performing those other transforms, perhaps there are several translations, but also rotations, scales, and maybe a projection. We'd compute $\mathbf{M} = \prod_{i=1}^{n} \mathbf{M}_i$, requiring $64(n-1)$ multiplies. Further, in our situation we have V coordinates, $P_1, P_2, \ldots, P_V$. We'd transform them to $Q_1, Q_2, \ldots, Q_V$ using $Q_i = \mathbf{M}P_i$ for $0 < i \leq n$, requiring $16V$ multiplications.

Staple This Side
Staple This Side

Each $\mathbf{M}P_i$ multiply requires 16 multiplies. It doesn't matter how many matrices were originally multiplied to compute $\mathbf{M}$. So those 16 multiplies are not just for one translation, that includes everything else that needs to be done. Computing $\mathbf{M}$ requires multiplies, but so long as $V \gg n$ that will be small compared to the multiplies used to transform coordinates.

Staple This Side

Staple This Side

Problem 5: [15 pts] Answer the questions below.

(*a*) The code below inserts the coordinates and colors for a cylinder:

```
for ( int i=0; i<=n_segs; i++ ) {
    // Some code omitted.
    coors_os << c + cyl_axis << c;
    colors   << color_cyan   << color_cyan;
  }
```

☑ What would the cylinder look like if the frame buffer were written with the color `color_cyan`?

It would not look like a cylinder, it might look like a rectangle with rounded endcaps. That's because the roundness of the cylinder's surface would not be visible if the entire surface had exactly the same color—cyan.

☑ What should the frame buffer be written with, if not exactly `color_cyan`?

The frame buffer should be written with a lighted color computed using the light location, distance of the surface from the light the angle made by the surface normal and the light, and of course the color cyan. When done correctly the curvature of the cylinder surface would be easy to recognize.

(*b*) The `Our_3D` C++ class in the course CPU-only code is modeled very loosely on a Vulkan implementation, and `World` is modeled on something like an application (such as a game) a developer would write.

☑ Who writes Vulkan and OpenGL implementations? ☑ Why do most of them give away the implementations for free?

The Vulkan and OpenGL *implementations* are written by the GPU manufacturers. They know best how to convert routines written in OpenGL shader language or SPIR-V to code that runs fast on their particular devices.

GPU company X would give its Vulkan and OpenGL implementation away for free because the implementation can only be used on company X's products, so it would only benefit their customers who presumably paid for the video cards (containing the GPUs). They have an incentive to do a good job because it makes their video cards look good.

The Vulkan and OpenGL standards are managed by Khronos, a non-profit organization. Those standards describe what Vulkan and OpenGL code should do (okay, *shall do*) running on a conforming implementation. Khronos doesn't write the implementations, and they wouldn't want to since they would have to write one for every single video card.

(*c*) Rasterization and ray tracing in `Our_3D` differ in what their outer loops iterate over.

☑ What kind of an object does the outer loop of `Our_3D` rasterization code iterate over?

The outer loop iterates over triangles. The iteration variable, `i`, is used to index a vertex array. But the outer loop access three vertices at a time, forming a triangle.

☑ What kind of an object does the outer loop of `Our_3D` ray tracing code iterate over?

The outer loop (strictly speaking, the two highest loop levels) of our ray tracing code iterates over pixels.

Staple This Side

Staple This Side

18 Fall 2024 Solutions

Name Solution

Formatted For Two-Sided Printing

Staple This Side

GPU Programming
EE 4702-1
Midterm Exam

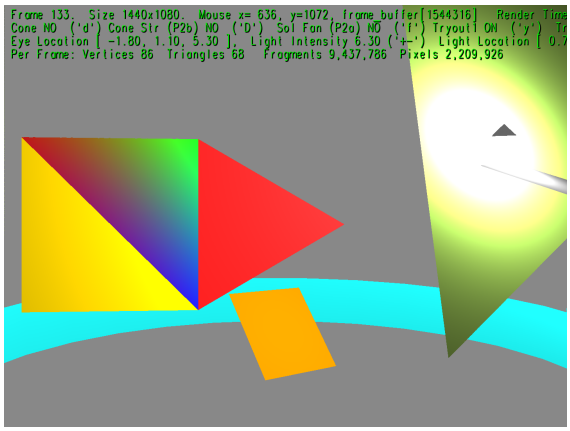
25 October 2024, 9:30-10:20 CDT

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 `p3` 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

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.

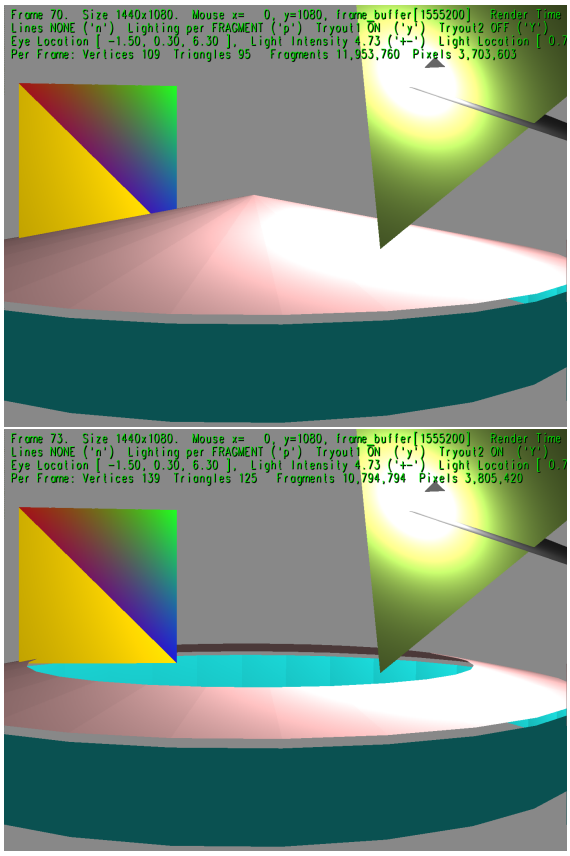
```
for ( size_t i = 0;  i < coors_os.size();  i += 3 )
  {
    pCoor p1 = coors_os[i], p2 = coors_os[i+1], p3 = coors_os[i+2];

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

Staple This Side

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

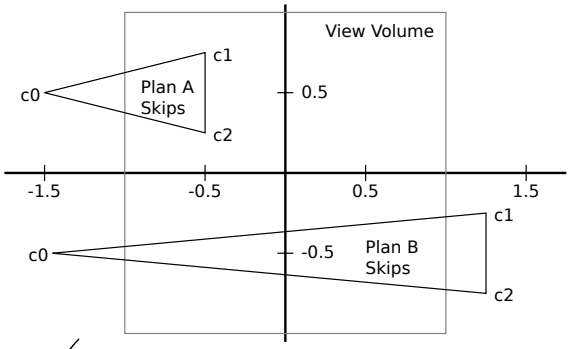
#ifdef    PLAN_A    /// Plan A
    //  If clip-space coords outside of view volume skip. (Not correct.)
    if ( c0.x < -1 || c0.y < -1 || c0.x > 1 || c0.y > 1 ) continue;

#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

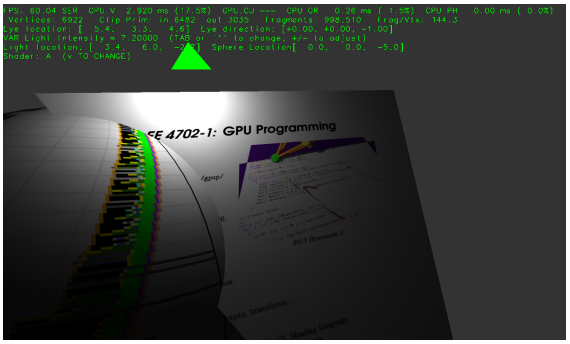
Staple This Side

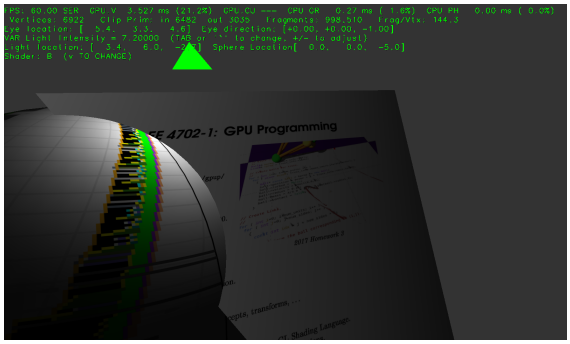
Problem 4: [35 pts] Answer each question below.

(*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
Staple This Side

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.

Staple This Side

Name Solution

Formatted For Two-Sided Printing

Staple This Side

GPU Programming
LSU EE 4702-1
Final Examination

Thursday, 12 December 2024 7:30-9:30 CST

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

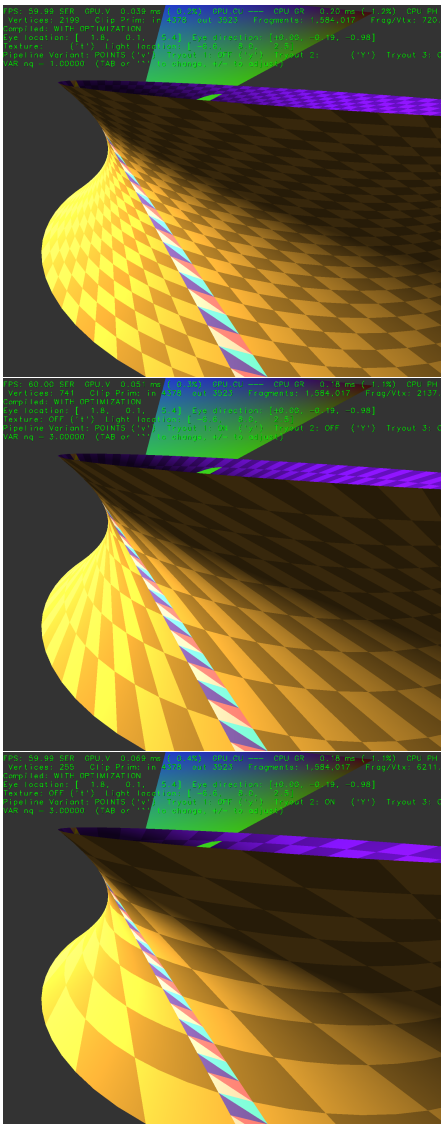
Alias Maybe

Good Luck!

Staple This Side

Staple This Side

Problem 1: [20 pts] The screenshots to the right show our hyperboloid rendered using code based on the Homework 5 solution. The top screenshot is from code nearly the same as the solution (other than the checkered appearance). Each invocation of the geometry shader (shown on the facing page) draws one *quad*. The lower-left of a quad, is at level `level0` and side `side0`, the upper right at level `level0+1` and side `side0+1` using variable names from the code. So the width of the quad is 1 and the height is 1. In the middle screenshot the height is 3 (level `level0` to `level0+3`), and width is 1, while in the lower screenshot both the width and height are 3. The 3×3 quad consists of $2 \times 3 \times 3 = 18$ triangles, it is **not** two big triangles.



On the facing page is the geometry shader code used by the points grouping (topology). Modify the shader code to draw quads of width `nq` and height `nq`, where `nq` is a variable. The CPU will record the draw with $\lceil n_s/n_q \rceil \lceil n_l/n_q \rceil$ vertices, where n_q is `nq`, n_l is `n_levels`, and n_s is `n_sides`. The unmodified code works correctly for $n_q = 1$.

☑ Correctly set `max_vertices`. It is okay to show an expression in terms of `nq`.

☑ Correctly set `level0` and `side0` in the shader so that they specify the lower-left of the quad that the shader will draw.

☑ Modify the code so that it draws a quad of width and height `nq`.
  ☑ Don't just emit two big triangles.

☑ Emit fewer than $6n_q^2$ vertices.

☑ Do not emit extra triangles, including zero-area triangles and triangle vertices that are not on the hyperboloid surface.

☑ Don't assume that n_q is a factor of either n_s or n_l. That is, don't assume $\lceil \frac{n_l}{n_q} \rceil n_q = n_l$ nor $\lceil \frac{n_s}{n_q} \rceil n_q = n_s$.

☑ Be sure that `EndPrimitive` is called at appropriate times.

Solution appears on the facing page. A new loop, `ss`, was added to iterate around the sides in a quad. The number of iterations in the existing `dl` loop was increased from 2 to `nq+1`. Each iteration of the `ss` loop draws on triangle strip. That strip is ended by a call to `EndPrimitive` at the end of the `ss` loop body.

Staple This Side

Staple This Side

```
 // SOLUTION
layout ( triangle_strip, max_vertices = (2 + 2 * nq) * nq) out;

void gs_points() {
  // SOLUTION - Compute number of quads per line (vertical) and per ring.
  int q_per_line = ( uc.n_levels + nq - 1 ) / nq;  // Note: This rounds up.
  int q_per_ring = ( uc.n_sides + nq - 1 ) / nq;   // Note: This rounds up.

  // Compute index of quad along line ..
  int q_level0 = ( gl_PrimitiveIDIn % q_per_line );
  // .. and use it to find the index of the level along the line.
  int level0 = q_level0 * nq;

  // Compute index of side around ring ..
  int q_side_0 = gl_PrimitiveIDIn / q_per_line;
  // .. and use it to find the index of the side around the ring.
  int side0 = q_side_0 * nq;

  // Return if no work. Not needed if draw uses correct number of vertices.
  if ( side0 >= uc.n_sides ) return;

  for ( int ss = 0; ss < nq; ss++ )   // SOLUTION - Outer loop to iterate over sides in quad.
    {
      if ( side0 + ss >= uc.n_sides ) break;

      // SOLUTION - Inner loop iterates over nq+1 levels, not just 2.
      for ( int dl = 0; dl<=nq; dl++ )
        {
          int level = level0 + dl;
          if ( level > uc.n_levels ) break;

          // SOLUTION - Still need this two-iteration loop for strip.
          for ( int ds = 0; ds < 2; ds ++ )
            {
              int side = side0 + ss + ds;
              int idx = level + side * uc.s_levels;

              // Retrieve the coordinate and normal.
              vec4 p = hyperb_coords[ idx ];
              vec4 n = hyperb_norms[ idx ];

              // Convert coordinate spaces.
              gl_Position = ut.clip_from_object * p;
              Out.vertex_e = ut.eye_from_object * p;
              Out.normal_e = normalize( mat3(ut.eye_from_object) * n.xyz );
              // Don't worry about color index and texture coordinates.
              EmitVertex();
            }
        }
      // SOLUTION - Call EndPrimitive at the end of the dl loop.
      EndPrimitive();
    }
}
```

Staple This Side

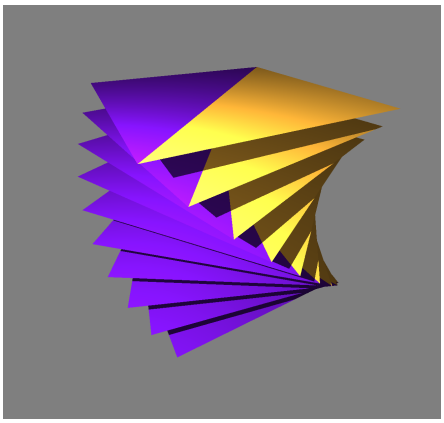
Problem 2: [15 pts] The code on the facing page emits the square shown in the upper screenshot in the first iteration of the loop. The coordinates `p0`, `p1`, `p2`, `p3` form a square, constructed using two triangles. After writing the buffer set the code applies transformation matrix `m` to these coordinates, 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.

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.



Let l be the length of a side of the square, and let q be the value of `n_squares`. Modify the code so that `m` rotates the square $\frac{\pi}{2(q-1)}$ radians around its center using the square's normal for the axis of rotation, and so that it moves the square along its normal by a distance of $\frac{l}{q-1}$, assume $q > 1$. See the lower screenshot. 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.

Solution appears on the facing page. Note that the normal, `n`, only needs to be computed once since a new square is updated by rotating an existing square around its normal (which so the rotated square has the same normal). Also, translating an object does not change its normals.

Note that transformations are applied from right to left.

Staple This Side

Staple This Side

Staple This Side

```
/// SOLUTION
//
// Construct vectors along sides of the square ..
//
pVect v01( p0, p1 ),  v02( p0, p2 );
//
// .. and use them to compute the normal, side length, and square center.
//
pNorm n = cross( v02, v01 );                          // ☑ Compute n.
float side_len = v01.mag();
pCoor square_center = p0 + 0.5 * ( v01 + v02 );

// Compute rotation angle and square separation specified in the problem.
//
float delta_theta = 0.5 * numbers::pi / max( 1, n_squares - 1 );
float delta_z = side_len / max( 1, n_squares - 1 );

// The transformation, m, consists of the following steps:
//
// - Translate so that the center of the square is moved to the origin.
// - Rotate by delta_theta degrees around the normal, n.
// - Translate the square back to its original location.
// - Translate the square up by delta_z.
//
// Construct the respective transformation matrices:
//
pMatrix_Translate m_to_origin( -square_center );
pMatrix_Rotation  m_rot( n, delta_theta );
pMatrix_Translate m_to_square( square_center );
pMatrix_Translate m_up( delta_z * n );

pMatrix m = m_up * m_to_square * m_rot * m_to_origin; // ☑ 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: [15 pts] On the facing page is the CPU code from Homework 5 showing a buffer set being prepared for an individual triangle grouping and a triangle strip grouping, followed by the geometry shader. The geometry shader (at the bottom of the page) uses color index 0 to detect the start of a triangle strip. For the questions below answer per frame.

☑ For the individual triangle grouping compute the number of bytes sent ☑ per vertex. ☑ State any assumptions about the size of data types. ☑ That's per vertex, not per draw.

By looking at the data inserted into buffer set `bset_hyperb` one can see that four kinds of data are inserted, a `pCoor`, such as `p00`, a `pTCoor`, such as `pTCoor(0,0)`, a `pNorm`, such as `n00`, and an `int`, such a `c1`. These four items are the attributes associated with a vertex. Their number of elements are 4, 2, 3, and 1, respectively. The total data per vertex is then: $\boxed{(4+2+3+1)4\,\mathrm{B} = 40\,\mathrm{B}}$.

☑ For the individual triangle grouping how many times will the vertex shader be invoked per draw (frame)?

First, count the number of vertices inserted for one loop body. To do that count any type of data, such as `pCoor`. Notice that six vertices are inserted in the loop body, the first group of lines inserts three, the next group inserts another three. Use n_l for `n_levels` and n_s for `n_sides`. The inner loop iterates n_l times, the outer loop n_s times, so the loop body executes $n_l n_s$ times, and the vertex shader $\boxed{\text{will be invoked } 6n_l n_s \text{ times}}$.

☑ For the individual triangle (triangle list) grouping how many times will the geometry shader be invoked per draw (frame)? ☑ How many triangles will be sent to the rasterizer per draw (frame)?

For an individual triangle (triangle list) grouping the geometry shader is invoked $3n_l n_s/3 = n_l n_s$ times. Each geometry shader invocation uses three vertices, and each vertex is used for exactly one geometry shader invocation. (When number of vertices is a multiple of 3, which is the case here.)

Assuming that clipping is done by the rasterizer, and not before it, the number of triangles sent to the rasterizer is also $n_l n_s$ because our geometry shader always semits one triangle when the color index is positive.

☑ For the triangle strip grouping how many times will the vertex shader be invoked per draw (frame)? ☑ Pay attention to the loop bounds.

The number of vertices inserted in the loop body is two. The number of iterations is $n_s(n_l+1)$, and so $2n_s(n_l+1)$ vertices are inserted.

☑ For the triangle strip grouping how many times will the geometry shader be invoked per draw (frame)? ☑ Don't forget color index 0. ☑ How many triangles will be sent to the rasterizer per draw (frame)?

Assuming the geometry shader always emitted exactly one triangle (which is not the case here), a draw with V vertices would result in $V-2$ triangles. In our case the number of times the $\boxed{\text{geometry shader is invoked is } 2n_s(n_l+1)-2}$.

Notice that the given geometry shader does not emit a triangle if the last vertex has a color index of zero, and so one needs to look more closely to determine the number of triangles sent to the rasterizer (emitted by the geometry shader). (For this code if the color index is zero a new strip is being started and that some vertices sent to the geometry shader are from the prior strip and we don't want to form a triangle using vertices from the prior and current strip.) So for each strip two invocations are ignored, those are the invocations due to the vertices inserted when `lev=0`. So the total number of triangles $\boxed{\text{sent to the rasterizer is just } 2n_s n_l}$.

☑ How much better is the code using the triangle strip grouping than the code using individual triangles at reducing the number of times the same vertex is sent to the rendering pipeline? ☑ Is there room for further reductions in this redundancy? ☑ Explain.

Roughly one third as many vertices are sent using a triangle strip compared to the individual triangles topology. By ANSI standard 2011-04-01 for performance class descriptors, triangle strips are a lot better. Even with triangle strips a particular vertex is inserted multiple times. Consider two iterations, `side=0;lev=0;` and `side=1;lev=1`. Vertex `p11` from the `side=0;lev=0;` iteration is the same as vertex `p00` from the `side=1;lev=1` iteration, so there is still redundancy. An indexed draw would remove all redundancy of this type.

Staple This Side
Staple This Side

Staple This Side

```
    switch ( pipeline_variant ) {  // CPU Code

    case PV_Individ:
      for ( int side = 0; side < n_sides; side++ )
        for ( int lev = 0; lev < n_levels; lev++ ) {
            pCoor p00 = coords[ idxf( lev,   side ) ],  p01 = coords[ idxf( lev,   side+1 ) ];
            pCoor p10 = coords[ idxf( lev+1, side ) ],  p11 = coords[ idxf( lev+1, side+1 ) ];
            pNorm n00 = norms[ idxf( lev,   side ) ],   n01 = norms[ idxf( lev,   side+1 ) ];
            pNorm n10 = norms[ idxf( lev+1, side ) ],   n11 = norms[ idxf( lev+1, side+1 ) ];

            bset_hyperb << p00 << p01 << p10;
            bset_hyperb << pTCoor(0,0) << pTCoor(0,1) << pTCoor(1,0);
            bset_hyperb << n00 << n01 << n10;

            bset_hyperb << p01 << p11 << p10;
            bset_hyperb << pTCoor(0,1) << pTCoor(1,1) << pTCoor(1,0);
            bset_hyperb << n01 << n11 << n10;

            int c1 = side==0 ? ( lev%2 ? 5 : 2 ) : 1,  c2 = side==0 ? ( lev%2 ? 3 : 4 ) : 1;
            bset_hyperb << c1 << c1 << c1 << c2 << c2 << c2;
          }
      break;

    case PV_Strip:
      for ( int side = 0; side < n_sides; side++ )
        for ( int lev = 0; lev <= n_levels; lev++ ) {
            pCoor p00 = coords[ idxf( lev, side ) ],  p01 = coords[ idxf( lev, side + 1 ) ];
            pNorm n00 = norms[ idxf( lev, side ) ],  n01 = norms[ idxf( lev, side + 1 ) ];

            bset_hyperb << p00 << p01;
            bset_hyperb << pTCoor(0,lev%2) << pTCoor(1,lev%2);
            bset_hyperb << n00 << n01;

            int c1 = lev == 0 ? 0 : side == 0 ? ( (lev+1)%2 ? 5 : 2 ) : 1;
            int c2 = lev == 0 ? 0 : side == 0 ? ( (lev+1)%2 ? 3 : 4 ) : 1;
            bset_hyperb << c1 << c2;
          }
      break;


void gs_main_clean() {   // Geometry Shader
  int i_last = 2 - gl_PrimitiveIDIn % 2;
  if ( In[i_last].color_idx == 0 ) return;
  for ( int i=0; i<3; i++ ) {
      gl_Position = In[i].vertex_c;
      Out.color_idx = In[i_last].color_idx;
      Out.vertex_e = In[i].vertex_e;
      Out.normal_e = In[i].normal_e;
      Out.tcoor = In[i].tcoor;
      EmitVertex();
    }
  EndPrimitive();  }
```

Staple This Side

Problem 4: [20 pts] Appearing on the facing page is an excerpt from the ray generation shader used in class, followed by the miss shader.

(*a*) The ray generation shader does not write the frame buffer if the ray does not intersect geometry. (The frame buffer is initialized with a background color before ray tracing starts.) Modify the code so that if a ray does not intersect geometry the frame buffer is written with `color_tan` (thus making initialization unnecessary). There are several ways to do this, do it in the most efficient way possible.

☑ Write the frame buffer with `color_tan` if a ray misses ☑ efficiently. (Efficiently means with as little extra code as possible, or even with less code than is here now.)

Solution appears on the facing page. The ray payload has been initialized to tan, but the alpha component still set to zero. After `traceNV` returns write the frame buffer with the ray payload. If there was a miss it will still be tan.

(*b*) The code sets `tmin` to `1`, which is correct, but it sets `tmax=10000`, which works but probably does not match the far plane from `ut.object_from_clip`.

☑ Explain what `tmin` and `tmax` do.

The specify the range of distances to consider for intersections. The distances are lengths of the ray argument to `traceNV`, which is `eye_to_pixel_g` in the code shown. An intersection at a distance less than `tmin` or more than `tmax` will be ignored.

☑ Explain why increasing `tmax` would increase the execution time of real ray tracing code ☑ and why it would not have much of an impact on the course CPU-only ray tracing code. Assume that both codes render the same scene and do so correctly and that the application programmer is satisfied with the results (other than execution time).

Our ray tracing code looks at every triangle, so it does not matter what `tmax` is. In a practical ray tracing implementation the triangles are organized to reduce the number of intersection tests. One way is to sort triangles into boxes (hierarchically). If the closest face of one such box is already at distance greater than `tmax` from the eye then there is no need to test the triangles in the box, saving time.

☑ Set `tmax` to the correct value based on `ut.object_from_clip`.

Hint: Pay attention to how `pixel_c` *and* `pixel_g` *are computed and think about clip space. It may be helpful to use function* `length(MYVECTOR)` *to find the length of a vector.*

The solution appears on the facing page.

Staple This Side

Staple This Side

Staple This Side

```
void main() {
  // Code above omitted for brevity.

  // Prepare clip-space coordinate of pixel on near plane ..
  vec4 pixel_c = vec4( pixel_c2d.x, pixel_c2d.y,  0,  1 );
  // .. and convert to global space.
  vec4 pixel_g = homogenize( ut.object_from_clip * pixel_c );


  // Compute global- (object-) space coordinates of eye.
  //
  vec4 eye_e = vec4(0,0,0,1);
  vec4 eye_g = ut.object_from_eye * eye_e;


  // Compute vector from eye to pixel.
  //
  vec3 eye_to_pixel_g = pixel_g.xyz - eye_g.xyz;
  //
  // This is the ray to be cast by this shader.

  float tmin = 1;

  // SOLUTION (b) -- Compute global space coordinates of pixel on far plane ..
  //
  vec4 pixel_cf = vec4( pixel_c2d.x, pixel_c2d.y,  1,  1 );
  vec4 pixel_gf = homogenize( ut.object_from_clip * pixel_cf );
  //
  // .. and compute vector from eye to far pixel.
  //
  vec3 eye_to_pixel_gf = pixel_gf.xyz - eye_g.xyz;
  //
  // Now set tmax to length of eye_to_pixel_gf scaled to eye_to_pixel_g:
  //
  tmax = length( eye_to_pixel_gf ) / length( eye_to_pixel_g );

  // SOLUTION (a) -- Initialize payload to tan.
  //
  rp_color = color_tan;   rp_color.a = 0;

  traceNV
   ( topLevelAS,      // Acceleration Structure: Holds optimized geometry info.
     gl_RayFlagsOpaqueNV, 0xff, // Types of objects to consider (or ignore).
     0, 0, 0,         // Specify which sets of shaders to call.
     eye_g.xyz,       // Ray Origin
     tmin,
     eye_to_pixel_g,  // Ray Direction.
     tmax,
     0   );          // Payload Location.

  // if ( rp_color.a == 0 ) return; // Removed for part (b) solution.

  imageStore(fb_image, ivec2(gl_LaunchIDNV.xy), vec4(rp_color.rgb, 0.0));
```

Staple This Side

Problem 5: [10 pts] A retroreflective coating reflects light back towards its source. They are often used on highway signs to reflect the light from a car's headlights back towards the car.

The fragment shader below sets `retro` to `true` if the material property is retroreflective. Note that `vertex_e` and `uni_light.position` are in eye space.

Staple This Side

☑ Modify the fragment shader so that when `retro` is true the lighted color accounts for the retroreflective coating taking into account the location of the ☑ light, ☑ vertex, and ☑ eye. It is okay to assume the coating reflects light *exactly* to its source, even though that would not help a driver.

Solution appears below. Vector `vec_ve` is a vector from the eye to the vertex (fragment) (remember that in eye space the eye is at the origin). The dot product of this vector and the vector from the vertex to the light indicates how much they point in the same direction. Raising it to a power enhances this, meaning that `align` will only be close to one if the two vectors nearly point in the same direction. The value of `align` is used to increase the amount of light, `attenuation`.

```
void fs_retro() {
  vec4 texel = texture(tex_unit_0,tcoor);
  bool retro = uc.retroreflective[color_idx];

  // Vector from fragment to light.  Note: light position is in eye space.
  vec3 vec_vl = uni_light.position.xyz - vertex_e.xyz;

  // Distance squared.
  float dist_sq = dot( vec_vl, vec_vl );

  // False if the light illuminates the side we can't see.
  bool lit_side = dot( normal_e, vec_vl )>0 == dot( normal_e, -vertex_e.xyz )>0;

  // Amount of light reaching the fragment.
  float attenuation = lit_side ? abs( dot( normal_e, vec_vl ) / dist_sq ) : 0;

  // Material color.
  vec4 mat_color = gl_FrontFacing ? uc.front[color_idx] : uc.back[color_idx];

  /// SOLUTION
  vec3 vec_ve = vertex_e.xyz;
  float align = pow( dot( normalize(vec_ve), normalize( vec_vl ) ), 11 );
  if ( retro && align > 0 ) attenuation += 10 * align * attenuation;

  // Compute lighted color.
  frag_color = texel * mat_color * uni_light.color * attenuation;
}
```

Staple This Side

Staple This Side

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

(*a*) Answer the following questions about colors.

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

A material property is what is commonly called color, it describes how the primitive interacts with light. A lighted color is how the primitive appears taking into account the material property and also the color of the light and the positioning of the primitive relative to the light. A material property might be one of the vertex attributes, or it could be common to all primitives in a draw. The lighted color is computed by the vertex or fragment shader and is written to the frame buffer (perhaps after being mixed with textures).

Consider three options: providing a color with each vertex (as an attribute), such as `color_gold`, providing a color index (as an attribute) as done in some homework assignments this semester, or just providing a color in a uniform variable.

☑ Describe a situation in which a vertex color attribute is the best of the three options. ☑ Explain the advantage.

A vertex color attribute works best when each vertex or triangle has its own unique color. The same effect could be obtained using a color index, but that would require more data since for each vertex one must move (perhaps from CPU to GPU) a color index and the color itself. It's not just the increase in data size, there would be the added delay from reading the color from a storage buffer.

☑ Describe a situation in which a color index attribute is the best of the three options. ☑ Explain the advantage.

A color index attribute works best when there is more than one color, but the number of different colors is small. Because a color index is $\frac{1}{4}$ the size of a color, the amount of data moved from CPU to GPU is smaller. (The amount of data moved from GPU memory to GPU cores would also be smaller due to caches, some thing not covered in this course but which is covered in LSU EE 7722.).

☑ Describe a situation in which a color in a uniform variable is the best of the three options. ☑ Explain the advantage.

This works best when all primitives in a draw are the same color. This uses the least amount of data, and has the added benefit that data in uniform variables are read quickly.

(*b*) Answer the following questions on homogeneous coordinates.

☑ Provide an example of a homogeneous coordinate ☑ and its homogenized value.

An example of a homogeneous coordinate is $P = \begin{bmatrix} 6 \\ 8 \\ 10 \\ 2 \end{bmatrix}$. The homogenized value of P is $\begin{bmatrix} 3 \\ 4 \\ 5 \\ 1 \end{bmatrix}$.

☑ Describe something that can be done with a homogeneous coordinate needed for rendering that can't be done with ordinary coordinates.

It is possible to apply a translation to a homogeneous coordinate by multiplying by a 4×4 matrix. It is also possible to apply a projection transformation by a 4×4 matrix (a frustum transform).

Staple This Side

19 Fall 2023 Solutions

Name Solution

Formatted For Two-Sided Printing

Staple This Side

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)

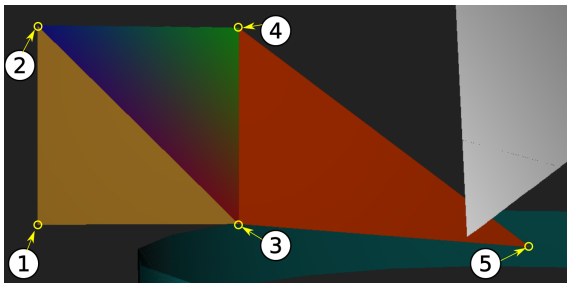
Alias Starship Launch !!?

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();
```


Staple This Side
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.
      }}
```

Staple This Side

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

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.
```

Staple This Side

```
// 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?

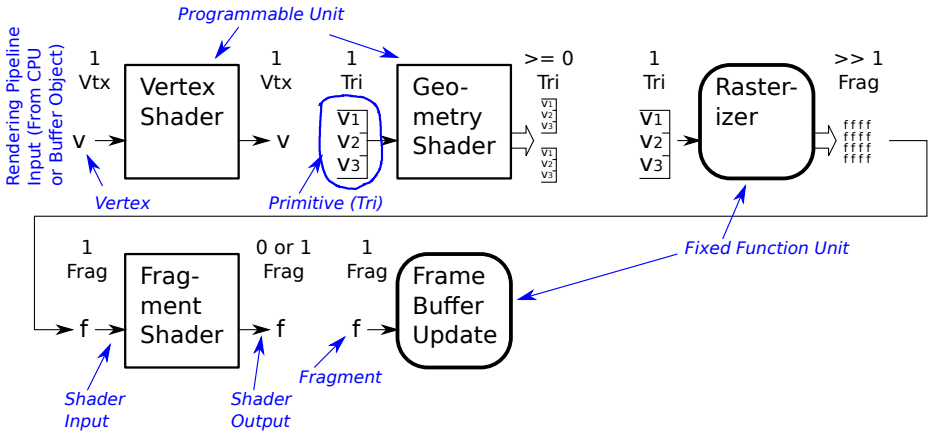
A material property is what is popularly known as a color. A leaf is green even if nobody is looking at it, or if it is completely dark outside. A lighted color is computed using the material property and information about a light source such as its location, brightness, etc, and the geometry of the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

Name Solution *Formatted For Two-Sided Printing*

Staple This Side

GPU Programming
LSU EE 4702-1
Final Examination

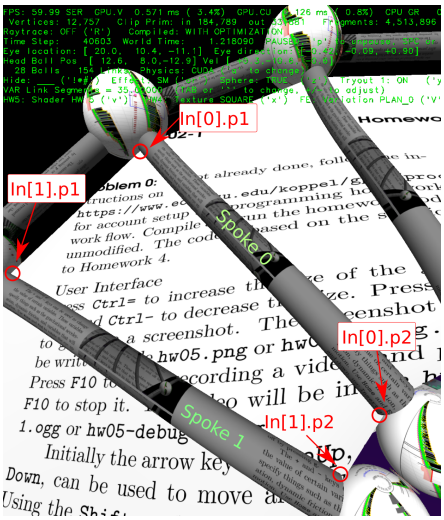
Thursday, 7 December 2023 7:30-9:30 CST

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

Alias Whitespace-Efficient

Good Luck!

Staple This Side



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 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.

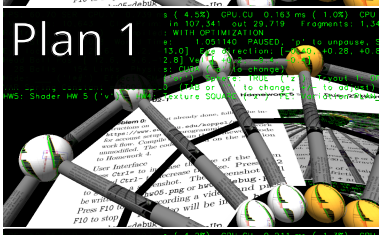
☑ Describe the appearance for `Plan_1`, draw a sketch. ☑ Explain.

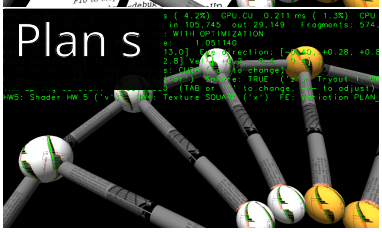
There should be a gap in the center of the texture, see the screenshot to the lower right labeled Plan 1. Suppose `opt_segments/2 = 10`. At iteration 10 two vertices are emitted and then `EndPrimitive()` is called, ending a triangle strip. In iteration 11 two more vertices are emitted, and these are 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 to 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();

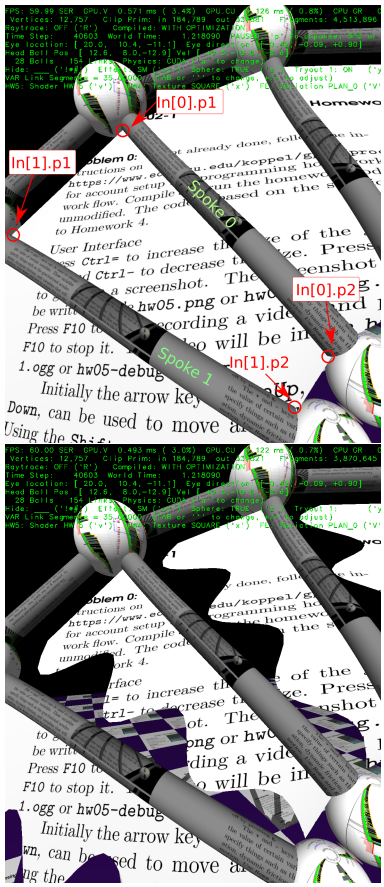
    switch ( variation ) {
    case Plan_0: break;  // Original. Same as Homework 5 solution.
    case Plan_1: if ( s == opt_segments/2 ) EndPrimitive(); break;
    case Plan_s: EndPrimitive(); break; }
  }
  EndPrimitive();
}
```


Staple This Side

Staple This Side

Staple This Side

(*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();
}
```

Staple This Side

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 $\boxed{n_s[(n_g+1)2+2]}$. 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 $\boxed{n_s+1}$.

☑ 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 $\boxed{n_s[(n_g+1)2+2]-2}$.

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 $\boxed{n_s}$.

☑ 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 $\boxed{n_s[(n_g+1)2+2]-2}$. For those wondering why the number of triangles isn't just $2n_sn_g$: 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 $\boxed{2n_sn_g}$.

☑ 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 $4\times4\,\mathrm{B}=16\,\mathrm{B}$. 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 $\boxed{n_s[(n_g+1)2+2]16\,\mathrm{B}}$.

Staple This Side
Staple This Side

Staple This Side

The Homework 5 code inserts only integers into the buffer set, each integer is $4\,\mathrm{B}$, the total amount of data due to the buffer set is $(n_s+1)4\,\mathrm{B}$. The Homework 5 code also uses storage buffers `pos1`, `pos2`, `v1`, and `v2`. Each element of each buffer is $16\,\mathrm{B}$. (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 $16n_s\,\mathrm{B}$. The four storage buffers together are $64n_s\,\mathrm{B}$. Because the links move it will be necessary to send those storage buffers to the GPU each frame.

The total mount of data form the Homework 5 code is the size of the buffer set and the storage buffers which is $\boxed{4(n_s+1)\,\mathrm{B}+64n_s\,\mathrm{B}}$.

```
/// Homework 4 CPU Code: Input topology is a triangle strip. Vertex attribute: pCoor
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;
    pCoor 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];
    pCoor 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;
        pCoor 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.
```

Staple This Side

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
```

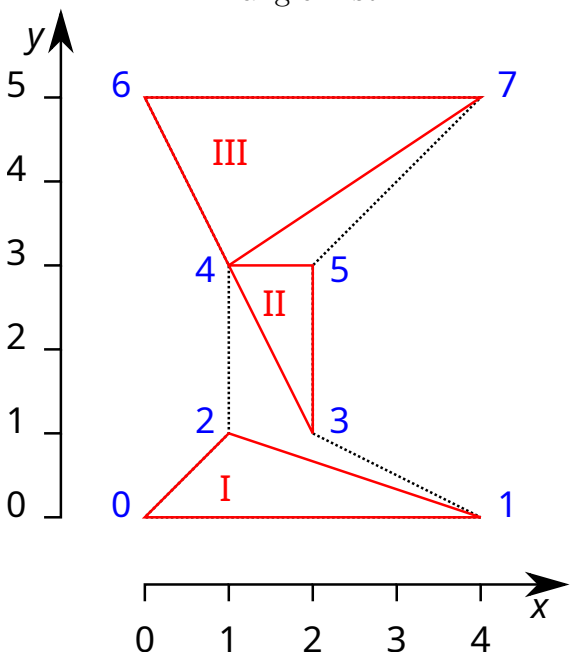
Do not try to fix the code. Instead show the result of rendering with these coordinates for the topologies requested below.

☑ 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.

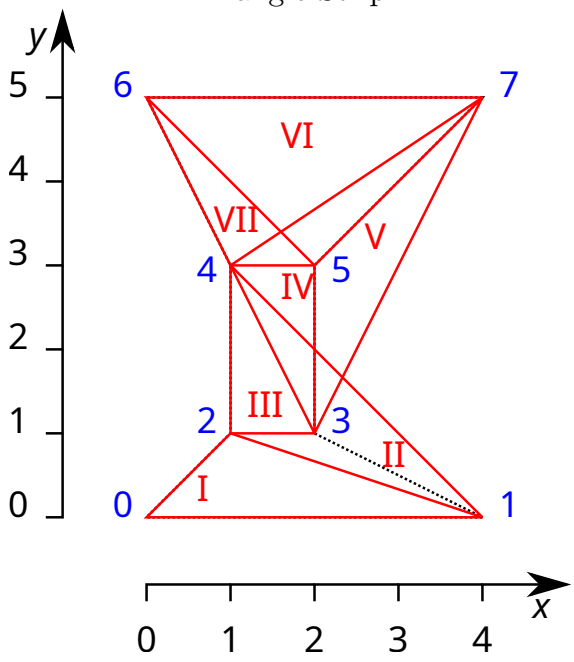
☑ On the diagram on the right sketch the triangles that will be rendered by the code below when the pipeline is set up for a triangle strip.

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

Triangle List



Triangle Strip



Staple This Side

Staple This Side

Staple This Side

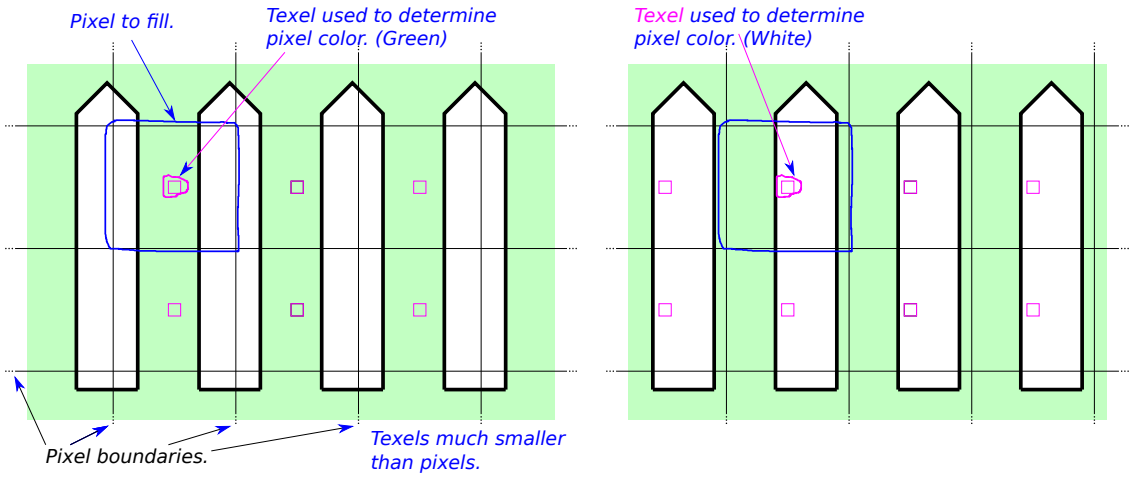
Problem 4: [15 pts] Answer the following texture questions.

(*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).

Staple This Side

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;
      pCoor 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;
     pCoor wf = b0*w0 + b1*w1 + b2*w2;  // Window-space coordinate of fragment.

     float bc0 = 1/(f.w0ow2z*(1-b0)/b0+1),   bc1 = 1/(f.w1ow2z*(1-b1)/b1+1),  bc2 = 1-bc0-bc1;

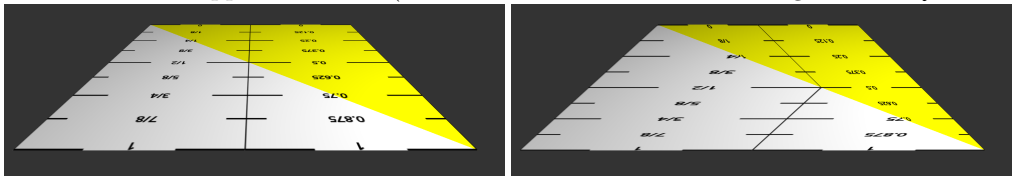
     if ( use_eye ) tc = bc0*tc0 + bc1*tc1 + bc2*tc2; // Eye-space interpolation.
     else           tc = b0*tc0 + b1*tc1 + b2*tc2;    // Window-space interpolation.
     texel = texture2D(tc); // Look up and filter texel.

     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`.



Staple This Side
Staple This Side

Staple This Side

(*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 by examined, which can greatly reduce the number of triangles to examine.

Staple This Side

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.

☑ $S(t) =$

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

☑ $t(x_1) =$

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

Staple This Side

Staple This Side

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.

Staple This Side

Staple This Side

20 Fall 2022 Solutions

Name Solution

GPU Programming
EE 4702-1
Midterm Exam

21 October 2022, 9:30-10:20 CDT

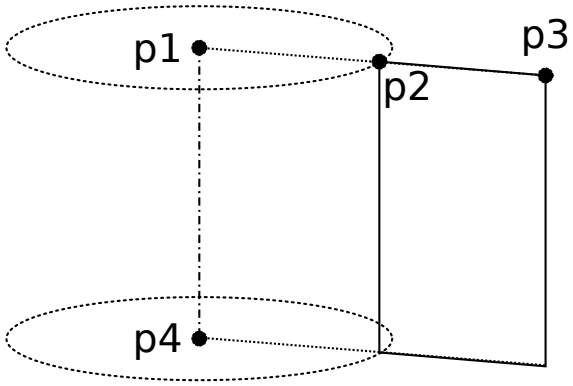
Problem 1		(30 pts)
Problem 2		(15 pts)
Problem 3		(15 pts)
Problem 4		(40 pts)
Exam Total		(100 pts)

Alias Artemis I?

Good Luck!

Problem 1: [30 pts] Appearing below is code based on the solution to Homework 2, in which a paddle wheel was to be rendered with the shape and positioning determined by given points `p1`, `p2`, `p3`, and `p4`. The code has been modified to work with the course Vulkan library and the shaders written for Homework 3.

(*a*) The last lines insert normals into the buffer set. These normals are obviously wrong. Modify the code to compute the correct normal.



☑ Modify code to compute correct normal.

The solution appears below. The normal is the cross product of the `vz` and `v`. One might think of `v` as a rotated version of `vx`. A common mistake was to take the cross product of `vz` and `vx`, which would only work for `i=0`.

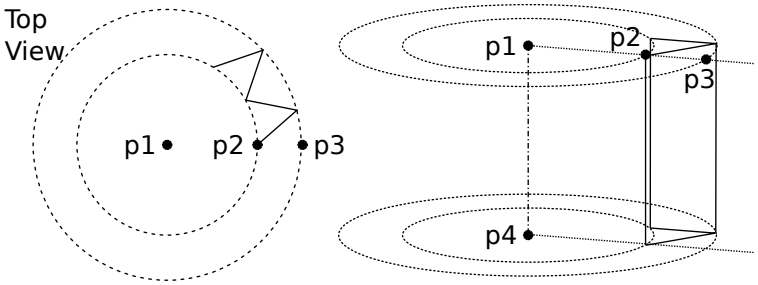
```
pVect vx(p1,p2), vz(p1,p4);
pNorm ax(vx), ay( cross(vz,ax) );
pVect vy = ay * ax.magnitude;
float r2or1 = pNorm(p1,p3).magnitude / ax.magnitude;
float delta_theta = 2 * M_PI / n_pieces;

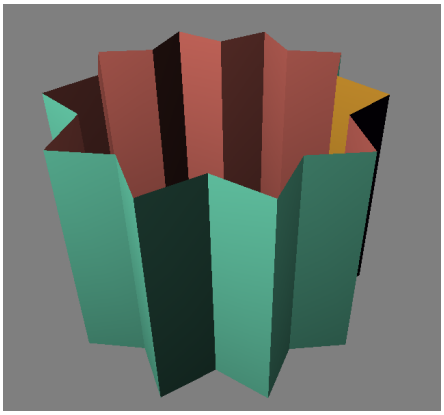
for ( int i=0; i<n_pieces; i++ ) {
    float theta = i * delta_theta;
    pVect v = vx * cosf(theta) + vy * sinf(theta);
    pVect v2 = v * r2or1;
    bset_mta <<  p1 + v   <<  p1 + v2  <<  p4 + v;
    bset_mta <<  p1 + v2  <<  p4 + v2  <<  p4 + v;

    // pNorm n = pVect(1,2,3);  // Obviously incorrect normal.

    // SOLUTION
    pNorm n = cross(vz,v);
    //
    // That's it, just one line.

    // Insert normals into buffer set.
    bset_mta << n << n << n;
    bset_mta << n << n << n;
}
```





(*b*) Appearing below again is the code from the Homework 2 solution. This time the goal is to render a (crude) gear positioned and oriented using the same four points, `p1`-`p4`. The points define two pairs of circles as illustrated above. The center illustration shows how one tooth (gear part) is constructed. There should be `n_pieces` teeth, positioned so they touch and go around the gear as shown on the screenshot to the right. Modify the code to draw the gear.

☑ Complete code so that it draws the figure. ☑ Avoid computationally wasteful code.

☑ Write coordinates and normals. Don't write colors.

Solution appears on the next page.

```
pVect vx(p1,p2), vz(p1,p4);
pNorm ax(vx), ay( cross(vz,ax) );
pVect vy = ay * ax.magnitude;
float r2or1 = pNorm(p1,p3).magnitude / ax.magnitude;
float delta_theta = 2 * M_PI / n_pieces;


for ( int i=0; i<n_pieces; i++ ) {
    float theta = i * delta_theta;
    pVect v = vx * cosf(theta) + vy * sinf(theta);
    pVect v2 = v * r2or1;




    bset_mta <<




    bset_mta <<
}
```

Solution to Problem 1b appears below.

If there are to be the same number of gear teeth as there were paddle wheel blades then we will need to iterate over twice the number of positions around the circle. For that reason compute a delta theta that is based on `2*n_pieces` and use `2*n_pieces` as the loop bound.

A gear tooth consists of two rectangles. (The two rectangles are shown using solid lines in the center illustration on the top of the previous page.) Each iteration of the loop below emits one rectangle. An iteration starts by constructing a vector from `p1` to the inner circle, `vc` (`c` is for current). If `i` is odd then that vector is lengthened so that it reaches from `p1` to the outer circle. The just-computed vector, `vc`, and the vector computed in the previous iteration, `vp`, are used to compute the four vertices of the rectangle, `pp_bot`, `pc_bot`, `pp_top`, and `pc_top`. Two triangles are emitted using these vertices. After that a normal is computed and emitted. Note that the same normal is used for all six vertices.

```
pVect vx(p1,p2), vz(p1,p4);
pNorm ax(vx), ay( cross(vz,ax) );
pVect vy = ay * ax.magnitude;
float r2or1 = pNorm(p1,p3).magnitude / ax.magnitude;
float delta_theta = 2 * M_PI / n_pieces;

// SOLUTION
float delta_theta_h = 2 * M_PI / ( 2 * n_pieces );

pVect vp = vx;  // Previous axis-to-cylinder vector.

for ( int i=1; i<=2*n_pieces; i++ ) {

    float theta = i * delta_theta_h;

    pVect vc = vx * cosf(theta) + vy * sinf(theta);
    if ( i & 1 ) vc *= r2or1;

    // Compute vertices of rectangle.
    pCoor pp_bot = p4 + vp,  pc_bot = p4 + vc;
    pCoor pp_top = p1 + vp,  pc_top = p1 + vc;

    // Emit two triangles making up the rectangle.
    bset_mta << pp_bot << pp_top << pc_top;
    bset_mta << pp_bot << pc_top << pc_bot;

    // Compute the normal of the rectangle.
    pNorm np = cross(pc_bot,pc_top,pp_top);

    // Emit the normals.
    bset_mta << np << np << np   << np << np << np;

    // Save vc for use in the next iteration.
    vp = vc;
  }
```

Problem 2: [15 pts] Coordinates `w0`, `w1`, and `w2` are the window-space coordinates of a triangle, call it `w`, ready to be rasterized. As pointed out in class, a triangle can be rasterized more efficiently if an edge is parallel to the x axis. Split triangle `w` into two triangles, `a` and `b`, such that `a` and `b` both have an edge parallel to the x axis and of course `a` and `b` cover the same points as `w`. To make things easier, assume that `w0.y > w1.y > w2.y`. Vectors between `w`'s vertices have been computed for convenience.

☑ Assign `a0` through `b2` so that `a` and `b` each have an edge parallel to the x axis and ☑ `a` and `b` together form `w`.

```
pCoor w0 = coors_w[i++];   // Assume w0.y > w1.y > w2.y
pCoor w1 = coors_w[i++];
pCoor w2 = coors_w[i++];

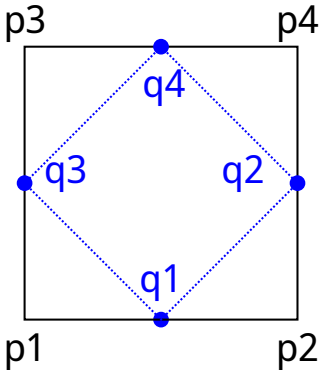
pVect v02(w0,w2);
pVect v01(w0,w1);
pVect v12(w1,w2);

// SOLUTION
pCoor w02 = w0 + v02 * v01.y / v02.y;
// A slightly more efficient solution: just one division.
pCoor w02b(  w0.x + v02.x * v01.y / v02.y,  w1.y,  w1.z );

pCoor a0 = w0;
pCoor a1 = w1;
pCoor a2 = w02;

pCoor b0 = w02;
pCoor b1 = w1;
pCoor b2 = w2;
```

Problem 3: [15 pts] The points p1 through p4 and points q1 through q4 in the diagram to the right each form a square, and both squares are in the same plane. Compute transformation matrix m (see below) that will transform points p1-p4 to points q1-q4. The code at the bottom of the page shows how the matrix will be used. Some transformations are shown for reference.



☑ Compute transformation m that rotates points as described.

The solution appears below. To transform the p points so that they are at the position of the q points we need to perform a rotation by 45° and then scale the points so that the square is smaller. The rotation should be around the center of the square, but typical rotation matrices rotate around the origin. So before applying such a rotation we need to translate the points so that the center of the square is over the origin. It is also important that the scaling be applied to points in which the center is at the origin.

So we need to apply the transformations in this order: (1) translate center to origin, (2) rotate, scale, (3) translate back to original location.

```
pCoor p1 = get_point(),  p2 = get_point();
pCoor p3 = get_point(),  p4 = get_point();

// Compute the coordinate of the center of the square.
pCoor ctr = p1 + pVect(p1,p4)/2;

// Prepare translation matrices that moves center to the origin and back.
pMatrix_Translate to_origin( -ctr );
pMatrix_Translate from_origin( ctr );

// Compute the rotation axis for use in rotating the points.
pNorm rot_axis = cross(p1,p2,p3);

// Prepare a rotation matrix that rotates by 45 deg  ( pi/4 ).
pMatrix_Rotation rot( rot_axis, M_PI/4 );

// Prepare a scale matrix that shrinks the p square to the q square's size.
pMatrix_Scale scale( 1/sqrt(2) );

// Combine these to form one transformation matrix.
//
pMatrix m = from_origin * scale * rot * to_origin;
//            Fourth      Third   2nd   First
//
// Note that transformations are applied from RIGHT to LEFT.

// Use m to compute the position of the q points.
pCoor q1 = m * p1;
pCoor q2 = m * p2;
pCoor q3 = m * p3;
pCoor q4 = m * p4;
```

Problem 4: [40 pts] Answer each question below.

(*a*) The vertex shader code below is getting the clip-from-object transformation matrix from a uniform variable. It is possible to supply this matrix as a vertex shader input. What would be the disadvantage of doing so?

```
// Conventional Approach
layout (location = LOC_IN_POS) in vec4 in_vertex_o;
void vs_main() {
  // Transform the vertex object-space coordinate to clip space.
  gl_Position = ut.clip_from_object * in_vertex_o;


// Alternative Approach
layout (location = LOC_IN_POS) in vec4 in_vertex_o;
layout (location = LOC_IN_MAT) in mat4 in_clip_from_object;
void vs_main() {
  // Transform the vertex object-space coordinate to clip space.
  gl_Position = in_clip_from_object * in_vertex_o;
```

☑ Disadvantage of sending matrix as shader input.

Answer: That would more than double the amount of data going into the vertex shader, which could hurt performance.

Explanation: Each matrix consists of 16 floats. Suppose the vertex shader inputs already consist of a coordinate, a color, and a normal. Those total $4 + 4 + 3 = 11$ floats. So sending the matrix along with them would more than double the amount of data.

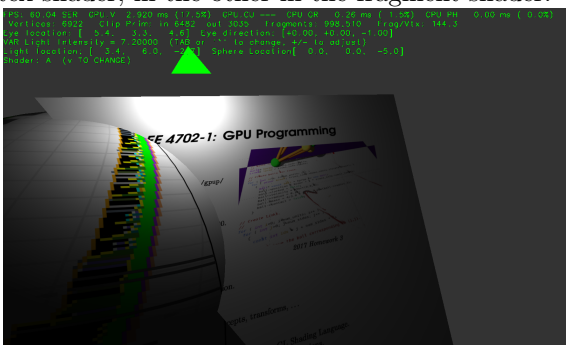
(*b*) In Homework 3 a set of colors was placed in a uniform variable. Would it be feasible to put vertex coordinates in a uniform variable also for typical rendering?

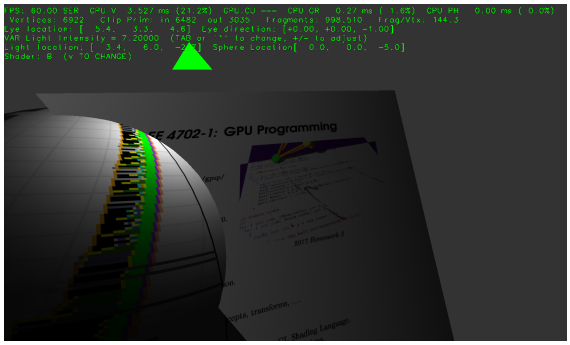
☑ Is putting vertex coordinates in a uniform variable ◯ *feasible* or ⊗ *not feasible* ?

☑ Explain.

It is not feasible because in a typical draw hundreds or more vertices are streamed in. There is a limit to the size of uniform variables (which varies by device) that size would be exceeded by using vertices.

(*c*) The two screenshots below show the same scene rendered with two different sets of shaders. The scene consists of one big triangle and a sphere (consisting of many triangles). In one image lighting is done in the vertex shader, in the other in the fragment shader.





☑ In which image is lighting being performed in the fragment shader? ⊗ *Shader A (left)* or ◯ *Shader B (right)* . ☑ Explain.

Shader A. Because the lighting is brightest *between* the big triangle's upper vertices. If lighting were done in the vertex shader a lighted color would have been computed for each of those two vertices and interpolated values would be used for the fragments between them. That's whats happening in the right figure, where the upper-left vertex looks whitish, but the upper-right vertex is dark, and the colors gradually change from whitish to dark from the left vertex to the right vertex.

☑ Performing lighting in the vertex shader ◯ *looks better* ⊗ *uses less computation* . ☑ Explain.

Answer: Less computation, because there typically fewer vertices than fragments.

Explanation: A triangle has three vertices, but can cover thousands or more fragments. The vertex shader is executed for each vertex, the fragment shader is executed for each fragment. The work needed to compute lighting would be done three times in the fragment shader but many times (hundreds of thousands for the images above, as those with sharp vision can see) so clearly there is much less work computing lighting in the vertex shader.

☑ Performing lighting in the fragment shader ⊗ *looks better* ◯ *uses less computation* . ☑ Explain.

Looks better because the lighting that is used is computed at the fragment, rather than an interpolation of values computed at the three vertices (for a triangle). See the answer to the In-which-image question.

(*d*) Suppose the big triangle in the images above were split into 64 triangles.

☑ How many additional vertices would there be when rendered using a triangle list.

Since a triangle list is being used, there would be $64 \times 3 = 192$ vertices which is 189 more than is used by the original triangle.

☑ How many additional fragments would there be?

Zero. The number of fragments does not change. (A fragment is generated if the center of a pixel is within the triangle, so there will not be fragments that are part of both triangles. Unless two triangles' edges pass over the same fragment centers. Because these are individual triangles there would be a fragment for each triangle. But that would require a perfect alignment. Since we're on the topic, if the two triangles were part of a triangle strip or fan then a fragment would be generated along the edges of just one of those triangles.)

☑ How would that affect the two images above?

The code using shader B (the vertex shader) would generate an image that would look more like shader A. The spot of light would look more like a polygon.

(*e*) One thing our CPU-only ray tracing code did not do was check for shadows. But it would be easy to do so. Suppose ray tracing code included a routine `cast_ray(ray_origin,ray_direction)` that returned the triangle closest to the ray origin. Using that routine, how can one check for shadows?

☑ With ray tracing can check for shadows by:

Let S denote the coordinates of the intersection of the eye ray with the closest triangle. Let L denote the coordinates of the light. Call `cast_ray(S,SL)`, where `SL` is the vector from S to L. If the closest intersection is the light itself, then S is illuminated, otherwise it is in the shade.

(*f*) In our CPU-only rasterization routines we had our own rasterization code:

```
for ( float b0=0; b0<=1; b0 += db0 )
  for ( float b1=0; b1<=1-b0; b1 += db1 ) {
      pCoor c = w2 + b0 * v20 + b1 * v21;  // Window-space coordinate.
      if ( uint(c.x) >= win_width || uint(c.y) >= win_height ) continue;
      demo_frame_buffer[ c.x + int(c.y) * win_width ] = color;
    }
```

The routine above was kept simple for teaching purposes, but we could have tuned it for the CPU or GPU the code was to run on. However in both Vulkan and OpenGL rasterization is a fixed-function stage, meaning we can't write our own specially tuned routine. Doesn't that hurt—not help—performance?

☑ Why is rasterization fixed-function? ☑ How does preventing us from tuning help performance?

The fixed-function rasterization is usually part of the GPU driver and has been written by the GPU manufacturer for the particular device it is running on. Our OpenGL or Vulkan code will then be using a good rasterizer wherever it is run. Had we taken the time to write a rasterizer for a particular device, it would only work well on that device, and would eventually become outdated. Unless we spent all of our time writing rasterizers for each new device that came on the market. But we rather be concentrating on more interesting parts of our real-time 3D application.

Name Solution

GPU Programming
LSU EE 4702-1
Final Examination

Wednesday, 7 December 2022 15:00-17:00 CST

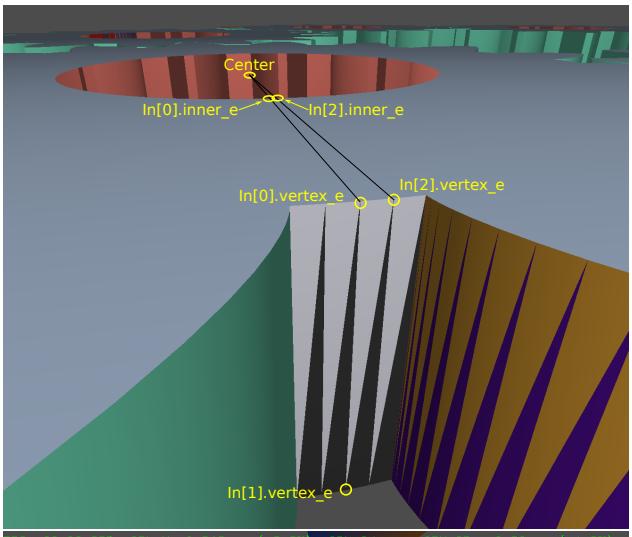
Problem 1		(20 pts)
Problem 2		(20 pts)
Problem 3		(15 pts)
Problem 4		(30 pts)
Problem 5		(15 pts)
Exam Total		(100 pts)

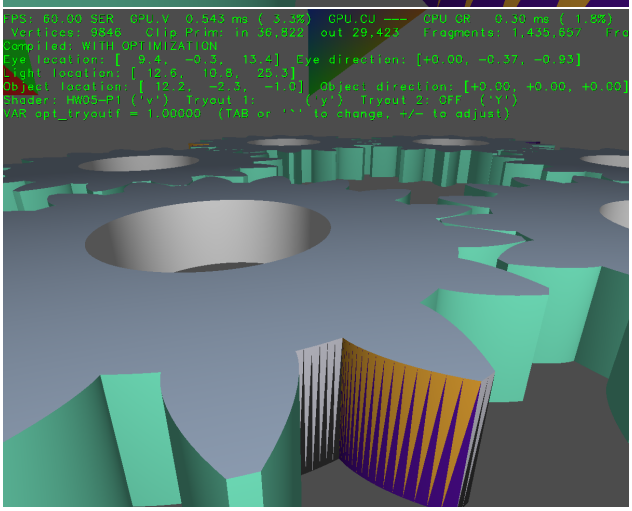
Alias Just say no to GPU scalpers.

Good Luck!

Problem 1: [20 pts] Appearing below is the geometry shader based on Homework 5, Problem 1, in which gears were rendered.

(*a*) The screenshot to the upper right shows how coordinates provided to the geometry shader, such as `In[0].inner_e`, relate to the gear. In the screenshot to the lower right the gear hole has walls, shown in white. Add code to the shader so that the walls are rendered. (The code rendering the sides of the gear [shown in gray] is not shown, and should be ignored for this problem.) *Hint: Don't write too much code. A loop is needed, but a correct solution does not need code before the loop.*





☑ Add code to render the walls.

☑ Be sure to correctly compute the normal, including for points on the lands (such as the purple/gold section).

☑ Use color index 7 for the walls.

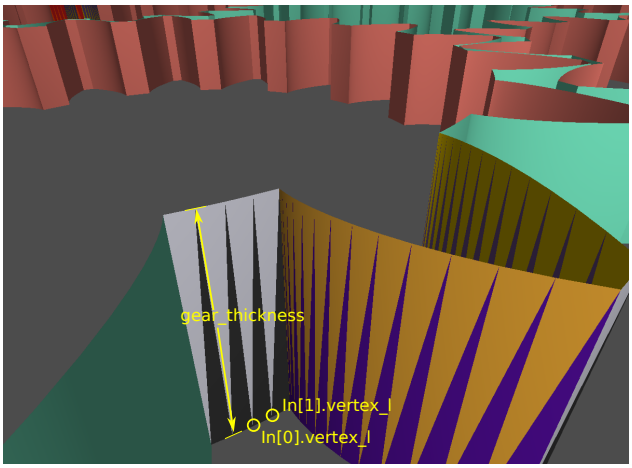
☑ Make sure that the front of the triangle is visible.

```
layout ( triangles ) in;
layout ( triangle_strip, max_vertices=8 ) out;
layout (location = 0) in Data_VG {
  vec4 vertex_c, vertex_e, inner_c, inner_e;
  vec3 normal_e;
  int color_idx;
} In[];

void gs_main_prob1() {  /// Geometry Shader
  for ( int i=0; i<3; i++ ) {    // Emit edge of gear (lands and faces).
      gl_Position = In[i].vertex_c;
      vertex_e = In[i].vertex_e;
      normal_e = In[i].normal_e;
      color_idx = In[i].color_idx;
      EmitVertex();
    }
  EndPrimitive();

  // SOLUTION -- Just use the inner coordinates.
  for ( int i=2; i>=0; i-- ) { // Iterate backward so triangle faces gear center.
      gl_Position = In[i].inner_c;   vertex_e = In[i].inner_e;
      color_idx = 7;
      // Coordinates of inner_e & vertex_e are on line through center, so use to compute normal.
      normal_e = normalize( In[i].inner_e.xyz - In[i].vertex_e.xyz );
      EmitVertex();
    }
  EndPrimitive();
```

(*b*) The geometry shader below correctly renders the gear (not the sides) and was written for a geometry shader input set to triangles (three vertices each). Modify the geometry shader so that it works when the geometry shader input is set to lines (two vertices each). The vertices provided are labeled in the screenshot to the right. These are vertices at $z = 0$ in local space. The (The output of the geometry shader remains a triangle strip.)



Input `vertex_l` is a local space coordinate along the bottom of the gear. The code correctly transforms these to eye and clip space. Modify the code so that it computes the coordinates along the top of the gear (`In[0] and In[2]` in the previous problem), and then emits triangles for the edge (but not the sides).

☑ Modify code so that it works for line strip input, and uses `gear_thickness` to compute the upper gear coordinates.

☑ Hint: Will need to update `max_vertices`. ☑ Hint: Not much more code is needed.

```
layout ( lines ) in; // Changed from triangles to lines.
layout ( triangle_strip, max_vertices = 3 ) out;  // [ ] Update as needed.
void gs_main_prob2() {   /// Geometry Shader
  int inst_idx = In[0].inst_idx;
  mat4 g_from_l = gears_xform[inst_idx] * rot;
  mat3 e_from_l = mat3(ut.eye_from_object) * mat3(g_from_l);
  vec4 avertex_e[3], vertex_c[3];
  vec3 anormal_e[3];
  float gt = gear_thickness; // Gear height in z axis in local space.

  for ( int i=0; i<3; i++ )
   {
      vec4 vertex_o = g_from_l * In[i].vertex_l;
      // Compute clip- and eye-space coordinates for vertex and normal.
      vertex_c[i] = ut.clip_from_object * vertex_o;
      avertex_e[i] = ut.eye_from_object * vertex_o;
      anormal_e[i] = normalize( e_from_l * In[i].normal_l );
    }



  for ( int i=0; i<3; i++ ) {
      gl_Position = vertex_c[i];
      vertex_e = avertex_e[i];
      normal_e = anormal_e[i];
      color_idx = In[i].color_idx;
      EmitVertex(); }
  EndPrimitive();
}
```

Solution appears below. The input topology class was changed to lines, and for that change the number of iterations in loops operating on `In` were reduced from 3 to 2. Since the upper vertex coordinate is no longer provided, it must be computed. That is done in the first `i` loop. The second `i` loop now emits two vertices per iteration, the lower one and the upper one.

```
// SOLUTION -- Change topology class from triangles to lines.
layout ( lines ) in;
// SOLUTION -- Change max_vertices from 3 to 4.
layout ( triangle_strip, max_vertices = 4 ) out;

void gs_main_fe_p1b() {   /// Geometry Shader
  int inst_idx = In[0].inst_idx;
  mat4 g_from_l = gears_xform[inst_idx] * rot;
  mat3 e_from_l = mat3(ut.eye_from_object) * mat3(g_from_l);

  // SOLUTION -- Change number of elements from 3 to 2, and remove vertex_c, add upper_vtx_e;
  vec4 avertex_e[2],  upper_vtx_e[2];
  vec3 anormal_e[2];
  float gt = gear_thickness; // Gear height in z axis in local space.

  for ( int i=0; i<2; i++ ) { // SOLUTION -- Change number of iterations from 3 to 2.
      vec4 vertex_o = g_from_l * In[i].vertex_l;

      // SOLUTION: Compute object-space coordinates of upper vertex (top of gear).
      vec4 upper_vertex_l = In[i].vertex_l + vec4(0,0,gt,0); // Just add thickness to z component.
      vec4 upper_vertex_o = g_from_l * upper_vertex_l;

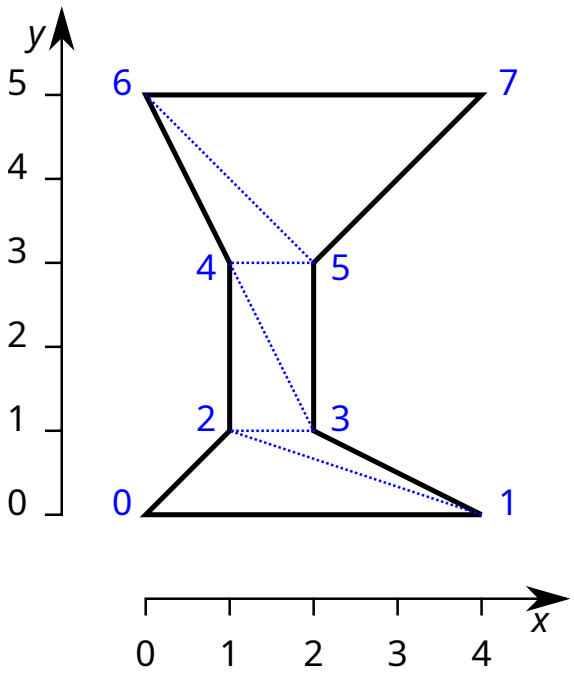
      // Compute eye-space coordinates for vertices and normal.
      avertex_e[i] = ut.eye_from_object * vertex_o;
      upper_vtx_e[i] = ut.eye_from_object * upper_vertex_o;

      // Note that both upper and original vertex have the same normal.
      anormal_e[i] = normalize( e_from_l * In[i].normal_l );
    }

  for ( int i=0; i<2; i++ ) {  // SOLUTION -- Change number of iterations from 3 to 2.
      // Emit Lower Vertex.
      vertex_e = avertex_e[i];
      gl_Position = ut.clip_from_eye * vertex_e;
      normal_e = anormal_e[i];
      color_idx = In[i].color_idx;
      EmitVertex();

      // SOLUTION -- Emit Upper Vertex.
      vertex_e = upper_vtx_e[i];
      gl_Position = ut.clip_from_eye * vertex_e;
      normal_e = anormal_e[i];
      color_idx = In[i].color_idx;
      EmitVertex();
    }
  EndPrimitive();
}
```

Problem 2: [20 pts] Answer the questions below about rendering an hourglass figure.



(*a*) Add coordinates to the buffer set below so that it renders the illustrated figure as a triangle strip. The z coordinate of all points is zero. Note that `pC(x,y)` can be used as an abbreviation for `pCoor(x,y,0)`. The first coordinate, `pC(0,0)`, is already inserted.

☑ Insert coordinates so that the buffer set renders the figure ☑ as a triangle strip.

☑ Don't overlap triangles.

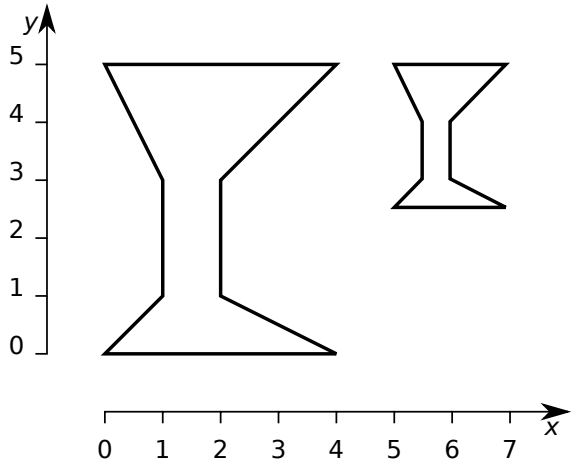
☑ Just insert coordinates, don't insert normal, colors, or anything else.

The solution appears below. In the code below vertex numbers are provided in the comments. In the diagram those same vertex numbers are shown, along with the resulting triangles (shown with blue dotted lines). A common mistake with a triangle strip is inserting coordinates in the wrong order, such as swapping vertex 2 and 3, but not changing the others.

```
bset_hg.reset(pipe_hg);
auto pC = [&](float x, float y)
  { return pCoor(x,y,0); };

// SOLUTION
bset_hg
<< pC(0,0) << pC(4,0)    // Vertex 0 and 1
<< pC(1,1) << pC(2,1)    // Vertex 2 and 3
<< pC(1,3) << pC(2,3)    // Vertex 4 and 5
<< pC(0,5) << pC(4,5);   // Vertex 6 and 7
```

(*b*) The figure to the right shows the hourglass from the previous problem in its original position (which is described by `bset_hg`), and a second time in a new position (which is to be described by `bset_hg2`). Buffer set `bset_hg` contains the coordinates for the hourglass in its original position. The code below inserts coordinates into `bset_hg2` using coordinates from `bset_hg` transformed using `m`. Complete the code so that `m` is assigned an appropriate transformation matrix. For reference, several standard transformation matrices are provided below.



☑ Compute `m` to transform larger hourglass to smaller hourglass.

☑ Note that there is a change in position and in size. ☑ Pay attention to the order in which transformations are applied.

```
// The declarations below are for reference.
float angle, scale_factor;
pVect vec;
pMatrix_Rotation mat_r(axis,angle);
pMatrix_Translate mat_t(vec);
pMatrix_Scale mat_s(scale_factor);
// The declarations above are for reference.

// Note: bset_hg already has coordinates of hourglass.



// SOLUTION

// First, scale the original hourglass to half the size.
//
pMatrix_Scale m_s( 0.5 );

// Next, move it to the new position.
//
pMatrix_Translate m_t( pVect( 5, 2.5, 0 ) );

// Compute the complete transformation matrix.
//
pMatrix m = m_t * m_s;



for ( pCoor c: bset_hg.pos ) bset_hg2 << m * c;
```

Problem 3: [15 pts] Answer the following questions about clip and eye space.

(*a*) Draw a diagram illustrating the view volume. The diagram should include the eye, the near plane, and the far plane.

☐ Diagram showing eye, view volume, near plane, and far plane.

Coming soon.

(*b*) Answer the following questions.

☑ The near plane distance was supposed to be set to 4.3. It is accidentally changed to 8.6. How does this affect the scene?

Note: The question in the original exam was: "What would happen if the near plane were too far from the eye?"

Objects that were visible before the change are no longer visible after the change. The z component of no longer visible objects are between -4.3 and -8.6 in eye space.

☑ What is the practical disadvantage of making the far plane too far from the eye?

This will result in very distant objects being rendered. Such objects are likely to be small, and so computation will be wasted on them.

(*c*) In a scene the user's eye is at location `eloc` (type `pCoor`) and the center of the user's monitor is at location `mloc` (type `pCoor`). The user is looking straight at the monitor, so it will appear as a rectangle, not as a trapezoid. The width of the user's monitor is `mwid` (type float). All of these dimensions are in global space. (For this problem consider object space equivalent to global space.)

Given this information plus needed assumptions compute the `eye_from_object` and `clip_from_eye` transformations. Some declarations have been provided for reference.

☑ Compute the `eye_from_object` transform. (OpenGL name: `gl_ModelViewMatrix`.)

☑ Compute the `clip_from_eye` transform. (OpenGL name: `gl_ProjectionMatrix`.)

☑ Make sure that `near` in the frustum transform is set correctly. ☑ Make sure that `height` in the frustum is set correctly, so that squares remain squares.

```
const int win_width = vh.s_extent.width;    // Frame buffer window width
const int win_height = vh.s_extent.height;
const float aspect = float(win_width) / win_height;

const float mwid = app.global_space_monitor_width;
const pCoor mloc = app.global_space_monitor_location;
const pCoor eloc = app.global_space_eye_location;

// Matrix examples below are for reference.
pMatrix_Rows m_r3( ax, ay, az );
// Rotate so that m_r3 * ax -> pVect(1,0,0), m_r3 * az -> pVect(0,0,1),...

pMatrix_Rotation m_r1( vec1, vec2 );
// Rotate so that m_r1 * vec1 = vec2;

pMatrix_Translate m_tlate(vec);
// Translate so that m_tlate * P -> P + vec;

pMatrix_Frustum m_frust( width, height, near, far );
// Width and height of "monitor" (small part of frustum).
// Near plane distance, far plane distance.

/// SOLUTION – Compute eye_from_object
//
// First the easy part, move eye to the origin.
pMatrix_Translate ctr(-eloc);

// Next, we need to compute a rotation matrix to rotate global
// space to eye space. Do so by finding eye-space axis in global space.

// The eye-space z-axis is from the eye to the monitor.
pNorm az(eloc,mloc);

// Assume that ay is the same in eye and global space.
pVect ay(0,1,0);

// With ay and az compute ax, and then construct the rotation matrix.
pNorm ax = cross( ay, az );
pMatrix_Rows m_to_monitor( ax, ay, az );
```

```
    // With the translation and rotation, construct eye_from_object.
    pMatrix eye_from_object = m_to_monitor * ctr;

    /// SOLUTION – Compute clip_from_eye
    //
    // Assume that the far plane should be 1000 times the near plane distance.
    pMatrix clip_from_eye = pMatrix_Frustum( mwid, mwid*aspect, az.magnitude, az.magnitude * 1000
);
```

Problem 4: [30 pts] Answer the following questions about rendering pipelines and shaders.

(*a*) Answer the following questions about shaders used for rasterization under Vulkan (or OpenGL).

☑ To render just one triangle in the whole scene, how many times would a vertex shader *typically* be called?

Three times, one for each vertex of course.

☑ Describe how it is possible to render T triangles with $T + 2$ invocations of a vertex shader.

By setting the input topology to a triangle strip.

☑ Describe how it is possible to render T triangles with T invocations of a vertex shader.

This would require the use of a geometry shader. The output of the geometry shader would be set to a `triangle_strip` grouping. The conceptually simplest input topology is a point. Each invocation of the geometry shader is provided one vertex (the point). From that it computes and emits three vertices, forming a triangle.

Another possibility is to set the input topology to a `triangle_list`, meaning a geometry shader invocation is provided three vertices. The geometry shader can then emit three triangles.

☑ To render one triangle in the whole scene, how many times would a geometry shader typically be invoked?

Once. That's typical. Non-typical might be invoking the geometry shader 100 times, and for 99 of those invocations it does not emit even one triangle.

☑ A geometry shader is invoked G times. Why is it not possible to determine how many triangles are rendered?

Because a geometry shader can emit zero or more triangles.

☑ What does one need to know about a triangle to determine how many times a fragment shader is invoked to render the triangle?

One needs to know its clip-space area and the number of pixels in the frame buffer. Or, equivalently, one just needs the window space area of the triangle.

(*b*) Answer the following questions about shaders in a Vulkan (NVidia or Khronos extension) ray tracing "pipeline".

☑ What does one need to know to determine how many times is the ray generation shader invoked? ☑ Provide an example.

The number of pixels in the window. For example, a window is 500 pixels wide by 300 pixels high. There are $500 \times 300 = 150000$ pixels. The ray generation shader would be invoked once per pixel. For each invocation a ray would be cast through the assigned pixel. Typically.

☑ The ray generation shader casts a ray. In what direction does it cast the ray?

It casts the ray from the eye to its assigned pixel.

☑ The ray generation shader used in class set its ray payload, a four-element vector, to zero. What does it do with the payload after the ray is cast?

It writes the payload to an image. That image will later be written to the frame buffer. (It might be composited with other images first.)

(*c*) Consider a scene that can be rendered using ray tracing or using rasterization. This was the case for many classroom examples. Consider a scene without shadows and without reflections.

☑ For the same scene, which is invoked more often, the fragment shader or the nearest hit shader? ☑ Explain.

The fragment shader. The nearest-hit shader is invoked at most once per cast ray. If there are no shadows or reflections there will be one ray cast per pixel. A fragment shader is invoked for each pixel covered by a primitive, for all primitives. A pixel can be covered by multiple primitives (when they overlap as viewed from the eye), so the fragment shader will be invoked more times.

Problem 5: [15 pts] Answer each question below.

(*a*) Consider Vulkan buffers bound to a pipeline as uniform buffers or as storage buffers.

☑ Both a uniform buffer and a storage buffer can hold arrays. ☑ Which is better for larger arrays?

Storage buffers. They have a much larger maximum size. Also, performance is poor when indexing uniform buffers with non-constant indices.

☑ Both uniform buffers and storage buffers can be used to store a single (meaning just one) object, such as a transformation matrix or small structure. ☑ Which is better for single objects, and why?

Uniform buffers. Reads from uniform buffers are faster.

(*b*) Textures are read using a built-in function in the fragment shader, such as `vec4 texel = texture(tex_unit_0,tex_coor);`.

☑ What are the range of values for `tex_coor`? How does that relate to the texture?

A `tex_coor` consists of two floats (typically). The range of coordinates in a texture is $[0, 1]$ along each dimension. (There are several ways of handling values outside this range, such as repeating the texture image. But that doesn't change the fact that the coordinate space is in $[0, 1]$.)

☑ How does the `texture` builtin function determine which MIPMAP level to use when retrieving texels?

It finds the MIPMAP levels for which the texel area is closest to the pixel area. To do that it might compare the texture coordinates of two adjacent fragments, and compare their difference to the number of pixels in the base image.

21 Fall 2021 Solutions

Name Solution_____

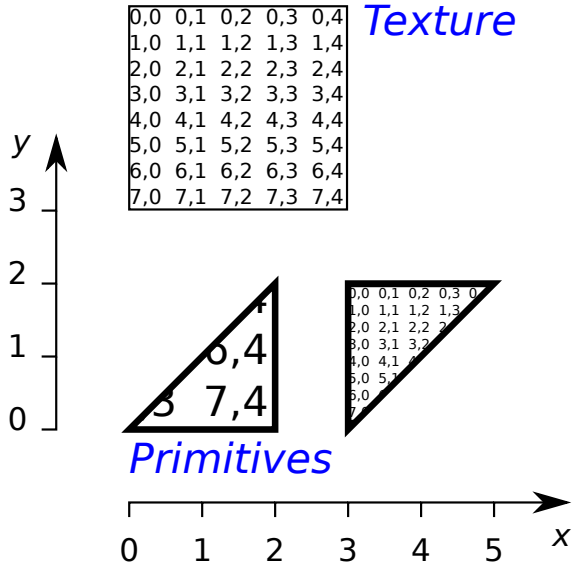
GPU Programming
EE 4702-1
Take-Home Pre-Final Examination

Due: 4 December 2021 at 23:59 CST

Problem 1	_____	(10 pts)
Problem 2	_____	(10 pts)
Problem 3	_____	(30 pts)
Problem 4	_____	(20 pts)
Problem 5	_____	(30 pts)
Exam Total	_____	(100 pts)

Good Luck!

Problem 1: [10 pts] Appearing to the right are two primitives (triangles) and a texture image. Further below is code that creates a pipeline for the two triangles and populates a buffer set with coordinates. Using convenience function `tC` insert texture coordinates so that the texture image appears as shown on the triangles.



☑ Insert texture coordinates into `bset_triss` so that the texture renders as shown.

☑ Just insert texture coordinates. Use the `tC` function to avoid writing out `pTCoor(foo,bar)`.

The solution appears below. Note that texture coordinates are in the range $[0, 1]$ and that the upper-left is $(0, 0)$. A solution in which the upper-left was assumed to be $(0, 1)$ would get full credit.

```
void World::render_tris(vk::CommandBuffer& cb) {
  if ( !pipe_tris )
    pipe_tris .init(vh.qs)
      .color_uniform_set( color_white * 0.9, color_red )
      .use_uni_light( uni_light )
      .use_texture( sampler, texid_tris )
      .topology_set( vk::PrimitiveTopology::eTriangleList )
      .create();

  auto pC = [&](float x, float y) { return pCoor(x,y,0); };
  auto tC = [&](float x, float y) { return pTCoor(x,y); };

  bset_triss.reset( pipe_tris );

  // SOLUTION
  bset_triss
    << pC(0,0) << tC( 0.8, 1 )

    << pC(2,0) << tC( 1,   1 )

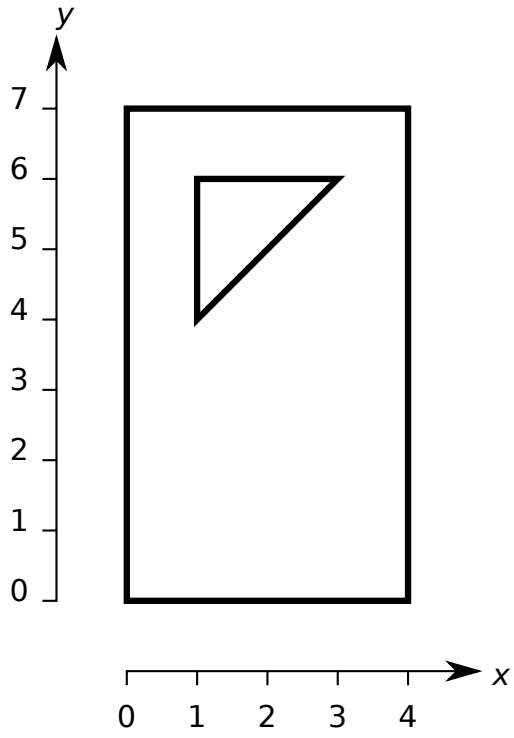
    << pC(2,2) << tC( 1,   0.8 )


    << pC(3,0) << tC( 0,   0.8 )

    << pC(5,2) << tC( 1,   0 )

    << pC(3,2) << tC( 0,   0 )
    ;
```

Problem 2: [10 pts] Appearing below is an illustration of a rectangle with a triangular hole, call it a card. Also in the illustration are x and y axes in a local coordinate space. For all points $z = 0$.



(a) Appearing below is code that prepares a pipeline and is to populate a buffer set for rendering the card. The first coordinate, $\begin{bmatrix} 0 \\ 7 \\ 0 \end{bmatrix}$, is inserted using the convenience function `pC`. Insert the remaining coordinates needed to render the card. Note that the pipeline expects vertices in triangle strip order.

☑ Describe the shape using one—just one—triangle strip.

☑ Use the coordinates given in the diagram. Omit the z coordinate.

☑ For this part just insert coordinates.

☑ No part of the hole should be covered by a triangle.

The solution appears below.

```
void World::render_card(vk::CommandBuffer& cb)
{
  if ( !pipe_card )
    pipe_card .init(vh.qs)
      .color_uniform_set( color_white * 0.9, color_red )
      .use_uni_light( uni_light ) .use_texture( sampler, texid_card )
      .topology_set( vk::PrimitiveTopology::eTriangleStrip )
      .create();

  auto pC = [&](float x, float y) { return pCoor(x,y,0); };

  bset_cards.reset( pipe_card );

  /// SOLUTION
  bset_cards
    << pC(0,7) << pC(1,6) << pC(4,7) << pC(3,6)
    << pC(4,0) << pC(1,4) << pC(0,0) << pC(1,6) << pC(0,7);
```

Problem 3: [30 pts] The code below renders a pointy p (from last year's pre-final and final exams) above each ball. The array `pts` holds the coordinates needed to draw a pointy p in its local coordinate space. (x and y only, all z values are zero). The `i` loop computes a transformation matrix, `p_xform`, that maps points from the pointy p local space to a position above ball `i`. The matrix is put into a storage buffer and the local-space coordinates are written to the buffer set. The z component of the coordinate is set to i, not zero. The vertex shader will use that value of i to retrieve the transformation matrix.

The solution to part b appears below in blue.

```
void World::render_p2(vk::CommandBuffer& cb) {
  if ( !pipe_p2 )
    pipe_p2.init(vh.qs)
      .storage_bind("BIND_PXFORM").uniform_bind( buf_uni_common, "BIND_UNI_COMMON" )
      .color_uniform_set( color_white * 0.9, color_red ).use_uni_light( uni_light )
      .shader_inputs_info_set<pCoor>()
      .shader_code_set
      ("mt-shdr-p.cc", "vs_main_p2(); ", "gs_main_p2();", "fs_main();")
      .topology_set( vk::PrimitiveTopology::eTriangleStrip )
      .create();

  // Coordinates needed to draw a pointy p using a triangle strip.
  vector<vec2> pts = { {0,0},{1,0},  {0,7},{1,6},  {4,5},{3,5},  {1,3},{1,4} };

  const int n_balls = balls.size();
  const bool n_changed = n_balls != n_balls_seen;
  n_balls_seen = n_balls;

  buf_p_xform.clear();    if ( n_changed ) bset_p2.reset( pipe_p2 );

  for ( int i=0; i<n_balls; i++ )
    {
      Ball* const ball = balls[i];
      pMatrix p_xform = pMatrix_Translate(ball->position) * ball->omatrix
        * pMatrix_Translate(pVect(0,ball->radius * 1.5,0))
        * pMatrix_Rotation(pVect(0,1,0),M_PI)
        * pMatrix_Scale( ball->radius / 3.5 );

      buf_p_xform << p_xform;

      // Emit the coordinates for a pointy p for ball i.
      // The value of i is put in the z coordinate so that the
      // shader can retrieve the transformation matrix.
      if ( n_changed ) for ( auto p: pts ) bset_p2 << pCoor(p.x, p.y, i);
    }

  buf_p_xform.to_dev();  // Send transformation matrices to the GPU.
  if ( n_changed ) bset_p2.to_dev();      // Send shader inputs to the GPU.

  pipe_p2.storage_bind( *buf_p_xform, "BIND_PXFORM" );

  transform.use_global_for( pipe_p2 );
  pipe_p2.record_draw(cb,bset_p2);
}
```

Problem 3, continued: The vertex and geometry shaders appear below.

```
// Interface block for vertex shader output / geometry shader input.
layout ( location = 0 ) out Data_to_GS
{
  vec3 normal_e;
  vec4 vertex_e;
  vec4 vertex_c;
};

void vs_main_p2() {
  vec4 vertex_o = vec4(in_vertex_o.xy,0,1);
  mat4 xform = p_xform[int(in_vertex_o.z)];
  vec3 normal_o = vec3(0,0,1);
  vec3 normal_g = mat3(xform) * normal_o;
  vec4 vertex_g = xform * vertex_o;

  vertex_c = gl_ModelViewProjectionMatrix * vertex_g;
  vertex_e = gl_ModelViewMatrix * vertex_g;
  normal_e = normalize(gl_NormalMatrix * normal_g );
}

layout ( location = 0 ) in Data_to_GS
{
  vec3 normal_e;
  vec4 vertex_e;
  vec4 vertex_c;

} In[];

layout ( triangles ) in;
layout ( triangle_strip, max_vertices = 3 ) out;

void gs_main_p2() {
  for ( int i=0; i<3; i++ )
    {
      gl_Position = In[i].vertex_c;
      normal_e = In[i].normal_e;
      vertex_e = In[i].vertex_e;
      EmitVertex();
    }
  EndPrimitive();
}
```

(*a*) Compute the amount of data sent from the CPU to the GPU for each call to `render_p2`. Don't include things that are done just once, such as the initialization of `pipe_p2`. Include the storage buffer and the buffer set. Base the size on the data inserted into the buffers by the routine. All data is inserted using the `<<` operator. In your answer let n denote the value of `n_balls` and let p denote the number of coordinates in `pts`. (We know $p = 8$ but use p anyway.)

☑ Amount of data, in bytes, sent for a typical call to `render_p2` in terms of n and c is:

Each `pMatrix`, a 4×4 matrix holds 16 floats for a total size of 16×4 B. Each `i` iteration inserts one matrix, so the total size is $64n$ B.

Each `pCoor`, a 4-element vector, is 4×4 B. In each `i` iteration p `pCoor` objects are inserted so for each `render_p2` call a total of np objects and a total size of $16np$ B for data in `bset_p2`.

The total amount of data is $n(64 + 16p)$ B.

(*b*) Assume that the value of `n_balls` does not change very often, but that the values of `ball->position` do change each time `render_p2` is called. Based on this observation, modify the code to reduce the amount of data sent from the CPU to the GPU. The variable `n_changed` should be useful, it is true when `render_p2` is called the first time and when the number of balls has changed.

Show your solution on the code several pages back.

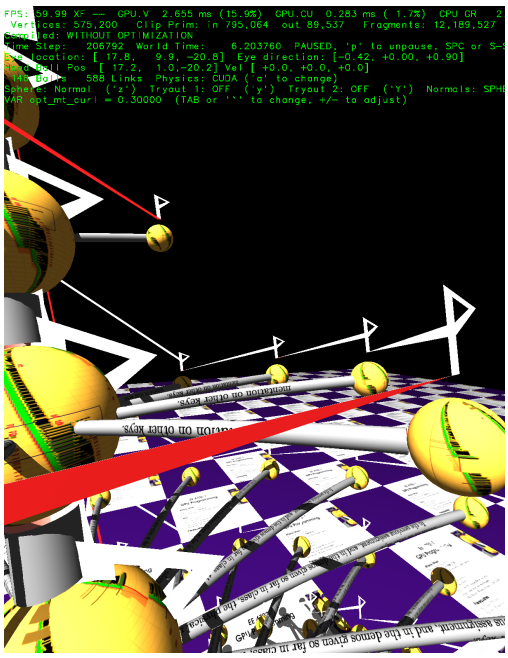
☑ Modify code to reduce CPU to GPU transfer when number of balls does not change.

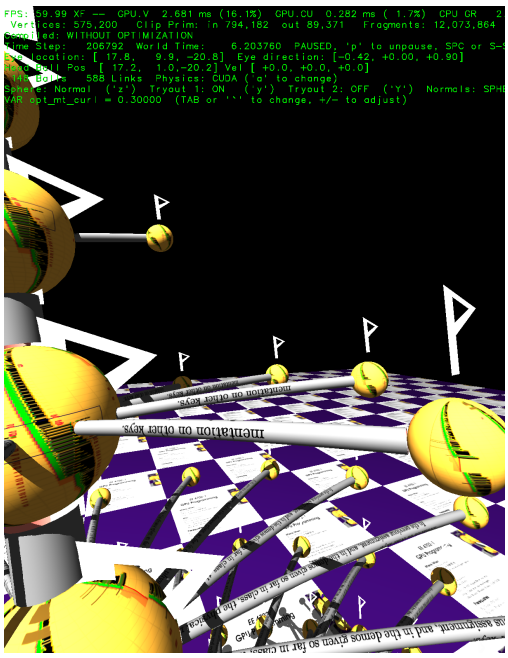
The calls to methods of `bset_p2` need to be guarded with `if ( n_changed )`. These changes are shown in the code in large blue type on the code several pages back.

☑ By what fraction has the amount of data been reduced?

Assuming that `n_changed` is false in the vast majority of calls to `render_p2`, then the fraction is $\frac{64}{64+16p}$ B.

(*c*) Because all the pointy p's are rendered using one triangle strip they are connected. See the screenshot on the lower left. Modify the shader code so that they are not connected, as in the screenshot on the right. *Hint: Use the value of* `i` *placed into the z component of the coordinate.*





☑ Modify shaders so that pointy p's are not connected.

Notice that one of the attributes that's sent with each vertex is the value of `i`, which is the ball number. So, to avoid inter-ball triangles have the geometry shader check whether all three vertices have the same value of `i`. If not, return without emitting a triangle. The solution code appears on the next page.

Solution, Problem 3c.

```
// Interface block for vertex shader output / geometry shader input.
layout ( location = 0 ) out Data_to_GS
{
  vec3 normal_e;
  vec4 vertex_e;
  vec4 vertex_c;

  int idx;                    // SOLUTION -- Pass i (as idx) to geometry shader.
};

void vs_main_p2() {
  vec4 vertex_o = vec4(in_vertex_o.xy,0,1);
  mat4 xform = p_xform[int(in_vertex_o.z)];
  vec3 normal_o = vec3(0,0,1);
  vec3 normal_g = mat3(xform) * normal_o;
  vec4 vertex_g = xform * vertex_o;

  vertex_c = gl_ModelViewProjectionMatrix * vertex_g;
  vertex_e = gl_ModelViewMatrix * vertex_g;
  normal_e = normalize(gl_NormalMatrix * normal_g );

  idx = int(in_vertex_o.z);   // SOLUTION -- Pass i (as idx) to geometry shader.
}

layout ( location = 0 ) in Data_to_GS
{
  vec3 normal_e;
  vec4 vertex_e;
  vec4 vertex_c;
  int idx;                    // SOLUTION
} In[];

layout ( triangles ) in;
layout ( triangle_strip, max_vertices = 3 ) out;

void gs_main_p2() {
  // SOLUTION - Don't emit triangle if values of i (idx) don't match.
  if ( In[0].idx != In[1].idx || In[0].idx != In[2].idx ) return;

  for ( int i=0; i<3; i++ )
    {
      gl_Position = In[i].vertex_c;
      normal_e = In[i].normal_e;
      vertex_e = In[i].vertex_e;
      EmitVertex();
    }
  EndPrimitive();
}
```

Problem 4: [20 pts] Appearing below is the code at the end of the default closest-hit shader, which is part of the course demo of the ray tracing extension to Vulkan and OGSL. (The code is in file `dev-vulkan/rt-shdr-main.cc`, and is in the `main` routine following `#ifdef _RT_CLOSEST_HIT_`.) This code casts a ray at the light and dims the color if the closest-hit vertex is shaded.

```
  // At this point rp_color is set to the lighted color, assuming no shadow.
  vec4 light_g = ut.mvi * ul.cgl_LightSource[0].position;
  vec3 vtx_to_light_g = light_g.xyz - vertex_g.xyz;
  float tmin = 0.001 / length(vtx_to_light_g),   tmax = 1.0;

  rp_shadowed = true; // The miss shader will set this to false, if invoked.

  traceNV
   ( topLevelAS,
     gl_RayFlagsTerminateOnFirstHitNV
     | gl_RayFlagsOpaqueNV | gl_RayFlagsSkipClosestHitShaderNV,
     0xfe, // Don't include lights in intersection test.
     0,0,  // Specify which kind of hit shader to use.
     1,    // Specify which miss shader to use.
     vertex_g.xyz,               // Ray Origin,
     tmin, vtx_to_light_g, tmax,  // Ray Vector, and Distance Range (tmin,tmax)
     2 );   // Ray payload location.

  rp_color = rp_shadowed ? 0.3f * c : c;
}
```

(*a*) Answer the following questions about, `tmin` and `tmax`, which are the minimum and maximum positions on the ray for ray traversal.

☑ Note that `tmax` is set to `1`. What would happen if `tmax` were set to `0.5`? ☑ Your answer should include a description of the flaw in rendered image due to this change.

If `tmax` were set to `0.5` shadows would not be cast by objects that were closer to the light than this vertex (`vertex_g`). So, an object would be well lit even when something is between the object and the light and closer to the light than the object.

☑ Care is being taken to set `tmin` to a value that's small, but not too small. What might go wrong if `tmin` were set to zero, or would certainly go wrong if `tmin=-.001`? ☑ Your answer should include a description of the flaw in rendered image due to this change.

If `tmin` were zero then ray traversal would find the same vertex (same value of `gl_Instance` and `gl_PrimitiveID`) which we are currently operating on. Since there would be no miss, the shader would assume a shadow. In the rendered image everything would be shadowed.

(*b*) The code above initializes `rp_shadowed = true` and uses a miss shader to set `rp_shadowed = false`. An alternative approach would be to initialize `rp_shadowed = false` and use a closest-hit shader to set `rp_shadowed = true`. This can easily be made to work, but it would have lower performance. Why?

☑ Why might the alternative approach, using a closest-hit shader, run more slowly than the code above, which uses a miss shader?

In the code above the hit shaders are never called because of the `gl_RayFlagsSkipClosestHitShaderNV` flag. Therefore, ray traversal would stop after finding one intersection. In the alternative approach, in which a closest-hit shader were to be called, the ray traversal process would have to find *all* intersections, and choose the one closest to `vertex_g`. Since ray traversal in the code above stops after the first intersection, it performs less work.

(*c*) In an instance using an intersection shader, the acceleration structure is prepared using axis-aligned bounding boxes (AABBs). For example, for our sphere shader (in file `rt-shdr-sphere.cc`) the AABB for a sphere of radius r is a cube with each edge of length $2r$, a perfect fit. Describe the impact on execution time and correctness if the AABBs were made too large, say, twice as large as they needed to be.

☑ Impact on execution time if AABBs were too large. ☑ Explain.

Execution time would be longer because more rays would intersect these larger AABBs, and so more intersection shader invocations would be made. Though none of these extra intersection shader invocations will report an intersection, they will still have to go through some computation before determining that there is no intersection.

☑ Impact on correctness if AABBs were too large. ☑ Explain.

There would be no impact on correctness. The image would look the same whether the AABBs were well chosen or too large. That's because an intersection with an AABB only results in the instance's intersection shader being called. It is up to the intersection shader to determine if there was an intersection with the geometry. There is no reason to assume that an intersection shader would give the wrong answer when a ray does not come close to the geometry.

Problem 5: [30 pts] Answer each question below.

(*a*) The two shaders below do the same thing, though slightly differently.

```
void vs_plan_a() {
  vertex_e = gl_ModelViewMatrix * in_vertex;
  gl_Position = gl_ProjectionMatrix * vertex_e;
}

void vs_plan_b() {
  vertex_e = gl_ModelViewMatrix * in_vertex;
  gl_Position = gl_ModelViewProjectionMatrix * in_vertex;
}
```

The shader code is provided a modelview matrix and a projection matrix at the beginning of the pipeline execution. In both `vs_plan_a` and `vs_plan_b` there are two matrix/vector multiplies, which each require $4^2 = 16$ multiply/add operations. But `vs_plan_b` uses `gl_ModelViewProjectionMatrix`, which is the product of the modelview and projection matrices. The product of these two matrices is computed using $4^3 = 64$ multiply/add operations. That brings the total to $16 + 16 + 64 = 96$ operations, much more than 32 for `vs_plan_a`. Therefore, `vs_plan_a` is more computationally efficient than `vs_plan_b`.

☑ Describe the flaw with this `vs_plan_a`-is-better-than-`vs_plan_b` argument.

The product is computed before pipeline execution, and it is computed at most once per pipeline execution. (That is because it is a uniform variable.) A pipeline execution is expected to have a large number of vertex shader invocations. Suppose that there are 1000 invocations. So the $4^3 = 64$ operations needed to compute the product is tiny compared to the computation performed by the vertex shaders, $32,000$ operations.

☑ Describe a case when the argument is correct, but explain why this case does not reflect typical use.

It would be correct if a pipeline execution processed just one vertex.

(*b*) Answer the following questions about view volumes.

☑ What is a view volume?

It is the part of the scene that is visible (projected on to the frame buffer). In Vulkan and OpenGL it is the part of the scene inside of a cube from clip-space coordinate (-1,-1,-1) to (1,1,1).

☑ It is easy to determine whether a vertex is in the view volume by using its coordinate in ◯ *object space*, ◯ *eye space*, or ⊗ *clip space* (check one).

☑ Let $P = \begin{bmatrix} x \\ y \\ z \\ w \end{bmatrix}$ be a coordinate in the coordinate space chosen for the previous answer. Show a mathematical expression or write code that tests whether P is in the view volume.

It is in clip space if $|x/w| \leq 1$, $|y/w| \leq 1$, and $|z/w| \leq 1$, or to avoid division $|x| \leq |w|$, $|y| \leq |w|$, and $|z| \leq |w|$.

☑ It is easy to determine whether some triangles are in the view volume. ☑ Provide an example of such a triangle and ☑ explain why.

It is easy if at least one vertex is in the view volume. To determine whether a vertex is in the view volume requires three comparisons. (See the answer to the previous part.) So the determination can be mode with nine comparisons (three per vertex).

☑ Provide an example of a triangle for which it is not so easy to determine if it is in the view volume. ☑ Illustrate with a diagram.

One in which all three vertices are outside the view volume. Though all vertices are outside the view volume part of the triangle can still be in the view volume and that requires extra steps to resolve. FINISH.

(*c*) Describe how suitable an OpenGL Shading Language uniform variable is for each of the following purposes.

☑ Explain whether this a suitable use for a uniform variable: To hold the lighted color computed by a vertex shader.

That won't work because uniform variables cannot be written by shaders, including vertex shaders. Even if the uniform were written by some other means, each vertex can have a different lighted color but the value of a uniform variable must be the same for every vertex in a pipeline execution.

☑ Explain whether this a suitable use for a uniform variable: To hold the location of a light source.

That is suitable because that would be the same for every vertex.

(*d*) Vertex coordinates are usually three dimensional but texture coordinates are usually two dimensional. Why? (Ignore the w component in your answer.)

☑ Texture coordinates have two, not three, dimensions because:

Because textures are mapped on to triangles, which are two dimensional.

(*e*) A homogeneous coordinate consists of four components, compared to just three for ordinary Cartesian coordinates. Homogeneous coordinates increase the amount of work needed for a matrix/vector multiply from 9 to 16 multiplications. Transformations are realized by multiplying a transformation matrix by a coordinate.

☑ Describe a transformation that cannot be done without homogeneous coordinates.

Translation.

☑ Describe a transformation that can be done using ordinary Cartesian coordinates.

Scale. Also rotation.

Name Solution

GPU Programming
LSU EE 4702-1
Take-Home Final Examination

Wednesday 8 December to Friday 10 December 2021 16:30 CST

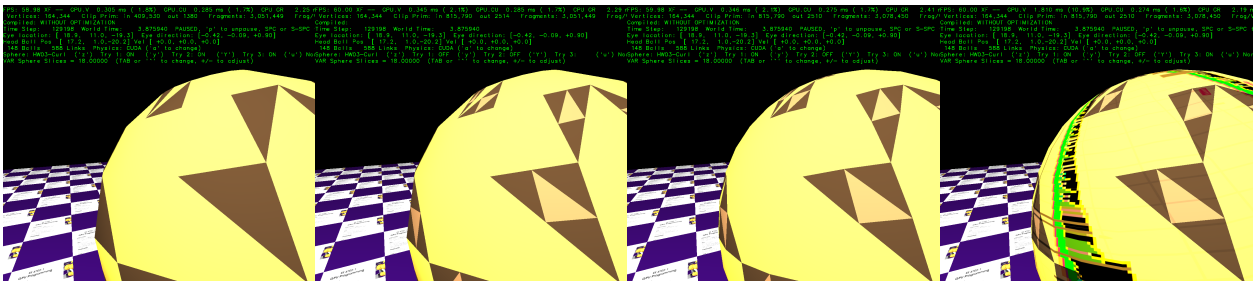
Work on this exam alone. Regular class resources, such as notes, papers, solutions, documentation, and code, can be used to find solutions. In addition outside Vulkan and OpenGL references and tutorials, other programming resources, and mathematical references can be consulted. Do not try to seek out references that specifically answer any question here. **Do not discuss this exam with classmates or anyone else**, except questions or concerns about problems should be directed to Dr. Koppelman.

Warning: Unlike homework assignments collaboration is not allowed on exams. Suspected copying will be reported to the dean of students. The kind of copying on a homework assignment that would result in a comment like "See ee4702xx for grading comments" will be reported if it occurs on an exam.

Problem 1	_____	(30 pts)
Problem 2	_____	(30 pts)
Problem 3	_____	(40 pts)
Exam Total	_____	(100 pts)

Good Luck!

Problem 1: [30 pts] An issue faced when approximating a sphere with triangles is the number of triangles to use. If too few are used the sphere looks like a polyhedron, especially around the edges. Using too many can slow things down. One possible solution is to split the triangles in the geometry shader—if necessary. The lower-left image shows a sphere that does not have enough triangles. Some triangles are highlighted (in brown) to make them apparent. Looking along the edge (limb) of the sphere one can see straight lines, which is not what we want to see. The second image shows what happens when each triangle is split into four triangles. (Remember that only some are highlighted in brown shades.) But, looking along the edge we see that this splitting hasn't helped. The reason is that the old triangle and the four new ones all fall in the same plane. The third image shows the way the triangles should be split. The three new vertices are positioned to be on the sphere surface, rather than the plane of the original triangle. The last shows the same sphere as the third, but with a texture applied.



(*a*) Modify the shader code on the next page so that it splits a triangle into three triangles as described above. The vertices of all triangles must be on the surface of the sphere. The code is provided with some routines that should be helpful. Routine `emit(i)`, to be called in the geometry shader, emits vertex `i`. It is used in the geometry shader main routine, `gs_main()`. Routine `emit(i,j)` emits one vertex, but the vertex attributes are interpolated from geometry shader inputs `i` and `j`. So, `emit(0,1)` will emit a vertex halfway between vertex 0 and 1. That vertex is **not** on the sphere surface.

Solve this problem by first calling `emit` so that the four triangles are emitted, though on the plane of the original triangle. Try to do so using no more than 8 vertices.

Then modify `emit(i,j)` so that the interpolated vertex lies on the sphere surface. Though it is possible to do so using only information currently provided to the geometry shader, it is okay if additional information is passed from the vertex shader, such as the sphere center.

☑ Modify `gs_main` so that it emits four triangles as illustrated.

☑ Modify `emit(i,j)` so the interpolated vertex lies on the sphere surface.

☑ If needed, pass additional information from the vertex shader to the geometry shader.

☑ Set `max_vertices` to the number of vertices emitted.

Solution appears below.

```
layout ( location = 0 ) in Data_to_GS {
  vec3 normal_e;
  vec4 color;
  vec2 tex_coor;
  // SOLUTION -- Send sphere center and radius to geo shader.
  vec4 center_e;
  float sphere_radius;
} In[];

void vs_main_instances_sphere()   // Vertex Shader
{
  vec4 pos_rad = sphere_pos_rad[gl_InstanceIndex];
  vec4 center_o = vec4(pos_rad.xyz,1);
  // SOLUTION -- Change declaration to just an assignment. (Remove vec4.)
  sphere_radius = pos_rad.w;
  mat4 rot = sphere_rot[gl_InstanceIndex];
  vec4 normr = rot * in_vertex_o;
  vec3 normal_o = normr.xyz;
  normal_e = normalize(gl_NormalMatrix * normal_o ); // gl_NormalMatrix same as mat3(eye_from_clip)
  // SOLUTION -- Remove declaration from center_e
  center_e = gl_ModelViewMatrix * center_o;          // gl_ModelViewMatrix same as eye_from_clip
  // SOLUTION -- vertex_e now computed in geo shader, so don't send it.
  // vertex_e = center_e + vec4(normal_e * sphere_radius,0);
  tex_coor = in_tex_coor;
  color = sphere_color[gl_InstanceIndex];
}

layout ( triangles ) in;
// SOLUTION -- Increase max_vertices to 8.
layout ( triangle_strip, max_vertices = 8 ) out;

void emit(int i0, int i1) {
  float f = i0 == i1 ? 0 : 0.5;

  normal_e = mix(In[i0].normal_e,In[i1].normal_e,f);

  // SOLUTION: Compute surface coordinate by adding scaled interpolated normal
  //           to the sphere center.
  //
  vertex_e = In[i0].center_e + vec4(In[i0].sphere_radius * normalize(normal_e),0);

  color = mix(In[i0].color,In[i1].color,f) * shade[tidx%2];
  tex_coor = mix(In[i0].tex_coor,In[i1].tex_coor,f);
  gl_Position = gl_ProjectionMatrix * vertex_e;      // gl_ProjectionMatrix same as clip_from_eye
  EmitVertex();
}
```

```
void gs_main() {
  // SOLUTION -- Comment out code emitting the original triangle.
  //     emit(0); emit(1); emit(2); EndPrimitive();

  // Emit three triangles in one strip.
  emit(1);   emit(1,2);
  emit(0,1); emit(0,2);
  emit(0);
  EndPrimitive();

  // And emit one more triangle in a second strip.
  emit(0,2);  emit(1,2);
  emit(2);
  EndPrimitive();
}
```

(*b*) The code in the previous part always splits a triangle. But there is no point splitting a triangle if it is already small. Consider a triangle small if an edge spans less than 20 pixels. Also, it is triangles that are near the edge that should be split. Those in the center of our view look good even when relatively large. So it would make sense only to split triangles near the edge.

Modify the geometry shader below so that it assigns `split_tri` to true if the triangle should be split based on the size (in pixels) and whether it is close to the edge.

To help determining the size the window width in height in pixels are provided. What remains is determining the length of an edge, measured in pixels. That can be done using a particular coordinate space.

To determine if something is on the edge use the vertex normal and other information. (It is not necessary to use the exact method of the sphere shader.)

☑ Set `is_large` to true iff an edge is larger than 20 pixels.

☑ Set `near_edge` to true if the triangle is close to the edge of the sphere as seen by the viewer.

Solution on next page.

To determine whether the triangle is large—an edge is more than 20 pixels long—the triangle dimensions are first converted to window space and then the length of each edge is computed.

Consider a vector from a triangle vertex to the eye, that vector is called `v_to_eye` in the code below. A triangle with normal `normal_e` is facing away from the user if `dot( v_to_eye, normal_e ) < 0` . If the dot product is zero the triangle can't be seen because it is perfectly edge-on with the eye. (The eye is in the same plane as the triangle.)

Recall that when rendering a sphere using triangles, the vertex normals are set to the normal of the sphere surface, not to the normal of the triangle. If a triangle crosses the edge of the sphere then at least one of its vertices is on a part of the sphere that is hidden and at least one is on a part of the sphere that is visible. Let P denote coordinate of a vertex on the sphere, n the normal, and E the coordinate of the eye. The vertex is visible if $n \cdot \overrightarrow{PE} > 0$. The code below checks the visibility of each triangle vertex, and sets `near_edge` to true if at least one vertex is visible and at least one vertex is not visible. Note that $\overrightarrow{PE}$ is just P is $-P$ is an eye-space coordinate.

A common mistake was to use $\begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix}$ instead of $\overrightarrow{PE}$. This is computationally efficient since $n \cdot \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix}$ is just the z component of n, but it only works if P is in the center of the screen.

```
int w_px = com.win_width;
int h_px = com.win_height;

// SOLUTION
float len_limit = 20;

// Compute window-space coordinates of triangle vertices.
vec2 vtx_w[3];
for ( int i=0; i<3; i++ )  {
    // Compute clip-space coordinate of triangle vertex.
    vec4 c = gl_ProjectionMatrix * In[i].vertex_e;

    // Extract x and y component and homogenize them.
    vec2 vtx_c = c.xy / c.w;

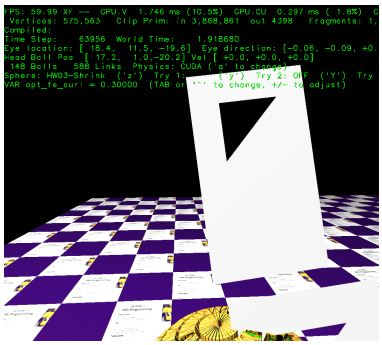
    // Convert to window space (but without recentering).
    vtx_w[i] = vtx_c * vec2(w_px,h_px);
  }

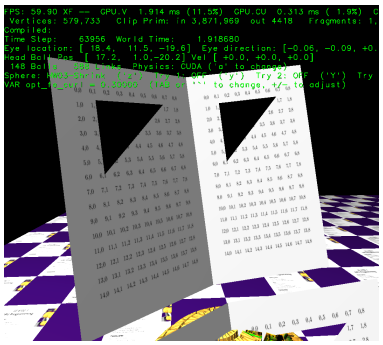
// Check whether any edge is longer than len_limit (20 pixels).
bool is_large = false;
for ( int i=0; i<3 && !is_large; i++ )
  if ( distance( vtx_w[i], vtx_w[(i+1)%3] ) >= len_limit )  is_large = true;

int n_ph_neg = 0;  // Number of hidden vertices.
int n_ph_pos = 0;  // Number of visible vertices.
for ( int i=0; i<3; i++ ) {
    vec3 v_to_eye = -In[i].vertex_e.xyz;
    if ( dot( In[i].normal_e, v_to_eye ) <= 0 ) n_ph_neg++; else n_ph_pos++;
  }
if ( n_ph_pos == 0 ) return;

bool near_edge = opt_tryout1 || n_ph_neg > 0 && n_ph_pos > 0;
bool split_tri = is_large && near_edge;
// Assume that remainder of geometry shader code is here.
```

Problem 2: [30 pts] The code below, once this problem is solved, will show the cards from the pre-final rendered in pairs as shown in the screenshot on the right. Each card shows the entire texture image (once per card). The screenshot on the left shows the original card, and without a texture applied.





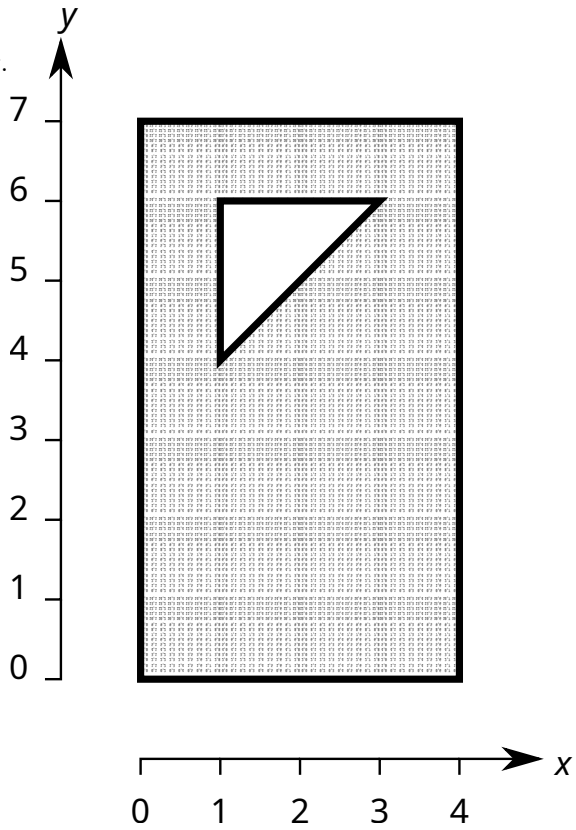
(*a*) First, get the texture right. The code below inserts texture coordinates into the buffer set for the card. Using the diagram show how the texture will appear based on this code. (It won't be blank.) The sampler has been set to repeat the texture.

☑ Using diagram above show texture mapped by code below.

```
World::render_card(vk::CommandBuffer& cb)
{
  auto pC = [&](float x, float y)
   { return pCoor(x,y,0); };
  bset_card.reset( pipe_card );
  bset_card
    << pC(0,7) << pC(1,6) << pC(4,7) << pC(3,6)
    << pC(4,0) << pC(1,4) << pC(0,0) << pC(1,6)
    << pC(0,7);

  for ( pCoor c: bset_cards.pos.vals )
    {
      float x = c.x,  y = c.y;
      bset_cards << pTCoor( x, y );
    }

  bset_cards.to_dev();
```



The texture coordinate space is $x, y \in [0, 1]$, with $\begin{bmatrix} 0 \\ 0 \end{bmatrix}$ being in the upper-left corner and $\begin{bmatrix} 1 \\ 1 \end{bmatrix}$ being in the lower-right corner of the texture image. When the sampler is set to repeat a texture coordinate such as $\begin{bmatrix} 1.2 \\ 3.4 \end{bmatrix}$ will be converted to $\begin{bmatrix} 0.2 \\ 0.4 \end{bmatrix}$. Because x ranges from 0 to 4 in the figure, the texture will be repeated four times from left to write. Similarly, the texture will be repeated seven times vertically but will be upside down.

(*b*) Modify the code below so that it shows the entire texture image on the card, as illustrated on the previous page.

☑ Modify code to show texture image as shown on right image.

```
World::render_card(vk::CommandBuffer& cb)
{
  auto pC = [&](float x, float y) { return pCoor(x,y,0); };
  bset_card.reset( pipe_card );
  bset_card
    << pC(0,7) << pC(1,6) << pC(4,7) << pC(3,6)
    << pC(4,0) << pC(1,4) << pC(0,0) << pC(1,6) << pC(0,7);

  for ( pCoor c: bset_cards.pos.vals )
    {
      // This loop iterates through the coordinates inserted above.
      // That is, on the first iteration c=pCoor(0,7,0), on the second
      // iteration c=pCoor(1,6,0), etc.

      float x = c.x,  y = c.y;

      // SOLUTION -- Scale coordinates to fit card. Also flip y axis.
      x = x/4;
      y = 1 - y/7;

      bset_cards << pTCoor( x, y );
    }

  bset_cards.to_dev();
```

(*c*) Modify the code below so that it shows the second card placed to the left of each original card and rotate 30 degrees. The right edge of the new card should touch the left edge of the old card. Do this by inserting a new transformation matrix into `buf_card_xform`. The code below starts off a new transform, `t2`. It needs to be finished and inserted into the buffer.

☑ Add a transform so that there is a new card to the left of each existing one, as described above.

The solution appears below. To move the new card to the left of the old card first translate it by 4 in the $-x$ direction. After this translation the right side of the card will be at $x = 0$. So rotate the card by $30\,\text{deg} = \pi/6\,\text{rad}$ around the y axis.

```
buf_card_xform.clear();
for ( auto& ball: balls ) {
    pMatrix ctform =
      pMatrix_Translate(ball->position)
      * ball->omatrix
      * pMatrix_Translate(pVect(0,ball->radius * 1.5,0))
      * pMatrix_Rotation(pVect(0,1,0),M_PI)
      * pMatrix_Scale( ball->radius / 3.5 );

    buf_card_xform << ctform;

    // SOLUTION
    pMatrix t2 = ctform
      * pMatrix_Rotation( pVect(0,1,0), M_PI/6 )
      * pMatrix_Translate( pVect(-4,0,0) );

    buf_card_xform << t2;
  }

buf_card_xform.to_dev();
pipe_card.storage_bind( buf_card_xform, "BIND_XFORM" );
transform.use_global_for(pipe_card);
pipe_card.record_draw_instanced(cb,bset_card,buf_card_xform.size());
```

Problem 3: [40 pts] Answer each question below.

(*a*) A piece of information that we often compute and send down the rendering pipeline is the eye-space normal, often called `normal_e`. What is that used for, and how would the image be affected if it were set to always point up.

☑ In the pipelines used in class the eye-space normal is used for:

Computing the lighted color.

☑ How might an image, say a sphere, appear if `normal_e` always pointed up.

Because the lighting computation does not take into account the actual direction the surface is facing, but instead is computed as though the surface is facing up, the sphere will not look spherical, it would look more like a circle.

(*b*) In ray tracing, the ray generation shader casts a ray from the eye through each pixel. For each ray the frame buffer is written for the geometry item closest to the eye using the color provided (typically) by a closest-hit shader. In rasterization rendering (the material before ray tracing) how do we insure that the item closest to the eye is written to the frame buffer?

☑ Rasterization rendering ensures the frame buffer is written with an item closest to the eye by:

Using a depth buffer. The depth buffer holds the distance to the eye for whatever is in the frame buffer. That is compared to the distance from the eye of the item to be written, the new item is written (typically) only if it is closer. The color along with the distance (depth) is written.

(*c*) In class we prepared a buffer set holding the vertices of a sphere, and used that for rendering multiple spheres in a scene. We did that for the entire sphere. (This problem is about the tessellated sphere, not the true-sphere shader.) That sounds wasteful because the whole sphere is never visible. Why not just store vertices for half of a sphere, say the front half ($-z$). Explain the flaw in the following argument:

One cannot save space by storing vertices for half of a sphere because the eye can be moved to any position around the sphere, and so the eye could move to the part in which the vertices were omitted.

☑ The flaw in the argument above is:

It ignores the fact that a transformation can be used to rotate the sphere coordinates so that they are facing the eye.

(*d*) How does the shape of the view volume in a rasterization rendering compare to the shape of the view volume using the following ray-generation shader:

```
void main() {
  const vec2 pixel_p = vec2(gl_LaunchIDNV.xy) + vec2(0.5);
  vec2 inUV = pixel_p/vec2(gl_LaunchSizeNV.xy);
  inUV.y = 1 - inUV.y;
  vec2 pixel_c = inUV * 2.0 - 1.0;

  vec4 eye_g = ut.object_from_eye * vec4(0,0,0,1);
  vec4 pixel_e = ut.eye_from_clip * vec4(pixel_c, -1, 1) ;
  vec4 eye_to_pixel_g = ut.object_from_eye * vec4(normalize(pixel_e.xyz), 0) ;

  uint rayFlags = gl_RayFlagsOpaqueNV, cullMask = 0xff;
  float tmin = 1, tmax = 10000.0;

  traceNV
   ( topLevelAS, rayFlags, cullMask,
     0 /* intersect/hit sbtRecordOffset*/,
     0 /* intersect/hit sbtRecordStride*/,
     0 /*missIndex*/,
     eye_g.xyz, tmin, eye_to_pixel_g.xyz, tmax,
     0 /*payload location */);

  imageStore(fb_image, ivec2(gl_LaunchIDNV.xy), vec4(rp_color, 0.0));
}
```

☑ How do the shapes compare. Be specific, use a sketch.

In rasterization the shape of the view volume is a frustum (a pyramid with the top cut off). The view volume corresponding to the ray generation shader code above is close in shape to a frustum, except that rather than a near plane there is a spherical surface (not a whole sphere). The equivalent of the far plane is also a spherical surface. If `pixel_e` were not normalized, the shape would be a frustum.

(*e*) CPU code is compiled ahead of time. In a production setting, it is compiled by the developers and shipped in binary form to the customer. But shader code is typically compiled when the program is run. Why?

☑ Why is it important for shader code to be compiled at run time on the user's system while its okay for CPU code to be compiled in advance by the developer?

Because different GPUs, especially different GPU generations, vary by a great deal. It is important that code is optimized for each, and so code is best compiled when the exact GPU is known, which is typically at run time.

To speed things up a little, shader code can be converted into an intermediate language, usually SPIR-V for Vulkan, at build time. Then at run time the SPIR-V code is compiled.

(*f*) Our true sphere shader enabled us to render a perfect sphere. For each sphere one vertex was sent down the pipeline. The geometry shader emitted a rectangle framing the sphere, and the fragment shader computed which point on the sphere projects on to that fragment. A true cube shader could be designed along the same lines. Would it improve things over the conventional method of rendering cubes? (The conventional method is 12 triangles, perhaps using 6 triangle strips.)

☑ Would a true cube shader make a better-looking cube than the conventional rendering methods? ☑ Explain.

No. The conventional method renders a cube perfectly because two triangles can be chosen to exactly match a cube face.

☑ Would a true cube shader reduce computation over the conventional rendering method? ☑ Explain.

It would likely increase computation because the fragment shader would have to go through the process of finding where on the cube the fragment lies.

(*g*) A triangle strip is rendered with different colors, perhaps our card:

```
bset_card.reset( pipe_card );
bset_card
  << pC(0,7) << pC(1,6) << pC(4,7) << pC(3,6)
  << pC(4,0) << pC(1,4) << pC(0,0) << pC(1,6) << pC(0,7);

bset_card
  << color_red << color_white << color_green << color_orange
  << color_blue << color_lsu_spirit_purple << color_lsu_spirit_gold
  << color_turquoise << color_gray;
```

The interface block for the fragment shader input uses the `flat` interpolation qualifier for color:

```
layout ( location = 0 ) out Data_to_FS {
  vec3 normal_e;
  vec4 vertex_e;
  vec2 tex_coor;
  flat vec4 color;
};
```

☑ How would the appearance differ if the `flat` qualifier were omitted? Illustrate with a portion, not the whole thing.

If the flat qualifier were omitted the colors at each vertex would blend into each other. With the flat qualifier each triangle would be a solid color.

22 Fall 2020 Solutions

Name Solution

GPU Programming
EE 4702-1
Take-Home Pre-Final Examination

Due: 5 December 2020 at 23:59 CST

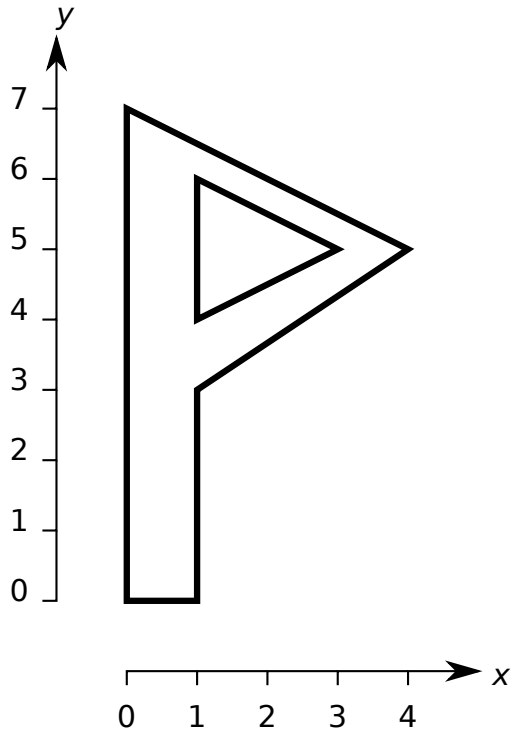
Work on this exam alone. Regular class resources, such as notes, papers, solutions, documentation, and code, can be used to find solutions. In addition outside OpenGL references and tutorials, other programming resources, and mathematical references can be consulted. Do not try to seek out references that specifically answer any question here. **Do not discuss this exam with classmates or anyone else**, except questions or concerns about problems should be directed to Dr. Koppelman.

Warning: Unlike homework assignments collaboration is not allowed on exams. Suspected copying will be reported to the dean of students. The kind of copying on a homework assignment that would result in a comment like "See ee4702xx for grading comments" will be reported if it occurs on an exam. Please do not take advantage of pandemic-forced test conditions to cheat!

Problem 1	_____	(20 pts)
Problem 2	_____	(20 pts)
Problem 3	_____	(30 pts)
Problem 4	_____	(30 pts)
Exam Total	_____	(100 pts)

Good Luck!

Problem 1: [20 pts] Appearing below is an illustration of a pointy p and an incomplete routine. Complete the routine so that it emits a green pointy p using a single triangle strip and `glVertex` calls.



☑ Add code to start and stop a triangle-strip rendering pass.

☑ Emit the shape using one—just one—triangle strip. Use `glV` as an abbreviation for `glVertex3f`.

☑ Use the coordinates given in the diagram. Omit the z coordinate.

☑ Set a correct normal and color.

```
void render_p() {  /// SOLUTION
  glBegin(GL_TRIANGLE_STRIP);

  glColor3f(0,.2,0);
  glNormal3f(0,0,1);

  glVertex3f(0,0,0);
  glVertex3f(1,0,0);

  glVertex3f(0,7,0);
  glVertex3f(1,6,0);

  glVertex3f(4,5,0);
  glVertex3f(3,5,0);

  glVertex3f(1,3,0);
  glVertex3f(1,4,0);

  glEnd();
}
```

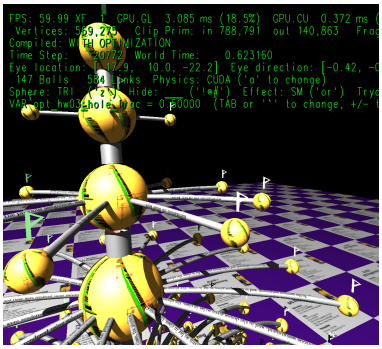
Problem 2: [20 pts] Consider `render_p` from the previous problem.

(*a*) Compute the amount of data sent from the CPU to the GPU each time `render_p` is called. If you did not solve the previous problem then assume that `render_p` calls `glVertex3f` eight times.

☑ Amount of data due to execution of `render_p`. ☑ Show units and state assumptions.

Each execution of `render_p` sends one normal, one color, and eight vertices from the CPU to the GPU and into the rendering pipeline. All of these send floats, which are 4 bytes each. Each sends three floats (as can be seen by the `3f` suffix). So the total amount of data is $(1 + 1 + 8) \times 3 \times 4 = 120\,\mathrm{B}$.

(*b*) The code below calls `render_p` for each ball. Before calling `render_p` the modelview matrix is modified so that the pointy p is rendered above the sphere. Suppose that the updated modelview matrix is sent to the GPU at the beginning of a rendering pass.



```
void World::render_prob1() {
  glMatrixMode(GL_MODELVIEW);
  for ( Ball *ball: balls ) {
      glPushMatrix();
      pNorm b_to_eye(ball->position,eye_location);
      pVect d_p = ball->omatrix * pVect(0,1,0);
      pCoor p_pos = ball->position + 1.5 * ball->radius * d_p;
      pVect az = b_to_eye;
      pNorm ax = cross(b_to_eye,pVect(0,-1,0));
      pVect ay = cross(az,ax);
      float s = 0.2 * ball->radius;
      pVect vx = s * ax, vy = s * ay, vz = s * az;
      pMatrix xform(vx,vy,vz,p_pos); // Concatenate vectors into a matrix.
      glMultTransposeMatrixf(xform);
      render_p();
      glPopMatrix();
    }}
```

Consider an alternative implementation where the vertex shader computes and uses the updated modelview matrix rather than having it computed by the routine above.

☑ How much data (in bytes) would this vertex shader invocation need to read in order compute the new matrix?

The vertex shader would need to read the values of the following variables: `ball->position`, `ball->omatrix`, `ball->radius`, `eye_location`, and implicitly, the current value of the modelview matrix.

Variable `ball->position` and `eye_location` are each four-element vectors, and so $4 \times 4 = 16\,\mathrm{B}$ each. Variable `ball->omatrix` is a 3×3 vector of floats, its total size is $3 \times 3 \times 4 = 36\,\mathrm{B}$. The modelview matrix is a 4×4 vector of floats, its total size is $4 \times 4 \times 4 = 64\,\mathrm{B}$. Finally, `ball->radius` is one float which is $4\,\mathrm{B}$.

Not all of this data needs to be sent. Only one column of `ball->omatrix` is used, reducing its contribution to $12\,\mathrm{B}$. Assuming the w component of `ball->position` and `eye_location` can assumed to be 1, only three floats each need to be sent. Presumably the modelview matrix will be sent anyway, so it would not be in the spirit of the problem to count it.

Taking these into account the amount of data is $12 + 12 + 12 + 4 = 40\,\mathrm{B}$.

Uniform variables could be used for `eye_location` and the modelview matrix since they are the same for every vertex in a rendering pass. The shading language compatibility profile automatically puts the modelview matrix in uniform variable `gl_ModelviewMatrix`.

Since `render_p` performs an entire rendering pass then uniform variables could be used for the other variables (`ball->position`, etc.) too. But if `render_p` did not stop and start the rendering pass, but instead that was done by code outside the `ball` loop, then the other variables would have to be shader inputs because they would be different at each iteration.

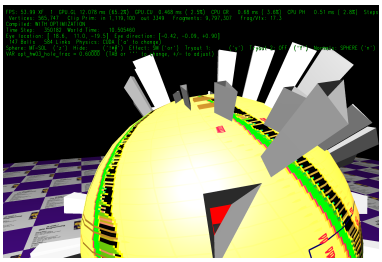
☑ Explain why the total amount of computation using this vertex shader would be larger.

In `render_prob1` the computation used to compute `xform` is done once per iteration (ball). But if `xform` were computed in the vertex shader then it would be done once per vertex, which is eight times as often for `render_p`.

☑ Explain why, even if the total computation were larger, execution time might still be faster.

Because the GPU is highly parallel so it can perform the computation more quickly. (Though in this case with only eight vertices in a rendering pass it won't make much of a difference.)

Problem 3: [30 pts] The shader code in routine `gs_monolith` below is based on the solution to Homework 3, in which the sphere was to be rendered with triangular holes.



(*a*) Modify the code so that there is a prism-shaped gray monolith over each hole. Render only the sides of the prisms, not the ends. A viewer looking down the axis of a prism should be able to see through the prism and into the sphere, perhaps to check whether it's full of stars. See the screenshot to the right. (Sorry, no stars.)

☑ Emit the prism faces. (The coordinates of one end of the prism are the same as the coordinates of the hole.)

☑ Determine the normal of each prism face, and emit the normal. ☑ All fragments of a face must have the same normal.

☑ Correctly set `max_vertices`. Don't set it larger than needed.

(*b*) The problem above could be solved using one triangle strip to render the prism, or using three triangle strips (one for each face). For the one-triangle-strip solution there would be a problem with the normals. Explain what that problem is, and how it might be fixed by modifying the fragment shader. (Not shown.)

☑ Problem with normals with a one-triangle-strip prism.

Normals are associated with vertices. In the prism shape rendered using a single triangle strip a vertex belongs to two faces of the prism. The normal can't be correct for both of them. The value of the normal seen by the fragment shader will be a blend of the normals provided at each edge.

☑ Key ideas (but not complete code) to fix normals by modifying fragment shader.

One solution would be to add two geometry shader outputs, `bool mono`, and `flat vec3 nflat_e`. When `mono` is false the fragment shader would use the usual normal, `normal_e`, but when `mono` is true it would use `nflat_e`. Because `nflat_e` is declared `flat` it won't be interpolated. Its value will be the one specified by the provoking (last) vertex of the triangle. The geometry shader can set `nflat_e` appropriately.

```
layout ( triangle_strip, max_vertices = 8 ) out;
void gs_monolith() {
  vec4 ctr_ce = AVG(vertex_e);       // Coord at triangle center.
  vec3 ctr_ne = AVG(normal_e);       // Normal at triangle center.
  vec2 ctr_tx = AVG(gl_TexCoord[0]); // Texture coord at triangle center.

  const float f = tryoutf.y; // Relative hole size.
  color = In[0].color;

  // Render the triangle-with-a-hole using a triangle strip that wraps
  // around the hole in the triangle.
  for ( int ii=0; ii<=3; ii++ ) {
      int i = ii % 3;  // Values of i:  0, 1, 2, 0.  (0 appears twice.)
      vec3 n_e = In[i].normal_e;

      // Prepare and emit inner vertex by blending outer vertex and center.
      normal_e =       f * n_e                  + (1-f) * ctr_ne;
      vertex_e =       f * In[i].vertex_e       + (1-f) * ctr_ce;
      gl_TexCoord[0] = f * In[i].gl_TexCoord[0] + (1-f) * ctr_tx;
      gl_Position = gl_ProjectionMatrix * vertex_e;
      EmitVertex();

      // Prepare and emit vertex of original triangle.
      normal_e = n_e;
      vertex_e = In[i].vertex_e;
      gl_TexCoord[0] = In[i].gl_TexCoord[0];
      gl_Position = In[i].gl_Position;
      EmitVertex();
    }
  EndPrimitive();

  color = vec4(0.5,0.5,0.5,1); // Change color to gray.
```

Solution code on next page.

```
    /// SOLUTION
    vec3 vtx_e[3], snorm_e[3];

    // Compute and save the coordinates of the top of the monolith.
    // Also save the surface normals.
    for ( int i=0; i<3; i++ ) {
        vec3 n_e = In[i].normal_e;
        snorm_e[i] = size * normalize(f * n_e + (1-f) * ctr_ne);
        vtx_e[i] = f * In[i].vertex_e.xyz + (1-f) * ctr_ce.xyz;
      }

    for ( int i=0; i<3; i++ ) {
        int ni = ( i + 1 ) % 3;  // Next i, possibly wrapped around.

        // Compute the normal of a face of the monolith.
        normal_e =
         normalize ( cross( In[ni].vertex_e.xyz - In[i].vertex_e.xyz, snorm_e[i].xyz ) );

        vertex_e.xyz = vtx_e[i];
        gl_Position = gl_ProjectionMatrix * vertex_e;
        EmitVertex();

        vertex_e.xyz = vtx_e[ni];
        gl_Position = gl_ProjectionMatrix * vertex_e;
        EmitVertex();

        vertex_e.xyz = vtx_e[i] + snorm_e[i];
        gl_Position = gl_ProjectionMatrix * vertex_e;
        EmitVertex();

        vertex_e.xyz = vtx_e[ni] + snorm_e[ni];
        gl_Position = gl_ProjectionMatrix * vertex_e;
        EmitVertex();

        EndPrimitive();
      }
}
```

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

(*a*) A program uses OpenGL and includes shader code. That program runs fine on a system using a GPU. Is it possible in principle to run that same code on a system without a GPU? The system still has a display which can show graphics. Explain why or why not.

☑ Can the program run on the system without a GPU?

☑ Explain why it can run or why it cannot ever run.

Yes it can run on the system without a GPU. That includes shader code, which can be compiled for the CPU just as easily as it can be compiled for a GPU. Though a GPU can execute many threads (or shader invocations) in parallel, that is not necessary for correctness. So the CPU could just have one thread compute everything.

(*b*) In class we described how vertices could be assigned different kind of colors (material properties) such as diffuse and specular using OpenGL calls such as `glMaterial`. From these a lighted color would be computed, accounting for the material properties and also light properties. Do current GPUs have special hardware to compute this lighted color? Explain.

☑ Do current GPUs (not just Ampere) have hardware to compute lighted colors? ☑ Explain.

No. Lighting would be computed by programmer-written shaders, or if the programmer did not provide shaders, in default shaders provided by the OpenGL implementation. There is no special hardware to compute lighting.

☑ Why was it necessary for OpenGL to have calls such as `glMaterial`, `glLight`, and so on?

Early GPUs were not programmable and had special hardware to compute lighting. The OpenGL lighting-related calls provided the data that this special hardware needed.

(*c*) Appearing below are three variations on a geometry shader. They are part of a rendering pipeline that also includes a fragment shader. The shader labeled `Original` is written clearly. A bored programmer came up with the `Stunt A` and `Stunt B` versions. Suppose that all variables are of type `vec4`, and that the names of the input variables accurately describe their contents. Explain whether each of the Stunt variations can be made to work by modifying the fragment shader.

```
void gs_typical() {  // Original
  for ( int i=0; i<3; i++ ) {
      gl_FrontColor = In[i].color;
      gl_BackColor = In[i].color;
      gl_Position = In[i].gl_Position;
      gl_TexCoord[0] = In[i].gl_TexCoord[0];
      EmitVertex();
    }
  EndPrimitive();
}

void gs_typical() { // Stunt A
  for ( int i=0; i<3; i++ ) {
      gl_FrontColor = In[i].color;
      gl_BackColor = In[i].gl_TexCoord[0];
      gl_Position = In[i].gl_Position;
      gl_TexCoord[0] = In[i].color;
      EmitVertex();
    }
  EndPrimitive();
}

void gs_typical() {  // Stunt B
  for ( int i=0; i<3; i++ ) {
      gl_FrontColor = In[i].color;
      gl_BackColor = In[i].gl_TexCoord[0];
      gl_Position = In[i].color;
      gl_TexCoord[0] = In[i].gl_Position;
      EmitVertex();
    }
  EndPrimitive();
}
```

☑ `Stunt A` ⊗ *can be made to work* or ◯ *cannot be made to work* .

☑ Explain why or why not.

It could only be made to work if only the back of every primitive faced the user. In that case the fragment shader would use `gl_Color` (which would get the value of `gl_BackColor` as the texture coordinate and `gl_TeXCoord` as the color. But it is not realistic to expect that.

☑ `Stunt B` ◯ *can be made to work* or ⊗ *cannot be made to work* .

☑ Explain why or why not.

It cannot be made to work because the rasterizer stage, which is between the geometry and fragment stages, uses `gl_Position`, and so it must be set correctly. In contrast, nothing between the geometry and fragment stages uses `gl_TeXCoord` nor `gl_BackColor` so swapping there values would go unnoticed.

(*d*) The code fragment below, taken from the classroom demo-7 code, updates the buffer object when `gpu_buffer_stale` is `true`. Variable `gpu_buffer_stale` is set to `true` initially, and again only when something about the sphere changes (for example, the number of triangles used to approximate it). The line commented `DISASTER` was added for this problem. Explain what will go wrong. *Hint: The problem occurs when* `gpu_buffer_stale` *is frequently true. Note: This problem also appeared on the 2019 final exam.*

```
if ( gpu_buffer_stale ) {
    gpu_buffer_stale = false;

    // Generate buffer id (name), if necessary.
    if ( !gpu_buffer ) glGenBuffers(1,&gpu_buffer); // ORIGINAL
    glGenBuffers(1,&gpu_buffer);                    // DISASTER

    // Tell GL that subsequent array pointers refer to this buffer.
    glBindBuffer(GL_ARRAY_BUFFER, gpu_buffer);

    // Copy data into buffer.
    glBufferData
      (GL_ARRAY_BUFFER,           // Kind of buffer object.
       coords_size*sizeof(pCoor), // Amount of data (bytes) to copy.
       sphere_coords.data(),      // Pointer to data to copy.
       GL_STATIC_DRAW);           // Hint about who, when, how accessed.

    // Tell GL that subsequent array pointers refer to host storage.
    glBindBuffer(GL_ARRAY_BUFFER, 0);
  }
```

☑ Explain the disaster that the `DISASTER` line causes.

Buffer leak. Each time block is executed a new buffer object is allocated and filled, but the old one is kept around and so memory or some other resource will eventually be used up.

(*e*) A vertex shader appears below, it is based on code from `links-shdr.cc` in the links demo and is used to render spheres using an instanced rendering pass. If there are 20 spheres in the scene and each sphere has 300 vertices, the vertex shader is called $20 \times 300 = 6000$ times. Label the variables in the routine below as follows:

☑ Label each OpenGL defined uniform variable with a U.

☑ Label each OpenGL defined input variable with an I.

☑ Label each OpenGL defined output variable with an O.

```
void vs_main_instances_sphere() {
  vec4 pos_rad = sphere_pos_rad[gl_InstanceID];
  float rad = pos_rad.w;
  mat4 rot = transpose(sphere_rot[gl_InstanceID]);
  vec4 normr = rot * gl_Vertex;
  normal_o = normr.xyz;
  vertex_o = vec4( pos_rad.xyz + rad * normal_o, 1 );

  gl_Position = gl_ModelViewProjectionMatrix * vertex_o;
  gl_TexCoord[0] = gl_MultiTexCoord0;
  color = sphere_color[gl_InstanceID];
}
```

(*f*) The storage type of `sphere_pos_rad` in `vs_main_instances_sphere` from the previous subproblem could be a shader input, a uniform variable, or a buffer object. Assume that all could be made to work correctly under some circumstances.

☑ Would it be ◯ *good* , ◯ *doable* , or ⊗ *bad* for `sphere_pos_rad` to be a shader input? ☑ Explain. If bad, how bad.

If the entire array were a vertex shader input that would likely be too large and even if it could fit it would be wasteful because the amount of data sent down the rendering pipeline would be the size of `sphere_pos_rad` **times** the number of vertices.

☑ Would it be ⊗ *good* , ◯ *doable* , or ◯ *bad* for `sphere_pos_rad` to be a uniform variable? ☑ Explain. If bad, how bad.

There is limited uniform space. If it would fit then it might work well. But, it would just be sent once per rendering pass, which is much better than sending it as a vertex shader input.

☑ Would it be ⊗ *good* , ◯ *doable* , or ◯ *bad* for `sphere_pos_rad` to be a buffer object? ☑ Explain. If bad, how bad.

This would work best because it could be any size and there would be less wasted data transfer.

Name Solution_____

GPU Programming
LSU EE 4702-1
Solve-Home Final Examination

Wednesday 9 December to Friday 11 December 2020 16:30 CST

Work on this exam alone. Regular class resources, such as notes, papers, solutions, documentation, and code, can be used to find solutions. In addition outside OpenGL references and tutorials, other programming resources, and mathematical references can be consulted. Do not try to seek out references that specifically answer any question here. **Do not discuss this exam with classmates or anyone else**, except questions or concerns about problems should be directed to Dr. Koppelman.

Warning: Unlike homework assignments collaboration is not allowed on exams. Suspected copying will be reported to the dean of students. The kind of copying on a homework assignment that would result in a comment like "See ee4702xx for grading comments" will be reported if it occurs on an exam. Please do not take advantage of pandemic-forced test conditions to cheat!

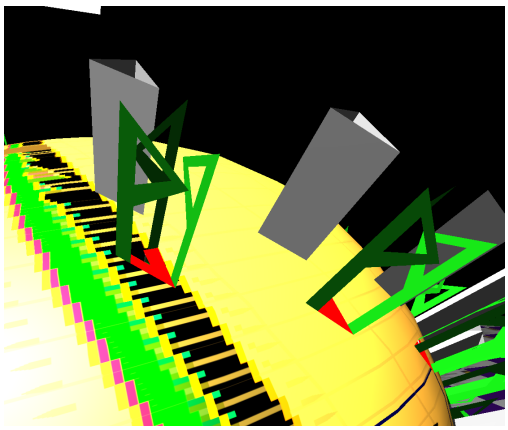
Problem 1	_____	(30 pts)
Problem 2	_____	(15 pts)
Problem 3	_____	(15 pts)
Problem 4	_____	(40 pts)
Exam Total	_____	(100 pts)

Good Luck!

Problem 1: [30 pts] The geometry shader below is based on the solution to the pre-final exam Problem 3. The geometry shader renders a prism centered on the triangle with a hole cut in it. The prism is also called a monolith in a playful reference to a recent news story. Notice that there is a `vec2` array, `pts`, declared in the shader. This array has the coordinates of the points on the pointy p from Pre-Final Exam Problem 1. The points are in the pointy p's local coordinate space and are in the right order to render the p as a triangle strip.

Modify the shader so that p's are rendered instead of monolith faces. Each p must be on the same plane as the monolith face and must be the same size. That is, the p can't extend outside where a face would be, the bottom, top and left edge must touch the respective edges of the face. The pointy part must touch or be near the right edge.

See the screenshot to the right. In the screenshot some holes have monoliths, and some have p's. In your solution just render p's.



☑ Modify the shader to render the p's where the monolith faces would go.

The solution is on the next page.

```
void gs_mono_p() {
  vec4 ctr_ce = AVG(vertex_e);       // Coord at triangle center.
  vec3 ctr_ne = AVG(normal_e);       // Normal at triangle center.

  // Render the triangle-with-a-hole using a triangle strip that wraps around hole.
  for ( int ii=0; ii<=3; ii++ ) {
      int i = ii % 3;
      normal_e =       f * In[i].normal_e       + (1-f) * ctr_ne;
      vertex_e =       f * In[i].vertex_e       + (1-f) * ctr_ce;
      gl_Position = gl_ProjectionMatrix * vertex_e;
      EmitVertex();     // Emit inner triangle vertex.
      normal_e = In[i].normal_e;
      vertex_e = In[i].vertex_e;
      gl_Position = In[i].gl_Position;
      EmitVertex();     // Emit original triangle vertex.
    }
  EndPrimitive();

  float size = length(In[0].vertex_e-In[1].vertex_e); // Height of monolith.
  color = vec4( 0.05, 0.5, 0.05, 1 );

  // Compute and save the coordinates of the top of the monolith, and surface normals.
  vec3 vtx_e[3], snorm_e[3];
  for ( int i=0; i<3; i++ ) {
      snorm_e[i] = size * normalize( f * In[i].normal_e + (1-f) * ctr_ne );
      vtx_e[i] = f * In[i].vertex_e.xyz + (1-f) * ctr_ce.xyz; }

  // Local x and y coordinates of the points on the pointy p.
  vec2 pts[] = { {0,0},{1,0},  {0,7},{1,6},  {4,5},{3,5},  {1,3},{1,4} };

  for ( int i=0; i<3; i++ ) {  // Iterate over faces of monolith.
      int ni = ( i + 1 ) % 3;  // Next i, possibly wrapped around.

      // Compute the normal of a face of the monolith.
      normal_e = normalize( cross( vtx_e[ni] - vtx_e[i], snorm_e[i].xyz ) );

      // Use a triangle strip to emit one face of the monolith.
      vec3 va[4] = { vtx_e[i], vtx_e[ni], vtx_e[i] + snorm_e[i], vtx_e[ni] + snorm_e[ni] };

      /// SOLUTION Below
      //  Compute local axes for drawing pointy p and scale them so that
      //  the pointy p fits on the monolith face.
      vec3 ax = ( vtx_e[ni] - vtx_e[i] ) / 4;
      vec3 ay = snorm_e[i] / 7;
      mat3 px = mat3(ax,ay,vec3(0));
      for ( int p=0; p<n_pts; p++ ) {  // Iterate over points describing p.
          vertex_e.xyz = vtx_e[i] + px * vec3(pts[p],0);
          gl_Position = gl_ProjectionMatrix * vertex_e;
          EmitVertex();
        }
      EndPrimitive();
      continue;
      /// SOLUTION Above
```

```
      for ( int j=0; j<4; j++ ) {
          vertex_e.xyz = va[j];
          gl_Position = gl_ProjectionMatrix * vertex_e;
          EmitVertex();
        }
      EndPrimitive();
    }
}
```

Problem 2: [15 pts] The vertex and geometry shaders below are used in rendering pipeline T, in which the input primitives are individual triangles, and rendering pipeline S, in which the input primitive is a triangle strip. In both cases the shaders work correctly, but there might be differences in performance.

(a) Modify the vertex and geometry shader to reduce the amount of data sent from the vertex shader to the geometry shader. Do so by moving some of the work performed by the vertex shader to the geometry shader. If necessary, declare new input and output variables. For this part the shaders will be run on pipeline S. Do not make changes that result in additional computation unless those changes also reduce vertex-to-geometry shader data transfer. Of course, your changes should not change what the shaders do.

☑ Modify shaders to reduce vertex-to-geometry shader data transfer.

☑ Don't make a change the has no impact on data transfer but does increase the amount of computation.

The solution appears below. Vertex output `gl_Position` is eliminated since that value can be computed in the geometry shader using `vertex_e`. There is certainly no need for both `gl_BackColor` and `gl_FrontColor` since they both carry the same value. The solution keeps `gl_BackColor`. Since `gl_FrontColor` and `gl_Position` are each of type `vec4` and a `vec4` is 16 bytes These changes reduce the vertex shader output data by 32 bytes.

```
void vs_main_basic() {  // The Vertex Shader
  // SOLUTION: REMOVE: gl_Position = gl_ModelViewProjectionMatrix * gl_Vertex;
  vertex_e = gl_ModelViewMatrix * gl_Vertex;
  normal_e = normalize(gl_NormalMatrix * gl_Normal);
  // SOLUTION: REPLACE: gl_BackColor = gl_FrontColor = gl_Color;
  gl_BackColor = gl_Color;  // SOLUTION. No need to send same color twice.
  tex_coord = gl_MultiTexCoord0.xy;
}
```

```
void gs_main_basic() {  // The Geometry Shader
  const bool type_a = glPrimitiveIDIn & 1;
  vec4 color_adjust = type_a ? vec4(0.5,0.5,0.5,1) : vec4(1);

  for ( int i=0; i<3; i++ )
    {
      gl_BackColor =  gl_BackColorIn[i] * color_adjust;
      // SOLUTION: REPLACE gl_FrontColor = gl_FrontColorIn[i] * color_adjust;
      gl_FrontColor = gl_BackColor; // SOLUTION: Just use back color.
      // SOLUTION: REPLACE gl_Position = gl_PositionIn[i];
      gl_Position = gl_ProjectionMatrix * In[i].vertex_e; // SOLUTION
      tex_coord = In[i].tex_coord;
      normal_e = In[i].normal_e;
      vertex_e = In[i].vertex_e;
      EmitVertex();
    }
  EndPrimitive();
}
```

(*b*) The requirement to *not make changes that result in additional computation unless those changes also reduce vertex-to-geometry shader data transfer* is much easier to comply with when the shaders are run on pipeline T. Explain why and include an example of such a change. The change should not change what the shaders do and should be useful. (That is, move something.)

☑ Why is it easier to avoid additional computation when the shaders are used in pipeline T than it is when the same shaders are used in pipeline S? ☑ Note: Don't compare the absolute performance of S to T, compare how much the change impacts computation on each pipeline.

It is easier to avoid additional computation in pipeline T because there is one execution of the `i`-loop body for each vertex shader invocation and so moving a computation from the vertex shader to the `i` loop in the geometry shader does not change the amount of computation (all other things being equal).

If the shaders were used with pipeline S then for each execution of the vertex shader (in all but two cases) there would be *three* executions of the `i`-loop body (each in a different geometry shader invocation).

☑ Provide an example.

For example, consider a rectangle rendered as 10 triangles. (Yes, 2 would suffice, but suppose there were 10.) In pipeline T the vertex shader would be executed $10 \times 3 = 30$ times and the geometry shader would be executed 10 times. In each execution of the geometry shader the `i` loop executes 3 iterations, and so the loop body executes a total of $10 \times 3 = 30$ times. In the solution to the previous part the a matrix/vector multiply is eliminated from the vertex shader and one is added to the `i` loop in the geometry shader. This move does not change the total number of matrix/vector multiplications.

If that same 10-triangle rectangle were rendered in S only $10 + 2 = 12$ vertices would be sent in to the rendering pipeline and so the vertex shader would execute only 12 times. The geometry shader would still execute 10 times and the `i` loop body would still execute 30 times. So moving the matrix/vector multiplication from the vertex shader to the geometry shader would increase the amount of work by a factor of $\frac{30}{12} = 2.5$.

So with pipeline S the modifications from the previous problem impact performance in two ways. The reduction in data being sent from the vertex to geometry shader will tend to improve performance but the increase in computation will tend to reduce performance. The 4×4 matrix/vector multiply consists of 16 multiply/add (MADD) operations. In current NVIDIA GPUs the time needed for the 16 MADD instructions is much less than the time needed to move 32 bytes across the GPU chip boundary, and so even with the additional computation the change would be worthwhile.

Problem 3: [15 pts] In Homework 3 we experimented with two ways to render a triangle with a hole in it. In the one-triangle method we relied on the fragment shader to render the hole. The geometry shader just emitted one triangle with little work. In the many-triangle method we emitted several triangles, forming a triangle-with-a-hole shape.

Based on performance measurements we found that the one-triangle method was faster. That must mean that the extra work done by the geometry shader in the many-triangle method had more of an impact than the extra work done by the fragment shader in the one-triangle method.

Let t_{g1} denote the time used by one invocation of the geometry shader for the one-triangle method, let t_{gm} denote the time used by one invocation of the geometry shader for the many-triangles method. Let t_{f1} and t_{fm} denote the times for one invocation of the respective fragment shaders.

Let n denote the number of spheres rendered, and let g denote the number of triangles in one sphere tessellation. Finally, let f denote the fraction of the triangle covered by the hole.

(a) Based on these, find an expression for the total time used by the geometry shaders during a render pass for each method.

☑ Total rendering pass time for geometry shader using one-triangle method:

Each invocation takes t_{g1}, and a there is one invocation for each tessellated triangle, so the total time is ngt_{g1}.

☑ Total rendering pass time for geometry shader using many-triangle method:

Each invocation takes t_{gm}, and a there is one invocation for each tessellated triangle, so the total time is ngt_{gm}.

(b) Find an expression for the time used by the fragment shaders. Use n_{f1} for the total number of fragment shader invocations in the one-triangle method. (But use it for both expressions.)

☑ Total rendering pass time for fragment shader using one-triangle method:

This is simply $n_{f1}t_{f1}$.

☑ Total rendering pass time for fragment shader using many-triangle method:

Since no n_{fm} has been provided it must be estimated: $n_{fm} = fn_{f1}$. So the total time is $fn_{f1}t_{fm}$.

(c) What does n_{f1} depend on? How can n_{f1} be made larger or smaller when viewing a scene without changing the primitives sent into the rendering pipeline. That is, one can't send fewer spheres or more spheres into the rendering pipeline.

☑ Quantity n_{f1} depends on:

It depends on the number of pixels covered by the spheres.

☑ It can be changed when viewing a scene by:

... by moving the eye further from the scene to reduce n_{f1} or closer to the scene to increase n_{f1}.

Problem 4: [40 pts] Answer each question below.

(*a*) The two shaders below do the same thing, though slightly differently.

```
void vs_plan_a() {
  vertex_e = gl_ModelViewMatrix * gl_Vertex;
  gl_Position = gl_ProjectionMatrix * vertex_e;
}

void vs_plan_b() {
  vertex_e = gl_ModelViewMatrix * gl_Vertex;
  gl_Position = gl_ModelViewProjectionMatrix * gl_Vertex;
}
```

OpenGL is provided a modelview matrix and a projection matrix at the beginning of a rendering pass. In both `vs_plan_a` and `vs_plan_b` there are two matrix/vector multiplies, which require $4^2 = 16$ multiply/add operations each. But `vs_plan_b` uses `gl_ModelViewProjectionMatrix`, which is the product of the modelview and projection matrices. The product of these two matrices is computed using $4^3 = 64$ multiply/add operations. That brings the total to $16 + 16 + 64 = 96$ operations, much more than 32 for `vs_plan_a`, right?

☑ Describe the flaw with this argument.

The product is computed before the rendering pass, and it is computed at most once per rendering pass. (That is because it is a uniform variable.) A rendering pass is expected to have a large number of vertex shader invocations. Suppose there are 1000 invocations. So the $4^3 = 64$ operations needed to compute the product is tiny compared to the computation performed by the vertex shaders, $32,000$ operations.

☑ Describe a case when the argument is correct, but explain why this case does not reflect typical use.

It would be correct if a rendering pass processed just one vertex.

(*b*) Answer the following questions about view volumes.

☑ What is a view volume?

It is the part of the scene that is visible. In OpenGL it is the part of the scene inside of a cube from clip-space coordinate (-1,-1,-1) to (1,1,1).

☑ It is easy to determine whether a vertex is in the view volume by using its coordinate in ◯ *object space*, ◯ *eye space*, or ◉ *clip space* (check one).

☑ Given the coordinate in that space, how can one tell whether it is inside or outside the view volume?

It is in clip space if the absolute value of each component is ≤ 1 after homogenization.

☑ It is easy to determine whether some triangles are in the view volume. ☑ Provide an example of such a triangle and ☑ explain why.

It is easy if all three vertices are in the view volume. In that case no part of the triangle can be outside the view volume.

☑ Provide an example of a triangle for which it is not so easy to determine if it is in the view volume. ☑ Illustrate with a diagram.

One in which all three vertices are outside the view volume. FINISH.

(*c*) Describe how suitable an OpenGL uniform variable is for the following:

☑ Explain whether this a suitable use for a uniform variable: To hold the lighted color computed by a vertex shader.

That won't work because uniform variables cannot be written by shaders, including vertex shaders. Even if the uniform were written by some other means, each vertex can have a different lighted color but the value of a uniform variable must be the same for every vertex in a rendering pass.

☑ Explain whether this a suitable use for a uniform variable: To hold the location of a light source.

That is suitable because that would be the same for every vertex.

(*d*) Vertex coordinates are usually three dimensional but texture coordinates are usually two dimensional. Why? (Ignore the w component in your answer.)

☑ Texture coordinates have two, not three, dimensions because:

Because textures are mapped on to triangles, which are two dimensional.

(*e*) In many of our sphere examples we put the coordinates into a buffer object.

☑ What are the advantages of a buffer object over using individual `glVertex` calls to feed a rendering pipeline?

Much lower overhead compared to calling `glVertex` to provide one vertex coordinate.

☑ What are the advantages of a buffer object over using a client (CPU) array to feed a rendering pipeline?

With a client array the data must be sent from the GPU to the CPU for each rendering pass, even if that data hasn't changed. A buffer object can be reused.

(*f*) A homogeneous coordinate consists of four components, compared to just three for ordinary Cartesian coordinates. Homogeneous coordinates increase the amount of work needed for a matrix/vector multiply from 9 to 16 multiplications. Transformations are realized by multiplying a transformation matrix by a coordinate.

☑ Describe a transformation that cannot be done without homogeneous coordinates.

Translation.

☑ Describe a transformation that can be done using ordinary Cartesian coordinates.

Scale. Also rotation.

23 Fall 2019 Solutions

Name Solution

GPU Programming
EE 4702-1
Take-Home Pre-Final Examination

Due: 5 December 2019 at 16:30

Work on this exam alone. Regular class resources, such as notes, papers, solutions, documentation, and code, can be used to find solutions. In addition outside OpenGL references and tutorials, other programming resources, and mathematical references can be consulted. Do not try to seek out references that specifically answer any question here. **Do not discuss this exam with classmates or anyone else**, except questions or concerns about problems should be directed to Dr. Koppelman.

Problem 1	_____	(30 pts)
Problem 2	_____	(25 pts)
Problem 3	_____	(5 pts)
Problem 4	_____	(25 pts)
Problem 5	_____	(15 pts)
Exam Total	_____	(100 pts)

Alias What's next?

Good Luck!

Problem 1: [30 pts] The `gs_points` shader in the solution to Homework 3, Problem 2 renders protrusions, which are blocks attached to the ring. The shader always emits five faces: the top (which we called the diamond), and the four sides. (Since it is attached to the ring there is no need to emit a bottom face.) Normally only the outside of the block is visible so the user can see at most three faces. That means at least three faces that can't be seen are emitted nevertheless. That can't be good. In this problem shader code will be modified to avoid those faces, and in the next problem the implications of this change will be analyzed.

The code on the next page is based on (but not identical to) the Homework 3 Problem 2 solution. The intention is that this be solved by hand, though solving by editing code is allowed. The code from the next page will be provided on request or may already be available.

(*a*) In the code below (meaning the next page) there is a 4-element Boolean array `visible` that is declared but not used. Add code that sets `visible[i]` to `true` is side `i` is visible to the user. By no coincidence the declaration of `visible` is just after code that sets `face_normal_e[i]` to the eye-space normal of side `i`. *Hint: Remember that in eye space the eye is at the origin (coordinate* $(0, 0, 0)$*). Also take advantage of the fact that eye-space coordinates and normals are available. Another Hint: A correct solution to this problem consists of a loop with a single statement,* `visible[i]=.x.`*, with* `.x.` *replaced by [am I giving away too much] a dot product.*

☑ Modify code so `visible[i]` set to `true` iff face `i` is visible.

(*b*) Modify the code below so that only the visible faces are emitted. This includes the diamond on top and the four sides of the protrusion. Feel free to use the `visible` array from the previous subpart (even is that subpart has not been solved). *Hint: Only one line is needed for the diamond on top. The changes needed to the code emitting the sides are more extensive.*

☑ Modify code so that diamond is not emitted if it is not visible.

☑ Modify code so that only visible sides are emitted.

Code is on the next page.

```
void gs_points() {
  /// Note: (Code setting ctop, etc. not shown.)
  // Prepare an array of object-space coordinates of vertices describing the
  // protrusion, and use that to initialize an array of eye-space coordinates.
  vec3 pts_o[8] =
    { ctop,       cr,       cbot,       cl,   ctop+nc*dr, cr+nr*dr, cbot+nc*dr, cl+nr*dr };
  vec4 pts_e[8];
  for ( int i=0; i<8; i++ ) pts_e[i] = gl_ModelViewMatrix * vec4(pts_o[i],1);
  int ord[4] = { 1, 0, 2, 3 }; // Order in which to emit vertices.

  // Emit the diamond-shaped protrusion top.
  color = color_diamond;
  normal_e = gl_NormalMatrix * nl;

  if ( dot( normal_e, pts_e[0].xyz ) < 0 ) // SOLUTION -- Part b (the if statement)
  for ( int i=0; i<4; i++ ) {
      vertex_e = pts_e[ord[i]];  // Retrieve vertices in the correct order.
      gl_Position = gl_ProjectionMatrix * vertex_e;
      EmitVertex();  }
  EndPrimitive();

  // Compute the eye-space normal of each protrusion face.
  vec3 face_normal_e[4];
  for ( int i0=0; i0<4; i0++ )
    face_normal_e[i0] = cross( pts_e[(i0+1) & 0x3].xyz - pts_e[i0].xyz,
                               pts_e[i0+4].xyz - pts_e[i0].xyz );

  bool visible[4];
  // SOLUTION -- Part a.
  for ( int i=0; i<4; i++ ) visible[i] = dot( face_normal_e[i], pts_e[i].xyz ) < 0;

  // SOLUTION -- Part b: Find the first visible side, set that to istart.
  int istart = -1;
  for ( int i=0; i<5; i++ ) if ( !visible[i&0x3] && visible[(i+1)&0x3] ) istart = i+1;
  if ( istart == -1 ) return;

  color = color_edge; // Color for sides.
  for ( int i=0; i<5; i++ ) {
      // int i0 = i & 0x3;          // The current edge.
      int i0 = ( i + istart ) & 0x3; // SOLUTION -- Part b: Start at istart.

      vertex_e = pts_e[i0+4];  gl_Position = gl_ProjectionMatrix * vertex_e;
      EmitVertex();
      vertex_e = pts_e[i0];    gl_Position = gl_ProjectionMatrix * vertex_e;
      EmitVertex();
      normal_e = face_normal_e[i0]; // This normal will be used for the next two vertices.

      if ( !visible[i0] ) break; // SOLUTION -- Part b.
    }
 EndPrimitive();
}
```

Problem 2: [25 pts] Assume that the previous problem has been solved correctly and so only those sides of the protrusions facing the user are emitted. Below, roughly estimate the benefit of this improvement when rendering protrusions by considering the amount of work done by the vertex, geometry, and fragment shaders. Make the estimate using the original (Homework 3 solution) vertex and fragment shaders, and the geometry shader obtained by solving the previous problem. When estimating, assume that the number of emitted faces per protrusion drops from 5 to 2.5. (The actual number of faces emitted for a particular protrusion with a particular eye location and orientation can range from zero [for example, if the viewer is in the center of the ring looking at some point on the ring] to three.)

When answering the problems below consider the change in the amount of work done by each shader (the change in the amount of work using the Homework 3 solution code to the amount of work using the geometry shader from the previous problem) invocation, and the change in the number of times the shader needs to be invoked.

(*a*) For this part assume a typical view of the ring. (Don't assume that no part of the ring is visible, or other special cases.)

☑ Compared with the Homework 3 solution code, the total work for the vertex shader stage after the change above is: ◯ *about half the amount of work,* ⊗ *no change in the amount of work,* ◯ *roughly the same amount of work,* ◯ *double the amount of work.*

☑ Explain. Consider ☑ work per shader and ☑ number of shader invocations.

The vertex shader is invoked once for each protrusion. The number of protrusions has not changed and the vertex shader has not changed so there is no change in the amount of work.

☑ Compared with the Homework 3 solution code, the total work for the geometry shader stage after the change above is: ◯ *about half the amount of work,* ◯ *no change in the amount of work,* ⊗ *roughly the same amount of work,* ◯ *double the amount of work.*

☑ Explain. Consider ☑ work per shader and ☑ number of shader invocations.

The geometry shader is invoked once for each protrusion. The number of protrusions has not changed but the geometry shader sure has. The main changes to the code are: computing the visibility of each face, determining the first side to render (value of `iside`), and skipping faces that do not need to be rendered. The first two increase the amount of work and the last reduces the amount of work. The geometry shader now has to compute visibility for each face, but that's just an additional five dot products. Finding the first side to render requires a five iteration loop with a simple loop body.

The number of faces emitted drops from 5 to 2.5. The code emitting a face performs two (the sides) to four (the diamond) 4×4 matrix/vector multiplications (the coordinate transformations), each requiring $4 \times 4 = 16$ multiply/add operations, in contrast to 3 multiply/add operations for a dot product of a 3-element vector. Therefor avoiding just one transformation more than makes up for the five dot products. The original code performed $8 + 4 + 5 \times 2 = 22$ transformations, the new code roughly performs $8 + \frac{1}{2}4 + 5 \times 2 = 15$ transformations, which is more than half the amount of work. When the trigonometric functions are taken into account the amount of work is about the same.

Grading Note: An answer of half the work with a justification similar to the one above would receive full credit.

☑ Compared with the Homework 3 solution code, the total work for the fragment shader stage after the change above is: ⊗ *about half the amount of work,* ◯ *no change in the amount of work,* ◯ *roughly the same amount of work,* ◯ *double the amount of work.*

☑ Explain. Consider ☑ work per shader and ☑ number of shader invocations.

The fragment shader has not been changed so the work per invocation is about the same. If half the number of faces are emitted then there will be half the number of fragment shader invocations. Assuming that the projected size of the visible faces is about the same as the invisible ones, the amount of work performed by the fragment shader will be about half.

(*b*) Suppose again that the ring is visible, and suppose that each invocation of the new geometry shader requires 10% more work than the geometry shader from the Homework 3 solution. (The extra time is needed to detect invisible faces.) What sort of view would make that extra work increase the total time needed for a rendering pass, and would make the extra work offset by a large reduction elsewhere?

☑ Characteristics of a view that would increase rendering time even with invisible faces omitted?

A view in which the projected size of the protrusions are small, say about three fragments each. A view like that can be obtained by moving the eye far away from the ring while still looking at it. In such a view for each geometry shader invocation there will be three fragment shader invocations. Let t_g be the amount of work done by an invocation of the original geometry shader and let t_f be the amount of work done for an invocation of the fragment shader (which hasn't changed). An invocation of the new geometry shader performs $1.1t_g$ work. Suppose that the original geometry shader emits six fragments for some view, and that the new one emits just three. The amount of work performed by the old one is $t_g + 6t_f$ and for the new one $1.1t_g + 3t_f$. For the rendering time to be increased $1.1t_g + 3t_f > t_g + 6t_f$, for the work to be equal $1.1t_g + 3t_f = t_g + 6t_f$, or $t_g = 30t_f$. So if the original geometry shader performs more than $30\times$ the work of the fragment shader, execution time will be increased with our new shader for views in which a protrusion covers just three fragments.

☑ Characteristics of a view that would sharply reduce rendering time with invisible faces omitted?

A view in which the projected size of the protrusions is large and so there are lots of fragments per primitive, say 1000 per primitive. In that case $1.1t_g + 1000t_f < t_g + 2000t_f$ will hold unless one geometry shader invocation does $10,000\times$ the work of a fragment shader invocation, which isn't very likely.

Problem 3: [5 pts] Appearing below is a correct solution to Homework 3 Problem 1, followed by a possible alternative solution. Indicate whether the alternative solution will render the strip as intended (the same way as the first solution) and whether there is any problem with it.

```
void gs_strip_plus() {  // Homework 3 Solution
  if ( In[2].normal_e == vec3(0,0,0) ) return;
  for ( int i=0; i<3; i++ )
    {
      normal_e = In[i].normal_e;
      vertex_e = In[i].vertex_e;
      color = In[i].color;
      gl_Position = In[i].gl_Position;
      EmitVertex();
    }
  EndPrimitive();
}

void gs_strip_plus() {  // A possible alternative solution.
  for ( int i=0; i<3; i++ )
    {
      if ( In[2].normal_e == vec3(0,0,0) ) return;
      normal_e = In[i].normal_e;
      vertex_e = In[i].vertex_e;
      color = In[i].color;
      gl_Position = In[i].gl_Position;
      EmitVertex();
    }
  EndPrimitive();
}
```

☑ Will the alternative solution render the strip correctly?

Yes.

☑ Explain the problem with the alternative solution.

The alternative solution will emit two vertices that will then be abandoned, wasting effort.

Problem 4: [25 pts] Answer each question below.

(*a*) Any uniform variable that a vertex shader uses can be replaced by a vertex shader input. Why is that not a good idea? Explain the impact on performance.

☑ Uniform variables should not be replaced by shader inputs because:

... because the driver will have no choice but to transfer v copies of each uniform, where v is the number of vertices. That will slow down execution. Also, uniform variables use constant memory, and so can be accessed faster.

☑ How might one estimate the impact on performance for such a replacement?

Let u denote the size of the uniform variables to convert. For example, a 4×4 float transformation matrix consumes $4\times4\times4 = 64\,\mathrm{B}$, so if 2 transformation matrices are converted from uniforms to vertex shader inputs $u = 2\times64 = 128\,\mathrm{B}$. Let v denote the number of vertices in a rendering pass and Θ denote the CPU to GPU data bandwidth, for example, $\Theta = 500\,\mathrm{GB/s}$. Then the additional time per rendering pass would be $\frac{uv}{\Theta}$.

The two matrices in a 5000-vertex rendering pass on the $\Theta = 500\,\mathrm{GB/s}$ device would add $\frac{128\,\mathrm{B}\times5000}{500\,\mathrm{GB/s}} = 1.28\,\mu\mathrm{s}$ to the pass if sourced from the CPU.

(*b*) In some cases a vertex shader input can be replaced by a uniform variable. Describe a situation in which a vertex shader input can be replaced by a uniform, and in which a shader input cannot be replaced by a uniform. An example can be found in the shaders for Homework 3 and 4.

☑ Situation where vertex shader input can be replaced by a uniform. ☑ Explain.

Answer: In the Homework 3 and 4 code all of the diamonds were the same color. So the color could have been assigned to a uniform.

Discussion: Instead it was provided using `glColor`. At least for the `glBegin/glEnd` rendering passes the driver had no way of knowing in advance that `glColor` would not be changed during the rendering pass and so it would provide it as a vertex shader input. For the `glDraw` rendering passes the driver could determine that the color would not change but it would need to recompile our programmable shaders to take advantage of that fact.

☑ Situation where vertex shader input **can not** be replaced by a uniform. ☑ Explain.

When the value won't be the same for every vertex. For example, the vertex normal in most of our examples. Since a uniform variable cannot be changed during a rendering pass it cannot be used for values that change.

(*c*) The geometry shader must write clip-space coordinates to output variable `gl_Position`. Anything else the geometry shader needs to write, such as a surface normal, can be written to variables of the programmer's choosing. Why does OGSL require clip-space coordinates in that way?

☑ Why must the geometry shader write clip-space coordinates?

☑ Why must it write them to a variable with a specific name, `gl_Position`?

In the rendering pipeline the rasterization stage follows the geometry shader. The rasterization stage assembles a primitive (using vertices emitted by the geometry or vertex shader) and then finds the pixels covered by that primitive. For each primitive it emits a fragment. To find the pixels covered by the primitive it needs clip space coordinates (which can be easily converted into pixel coordinates), which is why the geometry shader needs to emit clip space coordinates.

The geometry and fragment shaders (as well as vertex, tessellation, and compute shaders) are written by programmers using the OpenGL API. In contrast the rasterization stage is part of the fixed functionality, meaning it cannot be programmed, at least by OpenGL/OpenGL Shading Language users.

Let's suppose that the geometry shader emits coordinates (which it has to) and a color (which it "wants to") for each vertex. The color is read by the fragment shader which will do something with it, probably compute a lighted color. The geometry and fragment shaders were written by the same person or team, and so they will have agreed on a name for the variable holding the color, say `coler`. The rasterization stage only sees `coler` as a four-element vector, but has no idea what it's for and so the name is not important.

In contrast, the rasterization stage needs clip-space coordinates and so those must be written to a variable with a specific name.

(*d*) The true sphere shader presented in class rendered a sphere perfectly. Why would there be no advantage in a true cube shader?

☑ No advantage to a true cube shader because:

The true sphere shader renders a perfect sphere, meaning that exactly the pixels covered by the sphere are written. In contrast, when a sphere is approximated using triangles (the first method presented in class) the outline of the sphere will be a polygon rather than a circle unless the a very large number of triangles are used.

There is no similar problem when rendering a cube. A cube can be modeled using 12 triangles, and they will perfectly match the surface of a cube (if done correctly). Cube normals are easy too, there are only six of them. Since a cube can be perfectly rendered using triangles, there is no need for a true cube fragment shader.

(*e*) The shadow volume code and our lighting code took advantage of built in variables describing how user code set lighting state such as
`gl_LightSource[i].position`, which provides the position of light `i`. Light position and other lighting variables are deprecated, meaning they may be removed in future versions of OpenGL.

☑ Why are they deprecated?

The are deprecated because lighting can easily be computed by vertex or fragment shaders. Those shaders can declare their own uniform variables for light positions and whatever else they need, and use those to compute lighted colors.

☑ Why were they there in the first place?

OpenGL is intended to provide a uniform API that can be used on a variety of 3D graphics accelerators.

Early graphics accelerators were not programmable (one could not write shaders) and so one had no choice but to use built-in lighting functionality. Execution of an OpenGL command such as `glLightfv(GL_LIGHT0, GL_POSITION, light_location)` might write a special hardware *register* (actually a memory location that the hardware monitors) with the coordinates on one device, but use a totally different method to set position on another device. The device might have had specialized hardware to compute lighting or might have been programmable, but only by the manufacturer.

GPUs became programmable, and over time their instruction sets became more general purpose, making specialized hardware for lighting and other graphics tasks unnecessary.

Problem 5: [15 pts] Many of our fragment shaders make library function calls such as `texture(tex_unit_0,TexCoord)` to obtain a filtered texel located at coordinate `TexCoord`. Coordinate `TexCoord` is something the shader code provided. Consider an alternative method of obtaining a texel in which the user prepares an array of texels bound to a buffer object named `my_texture`. The fragment shader might obtain the texel using code such as `my_texel[twid*TexCoord.x + twid*tht*TexCoord.y]`, where `twid` is the width of the texture image and `tht` is the height of the image.

(*a*) Explain why the code fragment `my_texel[twid*TexCoord.x + twid*tht*TexCoord.y]` might return a texel, but not a filtered texel. In particular, what would need to be done for it to be filtered, and why could it not provide the benefit of mipmap levels. Remember that `my_texel` is an ordinary array of RGBA values from a texture image (called texels). Each element is a `vec4`, the `vec4` members are `.r`, `.g`, `.b`, and `.a`.

☑ `my_texel[twid*TexCoord.x + twid*tht*TexCoord.y]` could not return a filtered texel because:

To compute a filtered texel one needs to take some kind of a weighted average of texels that cover a pixel. A simple array lookup will just return one texel. There's no way to filter in advance since one can't usually predict pixel size and placement in advance, and there would be way too many possibilities anyway.

☑ Roughly, how could code accessing the texel array `my_texel` return a filtered texel?

The code accessing `my_texel` could find the indices (values of `TexCoord.x` and `TexCoord.y`) of all texels covering the pixel of interest, retrieve those, and then compute a weighted average of them. (Some texels would only partially cover the pixel.)

☑ Explain why accessing the single texel array could not provide the benefit of multiple mipmap levels.

Because each MIPMAP level would need to be a different array, at least if a single index were used.

(*b*) The library function `texture` is part of the OpenGL shading language. Unlike `gl_LightSource`, `texture` is not deprecated, meaning that it won't be removed from OpenGL anytime soon. Why is it that lighting-related features are deprecated while texture accesses and filtering are not deprecated?

☑ The `texture` function and most other texture-related features are not deprecated because:

. . . because texel filtering is compute intensive. GPUs have specialized hardware, texture units, for filtering texels, and so driver-specific routines are needed. If a user were to write code that made use of the texture units, that code would have to be specific to a particular GPU or would have to be written for all of the GPUs that the code might run on. Using the `texture` function is far more convenient.

24 Fall 2018 Solutions

Name Solution

GPU Programming
EE 4702-1
Take-Home Pre-Final Examination

Due: 30 November 2018 at 16:30

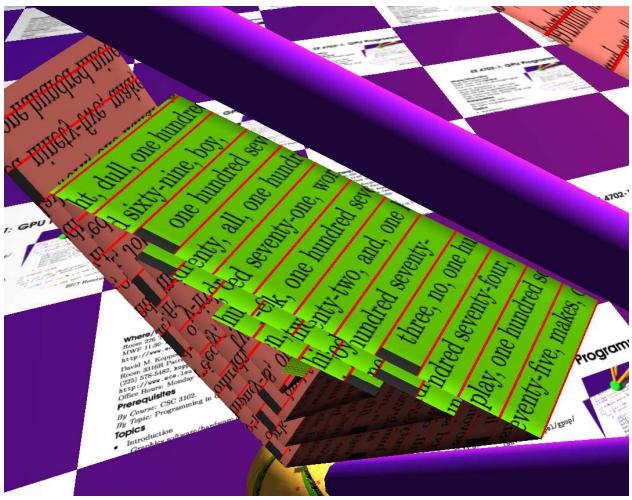
Work on this exam alone. Regular class resources, such as notes, papers, solutions, documentation, and code, can be used to find solutions. In addition outside OpenGL references and tutorials, other programming resources, and mathematical references can be consulted. Do not try to seek out references that specifically answer any question here. **Do not discuss this exam with classmates or anyone else**, except questions or concerns about problems should be directed to Dr. Koppelman.

Problem 1	_____	(33 pts)
Problem 2	_____	(34 pts)
Problem 3	_____	(33 pts)
Exam Total	_____	(100 pts)

Alias Think of something, quick!

Good Luck!

Problem 1: [33 pts] Appearing on the next page is the fragment shader code taken from the solution to Homework 3, in which a texture is applied to the triangular spiral in such a way that it looks like the texture image was cut into strips and glued together.



(*a*) Modify the shader code so that the text is rotated by 90°, as shown in the screenshot above. The text aspect ratio must be consistent along segments. As with the homework, the text must be readable without skipped or repeated sections. **Do not** assume or make any changes to the host code. In particular, the fragment shader should work with the same primitives as emitted for the Homework 3 code.

(*b*) Modify the shader code so that red lines are drawn separating the lines of text.

It is okay to try out your solution on the Homework 3 package, but put the solution in this exam. That is, the solution **will not** be collected by the TA-bot.

See the Homework 3 solution for detailed comments explaining the shader code.

The solution has been checked into the repo with the Homework 3 code. The shader code is in file `pre-fin-shder.cc` and the code can be run by building and executing `pre-fin`.

Use next page for solution.

☑ Modify shader so text rotated 90°.

☑ Draw red lines between groups of text.

☑ Text aspect ratio and size must be consistent on all segments and ☑ there must be no gaps or skipped sections.

See next page for solution.

```
void fs_main_hw03() {
  float line_num = floor(tex_coord.x);
  vec2 tc = vec2( tex_coord.x, ( line_num + tex_coord.y ) * tex_ht );
```

```
  vec4 color =
   gl_FrontFacing ? gl_FrontMaterial.diffuse : gl_BackMaterial.diffuse;
```

```
  // No need to modify code below.
  vec4 texel = texture(tex_unit_0,tc);
  vec3 nne = normalize(normal_e);
  float edge_dist = tex_coord.y;
  vec3 bnorm = tryout.x ? nne : edge_dist > 0.9
    ? normalize(mix(nne,spiral_normal,2*(edge_dist-0.9))) : edge_dist < 0.1
    ? normalize(mix(-spiral_normal,nne,0.8+2*edge_dist)) : nne;
  vec4 lighted_color =
    generic_lighting( vertex_e, color, bnorm, gl_FrontFacing );
  gl_FragColor = texel * lighted_color;
  gl_FragDepth = gl_FragCoord.z;
}
```

Solution appears below. A common mistake was not continuing a line.

```
void
fs_main_hw03()
{
  /// Pre-Final Problem 1 SOLUTION

  int ncol = 6;     // Number of pieces a line is split into.
  int nlines = 60;  // Number of lines per page.

  float s_tex_coord_x = tex_coord.x * tex_ht * nlines;
  float s_line_num = floor(s_tex_coord_x);
  float s_line_offs_01 = fract(s_tex_coord_x);
  float s_line_offs_cols = s_line_offs_01 * ncol;
  float s_col_num = floor(s_line_offs_cols);
  float s_col_offs = fract(s_line_offs_cols);
  float tex_x = ( s_col_num + tex_coord.y ) / ncol;
  float tex_y = ( s_line_num + s_col_offs ) / nlines;

  vec2 tc = vec2(tex_x,tex_y);

  vec4 color =
    tex_x < 0.01 ? vec4(0.1,0.1,0.1,1) :
    s_col_offs < 0.03  s_col_offs > 0.97 ? vec4(1,0,0,1) :
    gl_FrontFacing
    ? gl_FrontMaterial.diffuse : gl_BackMaterial.diffuse;


  // No need to modify code below.
  vec4 texel = texture(tex_unit_0,tc);
  vec3 nne = normalize(normal_e);
  float edge_dist = tex_coord.y;
  vec3 bnorm = tryout.x ? nne : edge_dist > 0.9
    ? normalize(mix(nne,spiral_normal,2*(edge_dist-0.9))) : edge_dist < 0.1
    ? normalize(mix(-spiral_normal,nne,0.8+2*edge_dist)) : nne;
  vec4 lighted_color =
    generic_lighting( vertex_e, color, bnorm, gl_FrontFacing );
  gl_FragColor = texel * lighted_color;
  gl_FragDepth = gl_FragCoord.z;
}
```

Problem 2: [34 pts] Recall that the true sphere code covered in class emits a square (two triangles) for each sphere to be rendered. The square is chosen so that it perfectly frames the sphere as seen by the viewer. The square is perfect in the sense that if it were made any smaller parts of the sphere would not be visible. But it is still wasteful because the sphere is not visible in the corners of the square and so those fragment shader invocations are wasted.

(*a*) Modify the true-sphere geometry shader below so that it emits a $2s$-sided polygon rather than a square. Show the code for computing the polygon coordinates, **do not** use the assumed functions from the next problem. Note that when $s = 2$ the code should emit a bounding square, which is what the original code does.

Use next page for solution.

☑ Modify code to emit $2s$-sided bounding polygon. ☑ Take advantage of the fact that $2s$ is an even number.

See next page for solution.

```
const int s = 8;     // Don't assume that s is always 8!!
layout ( points ) in;
layout ( triangle_strip, max_vertices = 4 ) out;

void gs_main() {
  vertex_id = In[0].vertex_id;
  mat4 mvp = trans_proj * gl_ModelViewMatrix;
  vec4 pos_rad = sphere_pos_rad[vertex_id];
  vec3 ctr_o = pos_rad.xyz;
  float r = pos_rad.w;
  vec3 e_o = vec3(gl_ModelViewMatrixInverse*vec4(0,0,0,1));  // Eye location in object space.

  vec3 ec_o = ctr_o - e_o;  // Eye to Center (of sphere).
  float ec_len = length(ec_o);
  vec3 ne = ec_o / ec_len;

  // Find vectors orthogonal to ec_o.
  vec3 atr_o = abs(ec_o);
  int min_idx = atr_o.x < atr_o.y ? ( atr_o.x < atr_o.z ? 0 : 2 )
    : ( atr_o.y < atr_o.z ? 1 : 2 );
  vec3 nx_raw = min_idx == 0 ? vec3(0,-ec_o.z,ec_o.y)
    : min_idx == 1 ? vec3(-ec_o.z,0,ec_o.x) : vec3(-ec_o.y,ec_o.x,0);
  vec3 nx = normalize(nx_raw),  ny = cross(ne,nx);

  // Compute center of the limb, lmb_o. (The limb is the sphere outline.)
  float sin_theta = r / ec_len;
  float a = r * sin_theta;
  vec3 lmb_o = ctr_o - a * ne;

  // Compute axes to draw the limb.
  float b = r * sqrt( 1 - sin_theta * sin_theta );
  vec3 ax = b * nx,  ay = b * ny;

  // Emit a bounding square for the limb.  --- Solution should start here.
  for ( int i = -1; i < 2; i += 2 ) for ( int j = -1; j < 2; j += 2 ) {
        vertex_o = lmb_o + ax * i + ay * j;
        gl_Position = mvp * vec4(vertex_o,1);
        EmitVertex();  }
  EndPrimitive();




}
```

Solution appears below. A single triangle strip is used for the entire polygon taking advantage of the fact that the number of sides is even. First, a bounding circle for the polygon is found. A loop iterates over an angle effectively from $\theta = \pi/(4s)$ to $\theta = \pi(1 - 1/4s)$. Coordinates of points on the circle at both $-\theta$ and θ are computed and emitted. In this way only $2s$ vertices need to be emitted. A brute-force approach would emit $2s$ individual triangles (pie slices) of 3 vertices each.

```
void gs_main()
{
  vertex_id = In[0].vertex_id;
  vec4 pos_rad = sphere_pos_rad[vertex_id];
  vec3 ctr_o = pos_rad.xyz;
  float r = pos_rad.w;

  if ( mirrored ) ctr_o.y = -ctr_o.y;

  // Eye location in object space.
  vec3 e_o = vec3(gl_ModelViewMatrixInverse * vec4(0,0,0,1));

  // Vectors from eye to sphere center.
  vec3 ec_o = ctr_o - e_o;  // Eye to Center (of sphere).
  float ec_len = length(ec_o);
  vec3 ne = ec_o / ec_len;

  // Vectors orthogonal to ec_o.
  //
  vec3 atr_o = abs(ec_o);
  int min_idx = atr_o.x < atr_o.y ? ( atr_o.x < atr_o.z ? 0 : 2 )
    : ( atr_o.y < atr_o.z ? 1 : 2 );
  vec3 nx_raw = min_idx == 0 ? vec3(0,-ec_o.z,ec_o.y)
    : min_idx == 1 ? vec3(-ec_o.z,0,ec_o.x) : vec3(-ec_o.y,ec_o.x,0);
  vec3 nx = normalize(nx_raw);
  vec3 ny = cross(ne,nx);

  // Compute center of the limb, lmb_o, the circle formed by the most
  // distant visible parts of the sphere surface.
  float sin_theta = r / ec_len;
  float a = r * sin_theta;
  vec3 lmb_o = ctr_o - a * ne;

  // Compute axes to draw the limb.
  float b = r * sqrt( 1 - sin_theta * sin_theta );
  vec3 ax = b * nx;
  vec3 ay = b * ny;

  mat4 mvp = trans_proj * gl_ModelViewMatrix;

#if 0
  // Emit a bounding square for the limb.
  for ( int i = -1; i < 2; i += 2 )
    for ( int j = -1; j < 2; j += 2 )
      {
        vertex_o = lmb_o + ax * i + ay * j;
```

```
        gl_Position = mvp * vec4(vertex_o,1);
        EmitVertex();
      }
  EndPrimitive();
#else

  /// Pre-Final Solution
  //
  const int n_sides = s * 2;

  // Find radius of bounding circle for 2s-sided regular polygon.
  //
  const float pi = 3.1415926535;
  const float delta_theta = pi / s;
  const float delta_theta_h = delta_theta/2;
  const float b_outer = b / cos(delta_theta_h);  // Radius of bounding circle.

  // Compute axes for this circle.
  //
  const vec3 bx = b_outer * nx;
  const vec3 by = b_outer * ny;

  for ( int i=0; i<s; i++ )
    {
      const float theta = delta_theta_h + i * delta_theta;

      // Compute vertex coordinates for "left" and "right" half of circle.
      //
      for ( int j=-1; j<2; j+=2 )
        {
          vertex_o = lmb_o + bx * cos(theta) - j * sin(theta) * by;
          gl_Position = mvp * vec4(vertex_o,1);
          EmitVertex();
        }
    }
  EndPrimitive();
#endif
}
```

(*b*) Let t_f denote the execution time of 1 invocation of the true-sphere fragment shader when the sphere covers the fragment, and let t_n denote the execution time of 1 invocation when the sphere does not cover the fragment. Assume that $t_f > t_n$. Let $t_{g0} + st_{g1}$ denote the execution time of one invocation of the true-sphere geometry shader that emits a $2s$-sided polygon.

In general, emitting a polygon with more sides reduces the work performed by the fragment shader but increases the work done by the geometry shader, improving overall performance. But there will be a point at which increasing s *reduces* performance.

Using the symbols defined above **plus other information**, find an expression for the time needed to render a sphere. The other information is needed to compute a sphere rendering time using the timings above. Of course, no credit will be given if that other information *is* the execution time. Also determine the value of s that will minimize the total time used by the geometry and fragment shaders for rendering a sphere. When computing the s that minimizes time it is okay to apply simplifications based on s being large. (Otherwise a closed-form solution can't easily be found.) *Hint: Yes, the solution requires calculus.*

☑ Indicate the other information that is needed.

The other information is the number of fragments. Let r denote the number of fragments along the limb radius (from the center of the projected sphere to an edge). This value of r is determined by the sphere size, the location of the sphere and user, and the projection matrix.

☑ Show an expression for the time needed to render a sphere using a $2s$-sided bounding polygon in terms of t_f, etc., and the other information.

The number of fragment within the circle, taking t_f time, is πr^2. The number within the square bounding box is $4r^2$. The number of fragments within a $2s$-sided regular polygon with an inner radius of r is $2s \tan \frac{\pi}{2s} r^2$.

When rendering a sphere the geometry shader will be invoked just once and the fragment shader will be invoked $2s \tan \frac{\pi}{2s} r^2$ times, of which πr^2 invocations will take t_f and the remainder t_n. The total time needed, $t(s)$, is then:

$$t(s) = t_{g0} + st_{g1} + \pi r^2 t_f + \left(2sr^2 \tan \frac{\pi}{2s} - \pi r^2\right) t_n.$$

☑ Find an expression for s that will minimize time in terms of t_f, etc., and the other information.

To minimize $t(s)$ solve $\frac{d}{ds} t(s) = 0$ for s.

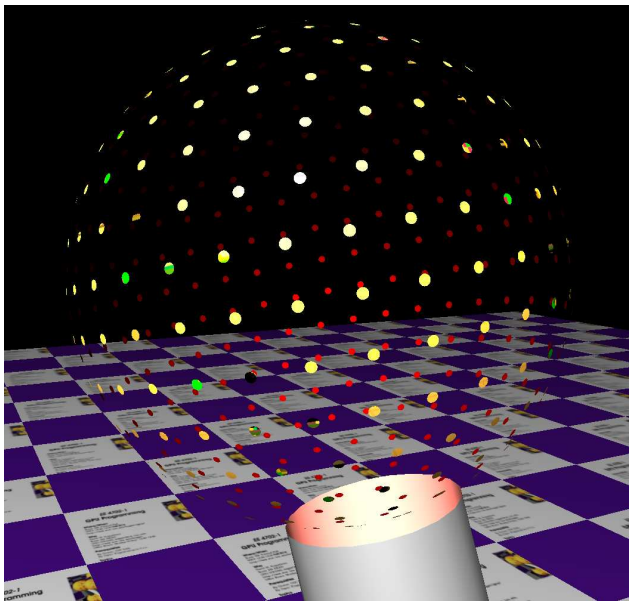
$$\frac{d}{ds} t(s) = t_{g1} + 2t_n r^2 \tan \frac{\pi}{2s} - \frac{\pi t_n r^2}{s \cos^2 \frac{\pi}{2s}} = 0$$

Assuming that s is large reduces the equation above to $st_{g1} = 0$, which isn't useful. *Note: I made an error taking the derivative when constructing the problem, with my error it would be possible to solve for s. Any solution attempting to differentiate the expression for time, setting it to zero, and solving for s would get full credit.*

A better solution approach is to approximate the number of non-sphere fragments using a bounding circle for the polygon, one of radius $r/\cos(\pi/2s)$.

Problem 3: [33 pts] The screenshot below is based on the code from Homework 4. The code is in *lenses* mode, meaning that the sphere is visible only at certain points. Unlike Homework 4, the lenses are much further apart from each other, and therefore much less of the sphere is visible. If the code in the Homework 4 solution were used then most invocations of the fragment shader would be wasted because they end up discarding the fragment.

There would be much less waste if a separate primitive were emitted for each lens, rather than one primitive for the entire sphere.



Suppose that each sphere has the same number of lenses, call that number h, and refer to the lenses as lens 0, lens 1, etc. Assume that a library of functions is available for computing positions for each lens. Among these is a function `lens_get_ctr_o(i,sphere_info)` which returns the object-space coordinates of the center of lens `i`. Variable `sphere_info` holds the value of the sphere center, radius, and orientation. Assume that `sphere_info` has already been prepared. You may assume that related functions exist such as `lens_get_ctr_e(i,sphere_info)` for eye-space coordinates. Also convenient for this problem is function `lens_get_bb_o(i,j,n,sphere_info)` which returns the object-space coordinates of point `j` of an `n`-sided bounding polygon for lens `i`. The bounding polygon is like the bounding square used in the true-sphere code: it tightly fits around the lens as seen by the viewer (or equivalently, when projected on to the frame buffer). The points wrap around the lens in a clockwise direction with increasing `j`.

Show changes needed to the geometry and fragment shaders on the following pages to render the lenses efficiently for cases in which h is relatively small, say on the order of 10.

Problem 3, continued:

(*a*) Appearing below is a shortened version of the geometry shader from the Homework 4 solution. The comments have been removed, but feel free to look at comments in the posted solution. Modify the geometry shader below so that it emits a primitive for each lens and provides information needed by the fragment shader (see next problem). Use the functions described above to find coordinates of bounding polygons for the lenses.

☑ Modify so that it emits a primitive for each lens.

☑ If necessary, change layout and output declarations.

☑ Provide information needed by fragment shader (see next subproblem).

The solution appears after the unmodified code.

```
out Data_to_FS // Data to fragment shader
{
  flat int vertex_id;
  vec3 vertex_o;


};

const int h = 8; // Number of holes. Don't assume it is always 8.
layout ( points ) in;
layout ( triangle_strip, max_vertices = 4 ) out;

void gs_main() {
  vertex_id = In[0].vertex_id;
  vec4 pos_rad = sphere_pos_rad[vertex_id];
  vec3 ctr_o = pos_rad.xyz;
  float r = pos_rad.w;
  Sphere_Info sphere_info = make_sphere_info(pos_rad,sphere_rot[vertex_id]);

  // Eye location in object space.
  vec3 e_o = vec3(gl_ModelViewMatrixInverse * vec4(0,0,0,1));

  // Vectors from eye to sphere center.
  vec3 ec_o = ctr_o - e_o;  // Eye to Center (of sphere).
  float ec_len = length(ec_o);
  vec3 ne = ec_o / ec_len;

  // Vectors orthogonal to ec_o.
  vec3 atr_o = abs(ec_o);
  int min_idx = atr_o.x < atr_o.y ? ( atr_o.x < atr_o.z ? 0 : 2 )
    : ( atr_o.y < atr_o.z ? 1 : 2 );
  vec3 nx_raw = min_idx == 0 ? vec3(0,-ec_o.z,ec_o.y)
    : min_idx == 1 ? vec3(-ec_o.z,0,ec_o.x) : vec3(-ec_o.y,ec_o.x,0);
  vec3 nx = normalize(nx_raw);
  vec3 ny = cross(ne,nx);
```

```
    // Compute center of the limb, lmb_o, the circle formed by the most
    // distant visible parts of the sphere surface.
    float sin_theta = r / ec_len;
    float a = r * sin_theta;
    vec3 lmb_o = ctr_o - a * ne;

    // Compute axes to draw the limb.
    float b = r * sqrt( 1 - sin_theta * sin_theta );
    vec3 ax = b * nx, ay = b * ny;
    mat4 mvp = trans_proj * gl_ModelViewMatrix;

    // Emit a bounding square for the limb.
    for ( int i = -1; i < 2; i += 2 ) for ( int j = -1; j < 2; j += 2 ) {
          vertex_o = lmb_o + ax * i + ay * j;
          gl_Position = mvp * vec4(vertex_o,1);
          EmitVertex();
        }
    EndPrimitive();









}
```

The solution appears below. The geometry shader emits `h` primitives using the convenience functions to find the primitive limits. For each lens the shader also sends the object-space coordinates of the lens center and whether the lens faces the viewer. Note that it is now the geometry shader that computes the lens location, not the fragment shader. That's possible because one primitive is emitted for each lens.

```
/// SOLUTION

out Data_to_FS
{
  flat int vertex_id;
  vec3 vertex_o;

  /// SOLUTION – Add variables for lens center and face.
  //
  flat vec3 lens_ctr_o;
  flat bool lens_front;

};

// Type of primitive at geometry shader input.
//
layout ( points ) in;

// Type of primitives emitted geometry shader output.
//
 /// SOLUTION – Increase number of vertices.
layout ( triangle_strip, max_vertices = h * 4 ) out;

void
gs_main()
{
  vertex_id = In[0].vertex_id;
  vec4 pos_rad = sphere_pos_rad[vertex_id];
  vec3 ctr_o = pos_rad.xyz;
  float r = pos_rad.w;

  if ( mirrored ) ctr_o.y = -ctr_o.y;

  // Eye location in object space.
  vec3 e_o = vec3(gl_ModelViewMatrixInverse * vec4(0,0,0,1));

  // Vectors from eye to sphere center.
  vec3 ec_o = ctr_o - e_o;  // Eye to Center (of sphere).
  float ec_len = length(ec_o);
  vec3 ne = ec_o / ec_len;

  // Vectors orthogonal to ec_o.
  //
  vec3 atr_o = abs(ec_o);
  int min_idx = atr_o.x < atr_o.y ? ( atr_o.x < atr_o.z ? 0 : 2 )
```

```
    : ( atr_o.y < atr_o.z ? 1 : 2 );
  vec3 nx_raw = min_idx == 0 ? vec3(0,-ec_o.z,ec_o.y)
    : min_idx == 1 ? vec3(-ec_o.z,0,ec_o.x) : vec3(-ec_o.y,ec_o.x,0);
  vec3 nx = normalize(nx_raw);
  vec3 ny = cross(ne,nx);

  // Compute center of the limb, lmb_o, the circle formed by the most
  // distant visible parts of the sphere surface.
  float sin_theta = r / ec_len;
  float a = r * sin_theta;
  vec3 lmb_o = ctr_o - a * ne;

  // Compute axes to draw the limb.
  float b = r * sqrt( 1 - sin_theta * sin_theta );
  vec3 ax = b * nx;
  vec3 ay = b * ny;

  mat4 mvp = trans_proj * gl_ModelViewMatrix;

  /// SOLUTION – Remove original code emitting primitive.
#if 0
  // Emit a bounding square for the limb.
  for ( int i = -1; i < 2; i += 2 )
    for ( int j = -1; j < 2; j += 2 )
      {
        vertex_o = lmb_o + ax * i + ay * j;
        gl_Position = mvp * vec4(vertex_o,1);
        EmitVertex();
      }
  EndPrimitive();
#endif

  /// SOLUTION – Use convenience functions to find bounding square ..
  ///              .. of each lens.
  //
  //  Provide lens center and face to fragment shader.

  for ( int l=0; l<h; l++ )
    {
      lens_ctr_o = lens_get_ctr_o(l,sphere_info);
      vec3 norm_o = lens_ctr_o - ctr_o;
      lens_front = dot(ec_o,norm_o) < 0;
      // Emit a bounding square for the limb.
      int indices[] = {0, 1, 3, 2};
      for ( int s=0; s<4; s++ )
        {
          vertex_o = lens_get_bb_o(l,indices[s],4,sphere_info);
          gl_Position = mvp * vec4(vertex_o,1);
          EmitVertex();
        }
      EndPrimitive();
```

```
    }
}

#endif

}
```

(*b*) Appearing below is the fragment shader from the Homework 4 solution. The comments have been removed for brevity, feel free to look at the full Homework 4 solution. Show the changes needed to the fragment shader. The shader should be substantially simpler.

☑ Changes needed, based on geometry shader from previous part.

☑ The fragment shader should render lenses only, not full spheres.

☑ Simplify based on the fact that geometry shader is providing lens information.

The solution appears after the original code.

```
void fs_main()
{
  vec4 pos_rad = sphere_pos_rad[vertex_id];
  vec3 ctr_o = pos_rad.xyz;
  float rsq = pos_rad.w * pos_rad.w;
  vec3 e_o = gl_ModelViewMatrixInverse[3].xyz;
  vec3 ef = vertex_o - e_o;   // Eye to fragment.
  vec3 ce = e_o-ctr_o;
  float qfa = dot(ef,ef), qfb = 2 * dot(ef,ce), qfc = dot(ce,ce) - rsq;
  float discr = qfb*qfb - 4 * qfa * qfc;

  if ( discr < 0 ) discard;

  const float pi = 3.14159265359;
  const float two_pi = 2 * pi;
  const bool holes = opt_holes == OHO_Holes;    // If true sphere has holes
  const bool lenses = opt_holes == OHO_Lenses;
  const bool solid = !holes && !lenses;         // If true show complete sphere.

  vec4 sur_o;  // Global surface cordinates (front or back) of sphere.
  vec3 normal_o, sur_l;
  bool front;          // If true, cordinates are of front of sphere.
  float theta, eta;    // Local angular cordinates of sphere.
  bool found_hole = false;

  for ( float dir = -1;  dir <= 1;  dir+=2 ) {
      front = dir == -1;
      float t = ( -qfb + dir * sqrt( discr ) ) / ( 2 * qfa );

      sur_o = vec4(e_o + t * ef, 1);
      normal_o = normalize(sur_o.xyz - ctr_o);
```

```
      if ( !front ) normal_o = -normal_o;

      sur_l = normalize(mat3(sphere_rot[vertex_id]) * normal_o);
      theta = atan(sur_l.x,sur_l.z);
      eta = acos(sur_l.y);

      if ( solid ) break;

      const float hole_frac = 0.2;
      const float radians_per_hole_eq = two_pi / opt_n_holes_eqt;
      const float hole_radius = 0.5 * hole_frac * radians_per_hole_eq;

      float eta_hole = round( eta / radians_per_hole_eq ) * radians_per_hole_eq;
      float r = sin(eta_hole);
      const float n_holes = floor( two_pi * r / radians_per_hole_eq );
      if ( n_holes < -1 )
        {
          if ( holes ) break;
          if ( dir == 1 ) discard;
        }

      const float radians_per_hole = two_pi / n_holes;
      float theta_hole = round( theta / radians_per_hole ) * radians_per_hole;
      vec3 hole_dir_l =
        vec3( r * sin(theta_hole), cos(eta_hole), r * cos(theta_hole) );

      float dist = distance(hole_dir_l,sur_l);
      found_hole = dist < hole_radius;
      if ( lenses && found_hole  holes && !found_hole ) break;
      if ( !front ) discard;
    }

  vec3 normal_ee = normalize(gl_NormalMatrix * normal_o);
  vec4 sur_ee = gl_ModelViewMatrix * sur_o;

  // Compute clip-space depth. Take care so that compiler avoids full
  // matrix / vector multiplication.
  vec4 sur_e = gl_ModelViewMatrix * sur_o;
  vec4 sur_c = trans_proj * vec4(0,0,2*sur_e.z,1);
  gl_FragDepth = sur_c.z / sur_c.w;

  vec2 tcoord = vec2( ( 1.5 * pi + theta ) / two_pi, eta / pi );
  vec4 texel = front ? texture(tex_unit_0,tcoord) : vec4(1);
  vec4 color2 = front ? sphere_color[vertex_id] : vec4(0.3,0,0,1);
  gl_FragColor = texel * generic_lighting(sur_ee, color2, normal_ee);
}
```

The solution appears below. The code for computing the hole center coordinates has been removed. Instead the fragment shader uses `lens_ctr_o` provided by the geometry shader. Also, the `dir` loop has been removed. It is not needed because the geometry shader has determined whether the front or back of a lens is visible.

```
void
fs_main()
{
  /// SOLUTION - Pre-final Problem 3b

  vec4 pos_rad = sphere_pos_rad[vertex_id];

  // Center of sphere in original object-space cordinates.
  vec3 ctr_o = pos_rad.xyz;
  float rsq = pos_rad.w * pos_rad.w;

  // Eye location in object-space cordinates.
  vec3 e_o = gl_ModelViewMatrixInverse[3].xyz;

  // Prepare to compute intersection of ray from eye to through fragment with
  // sphere. That intersection is the point on the sphere corresponding
  // to this fragment.
  //
  vec3 ef = vertex_o - e_o;   // Eye to fragment.
  vec3 ce = e_o-ctr_o;
  float qfa = dot(ef,ef);
  float qfb = 2 * dot(ef,ce);
  float qfc = dot(ce,ce) - rsq;
  float discr = qfb*qfb - 4 * qfa * qfc;

  // If outside the limb, return.
  if ( discr < 0 ) discard;

  const float pi = 3.14159265359;
  const float two_pi = 2 * pi;

  vec4 sur_o;  // Global surface cordinates (front or back) of sphere.
  vec3 normal_o, sur_l;
  float theta, eta;    // Local angular cordinates of sphere.

  /// SOLUTION – No changes made to code above this point ..
  //  .. except for deleting unneeded vars.


  /// SOLUTION - Geometry shader determines which side faces eye.
  //
  bool front = lens_front;

  /// SOLUTION:  Remove loop. Value of dir based on lens_front from geo shader.
  {
    float dir = front ? -1 : 1;
```

```
  // Finish computing coordinate of point for this fragment.
  //
  float t = ( -qfb + dir * sqrt( discr ) ) / ( 2 * qfa );

  // Compute true sphere surface coordinate.
  //
  sur_o = vec4(e_o + t * ef, 1);

  // Compute possibly reflected sphere surface coordinate.
  //
  normal_o = normalize(sur_o.xyz - ctr_o);

  //  Reverse normal if this is the back surface of the sphere.
  //
  if ( !front ) normal_o = -normal_o;

  // Use sphere-local coordinates to compute texture coordinates.
  //
  sur_l = normalize(mat3(sphere_rot[vertex_id]) * normal_o);
  theta = atan(sur_l.x,sur_l.z);
  eta = acos(sur_l.y);

  const float hole_frac = 0.4;
  const float radians_per_hole_eq = pos_rad.w * two_pi / h;
  const float hole_radius = 0.5 * hole_frac * radians_per_hole_eq;

  /// SOLUTION – Removed code computing hole location.
  //  Instead use lens_ctr_o provided by geo shader.
  //
  // Find distance from center of lens to sphere surface point ..
  //
  float dist = distance(lens_ctr_o,sur_o.xyz);
  //
  // .. and check whether surface point is in lens.
  //
  bool found_lens = dist < hole_radius;
  //
  // If this part of the sphere is not visible, we're done.
  //
  if ( !found_lens ) discard;
}

/// SOLUTION – No changes made to code below this point.

vec3 normal_ee = normalize(gl_NormalMatrix * normal_o);
vec4 sur_ee = gl_ModelViewMatrix * sur_o;

// Compute clip-space depth. Take care so that compiler avoids full
// matrix / vector multiplication.
vec4 sur_e = gl_ModelViewMatrix * sur_o;
vec4 sur_c = trans_proj * vec4(0,0,2*sur_e.z,1);
```

```
  gl_FragDepth = sur_c.z / sur_c.w;

  vec2 tcoord = vec2( ( 1.5 * pi + theta ) / two_pi, eta / pi );

  // Get filtered texel.
  //
  vec4 texel = front ? texture(tex_unit_0,tcoord) : vec4(1);

  vec4 color = sphere_color[vertex_id];
  vec4 color2 = front ? color : vec4(0.3,0,0,1);

  // Multiply filtered texel color with lighted color of fragment.
  //
  gl_FragColor = texel * generic_lighting(sur_ee, color2, normal_ee);
}
```

Name Solution

GPU Programming
LSU EE 4702-1
Final Examination

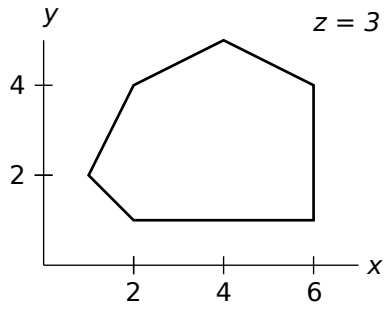
Tuesday, 4 December 2018 12:30–14:30 CST

Problem 1		(15 pts)
Problem 2		(30 pts)
Problem 3		(15 pts)
Problem 4		(40 pts)
Exam Total		(100 pts)

Alias em dash

Good Luck!

Problem 1: [15 pts] Appearing below is a geometric figure.



(*a*) Complete the individual-triangle rendering pass below so that it renders the figure with all triangles facing in the positive z direction and without overlapping triangles. Use the provided abbreviation `glV`. *Note: The part about facing the $+z$ direction was not in the original exam.*

☑ Complete code to render shape using individual triangles.

```
#define glV glVertex3f
 glBegin(GL_TRIANGLES);

 glV(2,1,3); glV(2,4,3); glV(1,2,3);  // SOLUTION
 glV(6,4,3); glV(4,5,3); glV(2,4,3);
 glV(2,1,3); glV(6,4,3); glV(2,4,3);
 glV(2,1,3); glV(6,1,3); glV(6,4,3);

 glEnd();
```

(*b*) Complete the triangle-strip rendering pass below so that it renders the figure.

☑ Complete code to render shape using a triangle strip.

```
#define glV glVertex3f
 glBegin(GL_TRIANGLE_STRIP);

 glV(2,1,3); glV(1,2,3);  // SOLUTION
 glV(6,1,3); glV(2,4,3);
 glV(6,4,3); glV(4,5,3);

 glEnd();
```

Problem 2: [30 pts] Appearing below is shortened host and shader code based on Homework 3, in which text was drawn on the triangular spiral. One drawback of this code is that it uses old-fashioned, deprecated, inefficient `glVertex` calls. On the following pages are routines that will implement a more efficient version of this code in which data such as `ctr` are placed in buffer objects and a rendering pass is performed *for the entire chain*, not just one triangular spiral.

```
for ( int i=2; i<chain_length; i++ ) {  // Host Code
    pCoor p0 = balls[i-2].position, p1 = balls[i-1].position, p2 = balls[i].position;
    pCoor ctr = (p0+p1+p2)/3; // Compute location of triangle center.
    // --- CODE REMOVED FOR BREVITY ---
    pCoor pprev = ctr;
    float delta_a = 0.6 / opt_n_segs;

    if ( opt_shader == SO_HW03 ) {
        glBegin(GL_TRIANGLE_STRIP);  // Render spiral using 1 triangle strip.

        for ( int j=0; j<opt_n_segs; j++ ) {
            pVect v = va[j%3];                // Vector from center to fold.
            pCoor p = ctr + j * delta_a * v;  // Coordinate of fold.
            pNorm n = cross( p - pprev, vz ); // Normal of segment.
            float tex_x = total_len_compute(j,delta_a,distv) * tex_scale;

            glNormal3fv(n);
            glTexCoord2f(tex_x,0);  glVertex3fv(p + vz);
            glTexCoord2f(tex_x,1);  glVertex3fv(p - vz);
            pprev = p;
          }
        glEnd();
      } }

void vs_main_hw03() { // Vertex Shader Code
  gl_Position ⟸ O = gl_ModelViewProjectionMatrix ⟸ U * gl_Vertex ⟸ I;
  vertex_e = gl_ModelViewMatrix ⟸ U * gl_Vertex ⟸ I;
  normal_e = normalize(gl_NormalMatrix ⟸ U * gl_Normal ⟸ I);
  tex_coord = gl_MultiTexCoord0.xy ⟸ I;   }
```

(*a*) But first let c denote the value of `chain_length` and n denote the value of `opt_n_segs`. Determine the amount of data, in bytes, sent from the CPU to the GPU for the code shown above. (Do not consider uniforms and other hidden code.)

☑ Amount of data sent from CPU to GPU in terms of c and n, in bytes.

The amount of data sent is $\boxed{(c-2)n(3+2(2+3))4\,\mathrm{B}}$, where the $3+$ term gives the number of floats sent by `glNormal3fv`, and the $2(2+3)$ term gives the number of floats sent by the calls to `glTexCoord2f` and `glVertex3fv`, the $4\,\mathrm{B}$ factor accounts for the size of a float, and $(c-2)n$ is the number of executions of the inner loop body.

(*b*) In the vertex shader above, label variables with the appropriate letter as requested below.

☑ Label uniform variables with a **U**, ☑ shader input variables with an **I**, and ☑ fixed-function shader outputs with an **O**.

Solution appears above, look for ⟸ These.

Problem 2, continued: Appearing below is the improved triangular spiral host code and shaders. The host code prepares four buffer objects and then starts an instanced rendering pass with line strips as the input primitive. The rendering pass uses a vertex shader and a geometry shader to complete the primitives. (The fragment shader is not a part of this problem.) The only inputs to the vertex shader are the vertex and instance IDs, which should be used to retrieve or compute information about the spiral segments.

```
// Host Code -- DO NOT MAKE OR ASSUME MODIFICATIONS TO THIS CODE.
      for ( int i=2; i<chain_length; i++ ) {
          // Put ball structures and coordinates into convenient variables.
          pCoor p0 = balls[i-2].position, p1 = balls[i-1].position, p2 = balls[i].position;
          pCoor ctr = (p0+p1+p2)/3;   ctr_a.push_back(ctr);

          // -- CODE PREPARING va_a, vz_a, and distv_a NOT SHOWN --
        }
      glUniform1i(5, opt_n_segs);
      TO_BUFFER_OBJECT(ctr_a,1);      TO_BUFFER_OBJECT(va_a,2);
      TO_BUFFER_OBJECT(distv_a,3);    TO_BUFFER_OBJECT(vz_a,4);
      glBindBuffer(GL_ARRAY_BUFFER,0);

      glDrawArraysInstanced(GL_LINE_STRIP, 0, opt_n_segs, ctr_a.size() );
```

(*c*) Modify the vertex and geometry shaders on the following pages to efficiently render the triangle spirals, make any needed changes to the interface blocks, but **do not modify** or assume modifications to the host code (above). For your convenience each shader contains a copy of the host code. Cross out or modify that code as needed. Use the handy abbreviations at the top of the page.

☑ Update the interface blocks as needed.

```
#ifdef _VERTEX_SHADER_
out Data {  // Out of Vertex Shader to Geometry Shader
 int ins_id, vtx_id;
 //  ☑   Add any needed declarations here.

 // SOLUTION
 vec4 p_e[2], p_c[2]; // Send pre-computed eye- and clip-space coords.
 float tex_x;         // Position of texture at fold.

};
#endif
#ifdef _GEOMETRY_SHADER_
in Data {  // In to Geometry Shader from Vertex Shader
 int ins_id, vtx_id;
 //  ☑   Add any needed declarations here.

 // SOLUTION
 vec4 p_e[2], p_c[2]; // Send pre-computed eye- and clip-space coords.
 float tex_x;         // Position of texture at fold.

} In[2];

// Out of Geometry Shader to Fragment Shader -- No changes needed
out Data {   vec3 normal_e;   vec4 vertex_e;   vec2 tex_coord;  };
#endif
```

Problem 2, continued: Vertex shader code on this page.

☑ Cross out unneeded code. ☑ Cross out data type for shader outputs. ☑ Avoid redundant computation.

☑ Update the interface blocks as needed.

```
const mat4 mv = gl_ModelViewMatrix, mvp = gl_ModelViewProjectionMatrix;
const mat3 nm = gl_NormalMatrix;

void vs_main_lines() {  // Vertex Shader Main Routine
  int ins_id = gl_InstanceID, vtx_id = gl_VertexID;

  int j = gl_VertexID;
  float delta_a = 0.6 / opt_n_segs;
  vec4 distv = distv_a[ins_id];
  vec3 vz = vz_a[ins_id].xyz;
  float tex_scale = 0.2;

  // SOLUTION
  vec3 v = va_a[ins_id][j%3].xyz;
  vec3 p = ctr_a[ins_id].xyz + j * delta_a * v;

  for ( int i=0; i<2; i++ ) // Compute +vz and -vz coords.
    {
      vec4 po = vec4( p + ( i == 0 ? vz : -vz ), 1 );
      p_e[i] = mv * po;
      p_c[i] = mvp * po;
    }

  tex_x = total_len_compute(j,delta_a,distv) * tex_scale;

}
```

Solution appears above. The vertex shader is computing the eye- and clip-space coordinates, saving the geometry shader the need to do so. Note that because the geometry shader primitive is a line strip, each vertex is used for two primitives. So, by doing coordinate space conversions in the vertex shader rather than the geometry shader cuts in half the number of coordinate space conversions needed. Whether this improves performance is a more complex question. However a requirement for this problem was to avoid redundant computation.

Problem 2, continued: Geometry shader code on this page.

☑ Cross out unneeded code. ☑ Cross out data type for shader outputs. ☑ Avoid redundant computation.

☑ Update the interface blocks as needed.

```
layout ( lines ) in;
layout ( triangle_strip, max_vertices = 4 ) out;

const mat4 mv = gl_ModelViewMatrix, mvp = gl_ModelViewProjectionMatrix;
const mat3 nm = gl_NormalMatrix;

void gs_main_lines() {  // Geometry Shader Main Routine

  // SOLUTION
  int ins_id = In[0].ins_id;

  vec3 vz_e = In[0].p_e[0].xyz - In[0].p_e[1].xyz;
  normal_e = normalize(cross( In[1].p_e[0].xyz-In[0].p_e[0].xyz, vz_e ));

  for ( int i=0; i<2; i++ )
    {
      for ( int j=0; j<2; j++ )
        {
          tex_coord = vec2(In[i].tex_x,j);
          gl_Position = In[i].p_c[j];
          vertex_e = In[i].p_e[j];
          EmitVertex();
        }
      }
  EndPrimitive();
}
```

Solution appears above. Notice that the geometry shader input consists of two vertices, `In[0]` and `In[1]`. The `In[0]` vertex corresponds to the vertex used to compute `pprev` on the host code. Notice how the loop nest avoids the need to duplicate code.

Problem 3: [15 pts] Answer each CUDA question below.

(*a*) Both CUDA kernels below do the same thing, but one will execute much less efficiently. Explain why in terms of the minimum request size.

```
__global__ void kmain_simple(float4 *d_in, float *d_out) {
  const int tid = threadIdx.x + blockIdx.x * blockDim.x;
  const int elt_per_thread = ( d_app.array_size + d_app.num_threads - 1 ) / d_app.num_threads;
  const int start = elt_per_thread * tid;
  const int stop = start + elt_per_thread;

  for ( int h=start; h<stop; h++ ) {
      float4 p = d_in[h];
      float sos = p.x * p.x + p.y * p.y + p.z * p.z + p.w * p.w;
      d_out[h] = sos;
    }
}

__global__ void kmain_efficient(float4 *d_in, float *d_out) {
  const int tid = threadIdx.x + blockIdx.x * blockDim.x;
  for ( int h=tid; h<d_app.array_size; h += d_app.num_threads ) {
      float4 p = d_in[h];
      float sos = p.x * p.x + p.y * p.y + p.z * p.z + p.w * p.w;
      d_out[h] = sos;
    }
}
```

☑ Problem with inefficient kernel due to request size.

Each access `d_in[h]` is to 16 bytes of data. Threads in a warp make such an access together. Assume that `elt_per_thread = 200` and consider two consecutive threads in such an access with `tid=10` and `tid=11`. The initial value of `h` for these threads will be 2000 and 2200. That means the starting address of the data access by these two threads differs by $200 \times 16 = 3200$ bytes and so a request will have to be generated for each thread. Since the minimum request size for current NVIDIA devices is 32 B only half of the request will be used.

☑ What is the maximum request size that will avoid this inefficiency? ☑ Explain.

For the access to `d_in[h]` the maximum request size is 16 B, for the write to `d_out[h]` the maximum is 4 B, so overall the maximum is 4 B. This guarantees that every byte of every request will be used in the simple kernel above.

(*b*) A CUDA kernel is to run on a GPU with 8 SMs (MPs). Configuration A consists of 4 blocks of 32 threads each. Configuration B consists of 8 blocks of 16 threads each. Neither is very good. Explain how each one underutilizes the hardware on current NVIDIA GPUs.

☑ Configuration A underutilizes hardware by . . .

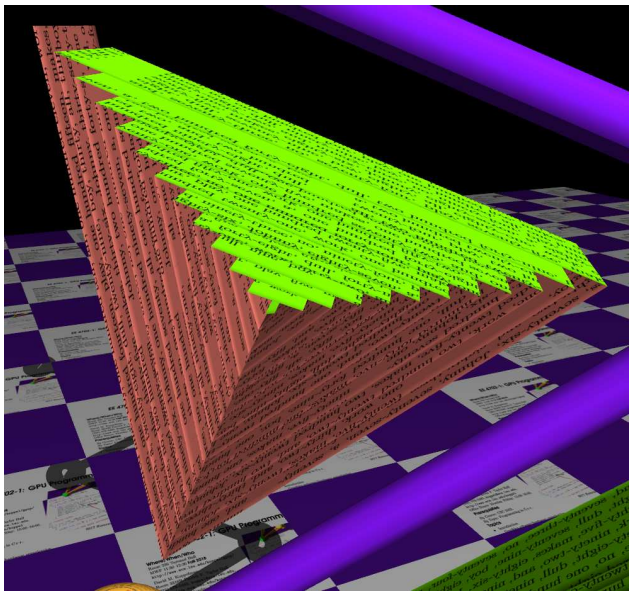
. . . leaving four SMs idle. Since a block can't run on more than on SM at a time with 4 blocks and 8 SMs, at least 4 will be idle.

☑ Configuration B underutilizes hardware by . . .

. . . leaving half of the warp unoccupied. Threads in an NVidia GPU are managed in units called warps, and the warp size up to now is 32. With less than 32 threads some functional units will go unused. The truth is even with 32 threads hardware will go unused since for Kepler (CC 3.x) up to Volta/Turing (CC 7.x) at least two and more often four warps per SM are needed to use all functional units (and often many more than that).

Problem 4: [40 pts] Answer each question below.

(*a*) The screenshot below is the Homework 3 triangular spiral with 37 segments per spiral. Imagine a spiral with even more segments, say 1000.



With a large number of segments there will usually be a large computational load on both the vertex and fragment shaders. In one of these shaders the computational load can be considered wasted, depending on the eye location, **even when the spiral is visible**. In which shader is computation wasted, and why.

☑ Which shader wastes computation? ☑ Why?

The fragment shader wastes computation because in a typical view, such as the one above, most fragments are discarded (since they are blocked from view by segments closer to the viewer).

Sketch two views in which the spiral is visible. In one the computational load is high and mostly wasted. In the other the computational load is lower and not wasted.

☑ View with lower load and little waste. ☑ View with high load and waste.

There will be little waste when spiral normal is aligned with the z axis. In such a view all or most fragments are visible (and there won't be many of them). There will be lots of waste when the spiral normal is orthogonal to z. In that case only one or two segments are visible, fragments for the rest of the segments are discarded.

NEED TO ADD SKETCH. If you are preparing for the 2019 final exam, ask for the sketch to be put in this solution. (Not the one you may have printed, the one on the Web site.)

(*b*) The OpenGL call `glColor` is used to specify a color, say purple `glColor3f(1,0,1);`. In typical use is that the color that will be written to the frame buffer? Explain.

☑ Are the arguments to `glColor` the exact color to be written to the frame buffer? ☑ Explain.

No. In typical use the value sent by calling `glColor` is used by the vertex shader to compute a lighted color. The fragment shader mixes the lighted color, actually a value interpolated from the lighted colors provided by all of the vertices making the primitive, with texture values. The result of that will be written to the frame buffer (if tests, such as the z test, pass).

(*c*) Describe the difference between the `flat`, `noperspective`, and `smooth` interpolation qualifiers.

☑ The `flat` qualifier's feature, ☑ the `noperspective` qualifier's, feature, ☑ the `smooth` qualifier's, feature.

Short Answer: `flat`, value is taken from the provoking vertex; `noperspective`, value is linearly interpolated between the vertices based on coordinates in clip space; `smooth`, value is linearly interpolated between the vertices based on coordinates in object space.

Explanation: The values provided to the fragment shader from variables declared using the `flat` qualifier will be the values written by the provoking (last) vertex of the primitive. For example, suppose color were declared `flat vec4 color;`, and the colors sent for the three vertices of a triangle were red, green, and blue. The triangle would be blue.

For both `noperspective` and `smooth` the value is interpolated linearly from the values provided for each vertex. With `smooth` the interpolation is based on object space coordinates, which is realistic but computationally expensive, while with `noperspective` the interpolation uses window space coordinates, which is easier to do but won't look right. Consider a line primitive with an attribute `tex_x`. Suppose the value for first vertex is `tex_x=0` and for the second vertexi is `tex_x=1`. For the declaration `noperspective float tex_x`, a value of 0.5 would be find at the point halfway between the two vertices based on a distance measured from the monitor. (Don't do this at home, you might scratch your monitor.) For the declaration `smooth float tex_x` the 0.5 value is halfway between the two vertices measured object space.

(*d*) Consider a rendering pass using triangles as the primitive. OpenGL (compatibily profile) allows one to specify a normal for each triangle vertex, but as we all know a triangle, geometrically, has just one normal. Why would one specify different normals for each vertex? Explain how such normals are chosen.

☑ Why might one choose different normals for each vertex of a triangle?

☑ Give an example and describe how those normals are chosen.

Because the triangle is approximating a surface, so by providing the normal of that surface, rather than that of the triangle, lighting will be more realistic. For example, if the triangles are approximating a cylinder use the cylinder surface normal.

(*e*) Describe what the inputs to the rasterization stage are, what the rasterization stage does, and what its outputs are.

☑ Rasterization stage input, ☑ rasterization stage job, ☑ rasterization stage output:

The input to rasterization is a primitive, which is the set of vertices corresponding to a primitive (say, three vertices for a triangle). The rasterization stage emits one fragment for each pixel that the primitive covers. The fragment consists of the attributes interpolated based on the fragment's position within the primitive.

(*f*) The true-sphere shader used in class rendered spheres perfectly. Would it make sense to use a similar approach to write a true-cube shader that can perfectly render a cube?

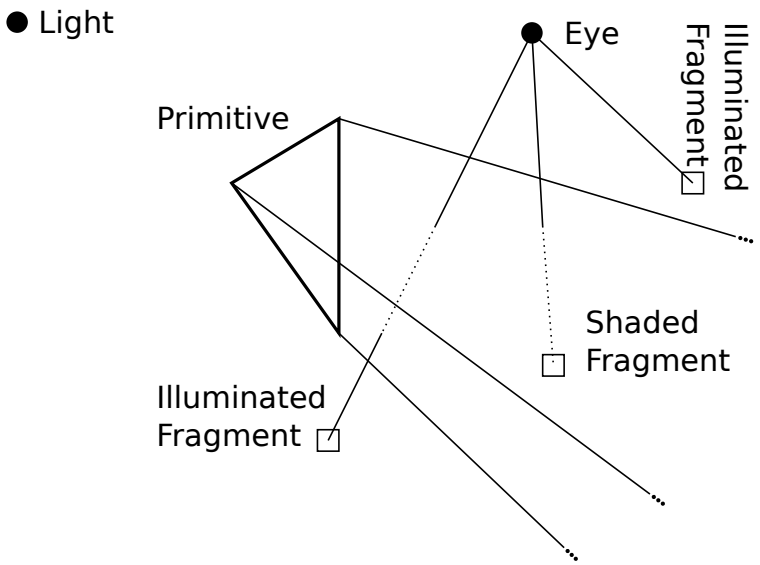
☑ Does a true cube shader make sense? ☑ Explain.

It does not make sense because a cube can be perfectly rendered using triangles. That was not possible for a sphere because, of course, the surface of a sphere is round and the flat surface of a triangle would only approximate the sphere surface.

(*g*) The unlabeled diagram below shows how shadow volumes can be used to render shadows. On the diagram show the location of the eye and light source, fragment(s) found to be in the shade, and fragment(s) found to be illuminated.

☑ Show: ☑ eye, ☑ light source, ☑ shaded fragment(s), ☑ illuminated fragment(s).

Solution appears below.



25 Fall 2017 Solutions

Name Solution

GPU Programming
EE 4702-1
Midterm Examination

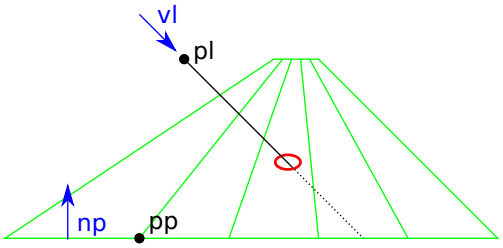
Wednesday, 18 October 2017 11:30–12:20 CDT

Problem 1		(30 pts)
Problem 2		(20 pts)
Problem 3		(25 pts)
Problem 4		(25 pts)
Exam Total		(100 pts)

Alias Think of something, quick!

Good Luck!

Problem 1: [30 pts] The illustration to the right shows a ring surrounding the point at which a line crosses a plane. When completed the code fragment below will render the ring. The line is defined by parametric equation $P(t) = \texttt{pl} + t \times \texttt{vl}$. Point `pp` is on the plane and vector `np` is normal to the plane. The ring is circular and is on the plane.



(*a*) Modify the code below so that `t` is assigned the value for which `pc` is on the plane.

☑ Compute t such that $P(t)$ is on the plane.

This is of course a line/plane intercept problem. We need to find t such that $\overrightarrow{P(t)P_p} \cdot n_p = 0$, where P_p is `pp` and n_p is `np`. First, separate t,

$$\begin{aligned} 0 =& \overrightarrow{P(t)P_p} \cdot n_p \\ =& (P_p - P(t)) \cdot n_p \\ =& (P_p - P_l - tv_l) \cdot n_p \\ =& \left(\overrightarrow{P_lP_p} - tv_l\right) \cdot n_p \\ =& \overrightarrow{P_lP_p} \cdot n_p - tv_l \cdot n_p \end{aligned}$$

then finish solving for t:

$$t = \frac{\overrightarrow{P_lP_p} \cdot n_p}{v_l \cdot n_p}$$

The code below computes t this way.

(*b*) Modify the code below to render the ring. Use `r1` for the inner radius and `r2` for the outer radius. Render it using `slices` sets of triangles. Use variable `pc` even if you haven't yet solved the previous part.

☑ Include: ☑ Code to set up the rendering pass, ☑ to set the color (red), ☑ to set the correct normal, ☑ and to compute and set vertex coordinates.

Solution appears below.

```
World::render_ring( pCoor pl,  pVect vl,     // Line definition.
                    pCoor pp,  pNorm np,     // Plane definition.
                    float r1,  float r2 ){  // Ring radii.

  pVect vlp(pl,pp);  // SOLUTION


  float t = dot(vlp,np) / dot(vl,np); // SOLUTION

  pCoor pc = pl + t * vl;

  // SOLUTION - Compute axis for drawing ring.
  pNorm ax(pc,pp);
  pNorm ay = cross(np,ax);

  glBegin(GL_TRIANGLE_STRIP);

  glNormal3fv(np);
  glColor3fv( color_red );

  const int slices = 100;
  const float delta_theta = 2 * M_PI / slices;
  for ( int i=0; i<=slices; i++ ) {
      const float theta = i * delta_theta;


      pVect d = ax * cos(theta) + ay * sin(theta);
      glVertex3fv( pc + r1 * d );
      glVertex3fv( pc + r2 * d );

    }
}
```

Problem 2: [20 pts] Recall that in Homework 2 a sphere was rendered using a spiral slice. The coordinates of the slice were placed in a buffer object and a sphere was rendered by performing multiple rendering passes each using the same buffer object but a different transformation. Let c denote the number of coordinates in the buffer object, and s denote the number of slices per sphere.

(a) What is the size of the buffer object, in bytes? *Hint: This is an easy question.*

☑ Size of buffer object, in bytes. ☑ State any assumptions made.

Short answer: $4 \times 4c\,\mathrm{B} = 16c\,\mathrm{B}$, with `pCoor` used for a coordinate, which I assume I remember correctly.

Explanation: The buffer object holds c coordinates, each coordinate is a `pCoor`. A `pCoor` is a structure holding four `floats`, so its size is $4 \times 4\,\mathrm{B}$.

(b) Appearing below is the code rendering a sphere, adapted from the Homework 2 solution. Based on this code, estimate the amount of data send from the CPU to the GPU to render the sphere, **not counting the buffer object**.

```
pMatrix_Rotation rot_xz( pVect(0,1,0), delta_theta ),  rot_yz( pVect(1,0,0), M_PI );
// Code to render one sphere.
glTranslatefv( ball->position );
glScale1f( ball->radius );
glMultTransposeMatrixf( pMatrix_Rotation(ball->orientation) );

for ( int j=0; j<2; j++ ) {
     for ( int i=0; i<s/2; i++ ) {
        glColor3fv( i & 1 ? color_lsu_spirit_purple : ball->color );
        glDrawArrays(GL_TRIANGLE_STRIP,0,c);
        glMultTransposeMatrixf(rot_xz);
      }
    glMultTransposeMatrixf(rot_yz);
  }
```

☑ In terms of s, amount of data sent to GPU to render sphere.

Short Answer: For each slice a new modelview matrix and color are sent to the GPU, for a total of $64 + 12 = 76\,\mathrm{B}$ per slice. For the entire sphere $\boxed{76s\,\mathrm{B}}$.

Explanation: The modelview matrix has $4 \times 4 = 16$ elements of size 4 bytes each, for a total size of $4^3 = 64\,\mathrm{B}$. The color size is $3 \times 4 = 12\,\mathrm{B}$. Because `glColor3fv` is called just before the rendering pass is started it should be clear that it is sent once per rendering pass, contributing $12s\,\mathrm{B}$ of data. The number of times the modelview matrix is sent to the GPU depends upon the OpenGL implementation. The commands `glTranslatefv`, `glScale1f`, and `glMultTransposeMatrixf` are all used to modify the default matrix (modelview in this case). An *eager* implementation might send the matrix each time one of these was called. A *lazy* implementation would update the copy of the modelview matrix on the CPU, but only send it to the GPU when a rendering pass is initiated. (In this situation lazy is better.) This solution assumes a lazy implementation. Since the loop body (which contains the command initiating the rendering pass, `glDrawArrays`) is executed $2 \times \frac{s}{2} = s$ times the array is sent s times and so the array contributes $64s\,\mathrm{B}$ of data per sphere. (An eager implementation would send the matrix $3 + 2(1 + \frac{s}{2})$ times, with the commands before the loop contribute the 3 term, and the $1+$ part contributed by the `glMultTransposeMatrixf` at the end of the `j` loop.)

(*c*) Buffer object BO2 holds coordinates to render an entire sphere, not just a slice, $s \times c$ coordinates total. Buffer object BO1 holds the slice discussed above. Suppose that initially all data is on the CPU. Let n be the number of sphere to be rendered. In terms of s and c, what is the smallest value of n for which using BO2 will require less data than BO1?

☑ Break-even value for n:

Using BO1 the total amount of data for n spheres is $16c + 76ns$. Using BO2 the total amount of data is $16sc + 76n$. Equating and solving for n yields $n = \frac{16c(s-1)}{76(s-1)} = \frac{16}{76}c$.

Problem 3: [25 pts] The diagram below shows a texture on the left-hand side and a scene to be rendered on the right. The scene consists of three triangles with the texture applied. The text in the texture, written by Mathy Vanhoef, is from the description of a widespread vulnerability in WiFi WPA2 implementations. See `https://www.krackattacks.com`, and please, Fall 2017 people, update your OS and wireless access point firmware. *Grading Note: The description of where the text came from was not in the original exam. The KRACK vulnerability became public two days before the midterm exam.*

Texture

We discovered serious weaknesses

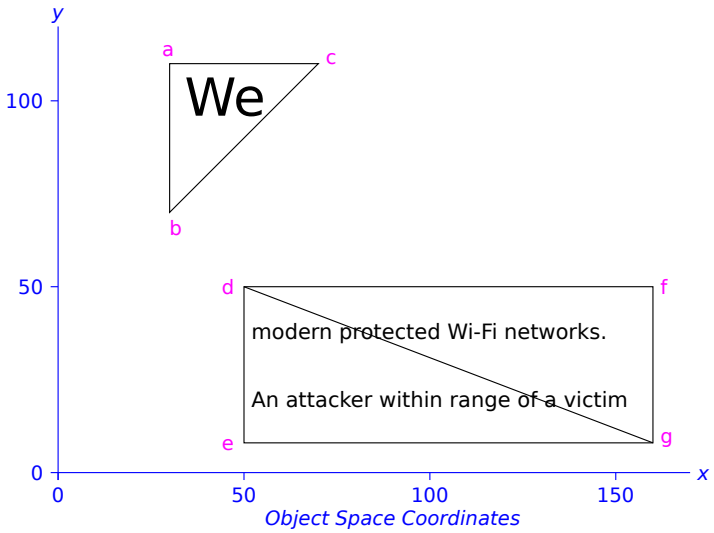
in WPA2, a protocol that secures all

modern protected Wi-Fi networks.

An attacker within range of a victim

can exploit these weaknesses using

key reinstallation attacks (KRACKs).



(*a*) Complete the code below so that it renders the triangles with the texture applied as shown. The color and normal have already been set, just specify vertex and texture coordinates. Use abbreviation `glV` for `glVertex2f` and `glT` for `glTexCoord2f`. Only specify x and y components for the vertex coordinates.

☑ Specify ☑ texture and ☑ vertex coordinates so that the scene is rendered as shown.

Solution appears below in blue. The solution uses the full function names, such as `glTexCoord2f`. On the exam it would be okay to use the abbreviations `glT` and `glV`.

```
glBegin(GL_TRIANGLES);
glNormal3f(0,0,1);
glColor3f(0.5,0.5,0.5);

glTexCoord2f(0,1);      // SOLUTION: Texture coord for Vtx a.
glVertex2f(30,110);     // Vertex a
// SOLUTION (below): Specify the remaining texture and vertex coords.
glTexCoord2f(0,.85);   glVertex2f(30,70);   // Vertex b
glTexCoord2f(0.1, 1);  glVertex2f(70,110);  // Vertex c

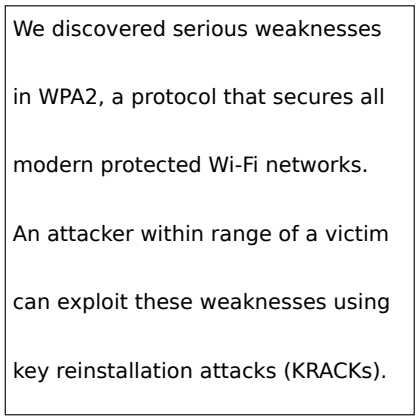
glTexCoord2f(0, .66);  glVertex2f(50,50);   // Vertex d
glTexCoord2f(0, .33);  glVertex2f(50,10);   // Vertex e
glTexCoord2f(1, .33);  glVertex2f(160,10);  // Vertex g

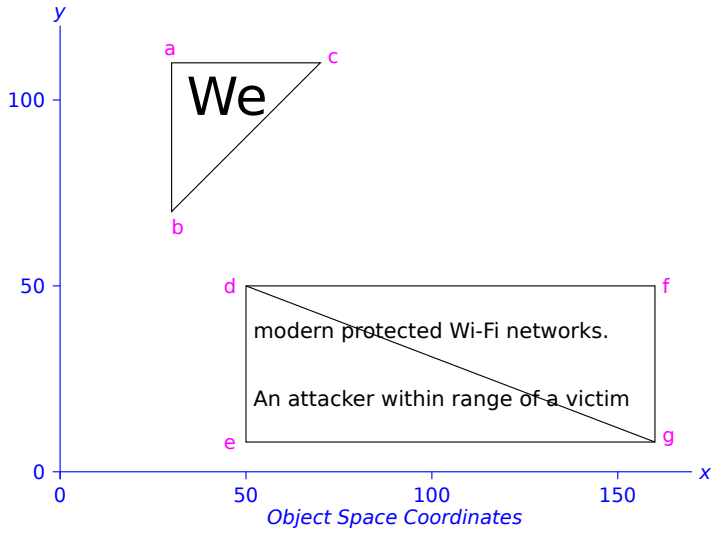
glTexCoord2f(0, .66);  glVertex2f(50,50);   // Vertex d
glTexCoord2f(1, .33);  glVertex2f(160,10);  // Vertex g
glTexCoord2f(1, .66);  glVertex2f(160,50);  // Vertex f


glEnd();
```

Problem 3, continued:

Texture





(*b*) Suppose that defg in the diagram above is four pixels across and two pixels down. Suppose that texturing were set up to use only a single mipmap level and nearest filtering. Explain how defg could appear all white in that case.

☑ At 4×2 pixels defg all white because:

Because the center of each pixel just happens to be over a part of the texture that's white. That's not so unlikely because most of the texture is white.

(*c*) Explain how the 4×2 rectangle would appear with linear filtering and mipmap levels. Explain the key points of linear texture filtering.

☑ Appearance with mipap levels and linear filtering.

The rectangle would appear gray.

☑ Key points of filtering.

Given a texture of width w, the width of the texture at mipmap level i is $w2^{-i}$. (Mipmap level 0 is the original image.) The color of a texel at mipmap level i might be the average of the four texels that it covers at mipmap level $i-1$. (This method of determining the color of a texel at a particular mipmap level is similar to the method used for determining the color of a filtered texel for a fragment.)

With linear-mipmap-linear filtering enabled, the filtered texel corresponding to a pixel is computed using the two closest mipmap levels, call them levels i and $i+1$. Level i is chosen so that the texels are smaller than the pixels and at level $i+1$ the texels are larger than the pixels. The filtered texel is based on a weighted average of up to four pixels at each of the two levels. The weight for a texel is based on the fraction of the pixel covered by the texel.

Problem 4: [25 pts] Answer each question below.

(*a*) Show a transformation matrix M that will translate a homogeneous coordinate P to $P + \begin{bmatrix} 7 \\ 6 \\ 5 \end{bmatrix}$. (That is, find M such that $MP = P + \begin{bmatrix} 7 \\ 6 \\ 5 \end{bmatrix}$.) Only show nonzero array elements.

☑ $M =$

The transformation matrix is:

$$M = \begin{pmatrix} 1 & 0 & 0 & 7 \\ 0 & 1 & 0 & 6 \\ 0 & 0 & 1 & 5 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

(*b*) Describe the "from" and "to" coordinate spaces for the following OpenGL matrices:

☑ The ModelView matrix maps a coordinate from ☑ object space to ☑ eye space.

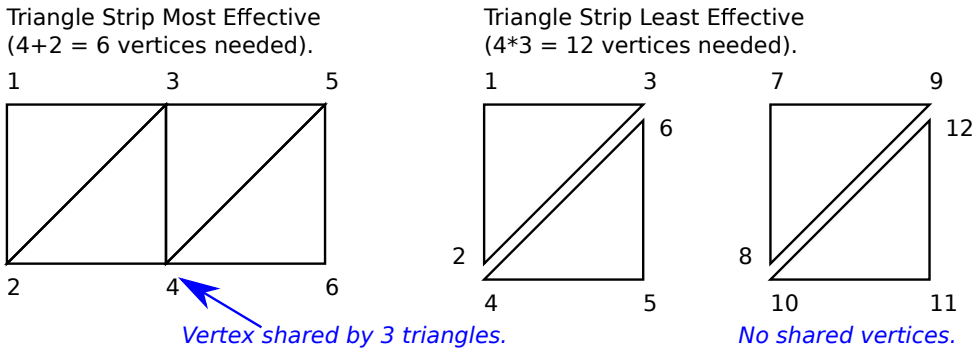
☑ The Projection matrix maps a coordinate from ☑ eye space to ☑ clip space.

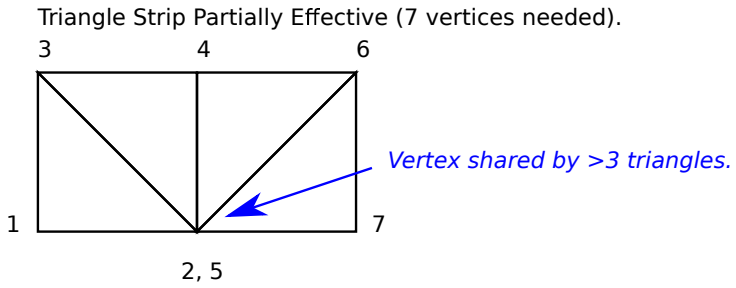
(*c*) Show two drawings, each consisting of four triangles. The one should be well-suited for rendering using a triangle strip, for the other a triangle strip should make no difference.

☑ Show a 4-triangle example well-suited to triangle strips and one in which a ☑ triangle strip doesn't help at all.

The well-suited triangle strip appears on the upper left. What makes this effective is that except for the starting (1,2) and ending (5,6) vertices, each vertex is shared by three triangles. Triangle strips don't help at all when there aren't any shared vertices, as in the drawing on the upper right. In the bottom drawing, labeled Partially Effective, triangle strips are better than individual triangles, but since 7 rather than 6 vertices are needed, they are not most effective. A triangle fan would be the best choice for this drawing.

Grading Note: Many answers to the doesn't-help part seemed to be based on the assumption that the triangles had to be touching in some way. If that were the case the question could be answered by having a vertex of one triangle touching an edge (but not a vertex) of another triangle.





(*d*) Suppose that buffer object o30 has 30 sphere coordinates and buffer object o1k has 1000 sphere coordinates. In both buffer objects the sphere coordinates are well chosen. They will be used to render a sphere which is fully visible. *Note: In the original exam the sizes were 1000 and 10,000.*

☑ Describe a situation in which the sphere would look better using o1k than using o30 and ☑ describe a situation in which the sphere would look the same with o1k as with o30. ☑ Explain.

Short Answer: o1k looks better when the projected sphere is large (covers alot of the window [frame buffer]), they look about the same when the projected sphere size is small. For the large sphere o30 can look bad because there are many pixels between triangle vertices so there will be straight lines where the modeled object is curved.

Long Answer: The sphere would look better using o1k than with o30 if the distance between triangle vertices using o30 were many pixels. A triangle edge would appear as a straight line and this would be most apparent along the limb (edge) of the sphere. With o1k these lines would be shorter, and so the curve would be better approximated. In contrast, if the entire sphere covered a small fraction of the window, then even using o30 the distance between vertices in a triangle would be only a pixel or two and so the edge of the sphere would appear properly curved either way.

Grading Note: Many answered that the situations differed in the use of surface normals for computing lighting. This is only partly correct. If triangle normals are used then the change in lighting from one triangle to another would be sudden making the triangle boundaries obvious. For that reason, o1k might look better since the triangles are smaller. In contrast, if sphere surface normals are used then the lighted color at points that are nearby but on different triangles would be almost the same (because being nearby there normals would be almost identical). So in this situation fewer vertices are needed for good appearance away from the sphere limb. However, the limb (edge) of the sphere would still look bad for o30 since it is constructed of relatively long straight lines.

Suppose that it takes much more time to render a sphere using o1k than with o30. The vertex, geometry, and fragment shader stages are running custom written code which could use some tuning.

☑ Which shaders should be tuned to potentially fix the performance problem? ☑ Which shaders be ignored? ☑ Explain.

The vertex shader is run once for each vertex. Since o1k has many more vertices the vertex shader could be responsible for the low performance. The same argument can be made for the geometry shader. In contrast, the fragment shader is run once per fragment and the number of fragments will be the same using o1k and o30 since the projected size of the sphere would be the same. For that reason there is no need to tune the fragment shader.

Name Solution

GPU Programming
EE 4702-1
Final Examination

Tuesday, 5 December 2017 12:30–14:30 CST

Problem 1		(15 pts)
Problem 2		(20 pts)
Problem 3		(30 pts)
Problem 4		(35 pts)
Exam Total		(100 pts)

Alias Walt.

Good Luck!

Problem 1: [15 pts] Appearing below is code based on Homework 5 (the tile cloud). First is code that launches the rendering passes, followed by the shaders. The shaders below were written to work with the rendering pass launched by `case 1`. In `case 3`, which is incomplete, the rendering pass input primitives are lines instead of triangles. Complete the `case 3` code and modify the shaders to work with your completed `case 3` code.

☑ Complete the `case 3` code. Use abbreviations such as `glV` (for `glVertex4f`), `glC`, and `glN`.

Solution appears below. Since a line has only two vertices, the five pieces of information, `color`, `pt_00`, `tact_pos`, `ax`, and `ay`, need to be sent using three attributes, `glColor`, `glNormal`, and `glVertex`, rather than the two attributes used in case 1.

```
case 1: {  pShader_Use use(s_hw05_tiles_1);
           glBegin(GL_TRIANGLES);
           for ( Tile* tile: tiles ) {  glColor4fv(tile->color);    glVertex4fv(tile->pt_00);
                                        glColor4fv(tile->tact_pos); glVertex4fv(tile->ax);
                                                                    glVertex4fv(tile->ay); }
           glEnd();  } break;
case 3: {  pShader_Use use(s_hw05_tiles_3);
           glBegin(GL_LINES);   //  <--- DON'T FORGET, LINES
           for ( Tile* tile: tiles ) {

             glColor4fv(tile->color);                        glVertex4fv(tile->pt_00);
             glColor4fv(tile->tact_pos); glNormal(tile->ax); glVertex4fv(tile->ay);

           }
           glEnd();  } break;
```

☑ Modify shader code (below) to work with `case 3` (above). Look at ☑ type of primitive (above), ☑ interface blocks, ☑ shader routines, and ☑ layout declarations.

Solution appears below.

```
out Data_to_GS { vec4 vertex_o;    vec4 color;       };

void vs_main_tiles_3() {  // Vertex shader routine.                 // CHANGE/ADD SOMETHING
  vertex_o = gl_Vertex;
  color = gl_Color;
  normal_o = gl_Normal;  // SOLUTION - Pass along third attribute.
}

// SOLUTION - Put normal_o in interface block.
in Data_to_GS { vec4 vertex_o;      vec4 color;   vec3 normal_o  } In[3];

// SOLUTION - Changed triangles to lines.
layout ( lines ) in;                                          // CHANGE/ADD ONE OF
layout ( triangle_strip, max_vertices = 4 ) out;                  // THE LAYOUT DECLARATIONS

void gs_main_tiles_3() {   // Geometry shader routine.             // CHANGE/ADD SEVERAL THINGS

  vec4 pt_00 = In[0].vertex_o;

  vec4 ax_o = vec4(In[1].normal_o,0);      // SOLUTION. Change to match new location.

  vec4 ay_o = vec4(In[1].vertex_o.xyz,0);  // SOLUTION. Change to new location.

  vec4 tact_pos = In[1].color;

  color = In[0].color;

  vec4 vtx_o[4];
  vtx_o[0] = pt_00;          vtx_o[1] = pt_00 + ax_o;
  vtx_o[2] = pt_00 + ay_o;   vtx_o[3] = pt_00 + ay_o + ax_o;
  // The code below does not need to be changed and so isn't shown.
```

Problem 2: [20 pts] Appearing below is the `render_tiles` routine from Homework 5.

(*a*) Estimate the amount of data sent from the CPU to the GPU for the rendering pass started by the code below. Use the following symbol: n, the number of tiles.

```
case 1: {
    pShader_Use use(s_hw05_tiles_1);
    glUniform2i(1, opt_tryout1, opt_tryout2);
    glUniform1i(2, light_state_get());
    glUniform1f(3, world_time);

    glBegin(GL_TRIANGLES);
    for ( Tile* tile: tiles ) {
        glColor4fv(tile->color);
        glVertex4fv(tile->pt_00);
        glColor4fv(tile->tact_pos);
        glVertex4fv(tile->ax);
        glVertex4fv(tile->ay);
      }
    glEnd();
  }
  break;
```

☑ Amount of data, in bytes:

Integer and float variables are each 4 bytes. There are a total of four variables sent as uniforms, for a total of $4 \times 4 = 16\,\mathrm{B}$. Note that uniforms are just sent once. Each `glColor` and `glVertex` call sends $4 \times 4 = 16\,\mathrm{B}$ ignoring overhead. For each tile $5 \times 16 = 80\,\mathrm{B}$ are sent. The total amount of data is $16\,\mathrm{B} + 80n\,\mathrm{B}$.

(*b*) The vertex shader below is used with the `case 1` code above. In terms of n, how much data is read by this vertex shader for a rendering pass?

```
void vs_main_tiles_1() {
  vertex_o = gl_Vertex;
  color = gl_Color;
}
```

☑ Total data read by shader above for a rendering pass:

An invocation of the vertex shader reads $2 \times 16 = 32\,\mathrm{B}$. Three vertices are sent down the rendering pipeline for each tile, and so the total amount of data read is $n \times 3 \times 32 = 96n\,\mathrm{B}$.

☑ Explain why the amount of data read by the vertex shader might be different than the amount of data sent from CPU to GPU when executing the `case 1` code.

The same vertex shader code is used for every invocation and it always reads two items. We know that the color is not set for the third vertex of each triangle, but the vertex shader code does not check to see if it is third (and it might not be possible) and so it reads two items every time. The host code however, does avoid a `glColor` call for one out of three vertices and that might reduce the amount of data sent.

Problem 2, continued: The rendering pass below provides the same data to the shaders as the one in the previous part, but there are significant differences.

```
#define TO_BO(name,num,update)                                                  \
  glBindBuffer(GL_ARRAY_BUFFER,bos_tiles[num]);                                 \
  if ( update ) glBufferData                                                    \
    (GL_ARRAY_BUFFER, name.size()*sizeof(name[0]), name.data(), GL_STREAM_DRAW); \
  glBindBufferBase(GL_SHADER_STORAGE_BUFFER,num,bos_tiles[num]);

    pShader_Use use(s_hw05_tiles_2);
    glUniform2i(1, opt_tryout1, opt_tryout2);
    glUniform1i(2, light_state_get());
    glUniform1f(3, world_time);

    TO_BO(pt_00,     1, pt_00_data_stale);
    TO_BO(ax,        2, axes_data_stale);
    TO_BO(ay,        3, axes_data_stale);
    TO_BO(color,     4, color_data_stale);
    TO_BO(tact_pos,  5, tact_data_stale);
    pt_00_data_stale = axes_data_stale = color_data_stale = tact_data_stale = false;

    glDrawArrays(GL_POINTS,0,tiles.size());
```

(*c*) Explain why the code above would be more efficient than the `case 1` code when rendering the very first frame. Also explain why the code above might be more efficient on the second frame than on the first, depending on how the tiles are used. *Note: The original exam did not clearly state that the second comparison was of two executions of the code above, the first on the first frame and the second on some later frame.*

☑ More efficient than `case 1` for the very first frame because:

It is inefficient to use the `glVertex` and related calls because they handle a small amount of data. Sending over all of the data as large buffer objects avoids this inefficiency.

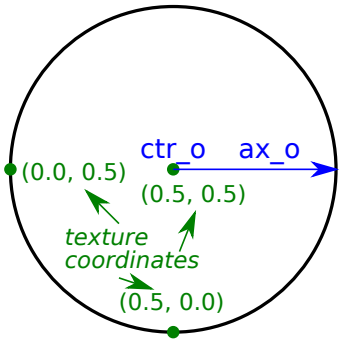
☑ Code above more efficient rendering second frame than first frame because ...

Only data that is changed needs to be re-sent from the CPU to the GPU.

☑ This efficiency can be assumed by use of variable such as ... because ...

Because of the variable names containing `stale`. That implies that only some data needs to be re-sent, the other data on the GPU is still fresh and so doesn't need to be replaced.

Problem 3: [30 pts] The code in this problem is similar to the Homework 5 tile code except that it will be used to render discs.



(*a*) Complete the geometry shader below so that it emits primitives for a disc (a filled-in circle) writing the geometry shader outputs shown. The provided shader code reads the disc center, `ctr_o`, disc normal, `nz_o`, and a vector to a point on the circumference, `ax_o` (see diagram), all in object space. The shader output includes `tcoord`, a texture coordinate. Assign this consistent with the texture coordinates shown on the diagram. Assume that the shader output primitive, a triangle fan, will work correctly even though it is not included in OpenGL Shading Language 4.5. A triangle fan makes the solution simpler. *Note: This triangle fan discussion was intentionally omitted from the original exam.*

☑ Emit primitives for the disc, make sure it's filled in.

☑ Don't forget to: ☑ Set `max_vertices`, ☑ set `normal_e`, ☑ set `tcoord`, and of course ☑ set `gl_Position`.

Solution appears on the next page.

Solution appears below.

```
out Data_to_FS { flat vec3 normal_e;  vec3 vertex_e;  vec2 tcoord; };
layout ( points ) in;

layout ( triangle_fan, max_vertices = slices + 1       ) out;  // SOLUTION

void gs_main_disc() {
  int vertex_id = In[0].vertex_id;
  vec4 ctr_o = ctrs[vertex_id];
  vec3 ax_o = axs[vertex_id].xyz;
  vec3 nz_o = nzs[vertex_id].xyz;

  const int slices = 10;
  const float pi = 3.1415926536;
  const float delta_theta = 2 * pi / slices;
// Abbrevs: glmvp, gl_ModelViewProjectionMatrix; glmv, gl_ModelViewMatrix; gln, gl_NormalMatrix

  // SOLUTION

  vec3 ay_o = cross(normalize(nz_o),ax_o);  // Compute y vector.


  // Compute the normal. Used for the entire disc.
  normal_e = gl_NormalMatrix * nz_o;

   // SOLUTION - Compute and emit attributes for the center coordinate.
  tcoord = vec2(0.5,0.5);
  gl_Position = gl_ModelViewProjectionMatrix * ctr_o;
  vertex_e = gl_ModelViewMatrix * ctr_o;
  EmitVertex();

  for ( int i=0; i<=slices; i++ )
    {
      float theta = i * delta_theta;    float costh = cos(theta), sinth = sin(theta);

      // SOLUTION - Compute attributes for point on circle and emit.
      vec4 pt_o = ctr_o + vec4( costh * ax_o + sinth * ay_o, 0 );
      tcoord = vec2( 0.5 + 0.5 * costh, 0.5 - 0.5 * sinth );
      vertex_e = gl_ModelViewMatrix * pt_o;
      gl_Position = gl_ModelViewProjectionMatrix * pt_o;
      EmitVertex();

    }
}
```

Problem 3, continued:

(*b*) Appearing below is the fragment shader for the code above. If variable `nevermind` were `true` the fragment shader would not write a fragment, but it's set to `false` in the code. (The `discard` keyword returns from the fragment shader without writing a fragment.)

The primitives emitted by the geometry shader (if solved correctly) will render a 10-sided polygon, which is not exactly a disc (circle). Modify the code so that it emits a perfect disc based on **the largest circle that can fit inside the polygon**. (For partial credit, a circle of radius 0.4 in texture coordinate units.) Use texture coordinates to determine whether a fragment is in the circle.

☑ Assign `radius` the correct value in terms of `slices`.

☑ Set `nevermind` so that a fragment is discarded if it's outside a radius-`radius` circle based on `tcoord`.

Solution appears below. Using texture coordinates makes it very easy to find the distance from the fragment to the disc center.

```
void fs_main_disc() {
  const int slices = 10;
  const float pi = 3.1415926535;
  const float delta_theta = 2 * pi / slices;

  // SOLUTION -- Set radius based on texture coordinates in diagram.
  float radius = 0.5 * cos(pi/slices);
  vec2 disc_center_t = vec2(0.5,0.5);

  // SOLUTION -- Find distance to center using texture coordinates.
  const bool nevermind = distance(tcoord,disc_center_t) > radius;


  vec4 texel = texture(tex_unit_0,tcoord);   // NO NEED TO CHANGE THIS CODE.
  vec4 color = colors[vertex_id];
  gl_FragColor = texel * generic_lighting(vertex_e, color, normal_e);
  gl_FragDepth = gl_FragCoord.z;
}
```

(*c*) The inputs to fragment shader `fs_main_disc` appear below. Explain the implications of removing the `flat` qualifier as described below.

```
in Data_to_FS  {  flat vec3 normal_e;   flat int vertex_id;
                  vec3 vertex_e;        vec2 tcoord;              };
```

☑ Explain impact on ☑ correctness and ☑ efficiency if `flat` were removed from `normal_e` and `vertex_id`.

Removing `flat` from `normal_e` would be wasteful but the code would still work correctly. It would be wasteful because the normal will be the same on every point on the disc and so the effort to interpolate it (between values at the three vertices) is wasted. On the other hand, `vertex_id` is an integer and OpenGL Shading Language won't interpolate integers, it's an error to even try

☑ Explain impact on ☑ correctness and ☑ efficiency if `flat` were added to `vertex_e` and `tcoord`.

Adding `flat` would reduce the amount of computation, however that doesn't mean anything if the results are wrong. Adding `flat` to `vertex_e` would affect the lighted color of the fragment, which would be noticeable if the disc were near a light source. Adding `flat` to `tcoord` would result in the texture not being applied and if the part-b code were used, would result in a polygon rather than a disc.

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

(*a*) Explain how the depth (z-buffer) test is used. Provide a diagram of an example in which sorting primitives by eye distance makes the depth test unnecessary, and another example diagram in which even with sorting a depth test is necessary for proper rendering.

☑ Explain how depth test used.

The depth test is applied just before a fragment is written to the frame buffer. The fragment's z value is compared to the z value of the corresponding pixel. Usually, if the fragment's z value is closer to the eye it gets written, otherwise it is abandoned.

☐ Diagram illustrating example in which sorting makes depth test unnecessary. *Hint: Example can have two primitives.*

☐ Diagram illustrating Example in which depth test necessary even with sorting. *Hint: Example can have three primitives.*

(*b*) Appearing below are sample uses of two procedures related to the stencil buffer. Explain what each one does in general (not necessarily in the example).

```
glStencilFunc(GL_EQUAL,4,-1);
glStencilOp(GL_REPLACE,GL_KEEP,GL_KEEP);
```

☑ The `glStencilFunc` procedure is used to ….

… specify the type of stencil test to perform. The value in the stencil buffer location corresponding to a fragment is compared to the reference value (the second parameter, 4 above). The type of comparison is specified by the first argument. In the example above the test passes if the stencil value equals 4.

☑ The `glStencilOp` procedure is used to ….

… specify under what conditions to update the stencil buffer and how to update it. The argument `GL_REPLACE` indicates that a new value should be written to the stencil buffer. The value to be written is the second argument to the most recent `glStencilFunc` call. Argument `GL_KEEP` indicates that the stencil buffer should not be changed. The first argument is used if the stencil test fails. The second if the depth test fails, and the third if the depth test passes.

(*c*) In general, why doesn't it make sense to access a texture in a vertex shader?

☑ Bad idea to access a texture in a vertex shader because ...

Accessing a texture returns a texel. In typical use a texel is associated with a frame buffer location (a pixel). Each fragment shader invocation is associated with a frame buffer location, but a vertex shader is not. A vertex shader invocation typically is associated with a coordinate that maps to a frame buffer location. But that vertex usually is part of a primitive which covers many locations, and so the texel would only apply to one of those locations.

(*d*) How is eye space defined? Describe the transformations that need to be applied to map from object space to eye space. Illustrate your answer using a sample scene.

☑ Defining features of eye space are ....

In eye space the viewer's eye is at the origin and facing in the $-z$ direction.

☑ In the following scene to transform from object to eye space the modelview matrix is used, which ...

... the product of a rotation and translation. The translation moves the eye to the origin, and the rotation moves the users monitor to face the $-z$ direction.

(*e*) An NVIDIA GPU has 10 SMs. Consider a kernel which evenly divides work among its threads, the usual assumption made in class. Further assume that there is a large amount of work. Let $t(G)$ denote the execution time when the kernel is launched with G blocks. In all cases the block size is 1024 threads.

Let $a = t(10)$, the time when launched with 10 blocks on the 10-SM GPU.

☑ Find an expression for $t(5)$ in terms of a.

$t(5) = 2a$. It takes twice as long because each block has twice as much work to do.

☑ Find an expression for $t(15)$ in terms of a.

$t(15) = 2\frac{10}{15}a$. With 15 blocks each block performs $\frac{10}{15}$ times as much work as for the $a = t(10)$ case. Since there are 10 SMs some SMs will have to compute two blocks. Since the block sizes are large the times for each of the two blocks are added. (If the block sizes were small, say 32 threads, then the time for two blocks on an SM would be about the same as the time for one.)

☑ Find an expression for $t(20)$ in terms of a.

$t(20) = 2\frac{10}{20}a = a$. The reasoning is the same as the previous case, but this time *all* SMs compute two blocks.

(*f*) Draw a sketch showing the shadow volume corresponding to a triangle. Include the triangle, the light source, some object in the shadow and some object seen through the shadow.

☐ Show: ☐ the light, ☐ triangle, ☐ shadow volume for the triangle, ☐ shadowed object, ☐ object seen through shadow.

26 Fall 2016 Solutions

Name Solution

GPU Programming
EE 4702-1
Midterm Examination

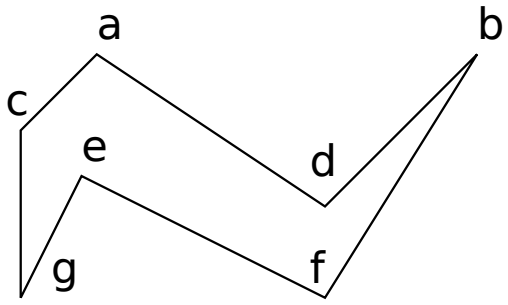
Friday, 28 October 2016 14:30–15:20 CDT

Problem 1		(20 pts)
Problem 2		(30 pts)
Problem 3		(15 pts)
Problem 4		(35 pts)
Exam Total		(100 pts)

Alias It Begins

Good Luck!

Problem 1: [20 pts] Appearing below is a figure with labeled vertices. In the code below `pa`, `pb`, ... are initialized to the coordinates of the corresponding vertices. Two arrays are declared, `coords_wrong` and `coords`.




```
void World::render_p1(pCoor p1, pCoor p2, pCoor p3) {
  pCoor pa(2,4,0);
  pCoor pb(7,4,0);
  pCoor pc(1,3,0);
  pCoor pd(5,2,0);
  pCoor pe(1.8,2.4,0);
  pCoor pf(5,0.8,0);
  pCoor pg(1,0.8,0);

  pCoor coords_wrong[] = { pa, pb, pc, pd, pe, pf, pg };

  /// SOLUTION – Part a
  pCoor coords[] = { pc, pg, pa, pe, pd, pf, pb };

  /// SOLUTION – Problem 1b. (Problem 1c also solves 1b.)
#if 0
  pVect vg1 = p1 - pg;
  pMatrix_Translate move(vg1);
  pMatrix m = move;
#endif

  glMatrixMode(GL_MODELVIEW);
  glPushMatrix();

  /// SOLUTION – Problem 1c. Step 1: Transform to a local space.
  //
  pMatrix_Translate to_origin(-pg);
  pMatrix_Scale scale( 1/(pb.x-pc.x), 1/(pc.y-pg.y), 1);

  // SOLUTION -- Problem 1c. Step 2: Rotate, scale, and translate for p1, p2, p3.
  //
  pVect ay(p1,p2);
  pVect v23(p2,p3);
  pVect az = cross(v23,ay);
  pNorm axn = cross(ay,az);
  float width = dot(axn,v23);
  pVect ax = width * axn;
  pMatrix_Cols rot_glo(ax,ay,az);
  pMatrix_Translate to_p1(p1);

  pMatrix m = to_p1 * rot_glo * scale * to_origin;
  glMultTransposeMatrixf(m);

  glBegin(GL_TRIANGLE_STRIP);
  for ( int i=0; i<coords.size(); i++ ) glVertex3fv( coords[i] );
  glEnd();    glPopMatrix();  }
```

(*a*) Initialize array `coords` so that the figure is rendered properly.

☑ Put `pa` ... `pg` into the `coords` initialization above in the correct order.

Solution appears above. Note that the primitive type is a triangle strip, so each vertex appears once.

(*b*) Complete the code for the modelview matrix so that the whole figure is moved such that vertex `g` is at coordinate `p1`. (Note: the problem below also solves this problem.)

☑ Assign the correct value for `m` in the code above and make any other necessary changes.

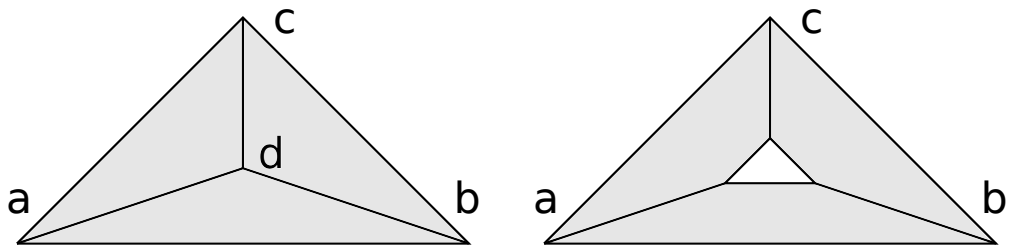
Solution appears above. A translation matrix is constructed using a vector from `pg` to `p1`. This code is guarded by an `#if 0` because part c also solves this problem.

(*c*) Set the modelview matrix so that the figure is rendered such that vertex `g` is at coordinate `p1`, vertex `c` is at `p2`, and all points are in the plane formed by `p1`, `p2`, and `p3`. This will move and rotate the figure, but won't change its shape. The matrix constructors `pMatrix_Rows rot1(v1,v2,v3)` and `pMatrix_Cols rot2(v1,v2,v3)` may come in handy. *Note: In the original version of the exam* `b` *was to be moved to* `p3`*.*

☑ Assign `m` to move the entire figure so that `g` is at `p1`, `c` is at `p2`, and `b` is at `p3`.

The solution appears above. The first step is to transform the points to a local space in which `pg` is at the origin, the distance from `pc` to `pg` is 1 unit along the y axis, and the distance from `pc` to `pb` is 1 unit along the x axis. The second step is to rotate the coordinates using the vector from `p1` to `p2` as the new y axis, etc. The final step is to translate the figure to `p1`.

Problem 2: [30 pts] Illustrated below are two possible outputs of a geometry shader processing input primitive *abc*. The geometry shader code shown below, when completed, renders the figure on the left (this page's problem) or right (next page's problem).




```
/// SOLUTION – Problem 2a – Change max_vertex from 3 to 8.
layout ( triangle_strip, max_vertices = 8 ) out;

void gs_main_1() {
  vec3 pt_sum = vec3(0);    vec3 n_sum = vec3(0);
  for ( int i=0; i<3; i++ ) { pt_sum += In[i].vertex_e.xyz; n_sum += In[i].normal_e; }
  vec4 center_e = vec4( pt_sum/3, 1 );
  vec3 center_norm_e = n_sum/3;

  /// SOLUTION – Problem 2b
  //  Transform coordinate of center point from eye space to clip space.
  vec4 center_c = gl_ProjectionMatrix * center_e;

  for ( int i=0; i<=3; i++ ) {
      int idx = i%3;
      vertex_e = center_e;   normal_e = center_norm_e;

      /// SOLUTION – Problem 2b
      //  Assign clip-space coordinate to geometry shader output.
      gl_Position = center_c;

      EmitVertex();

      normal_e = In[idx].normal_e;   vertex_e = In[idx].vertex_e;
      gl_Position = In[idx].gl_Position;
      EmitVertex(); }
  EndPrimitive(); }
```

(*a*) Fix the layout declaration and modify the code so that `gl_Position` is assigned a correct value for the center point (d in the diagram). Assume that `center_e` is already correct. *Hint: Don't just look at the figure for the layout declaration fix.*.

☑ Fix the layout declaration. ☑ Assign `gl_Position` correctly for the center point.

Solution appears above. The loop iterates four times and `EmitVertex` appears twice in the loop body, and so eight vertices are emitted.

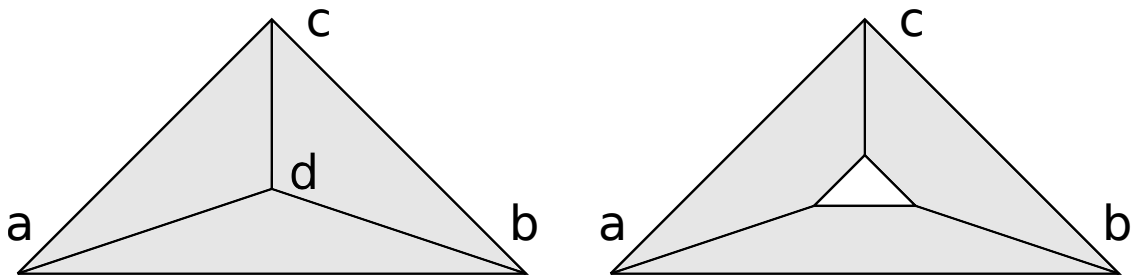
(*b*) Suppose that the goal is to split abc into three triangles that each have $\frac{1}{3}$ the fragments that abc would have had. Does the code above (assuming `gl_Position` is set correctly) achieve that goal? If not, briefly explain how to modify the code so that there will be three triangles with $\frac{1}{3}$ the vertices each.

☑ Are fragments equally split? If no, explain how to fix.

Short answer: They are not necessarily equally split because the number of fragments generated for a triangle depends upon its clip-space coordinates, and so it doesn't matter that abc has been split into three triangles of equal area in eye space (plus three triangles of zero area). It can be fixed by computing the center point using the clip space coordinates of a, b, and c.

More details: The shader above emits six triangles: three cover zero fragments (such as triangle dcd) while the number of fragments covered by the remaining three triangles depends on their clip space coordinates (computed using the projection matrix) and whether they are within the view volume. Even assuming that the entire triangle were in the view volume, the number of fragments could vary, for example, when the eye is closer to vertex a than b or c, triangle bdc will have fewer fragments than acd. To fix this problem the center point should be chosen based on clip space coordinates rather than eye space coordinates. Then the eye-space coordinate of the center point would be computed using the inverse of the projection matrix.

Problem 2, continued:




```
/// SOLUTION – Problem 2c
layout ( triangle_strip, max_vertices = 8 ) out;

void gs_main_1() {
  vec3 pt_sum = vec3(0);    vec3 n_sum = vec3(0);
  for ( int i=0; i<3; i++ ) { pt_sum += In[i].vertex_e.xyz; n_sum += In[i].normal_e; }
  vec4 center_e = vec4( pt_sum/3, 1 );
  vec3 center_norm_e = n_sum/3;




  for ( int i=0; i<=3; i++ ) {
      int idx = i%3;



      /// SOLUTION – Problem 2c
      vec3 ctr_to_vtx = In[idx].vertex_e.xyz - center_e.xyz;
      vertex_e = vec4( center_e + 0.5 * ctr_to_vtx, 1 );
      normal_e = ( center_norm_e + In[idx].normal_e ) * 0.5;
      gl_Position = gl_ProjectionMatrix * vertex_e;
      EmitVertex();

      normal_e = In[idx].normal_e;   vertex_e = In[idx].vertex_e;
      gl_Position = In[idx].gl_Position;
      EmitVertex(); }
  EndPrimitive(); }
```

(*c*) Modify the geometry shader so that it splits the triangle into the shape on the right, which is like the original triangle with a triangle-shaped hole in the center. The rendered primitives should only cover the area shown in gray. The exact area of the hole is not important, as long as it's not zero.

☑ Modify shader to render hole as shown in the shape on the right.

Solution appears above. Rather than emit the center point, the code emits a vertex halfway between the center point and a triangle vertex. The normal is also scaled.

Problem 3: [15 pts] The incomplete code below is supposed to test whether pairs of balls *interpenetrate*, meaning that there is a volume of space that both balls occupy, and if so apply a separation force. The separation force works like an ideal spring pushing apart the balls. *Note: in the original exam the phrase defining interpenetration read "they occupy the same space".*

```
for ( int i=0; i<chain_length; i++ ) for ( int j=i+1; j<chain_length; j++ )
  {
    Ball* const ball_i = &balls[i];
    Ball* const ball_j = &balls[j];
    pCoor pos_i = ball_i->position;
    pCoor pos_j = ball_j->position;
    float rad_i = ball_i->radius;
    float rad_j = ball_j->radius;

    /// SOLUTION – Part a
    pNorm pos_i_to_j(pos_i,pos_j);
    float ip_dist = rad_i + rad_j - pos_i_to_j.magnitude;
    bool interpenetrating = ip_dist > 0 ;

    if ( ! interpenetrating ) continue;

    /// SOLUTION – Part b
    pVect sep_force = ip_dist * sep_spring_constant * pos_i_to_j;

    ball_i->force += -sep_force;
    ball_j->force += sep_force;
  }
```

(*a*) Write code to determine whether the two balls are interpenetrating, and assign the result to `interpenetrating`.

☑ Set `interpenetrating` to true if balls are intersecting.

Solution appears above. The code computes an interpenetration distance, `ip_dist`, the length of the overlapping region on the line between their centers. The balls are intersecting if this quantity is positive.

(*b*) Write code to add on a separation forces to the balls. Use `sep_spring_constant` for the spring constant.

☑ Write code to add the separation force to balls' `force` members. ☑ Don't forget that force is a vector quantity.

Solution appears above. Apply a force along the line connecting their centers. The force is proportional to the amount of intersection.

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

(*a*) Convert homogeneous coordinate $\begin{bmatrix}1\\2\\3\\4\end{bmatrix}$ into an ordinary Cartesian coordinate.

☑ Cartesian coordinate:

The Cartesian coordinate is $\begin{bmatrix}1/4\\1/2\\3/4\end{bmatrix}$.

(*b*) Explain why `gl_Vertex`, the object-space vertex coordinate vertex shader input, is only in the compatibility profile (meaning that it's not really needed) but `gl_Position`, the output of the vertex and geometry shader stages is in the core profile (meaning that it is really needed).

☑ Input `gl_Vertex` **is not** really needed because:

Because `gl_Vertex` is only used by shader code, not by fixed functionality. So if the vertex shader code needs the object-space vertex coordinate, then that can be declared as an input (with whatever name the coder wants). There is no need for a pre-defined name for the object space vertex coordinate because none of the fixed functionality uses is.

☑ Output `gl_Position` **is** really needed because:

Because `gl_Position`, the vertex's clip-space coordinate is used by the fixed functionality, in particular by the rasterizer to perform clipping and to generate fragments. Therefore, the name of the shader output must be a name known to the rasterizer (something pre-defined) and not something made up by the person who wrote the shaders.

(*c*) The modelview matrix is predefined as a uniform variable. What would be the disadvantage of making it a vertex shader input?

☑ Disadvantage of making modelview a vertex shader input.

As a vertex shader input it would have to be copied and moved so that each shader invocation had its own value. And remember that the value consists of 16 floats. Consuming perhaps communication bandwidth and time. That's a waste because the value would be the same for an entire rendering pass. Furthermore, uniforms are probably placed in a higher speed memory than shader inputs.

(*d*) The code below updates a buffer object whenever its old contents becomes outdated. Two lines have been accidentally commented out. What are the consequences of each.

```
//      if ( gpu_buffer_stale )  // LINE A, ACCIDENTALLY Commented out.
   {
//          if ( !gpu_buffer )   // LINE B, ACCIDENTALLY Commented out.
              glGenBuffers(1,&gpu_buffer);
            glBindBuffer(GL_ARRAY_BUFFER, gpu_buffer);
            glBufferData
              (GL_ARRAY_BUFFER, coords_size*sizeof(float),
               coords, GL_STATIC_DRAW);

            glBindBuffer(GL_ARRAY_BUFFER, 0);
            gpu_buffer_stale = false;
          }
```

☑ Consequences of LINE A commented out, but LINE B not commented out, in particular on ☑ performance and ☑ stability.

The code would execute every time, not just when the data was stale. This would hurt performance because of the need to send more data from the CPU to the GPU. It would not affect stability.

☑ Consequences of LINE B commented out, but LINE A not commented out, in particular on ☑ performance and ☑ stability..

A new buffer object would be created each time, but without discarding the old buffer object. This would consume more and more memory, slowing execution and ultimately causing an execution error, or if the programmer was careful, an apology and an early exit.

(*e*) What are the typical operations performed in a fragment shader? What must the fragment shader do?

☑ Typical operations for fragment shader.

Fragment shaders typically read texels and blend them with the incoming color. Fragment shaders can also perform lighting.

☑ Mandatory operation for fragment shader.

The fragment shader must write the final color and depth (z value) to the fragment shader outputs.

Name Solution

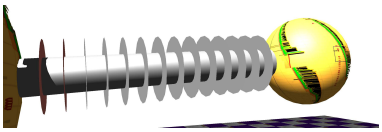
GPU Programming
EE 4702-1
Final Examination

Monday, 5 December 2016 17:30–19:30 CST

Problem 1		(20 pts)
Problem 2		(20 pts)
Problem 3		(15 pts)
Problem 4		(20 pts)
Problem 5		(25 pts)
Exam Total		(100 pts)

Alias Methane?

Good Luck!



Problem 1: [20 pts] The geometry shader code below is based on the solution to Homework 6 Problem 2 in which a curved link is rendered by having the vertex shader compute points and vectors related to a curve and by having the geometry shader, whose input primitive type is a line strip, emitting a cylinder between the locations described by its two input vertices. Input `In[].t` is the position along the curve, with $0 \leq t \leq 1$.

Add code to the shader so that it draws rings around the cylinder. The inner radius should be the same as the cylinder, the outer radius should be twice the cylinder radius. Do not emit a ring at the link endpoints (the parts that touch the spheres). See the illustration to the upper right.

☑ Complete the code so that it draws the rings. ☑ Can use suggested and other reasonable abbreviations.

☑ Set `normal_e` properly don't forget to set ☑ `vertex_e` and ☑ `gl_Position`.

☑ The code should be reasonably efficient. ☑ Don't overlap primitives.

Solution appears below. The solution code is in the repo in directory `hw/gpup/2016/fe`.

```
void gs_main_2() {
  const float rad = tex_rad[In[1].iid].z,  sides = sides_rad.x;
  for ( int j=0; j<=sides; j++ ) {
      const float theta = j * ( 2 * M_PI / sides );
      vec3 vect0 = cos(theta) * In[0].norm_e + sin(theta) * In[0].binorm_e;
      vec3 vect1 = cos(theta) * In[1].norm_e + sin(theta) * In[1].binorm_e;

      normal_e = vect1;    vertex_e = In[1].ctr_e + vec4( rad * vect1, 0);
      gl_Position = gl_ProjectionMatrix * vertex_e;  // SUGGESTED ABBREV: glP = glPM * v_e;
      EmitVertex();
      normal_e = vect0;    vertex_e = In[0].ctr_e + vec4(rad * vect0,0);
      gl_Position = gl_ProjectionMatrix * vertex_e;
      EmitVertex();   }
  EndPrimitive();
  float t0 = In[0].t,      t1 = In[1].t;       // Use if needed.
  vec4  c0 = In[0].ctr_e,  c1 = In[1].ctr_e;   // Use if needed.

  if ( In[0].t == 0 ) return;  // SOLUTION: Don't render ring at 1st segment.
  normal_e = cross(In[0].norm_e,In[0].binorm_e);  // SOLUTION: Normal is cross of axes.
  float rad2 = rad * 2;        // SOLUTION: Outer radius of ring. (Inner radius is rad)

  for ( int j=0; j<=sides; j++ ) {
      const float theta = j * ( 2 * M_PI / sides );
      vec3 v0 = cos(theta) * In[0].norm_e + sin(theta) * In[0].binorm_e; // Use if needed.
      vec3 v1 = cos(theta) * In[1].norm_e + sin(theta) * In[1].binorm_e; // Use if needed.

      vertex_e = In[0].ctr_e + vec4(rad * v0,0);     // SOLUTION: Inner edge of ring.
      gl_Position = gl_ProjectionMatrix * vertex_e;  // SOLUTION
      EmitVertex();                                  // SOLUTION

      vertex_e = In[0].ctr_e + vec4(rad2 * v0,0);    // SOLUTION: Outer edge of ring.
      gl_Position = gl_ProjectionMatrix * vertex_e;  // SOLUTION
      EmitVertex();                                  // SOLUTION

    }
}
```

Problem 2: [20 pts] Appearing below is a typical vertex shader.

```
in vec2 gl_MultiTexCoord0;  // Declare size.
void vs_main() {
  gl_Position = gl_ModelViewProjectionMatrix * gl_Vertex;
  vertex_e = gl_ModelViewMatrix * gl_Vertex;
  normal_e = normalize(gl_NormalMatrix * gl_Normal);
  gl_TexCoord[0] = gl_MultiTexCoord0;
  color = gl_Color;
}
```

(*a*) Suppose that the shader above was used in a rendering pass of v vertices. No buffer objects were used and all shader inputs were sourced from client arrays. Compute the amount of data sent from the CPU to the GPU for the rendering pass.

☑ CPU to GPU data for a rendering pass of v vertices counting shader inputs is:

Short Answer: counting `gl_Vertex`, `gl_Normal`, `gl_MultiTexCoord0`, and `gl_Color` the amount of data is

$$4(4+3+2+4)v\,\mathrm{B} = 52v\,\mathrm{B}.$$

☑ CPU to GPU data for a rendering pass of v vertices counting uniform variables is:

Short Answer: counting `gl_ModelViewProjectionMatrix`, `gl_ModelViewMatrix`, and `gl_NormalMatrix` the amount of data is $4(4^2+4^2+3^2)\,\mathrm{B} = 164\,\mathrm{B}$.

Explanation: Uniforms are sent once.

Problem 2, continued: The vertex shader below is used for an instanced draw of spheres. The vertex shader inputs are sourced from buffer objects and other buffer objects are accessed directly using the instance ID.

```
layout ( binding = 1 ) buffer sr { mat4 sphere_transform[]; };
layout ( binding = 3 ) buffer sc { vec4 sphere_color[]; };
void vs_main_instances_sphere() {
  mat4 transform = sphere_transform[gl_InstanceID INP];
  vec3 ctr = transform[3].xyz;          // Extract coordinate of sphere center.
  float radius = transform[3][3];       // Extract sphere radius.
  mat3 rot = mat3(transform);           // Extract rotation matrix.

  vec3 normal_o = rot * gl_Vertex.xyz INP;
  vec4 vertex_o = vec4( ctr + radius * normal_o, 1 );

  color = sphere_color[gl_InstanceID INP];
  gl_Position = gl_ModelViewProjectionMatrix UFM * vertex_o;
  vertex_e = gl_ModelViewMatrix UFM * vertex_o;
  normal_e = normalize(gl_NormalMatrix UFM * normal_o );
  gl_TexCoord[0] = gl_MultiTexCoord0 INP;  }
```

(*b*) In the code above label the vertex shader inputs and uniform variables.

☑ Label vertex shader inputs with a `I` and uniform variables with a `U`.

Solution appears above in blue. To make the labels easier to find vertex shader inputs are labeled INP and uniforms are labeled UFM.

(*c*) Compute the amount of data sent from the CPU to the GPU for an instanced draw rendering pass in which n instances (spheres) are rendered, each consisting of v vertices. Assume that initially all buffer objects are on the CPU. Note that `mat4` is a 4×4 matrix of floats.

☑ Amount of CPU to GPU data for n instances of v vertices, accounting for ☑ uniforms, ☑ buffer objects, ☑ and anything else that really needs to be moved.

Short Answer: $164\,\mathrm{B}$ for uniforms (see part a), $n(4^3+4^2)\,\mathrm{B} = 80n\,\mathrm{B}$ for bound buffer objects, and $v(4\times4+4\times2)\,\mathrm{B} = 24v\,\mathrm{B}$ for shader inputs.

Explanation: The vertex shader inputs are `gl_InstanceID`, `gl_Vertex`, `gl_MultiTexCoord0`. The size of a `gl_Vertex` is $4 \times 4\,\mathrm{B}$ and assuming a 2-component texture coordinate the size of `gl_MultiTexCoord0` is $4 \times 2\,\mathrm{B}$. It is reasonable to assume that the data for `gl_Vertex` and `gl_MultiTexCoord0` were put in buffer objects and sent to the GPU. The size of the two buffer objects would be $v(4\times4+4\times2)\,\mathrm{B} = 24v\,\mathrm{B}$. Input `gl_InstanceID` is just an ID number created by the GPU software, it's not something sent from the CPU to the GPU, and so zero data is sent for `gl_InstanceID`. (That is, it would be silly to send an array of values $0, 1, \ldots, n-1$ from the CPU to the GPU, the GPU could generate such values with a simple loop. All that's needed is the number of instances, n.)

The shader accesses two bound buffer objects, `sphere_transform` and `sphere_color`. Based on the layout declaration each element of `sphere_transform` is $4 \times 4 \times 4\,\mathrm{B}$ and each element of `sphere_color` is $4 \times 4\,\mathrm{B}$. Since these arrays are indexed using `gl_InstanceID` it is reasonable to assume that they each have n elements. So the total amount of data for the bound buffer objects is $n(4^3 + 4^2)\,\mathrm{B} = 80n\,\mathrm{B}$.

The shader accesses three uniforms, the same uniforms as are accessed in part a. So their total data is $164\,\mathrm{B}$.

Problem 3: [15 pts] Consider our well written CUDA example (slightly simplified):

```
__global__ void kmain_efficient() {
  const int tid = threadIdx.x + blockIdx.x * blockDim.x;
  const int n_threads = blockDim.x * gridDim.x;

  // Note: There's no need to understand this code. Just remember
  // that each thread does about the same amount of work.

  for ( int h=tid; h<d_app.array_size; h += n_threads ) {
      float4 p = d_app.d_in[h];
      d_app.d_out[h] = dot(p,p);
    }
}
```

Let n denote the value of `array_size`, and let S denote the number of streaming multiprocessors (abbreviated SMs and sometimes MPs). Assume that n is large so that there's plenty of work to do for each thread.

(a) Determine a launch configuration in terms of n and S that uses the minimum number of blocks needed to maximize warp (or thread) occupancy on the SMs. Do this for a device with a block size limit of 1024 threads, an SM limit of 2048 threads, and an SM limit of 16 blocks. *Hint: There's no need to use both n and S.* Use B for the block size and G for the number of blocks (grid size).

☑ Block size $B =$ 1024

☑ Explain.

Short Answer: Use the maximum block size to minimize the number of blocks.

To maximize warp (thread) occupancy we need to have 2048 threads per SM. Since we need to *minimize* the number of blocks, we should make the blocks as large as possible. The problem states that the maximum block size is 1024 threads, so that's the size used.

☑ Grid size (number of blocks) $G = 2S$

☑ Explain.

It takes two 1024-thread blocks per SM to achieve full occupancy, so for all S SM's the number of blocks needed is $2S$.

(b) Let B and G be a solution to the problem above (meaning a good choice for B and G for S SMs) and let t_o be the execution time in a kernel launch of that configuration. Estimate the execution time for the configurations below. *Hint: Remember that G was chosen to maximize the number of warps per SM.*

☑ Estimate the execution time for a launch of G blocks of size $B - 1$. ☑ Explain.

Short Answer: assuming execution time is limited by computation, scale up the time: $\frac{B}{B-1}t_o$. Since the code above is bandwidth-limited the time would likely still be t_0 since the same amount of data is read.

Explanation: threads are assigned to hardware in groups of 32 (in NVIDIA devices so far) called warps. If a block has 1023 threads there will be 31 warps with 32 threads, and one warp with 31 threads. There is no way to give the hardware that would be used by the 1024'th thread to the 1023 remaining threads so the remaining threads work as fast as they would have before, but now they have slightly more work, $\frac{B}{B-1}$ times as much as before. If execution time were determined by the amount of work per thread execution would take $\frac{B}{B-1}$ times longer.

☑ Estimate the execution time for a launch of $G - 1$ blocks of size B. ☑ Explain.

Note: This answer went beyond what was covered in 2017. Short Answer: assuming execution time is limited by computation and 2048 threads were needed to hide latency, scale up the time: $\frac{2S}{2S-1}t_o$. Since the code above is bandwidth-limited the time would likely still be t_0 since the same amount of data is read. Either way the execution time doesn't change much.

☑ Estimate the execution time for a launch of $G+1$ blocks of size B. ☑ Explain.

Short Answer: $2t_0\frac{2S}{2S+1}$ because the one extra block must wait for the first $2S$ blocks to finish.

Explanation: Since $G = 2S$ blocks fully occupy a GPU, in a launch of $2S+1$ blocks the first $2S$ blocks first run, when they are finished the one remaining block starts running. Even though that one last block has the whole GPU to itself, it will run no faster (under certain assumptions) than if it were running with others. The amount of work done by each block in a $2S+1$-block launch in comparison to a $2S$-block launch is $\frac{2S}{2S+1}$, slightly less work. The first group of $2S$ blocks finishes in $\frac{2S}{2S+1}t_0$ time, and then the last block starts and it also takes $\frac{2S}{2S+1}t_0$. Note that this one extra block made things alot worse.

Problem 4: [20 pts] Consider the CUDA kernel below. Let n denote the value of `array_size`, B the block size, and G the number of blocks.

```
__global__ void prob_x(float2 *d_in, float *d_out) {
  const int tid = threadIdx.x + blockIdx.x * blockDim.x;
  const int n_threads = blockDim.x * gridDim.x;

  for ( int h=tid; h<d_app.array_size-1; h += n_threads ) {
      float2 p = d_in[h];
      float2 q = d_in[h+1];
      d_out[h] = dot(p,q);
    }
}
```

(a) Compute how much data is read from global memory during the execution of the kernel. Note that each element `d_in[i]` is accessed by two threads.

☑ Amount of data read:

☑ Explain.

About $2n2 \times 4\,\mathrm{B} = 16n\,\mathrm{B}$, twice the size of the array.

In a system with a cache the data read by thread τ in the `d_in[h]` access would be placed in a cache where it would be found by thread $\tau - 1$ when it performs the `d_in[h+1]` accesses. (Or, $\tau - 1$ might bring in data for τ.) The total amount of data would be $8n\,\mathrm{B}$ (each element is $2 \times 4\,\mathrm{B}$).

But, recent GPU generations (Kepler to Maxwell) don't automatically cache data, and even when they do the chances of a hit are smaller. For that reason both the `d_in[h]` and the `d_in[h+1]` accesses would bring in data from global memory (or maybe the L2 cache), doubling the amount of data read: $16n\,\mathrm{B}$.

What can be done about it? Shared memory could be used so that thread τ could read data loaded by thread $\tau + 1$. Or, the code could be written so that the accesses to `d_in` use the read-only cache (available starting at Kepler 3.5).

(*b*) The kernel below includes a declaration for shared memory intended to fix the inefficiency in the kernel above. Complete the kernel.

☑ Use shared memory to avoid waste.

```
__global__ void prob_x2(float2 *d_in, float *d_out) {
  const int tid = threadIdx.x + blockIdx.x * blockDim.x;
  const int n_threads = blockDim.x * gridDim.x;

  __shared__ float2 sm[1025];

  for ( int h=tid; h<d_app.array_size; h += n_threads ) {
      float2 p = d_in[h];

      // SOLUTION
      __syncthreads();
      sm[threadIdx.x] = p;
      if ( threadIdx.x == blockDim.x - 1 ) sm[threadIdx.x+1] = d_in[h+1];
      __syncthreads();

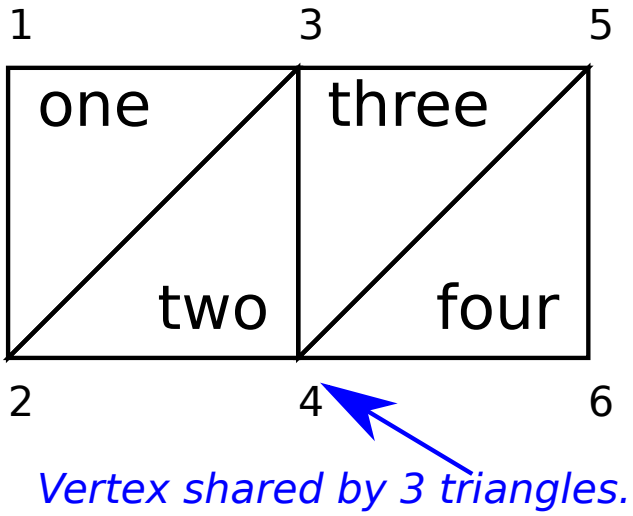
      float2 q = sm[threadIdx.x+1];  // SOLUTION
      d_out[h] = dot(p,q);
    }
}
```

Problem 5: [25 pts] Answer each question below.

(*a*) Show a diagram illustrating the relationship between vertices and primitives in a triangle strip. Number both the vertices and the triangles in the correct order. Show at least three triangles.

☑ Diagram with number vertices and triangles illustrating a triangle strip.

Solution appears below.



(*b*) Describe what is tested in the stencil test and what the stencil test might be used for.

☑ A stencil test checks:

The stencil test, applied to a fragment, compares a reference value provided in the OpenGL `glStencilFunc` call with the value in the stencil buffer location corresponding to the fragment. The stencil buffer is one of several frame buffer layers recognized by OpenGL. Others include the color buffers (which connect to a display) and the depth buffer.

☑ Typical use for a stencil test:

A stencil test might be used for a mirror effect. The stencil buffer is written in a pass where the primitive corresponds to mirror locations. Then in a subsequent pass other objects are rendered with a transform that maps their coordinates to their reflected location. The stencil test will only allow fragments from these objects to be written in the location at which there are mirrors.

(*c*) Interpolation qualifiers, `flat`, `noperspective`, and `smooth`, are used for the inputs of which rendering pipeline stage? What do they do?

☑ They appear at the input to: the fragment shader.

☑ They affect the qualified variable by:

Determining how to combine the values of the variable at the primitives vertices. For `flat` the value provided to the fragment shader is the value emitted by the provoking (last) vertex of the primitive. For `noperspective`, the value is linearly interpolated between the values at the vertices based on clip space coordinates. For `smooth`, the value is linearly interpolated base on object space coordinates (effectively). The `noperspective` interpolation is not geometrically accurate but is less costly to compute.

(*d*) Between which coordinate spaces does the OpenGL projection matrix transform.

☑ The projection matrix transforms coordinates ☑ from ☑ to

From eye space to clip space.

(*e*) What is the difference between a material property (including attributes emitted with `glColor3f`) and a lighted color?

☑ Difference between material property and lighted color?

A material property is a color given to the OpenGL rendering pipeline, it can apply to all vertices (as a uniform) or just one (as a vertex shader input). The rendering pipeline typically computes a lighted color by using the material property as well as light locations, the normal associated with the vertex, and other factors. If material properties were used instead of lighted colors then there would be no cues about shape and so, a sphere would look like a circle.

27 Fall 2015 Solutions

Name Solution

GPU Programming
EE 4702-1
Midterm Examination

Monday, 26 October 2015 13:30–14:20 CDT

Problem 1		(21 pts)
Problem 2		(25 pts)
Problem 3		(12 pts)
Problem 4		(18 pts)
Problem 5		(12 pts)
Problem 6		(12 pts)
Exam Total		(100 pts)

Alias Lets go Mets!

Good Luck!

Problem 1: [21 pts] In the code below three coordinates are defined, `ph` (P_h, head), `pt` (P_t, tail), and `ps` (P_s, surface).

(a) Complete the code below so that it renders a cylinder with axis $\overrightarrow{P_hP_t}$, with P_h at the top (flat part) of the cylinder, P_t at the bottom (the other flat part), and with P_s on the surface. Render only the curved parts, not the flat parts. The cylinder surface should be approximated by `slices` flat sides.

☑ Show code before the loop, such as for `ax` and `ay`.

☑ Show code in the loop that emits vertex ☑ coordinates and ☑ normals.

```
pCoor ph( 1.5, 0, -3.2 );
pCoor pt( 0,   5, -5 );
pCoor ps( 9,   6, -9 );
const int slices = 100;
const float delta_theta = 2 * M_PI / slices;
pNorm axis = pt - ph;
// Some of the solution goes here.

// SOLUTION BELOW
pVect hs = ps - ph;
float hl = dot(axis,hs);    // Distance from ph to
pCoor pa = ph + hl * axis;

pNorm ax = ps - pa;
pVect ay = cross(ax,axis);
// SOLUTION ABOVE




glBegin(GL_TRIANGLE_STRIP);
glColor3ub( 256, 89, 16 );
for ( int i=0; i<=slices; i++ )
  {
    const float theta = i * delta_theta;
    pVect n = cos(theta) * ax + sin(theta) * ay;

    // SOLUTION BELOW
    glNormal3fv(n);      // n is the normal.
    pVect rn = r * n;    // Scale n by the cylinder radius.
    pCoor p1 = ph + rn;
    glVertex3fv(p1);
    pCoor p2 = pt + rn;
    // SOLUTION ABOVE
    glVertex3fv(p2);




  }
glEnd();
```

Problem 2: [25 pts] The code below, taken from the solution to Homework 2, renders the surface of a truss. Recall that the truss might be a part of a spinning gyroscope.

```
glEnableClientState(GL_VERTEX_ARRAY);

// glVertexPointer(3, GL_FLOAT, sizeof(pCoor), coord_buffer);
glVertexPointer(3, GL_FLOAT, sizeof(pCoor), NULL);  // SOLUTION

glEnableClientState(GL_NORMAL_ARRAY);

// glNormalPointer(GL_FLOAT, 0, norm_buffer);
glNormalPointer(GL_FLOAT, 0, NULL);                 // SOLUTION

glColor3fv(strip->color);
glDrawArrays(GL_TRIANGLE_STRIP, 0, elts);
glDisableClientState(GL_NORMAL_ARRAY);
glDisableClientState(GL_VERTEX_ARRAY);
```

(*a*) How much data is sent from the CPU to the GPU to render a frame based on the code above. Your answer should be in terms of the variables above.

☑ The amount of data sent per frame is ... ☑ Indicate unit.

The color is sent once. That's 3 floats. The rendering pass consists of `elts` vertices. Each vertex consists of a coordinate (specified by `glVertexPointer`) which is 3 floats and a normal which is also 3 floats. Note that the number of components of a vertex coordinate is explicitly given as 3 in the call to `glVertexPointer`, whereas for normals it must be 3. It's possible to render with 2-, 3-, or 4-component vertex coordinates.

The total amount of data is $3 + (3 + 3)e = 3 + 6e$ floats or $(3 + 6e)4\,\mathrm{B} = (12 + 24e)\,\mathrm{B}$, where e is the same as `elts`.

(*b*) How could someone who is not familiar with Homework 2 be reasonably sure that the code above uses client arrays and not buffer objects? What would be different *in the parts shown* if buffer objects were used?

☑ We would guess code doesn't use buffer objects because ....

If buffer objects were used the last argument to `glVertexPointer` and `glNormalPointer` would likely be a zero or NULL (meaning start at the beginning of the buffer object).

☑ Show changes to parts shown if buffer objects were used.

Changes appear above. The original lines of code are shown commented out.

(*c*) Explain why using buffer objects in the code above would not make a big difference, whereas using buffer objects to render the balls (used to show nodes in the truss) in the same simulation does make a big difference?

☑ Buffer objects would not make a big difference for truss surface because ...

Buffer objects help when the same data needs to be used multiple times. Since the truss can be in constant motion the coordinates would change every frame, and so a buffer object would need to be sent every frame. The amount of coordinate and normal data sent would be the same whether or not buffer objects were used.

☑ Buffer objects do make a big difference for balls (part of truss) because ...

For rendering the ball a buffer object is used that contains the surface coordinates of a sphere with radius one and with a center at the origin. To render a ball at a particular location, in a particular orientation (if a texture is used), and of a particular size a

transformation matrix is constructed that moves the ball specified in the buffer object to the desired location, size, and orientation. When rendering a ball only the transformation matrix needs to be sent, which is 16 floats, rather than a whole set of coordinates. (The transformation matrix is used to update the modelview matrix.)

Problem 2, continued: The Homework 2 code from the previous page is repeated below.

```
glEnableClientState(GL_VERTEX_ARRAY);
glVertexPointer(3, GL_FLOAT, sizeof(pCoor), coord_buffer);
glEnableClientState(GL_NORMAL_ARRAY);
glNormalPointer(GL_FLOAT, 0, norm_buffer);
glColor3fv(strip->color);
glDrawArrays(GL_TRIANGLE_STRIP, 0, elts);
glDisableClientState(GL_NORMAL_ARRAY);
glDisableClientState(GL_VERTEX_ARRAY);
```

(*d*) Re-write the code so that it does not use client arrays, but instead uses calls like `glVertex`. *Hint: Only a few lines of code needed.*

☑ Re-write code so that calls like `glVertex` is used.

The solution appears below. The call to `glDrawArrays` is replaced by a `glBegin`/`glEnd` pair that encloses a loop which sends one vertex at a time to OpenGL.

```
// SOLUTION
glBegin(GL_TRIANGLE_STRIP);
glColor3fv(strip->color);
for ( int i=0; i<elts; i++ )
  {
    glNormal3fv(&norm_buffer[i]);
    glVertex3fv(&coord_buffer[i]);
}
glEnd();
```

Problem 3: [12 pts] The code fragment below is from a demo-10 vertex shader.

```
void vs_main_basic() {
  gl_Position = gl_ModelViewProjectionMatrix * gl_Vertex;
  // Other code removed.
}
```

(*a*) Identity the variable that uses the storage types given below.

☑ Which variable is a vertex shader input?

`gl_Vertex`

.

☑ Which variable is a uniform variable?

`gl_ModelViewProjectionMatrix`

.

☑ Which variable is a vertex shader output?

`gl_Position`

.

(*b*) Two of these variables appear in the OpenGL shading language compatibility profile but not in the undeprecated OpenGL SL standard, meaning that they aren't truly needed but are being kept so as not to break old software. Identify the variable that *is* truly needed and explain why.

☑ The undeprecated, truly needed variable is: It is `gl_Position`

☑ It is truly needed because ...

Shader output `gl_Position` is set to the clip-space coordinate of the vertex. That's something the fixed functionality needs to know to do clipping and to do rasterization.

Problem 4: [18 pts] Answer each question below.

(*a*) The interface block below is used as a fragment shader input. Among other things the fragment shader does lighting calculations. Explain what the `flat` qualifier does and why it is needed.

```
in Data_GF
{
  vec3 var_normal_e;
  vec4 var_vertex_e;
  flat vec4 color;
};
```

☑ The `flat` qualifier ...

Tells the fixed functionality not to interpolate the values of `color` at each triangle vertex when determining a value for the fragment shader input. Instead, the value of `color` sent to the fragment shader will be the `color` output produced by the vertex shader or geometry shader (if there is one) of the last vertex in the primitive.

☑ The `flat` qualifier helps this particular shader program by ...

eliminating computation wasted on interpolation (assuming that the color at each vertex is the same). The interpolation makes sense if the vertex shader performed lighting calculations because each vertex of a primitive would likely have a different lighted color.

(*b*) Suppose the depth (z) test was turned off. How would that affect the rendered image?

☑ Rendered image with depth test turned off would appear ...

Fragments of primitives that were written later would cover fragments of primitives that were written earlier. (With the depth test turned on fragments closer to the user would cover fragments further from the user.)

(*c*) Describe what textures might be used for in a graphics program.

☑ An example of what a texture might be used for is ...

A texture could give a primitive the appearance of being stone or wood (by using an image of stone or wood). It can be used to apply any image to a primitive.

Problem 5: [12 pts] Answer the following questions.

(*a*) The code below renders two triangles. Add the minimum amount of code to make the first one blue and the second one orange (RGB:1,.35,.06). *Hint: This is an easy problem, don't overthink it. Note: The shade of orange is from the New York Mets logo.*

☑ Add minimum amount of code so that the first triangle is blue and the second is orange.

```
glBegin(GL_TRIANGLES);
glColor3f(0, 0, 1);          // SOLUTION -- Blue
glVertex3f( 10,  20,  0 );
glVertex3f( 15,  20,  0 );
glVertex3f( 10,   0,  0 );
glColor3f(1, .35, 06);       // SOLUTION -- Orange
glVertex3f( 30,  20,  0 );
glVertex3f( 30,   0,  0 );
glVertex3f( 25,   0,  0 );

glend();
```

(*b*) The code below renders two triangles. Add texture coordinates so that when the two triangles are put together they show the entire texture. The texture has already been set up, just provide the coordinates.

☑ Add OpenGL calls to provide texture coordinates.

```
glBegin(GL_TRIANGLES);     //  SOLUTION BELOW
                             glTexCoord2f(0,1);
glVertex3f( 10,  20,  0 );   glTexCoord2f(1,1);
glVertex3f( 15,  20,  0 );   glTexCoord2f(0,0);
glVertex3f( 10,   0,  0 );
                             glTexCoord2f(1,1);
glVertex3f( 30,  20,  0 );   glTexCoord2f(1,0);
glVertex3f( 30,   0,  0 );   glTexCoord2f(0,0);
glVertex3f( 25,   0,  0 ); //  SOLUTION ABOVE

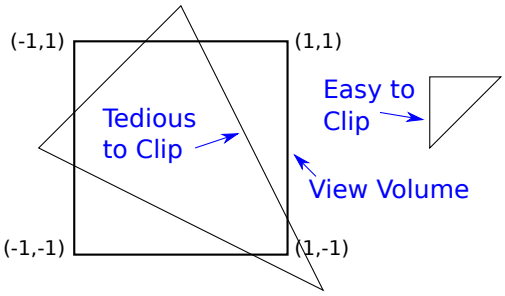
glend();
```

Problem 6: [12 pts] Clipping is performed between the output of the geometry shader and the input to the rasterizer.

(*a*) Provide a simple example of a triangle that can be completely clipped. The example should include the triangle's vertex coordinates in clip space.

☑ Easily completely clipable triangle coordinates:

A triangle with vertex clip-space coordinates $(2.3, .7, .5)$, $(3, .7, .5)$, $(2.3, 0, .5)$ is easy to clip because the x component of each vertex coordinate is strictly greater than 1 and therefore no part of it can be in the view volume. (The view volume is a cube with vertex coordinates $(\pm 1, \pm 1, \pm 1)$.) The Easy to Clip triangle appears in the illustration to the right. The illustration shows x and y coordinates only.



(*b*) Provide an example of a primitive for which clipping would be difficult or tedious.

☑ Illustration (not coordinates) of a triangle that is tedious to clip.

The triangle labeled Tedious to Clip in the illustration above.

☑ Briefly explain what makes it tedious.

Clipping results in a seven-sided polygon, which would have to be tessellated into triangles (assuming that the rendering pipeline can't handle polygons other than triangles, which is usually the case). The tedious part is finding the six intersections of the triangle edges with the view-volume faces, plus noticing that an edge of the view volume (point $(1, -1)$) contributes another vertex to the clipped primitive.

(*c*) Suppose due to some error the clipping stage doesn't clip anything. Explain why there would be no difference in the appearance of the rendered image.

☑ Image no different if clip stage doesn't clip because . . .

. . . fragments that are not in the window will be discarded somewhere after rasterization, probably between the rasterization and fragment shader.

(*d*) Again, suppose due to some error the clipping stage doesn't clip anything. Explain why performance would suffer.

☑ Performance suffers because . . .

Consider a triangle that is completely clipped. The clip stage only operates on the three vertices. Suppose that the triangle covers 100 pixels (but outside the window). If it's not clipped by the clip stage the rasterizer will have to generate 100 fragments. Even if they are discarded right after rasterization, work was performed to generate them.

Name Solution

GPU Programming
EE 4702-1
Final Examination

Friday, 11 December 2015 15:00–17:00 CST

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

Alias Methane?

Good Luck!

Problem 1: [20 pts] The code below, taken from Homework 4, performs the rendering pass used to draw the balls. Let n denote the number of balls and let s denote the number of vertices for each ball. It does so using an instanced draw in which a set of s vertices is sent into the rendering pipeline n times, for a total of sn vertices. Procedure `glDrawArraysInstanced(type,0,s,n)` sends a set of s vertices for rendering n times.

```
// CPU code for rendering pass.

  glBindBufferBase(GL_SHADER_STORAGE_BUFFER,1,balls_pos_rad_bo);
  glBindBufferBase(GL_SHADER_STORAGE_BUFFER,2,balls_color_bo);
  glBindBuffer(GL_ARRAY_BUFFER,sphere.points_bo);
  glVertexPointer(3,GL_FLOAT,0,0);
  glEnableClientState(GL_VERTEX_ARRAY);
  glDrawArraysInstanced(GL_TRIANGLE_STRIP, 0, num_vertices, num_balls );

// Vertex Shader Code for rendering pass.

void vs_main_instances() {
  vec4 center_o_rad = balls_pos_rad[gl_InstanceID ⟸ I ];
  float rad = center_o_rad.w;
  vec3 to_surface = rad * gl_Vertex.xyz ⟸ I ;
  vec4 position_o = vec4(center_o_rad.xyz + to_surface, 1f);

  gl_Position = gl_ModelViewProjectionMatrix ⟸ U  * position_o;
  vertex_e = gl_ModelViewMatrix ⟸ U  * position_o;
  normal_e = gl_NormalMatrix ⟸ U  * gl_Vertex.xyz ⟸ I ;
}
```

(*a*) In the shader code above indicate which variables are uniforms and which are vertex shader inputs. *Hint: This question can be answered without understanding what an instanced draw is.*

☑ Put a `U` next to all uniform variables. ☑ Put an `I` next to all vertex shader inputs.

(*b*) For each rendering pass indicate how much data must be sent from the CPU to the GPU due to this vertex shader. Assume that **all uniform variables** need to be **re-sent each rendering pass**. But, make an intelligent determination on whether vertex shader inputs and other data needs to be re-sent.

☑ Amount of CPU to GPU data for uniform variables per pass in terms of s and n.

Uniform sizes: `gl_ModelViewProjectionMatrix` and `gl_ModelViewMatrix`: since each is a 4×4 matrix, 16 floats each; `gl_NormalMatrix`: since it is a 3×3 matrix, 9 floats. Total: $4(32 + 9) = 164\,\mathrm{B}$.

☑ Amount of CPU to GPU data for vertex shader inputs per pass in terms of s and n.

Because `gl_InstanceID` is a consecutive numbering of instances it is something that the GPU can easily generate, and so there is no need to send values for this input from the CPU to the GPU. Input `gl_Vertex` is sourced from a buffer object. Recall that the buffer object contains the vertex coordinates of a sphere of radius 1 centered at the origin. We expect that this buffer object would be sent from the CPU to the GPU just once, and used as many times as needed. So in a typical pass the amount of data sent would be zero.

☑ Amount of CPU to GPU data for other data used by vertex shader per pass in terms of s and n.

Lets assume that the ball positions change each rendering pass but their colors remain the same. Amount of data: $16s\,\mathrm{B}$.

Problem 1, continued:

(*c*) Appearing below is the vertex shader used for spheres in an ordinary rendering pass, followed by the vertex shader for the instanced rendering. Both render the same balls, but using different CPU code.

```
void vs_main() {                // Vertex Shader for Ordinary Rendering
  gl_Position = gl_ModelViewProjectionMatrix * gl_Vertex;
  vertex_e = gl_ModelViewMatrix * gl_Vertex;
  normal_e = gl_NormalMatrix * gl_Vertex.xyz;
}

void vs_main_instances() {      // Vertex Shader for Instanced Rendering
  vec4 center_o_rad = balls_pos_rad[gl_InstanceID];
  float rad = center_o_rad.w;
  vec3 to_surface = rad * gl_Vertex.xyz;
  vec4 position_o = vec4(center_o_rad.xyz + to_surface, 1f);

  gl_Position = gl_ModelViewProjectionMatrix * position_o;
  vertex_e = gl_ModelViewMatrix * position_o;
  normal_e = gl_NormalMatrix * gl_Vertex.xyz;
}
```

Because it accesses `balls_pos_rad`, `vs_main_instances` *looks like* it's accessing more data than `vs_main`, but in reality less data is being sent to render all the balls.

☑ What data was sent from CPU to GPU in the original code that substituted for `balls_pos_rad`?

Short Answer: The modelview matrix.

Explanation: In all of the classroom examples and homework assignments, the vertices for a sphere are computed for a sphere of radius 1 with its center at the origin. These vertex coordinates and associated normals and texture coordinates are usually placed in buffer objects once, during initialization.

To show the sphere in the desired location at the desired size the usual way of doing things in class was to set the modelview matrix. This is the approach used in `vs_main`. One rendering pass is needed for each sphere rendered, and the modelview matrix would be set before each pass.

In contrast, with an instanced draw one can render multiple spheres in a single rendering pass. The pre-defined variable `gl_InstanceID` indicates which sphere (numbered starting at 0) is being rendered. Because the same modelview matrix is used during an entire rendering pass another method is needed to specify the size and position of each sphere. The code in `vs_main_instances` gets the size and position from the array `balls_pos_rad` and uses that to transform the local sphere coordinate in `gl_Vertex` (by doing a vector add rather than a matrix/vector multiply). Note that this method does not rotate the sphere, but that should not be a problem since texture coordinates are presumably not being used.

☑ How did this data compare in size to `balls_pos_rad`?

The size of the modelview matrix is $4 \times 4 = 16$ floats, the size of the normal matrix is $3 \times 3 = 9$ floats. They are sent once per sphere. The size of an element of `balls_pos_rad` is 4 floats. So the original code uses $\frac{16+9}{4} = 6.25\times$ more data.

(*d*) Explain why the instanced draw gives higher performance than ordinary rendering.

☑ Higher performance because ...

... the overhead of performing a rendering pass is incurred once per n balls rather than n times per n balls (spheres).

Long Answer: The original method does send $6.25\times$ more data, but that's still just 25 bytes per ball so the amount of data will not have a significant impact. The instanced draw does more computation: it needs to perform three multiplies (`rad *`

`gl_Vertex.xyz`) that the original code does not have to do. That's in the vertex shader and is dwarfed by the amount of computation needed to do lighting computations in the fragment shader. So the additional computation needed for the instanced draw is also not a significant factor. Also, each vertex shader invocation needs to read the 16 bytes from `balls_pos_rad`, but the invocations sharing an instance ID will read the same element, and so this will add little time. It's likely that the time to set up a rendering pass will be much larger than these other differences.

Problem 2: [15 pts] CUDA kernel A launched with 10 blocks of 1024 threads on a GPU with 10 streaming multiprocessors (abbreviated SMs or MPs) takes 27 s to run. Consider several scenarios in which the kernel is launched on a GPU with 9 SMs. The only difference between the GPUs is the number of SMs. In both GPUs the maximum number of threads per SM and the maximum number of threads per block is 1024.

(a) Suppose that kernel A is launched again with 10 blocks but this time on the GPU with 9 SMs.

☑ Run time for A in a 10-block launch on the 9 SM GPU? ☑ Explain.

Short Answer: It would take $2 \times 27 = 54$ s because one SM would be assigned two blocks and those blocks would run sequentially.

Long Answer: In this particular case, because of the thread limit, only one block at a time can run on an SM. When the kernel is launched each of the first 9 blocks will be sent to an SM. The tenth block will have to wait for one of the first nine blocks to finish. Assuming that all blocks take about the same amount of time, that should occur in 27 s. The tenth block will then finish 27 s later, for a total time of 54 s.

(b) How long would A take to run on the 9-SM GPU if launched with 9 blocks of 1024 threads, but doing the same amount of work as the 10-block launch. Kernel A was written by a skilled programmer.

☑ Run time for A in a 9-block launch on the 9 SM GPU? ☑ Explain.

Short Answer: The run time would be $\frac{10}{9}27\,\text{s} = 30\,\text{s}$.

Long Answer: Since the programmer is skilled and since the number of blocks matches the number of SMs, we would expect that the run time would be proportional to the amount of work divided by the amount of computing resources. If the amount of computing resources (the number of SMs) changes from 10 to 9 we would expect run time to increase by a factor of $\frac{10}{9}$ yielding a run time of $\frac{10}{9}27\,\text{s} = 30\,\text{s}$.

(c) CUDA kernel C launched with 10 blocks of **32 threads** takes 72 s on the 10-SM GPU. How long would the 10-block launch take on the 9-SM GPU?

☑ Run time for C in a 10-block launch on the 9 SM GPU?

☑ Explain how the low thread count makes part c (this question) different than part a.

It would still take about 72 s because with 32-thread blocks it's possible for two blocks to run simultaneously on the same SM. Since 32 threads are not nearly enough to fully utilize an SM's resources (for example, the 128 single-precision floating-point units that NVIDIA calls CUDA cores on a CC 4.X device) the two blocks can run at the same time and so they will take the same amount of time to run as they would if each were run on a separate SM, assuming that the code was not data-limited. (We can assume it's not data limited in this case because there are only 32 threads.)

Problem 3: [15 pts] The CUDA kernel below is based on a classroom example. Notice that the computation of `tid` has been changed. Indicate the change in execution time (relative to the commented out code) and whether the new code is correct (produces the same answer as the commented out code). *Note: The original exam contained a second part.*

(*a*) Consider the simple kernel:

```
__global__ void cuda_thread_start_simple() {
  // const int tid = threadIdx.x + blockIdx.x * blockDim.x;
  const int tid = threadIdx.x;                  // Changed code. Original line above.

  const int elt_per_thread = array_size / num_threads;
  const int start = elt_per_thread * tid;
  const int stop = start + elt_per_thread;

  for ( int h=start; h<stop; h++ ) {
      float4 p = d_v_in[h];
      d_m_out[h] = dot( p, p ); }}
```

☑ Due to change, execution time is ◯ *longer* ☑ *about the same or* ◯ *shorter* .

☑ Due to change, results are ☑ *incorrect* or ◯ *still correct.*

☑ Important: Explain by showing how array elements are assigned to threads.

Because of the change every block does the computation that block 0 would do in the original code. Since work is divided evenly, and because the number of blocks hasn't changed, the execution time should not change by much. The only possible reason for an increase in execution time is that each block writes the same array elements. However, the effect of this should be small because the number of writers to a particular location is equal to the number of active blocks (which at most would be on the order of 160 [with a block occupancy of 16 and 10 SMs]) and because for multiple writes to the same location to be a problem those writes would have to occur within cycles of each other, which is unlikely for even two writers because any of a number of factors could cause blocks on different SMs to be at different places in the code such as not starting at exactly the same time.

The results would be incorrect because the computation performed by blocks $1, 2, \ldots$ is not being done.

Problem 4: [20 pts] Answer the following questions about CUDA.

(*a*) In an execution of the code below an SM reads 32 times more data than it needs to due to the way the code is written.

```
__global__ void some_prob(char *in_data, int *out_data) {
  const int tid = threadIdx.x + blockIdx.x * blockDim.x;
  const int width = 128;
  const int start_id = tid * width;
  int sum = 0;
  for ( int i=0; i<width; i++ ) sum += in_data[start_id+i];
  out_data[tid] = sum;
}
```

☑ Code reads 32× more data than it needs because ...

Adjacent threads in a warp read global memory locations that differ by 128 bytes, and so a separate request will be needed for each thread. The minimum request size is 32 bytes and only one byte of the request is used (`in_data` is an array of `char`s, which are one byte). Since only 1 out of 32 bytes are used the code reads $32\times$ more data than is needed.

More Details: To see how adjacent threads are reading global addresses that are 128 bytes apart compute `start_id+i` for two cases: `threadIdx.x` = 0 and for `threadIdx.x` = 1. In both cases set `blockIdx.x` = 0 and `i` = 0. For thread 0: `start_id+i` = 0+0 = 0 and for thread 1 `start_id+i` = 1*128+0 = 128. So the distance between accesses is 128 elements, or 128 B because each element is one byte. It should be easy to convince yourself that the distance between accesses is 128 when comparing any `threadIdx.x` and `threadIdx.x+1`. Keep in mind that within a block (and so a warp) the two threads will have the same `blockIdx.x` and the same `i`. (The same `i` because within a warp they must be at the same point in execution.)

Suppose that the type of `in_data` were changed from `char*` to `int*`. Instead of reading 32× more data it would only be reading $x\times$ more data than needed.

☑ What is the value of x?

Suppose that the size of an `int` is four bytes (which is common). Then the thread would be using 4 of 32 bytes, and so it would only be reading $\frac{32}{4} = 8\times$ more data than necessary. That is, $x = 8$.

(*b*) The code sample below is based on the shared memory classroom demo. *Note: The second* `syncthreads` *did not appear in the original exam.*

```
__shared__ int sum;

if ( threadIdx.x == 0 )   sum = 0;
__syncthreads();
if ( threadIdx.x == 40 )  sum += 40;
if ( threadIdx.x == 70 )  sum += 70;
if ( threadIdx.x == 200 ) sum += 200;
__syncthreads();
out_data[tid] = sum;
```

☑ Show at least four different possible values for `sum` that the code above can write to `out_data`.

They are: 40, 70, 200, 110, 240, 270, 310.

☑ Explain.

At the first `__synchthreads` we know for certain that `sum` is zero. Statements like `sum += 40;` consist of a shared memory read, an add, and a shared memory write. Let R40 denote the shared memory read performed by thread 40, and define W40, R70, etc. similarly. Perhaps we would like the ordering of these events to be R40, W40, R70, W70, R200, W200. If so `sum` would be

set to 310. All we know for certain is that within a thread the write will come after the read. But the ordering between threads is not guaranteed because each of the participating threads is on a different warp. Consider the following possible ordering: R40, R70, R200, W200, W70, R40. In this case all threads read zero, and since 70 is the last to write, `sum` will be set to 70. Consider another possible ordering: R70, W70, R40 R200, W200, W40. Here, threads 40 and 200 will read the 70 written by thread 70, since 40 is the last to write `sum` will be 110.

Here are all possibilities: 40, 70, 200, 110, 240, 270, 310.

(*c*) Consider the use of `__syncthreads()` in the CUDA code below and in general.

☑ In general, what does `__syncthreads` do?

It causes each thread to wait until all threads in the block reach `__syncthreads`, after which threads can proceed. In class this was described as a room with an entrance door and an exit door. Initially the entrance door is open. Threads enter the room when their execution reaches `__syncthreads`. When all of the threads in the block are in the room the entrance door closes and the exit opens, which means that execution can proceed to the statement after `__syncthreads`.

☑ What might go wrong if `__syncthreads` were removed from the code below?

```
__shared__ int our_data[1025];
our_data[threadIdx.x] = my_element;
__syncthreads();
output_data[tid] = my_element + our_data[threadIdx.x + 1];
```

Thread 31 might read element 32 of `our_data` before thread 32 writes it.

☑ What's wrong with the use of `__syncthreads` below?

```
if ( threadIdx.x != 20 ) __syncthreads();
```

All threads in the block must execute `__syncthreads`. In the example above thread 20 does not execute it and so the threads that do execute `__syncthreads` will be stuck in `__syncthreads`.

Problem 5: [20 pts] The geometry shader below, from Homework 4, passes a triangle unchanged.

```
layout ( triangles ) in;
layout ( triangle_strip, max_vertices = 6 ) out; // SOLUTION 3 -> 6

void gs_main_simple() {

  /// SOLUTION
  vec3 tnorm = normalize( cross                                  // Filled In
    ( In[1].vertex_e.xyz - In[0].vertex_e.xyz,
      In[2].vertex_e.xyz - In[1].vertex_e.xyz ) );

  for ( int i=0; i<3; i++ )
    {
      normal_e = In[i].normal_e;
      vertex_e = In[i].vertex_e;
      color = In[i].color;
      gl_Position = In[i].gl_Position;
      EmitVertex();
    }
  EndPrimitive();

 /// SOLUTION:
  color = vec4(0,0,1,1); // Blue
  normal_e = tnorm;

  for ( int i=0; i<3; i++ ) {
      vertex_e = In[i].vertex_e + tnorm;
      gl_Position = gl_ProjectionMatrix * vertex_e;
      EmitVertex();
    }
  EndPrimitive();
}
```

(*a*) Add code so that `tnorm` is assigned the triangle's eye-space geometric normal.

☑ Set `tnorm` to eye-space geometric normal of triangle.

Solution appears above.

(*b*) Modify the shader so that it emits a second triangle of the same shape but displaced one unit in the direction of `tnorm` (see above). (For `normal_e` see next part.) The new triangle should be blue. The following library functions are available: `cross`, `dot`, `normalize`, and `length`. A library of colors **is not** available.

☑ Add code to emit second triangle.

☑ Be sure to assign ☑ `vertex_e`, ☑ `color` to blue, and (important) ☑ `gl_Position`.

☑ Modify the `layout` declarations, if necessary.

Solution appears above. Notice that the `color` and `normal_e` shader outputs are only assigned once since they don't change. Also notice that the eye-space coordinate of the new triangle, `vertex_e`, had to be converted to clip space, something which is done in the vertex shader for the vertices of the original triangle. In the layout declaration `max_vertices` was increased from 3 to 6.

(*c*) What about `normal_e` for the new triangle? Suppose we are rendering a sphere. Compare the appearance

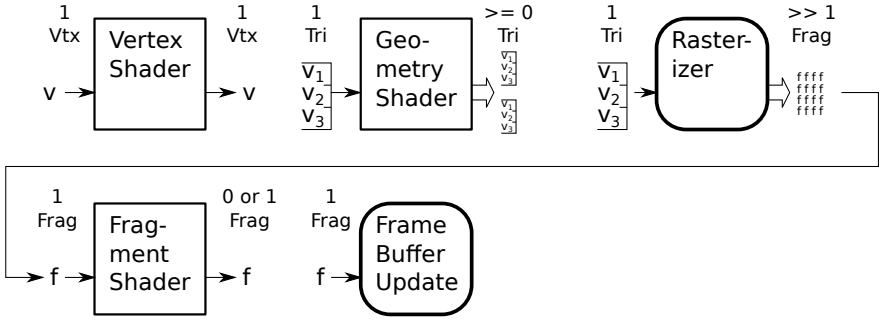
of the new triangle with `normal_e` set to `tnorm` to the appearance when `normal_e` is set to `In[i].normal_e`.

☑ Describe difference in appearance for `normal_e = tnorm` versus `normal_e = In[i].normal_e`.

With `normal_e = tnorm` the sphere will look like it is covered with triangles, that's because two nearby points on different triangles will have different normals and so will be lighted differently, making the triangle boundaries obvious. In contrast, if the normal is based on the sphere normal at those two nearby points, then the two normals will be almost the same and the lighting will be almost the same, making it hard to see the triangles.

Problem 6: [10 pts] Answer the following questions.

(*a*) The depth test (also called the z test) in an important part of the rendering pipeline.



☑ Where in the rendering pipeline is the depth test performed?

It is performed in the frame buffer update stage.

☑ Why would it make no sense to perform the depth test in the vertex shader?

Because the vertex shader only has the coordinate of one vertex, not the other vertices (if any) in the primitive. Even if it had this information, that would not help because some fragments might pass the depth test while other fragments would not pass. It would also force the vertex shader to compute the location of each fragment, reproducing the work that the rasterizer does.

(*b*) Switching from a triangle strip to individual triangles increases the amount of work performed by the vertex shader by a factor of 3, but does not change the amount of work performed by the geometry and fragment shaders.

☑ Why does the vertex shader do 3× more work?

Because in a triangle strip a single vertex, and so the work of a single vertex shader invocation, can be used for up to three triangles.

☑ Why do the geometry and fragment shaders do the same amount of work?

Because the number of primitives, triangles, is the same, and so the number of geometry shader invocations will be the same. Because the number and size of the triangles sent to the rasterizer are the same in both cases the number of fragment shader invocations will be the same.

28 Fall 2014 Solutions

Name Solution

GPU Programming
EE 4702-1
Midterm Examination

Wednesday, 12 November 2014 9:30–10:20 CST

Problem 1		(18 pts)
Problem 2		(12 pts)
Problem 3		(24 pts)
Problem 4		(12 pts)
Problem 5		(14 pts)
Problem 6		(20 pts)
Exam Total		(100 pts)

Alias Phylæ has landed!

Good Luck!

Problem 1: [18 pts] Appearing below is a simplified version of the vertex shader `vs_main` from Homework 4. One way to reduce the amount of computation it must perform is by pre-computing values and making them available as `vs_main` inputs. For example, rather than compute `radial_idx` each time, it can instead be pre-computed and stored in the `w` component of `gl_Vertex` or it can be stored in a user-defined vertex shader input named `radial_idx`.

```
void vs_main() {
  int bidx = gl_Vertex.x;              int ti = gl_Vertex.y;

  int radial_idx = bidx * opt_segments + ti;
  float theta = (1.0/opt_segments) * radial_idx * omega;

  vec3 pos1 = balls_pos[bidx-1].xyz;   vec3 pos2 = balls_pos[bidx].xyz;
  vec3 v12 = pos2.xyz - pos1.xyz;
  vec3 p = pos1 + float(ti) / opt_segments * v12;

  vec3 va = spiral_radius * normalize( vec3(0,v12.z,-v12.y) );
  vec3 vb = spiral_radius * normalize( cross(v12,va) );

  vec3 radial = va * cos(theta) + vb * sin(theta);
  vec3 p_outer = p + radial;           vec3 p_inner = p + 0.5 * radial;
```

Consider two candidates for new vertex shader inputs: `va/vb` (count these as one candidate) and `theta`.

(*a*) Estimate the amount by which each candidate would reduce the amount of computation to be performed **by the vertex shader**.

☑ Computation reduction by making `va/vb` an input:

☑ Computation reduction by making `theta` an input:

To solve this problem we need to estimate the amount of computation performed for each candidate. As done in class we will estimate computation by tallying the number of arithmetic operations needed.

The easier computation is for `theta`. Look at the line that assigns `theta` and the line that assigns `radial_idx`. They perform three multiplies, a divide, and an addition. These computations would not be necessary if `theta` were an input to the shader.

For `va` first consider the normalize. With optimization that recognizes that the x component is zero, that would be two multiplies and an add to compute $l^2 = y^2 + z^2$. A reciprocal square root and two multiplies are needed to compute $(0, yl^{-1}, zl^{-1})$, and two more multiplies are needed for `spiral_radius`. The total is 6 multiplies, an add, and a reciprocal square root (to compute $l^{-1} = 1/\sqrt{y^2+z^2}$). For `vb` we need to do a cross product which is 6 multiplies and four adds, plus 6 more multiply/adds and a reciprocal square root for the normalize and `spiral_radius` multiplication. The total for `va` and `vb` is 24 multiplies or multiply/adds, 3 adds and two reciprical square roots.

Grading Note: Many students computed the amount of data rather than the amount of computation.

(*b*) One of the candidates would result in a large increase in CPU to GPU data transfer. Which one is it, and why?

☑ Candidate causing a big increase in CPU to GPU data transfer: Variables `va` and `vb`.

☑ Reason for **big** increase:

Variables `va` and `vb` will result in a big increase in CPU to GPU traffic because, unlike `theta`, they need to be changed every frame. Notice that `theta` only depends upon constants, such as `opt_segments`, and things that are always the same for a particular vertex, such as `bidx`. Therefore the value of `theta` for each vertex can be put in a buffer object and used for a new vertex shader

input. The buffer object would be transferred to the GPU just once. In contrast, `va` and `vb` depend on `balls_pos` which changes each frame.

(*c*) In the code above there is a much better candidate for a new vertex shader input other than `va/vb` or `theta`. It would reduce computation by a significant amount and would not have a big impact on data transfer. *Hint: it's not a variable, but an expression, or several expressions.*

☑ The much better candidate for an input attribute is:

Rather than put `theta` in a buffer object, which would only save a small amount of computation, put values for `sin(theta)` and `cos(theta)`. Like `theta`, these don't change from frame to frame, and unlike `theta` they save much more computation because `sin` and `cos` are expensive to compute.

Problem 2: [12 pts] Appearing below is the fragment shader from the solution to Homework 4. Recall that this shader does not render the spiral surface in places where the texture is dark, giving the appearance of holes.

(*a*) Modify the shader code so that the texture itself is only applied to the front side of the spiral, and so that the holes still appear on **both** sides. *Hint: Can be done with a line or two of code.*

☑ Modify so holes are on both sides, but texture just on front.

```
void fs_main() {
  vec4 color = is_edge ? gl_FrontMaterial.ambient :
       gl_FrontFacing ? gl_FrontMaterial.diffuse : gl_BackMaterial.diffuse;

  vec4 texel = is_edge ? vec4(1,1,1,1) : texture(tex_unit_0,gl_TexCoord[0].xy);

  bool hole = texel.r + texel.g + texel.b < 0.05;
  if ( hole ) discard;

  // SOLUTION: Only use texel to color surface for non-edge front-facing prims.
  vec4 texel_apply = is_edge || !gl_FrontFacing ? vec4(1,1,1,1) : texel;

  gl_FragColor = texel_apply * generic_lighting(vertex_e, color, normalize(normal_e));

  gl_FragDepth = gl_FragCoord.z;
}
```

(*b*) The fragment shader code below, based on Homework 4, is supposed to use different colors for the front, back, and edge of the spiral. The code below has an error and the commented-out line shows the correct code. How will this error change the appearance of the spiral?

☑ With this error spiral will appear...

Short answer: ... with the exact same color everywhere, that is, there will be no shading based upon light location, etc.

Long answer: The lighting routine is not being applied. The lighting routine returns a color based on `our_color` but taking into account the location and orientation of the vertex (the other input parameters) and the location and color of the light (which are uniform variables). In the erroneous code none of this is being done, instead `our_color` is being used and so the color of the spiral will be exactly the same everywhere.

```
void fs_main() {
  vec4 our_color = is_edge ? gl_FrontMaterial.ambient :
            gl_FrontFacing ? gl_FrontMaterial.diffuse : gl_BackMaterial.diffuse;

  vec4 texel = is_edge ? vec4(1,1,1,1) : texture(tex_unit_0,gl_TexCoord[0].xy);
  bool hole = texel.r + texel.g + texel.b < 0.05;
  if ( hole ) discard;

  //  gl_FragColor = texel * generic_lighting(vertex_e, our_color, normalize(normal_e));
  gl_FragColor = texel * our_color;

  gl_FragDepth = gl_FragCoord.z;
}
```

Problem 3: [24 pts] Answer each question below.

(*a*) An object at point P_1 is moving at velocity v_1 just before collision with a wall with normal $\hat{w}$. The collision is completely elastic, meaning the object's speed (the magnitude of velocity) will not change. Write an expression for the velocity after the collision?

☑ Expression showing velocity after collision:

Short answer: $v_1 - 2\left(v_1 \cdot \hat{w}\right)\hat{w}$.

The speed by which the point is moving towards the wall is found by taking the dot product of the velocity with the wall normal: $v_1 \cdot \hat{w}$. The approach velocity is the vector $\left(v_1 \cdot \hat{w}\right)\hat{w}$. This velocity gets reversed by the collision, so the velocity after collision is $v_1 - 2\left(v_1 \cdot \hat{w}\right)\hat{w}$.

(*b*) An object at P_1 is connected to an object at P_2 by an ideal spring having a relaxed distance of d and a spring constant h. Write an expression for the force on P_1 due to this spring.

☑ Expression giving force on P_1 due to spring:

The distance between the points is $||\overrightarrow{P_1P_2}||$. The magnitude of the force due to the spring is given by $||\overrightarrow{P_1P_2}|| - d$. (A positive force value pulls the points towards each other.) The force on P_1 is $\left(||\overrightarrow{P_1P_2}|| - d\right)\widehat{P_1P_2}$, where $\widehat{P_1P_2}$ is vector $\overrightarrow{P_1P_2}$ normalized.

(*c*) Point P_1 in 3D space has homogeneous coordinate $(3, 4, 6, 0.5)$. What is the corresponding Cartesian (ordinary every day) coordinate?

☑ Cartesian coordinate is:

The Cartesian coordinate is $(3/0.5, 4/0.5, 6/0.5) = (6, 8, 12)$.

(*d*) Homogeneous coordinates make it possible to express certain transformations using matrix multiplication that would not be possible with ordinary Cartesian coordinates. Name one such transformation.

☑ Transformation that's **not** possible using matrix multiplication with Cartesian coordinates:

There were two covered in class, translation and projection. Either answer would be correct. A scale transformation can be done using a feature of homogeneous coordinates, but scale transformations can also be done for Cartesian coordinates, and so scale transformation is not a correct answer.

Problem 4: [12 pts] Answer each question below.

(*a*) Our user's eye is at location (x, y, z) in our coordinate system. How do we provide this information to OpenGL (using the compatibility profile)? Be specific.

☑ User's location given to OpenGL by ...

Short Answer: We specify a modelview matrix that translates point (x, y, z) to the origin.

Long Answer: The location of a user's eye is specified using the OpenGL modelview matrix. To provide this information we must specify a modelview matrix that moves the user's eye to the origin. Moving the user's eye to the origin only requires a translation transformation, but for the modelview matrix we also want to rotate the scene so that our user's computer monitor is facing the $-z$ direction.

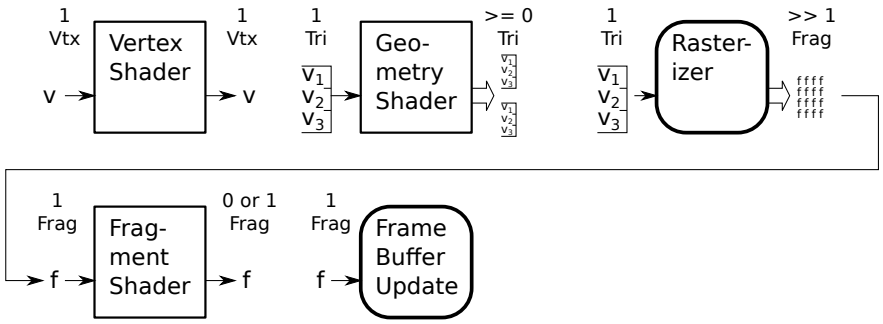
(*b*) The series of transformations performed by OpenGL (and shaders) maps 3D coordinates in our coordinate space to *window space* which are the 2D window coordinates corresponding to pixel locations. This transformation preserves coordinates' z component. Why is the z component needed, and how is it used?

☑ The z component is used for ...

The z component is used to decide which of two fragments located at the same pixel should be discarded (or how they should be blended together). Normally, the fragment further from the user would be discarded, but OpenGL allows for other possibilities.

This processes occurs after the fragment shader stage. The z component of the fragment to be written is compared to the z component already written into the frame buffer. If the fragment passes this depth test (the comparison) it is written, replacing (or blending with) the fragment that is already there (effectively discarding the old fragment). If the fragment fails the depth test it itself is discarded.

Problem 5: [14 pts] The diagram below shows a simplified form of the OpenGL rendering pipeline.



(*a*) The diagram shows 16 f's at the output of the rasterizer stage, but that's not necessarily the actual number.

☑ What does the actual number of f's depend on?

The number of f's depends upon the *projected* size of the triangle (the size of the triangle on the user's screen), and the resolution of the user's screen.

☑ How could one increase the number of f's without changing the input, $(V_1V_2V_3)$.

One could change the modelview matrix so that the user is closer to the triangle. Another option would be to increase the size of the window or render to a higher resolution device.

Grading Note: For some reason, most people answered "get a higher resolution monitor." Does something about the test remind people of Christmas?

(*b*) The boxes with rounded corners are part of what's called the *fixed functionality*. What does that mean, and why are those stages part of it.

☑ Fixed functionality means ...

The fixed functionality parts of the pipeline are parts that cannot be programmed. The Kronos Group (the organization defining the OpenGL standard) made those parts fixed function for performance reasons.

☑ The Rasterizer and Frame Buffer Update are fixed functionality because ...

... hardware is capable of performing the respective function better than software. Therefore making it programmable might improve flexibility, but it would slow down performance.

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

(*a*) The code below specifies vertices using a triangle strip. Modify the code so that it renders the same triangles using individual triangles.

☑ Modify code to use individual triangles ☑ without changing what's rendered.

```
glBegin(GL_TRIANGLES);

for ( int i=0   ;    i<num_v-2   ;    i++    )
  {

    glVertex3fv( coord[i] );
    glVertex3fv( coord[i+1] );
    glVertex3fv( coord[i+2] );

  }
glEnd();
```

(*b*) There is an important difference between specifying a color using `glColor` and `glMaterial`. One of them allows each vertex to have its own color, the other allows different properties to be set, such as ambient, diffuse, and specular but all vertices in a rendering pass must share these colors. *Note: This question applies only to the compatibility profile.*

☑ The command that allows each vertex to have its own color is:

The command is `glColor`, which specifies a vertex attribute.

☑ Why can't each vertex have its own set of color properties using either of these commands?

To limit the amount of data needed for each vertex. If each vertex *could* have its own full set of colors that would require more than eight 4-element vectors (values for the four types of material properties for each of two sides).

(*c*) Describe how the `GL_LINEAR_MIPMAP_LINEAR` minification method computes a filtered texel. Indicate how many texels are read from texture images, which images are read, and how they are combined.

☑ Number of texture images read:

An original texture image is used to construct a series of MIPMAP layers, with successively smaller widths, down to a width of one pixel. For `GL_LINEAR_MIPMAP_LINEAR` filtering two layers (texture images) are read, the two whose texels are closest in size to the pixel to be written.

☑ Number of texels read:

Four texels are read from each layer, for a total of eight texels.

☑ How they are combined:

A weighted average of the texel values is computed. The weight is based on how much of the texel covers the pixel to be written.

Name Solution

GPU Programming
EE 4702-1
Final Examination

Tuesday, 9 December 2014 7:30–9:30 CST

Problem 1		(20 pts)
Problem 2		(15 pts)
Problem 3		(20 pts)
Problem 4		(20 pts)
Problem 5		(25 pts)
Exam Total		(100 pts)

Alias Methane?

Good Luck!

Problem 1: [20 pts] Part of the vertex shader from Homework 6 (which is similar to the vertex shader in earlier assignments) appears below. For the questions below let c denote the chain length (the number of balls), let s denote the value of `opt_segments` (the number of iterations of the `t` loop in the CPU code), and let $n = cs$ denote the number of "vertices" sent to the vertex shader.

```
in ivec2 gl_Vertex; // Vertex Shader Inputs

out Data_to_GS {    // Interface block for vertex shader output / geometry shader input.
  vec4 vertex_e[2][2];  // Vertex coordinates in eye space.
  vec4 position[2][2];  // Vertex coordinates in clip space.
  vec2 texCoord[2];
  vec3 normal_e[2];
  vec3 radial_e;  // Normal for edge primitives.
};

void vs_main_lines() {
  const int bidx = gl_Vertex.x;
  const int ti = gl_Vertex.y;
  const int radial_idx = bidx * opt_segments + ti;
  const float delta_t = 1.0 / opt_segments;
  const float t = float(ti) * delta_t;
  const float theta = delta_t * radial_idx * omega;
  vec3 pos1 = balls_pos[bidx-1].xyz;                  // Buffer object access.
  vec3 pos2 = balls_pos[bidx].xyz;                    // Buffer object access.
  vec3 v12 = pos2.xyz - pos1.xyz;
  // ...
```

(a) For an entire rendering pass (execution of `glDrawArrays`) how much data is sent from the CPU to the GPU for this vertex shader? The answer should be in bytes and in terms of c, s, and n.

☑ Amount of data from CPU to GPU:

Short answer: Amount of $\boxed{\text{data is } (8n + 16c)\,\text{B}}$.

This particular vertex shader reads data two ways: as vertex-shader inputs (which includes data specified using `glVertex` and `glColor`) and via the access to `balls_pos`.

Though in general a vertex shader can read all kinds of input, such as the values provided by `glNormal` and `glTexCoord`, this vertex shader only reads inputs `gl_Vertex.x` and `gl_Vertex.y`. (Those who took the course in the Fall 2014 semester should remember that, because the vertex shader was used as an example of how one can compute attributes such as vertex coordinates, normals, and texture coordinates given just two integers and access to an array [`balls_pos`]. Future students who are reading this solution might suspect that the code following the ellipsis accesses additional vertex inputs. That would be a reasonable assumption but this solution will be based on the Homework 6 version of the code.)

As we can see from the declaration, the `gl_Vertex` input is a two-integer vector, and so has a size of $8\,\text{B}$. The vertex shader is called once per vertex, so the amount of data sent to the GPU due to the vertex shader inputs is $8n\,\text{B}$.

The vertex shader also reads the array `balls_pos`, which is bound to a buffer object. Because the physics simulation updates ball positions, the buffer object needs to be updated every frame. Each element of `balls_pos` is a four-float vector, which occupies $16\,\text{B}$. (We can see that the code above only uses three components, for example, by looking at `balls_pos[bidx].xyz`. However the presence of the swizzle operator (`.xyz`) implies that each element is a four-element vector.)

There is one element for each ball, so for c balls the total size of `balls_pos` (and the buffer object to which it is bound) is $16c\,\text{B}$.

Counting both the vertex shader inputs and the buffer object, the total amount of data sent is $(8n + 16c)\,\text{B}$.

(b) For an entire rendering pass (execution of `glDrawArrays`) how much data is output by the vertex shader for the geometry shader? The answer should be in bytes and in terms of c, s, and n.

☑ Amount of data output from vertex shader:

Based on the interface block, the amount of data written by each invocation of the vertex shader is as follows. Member `vertex_e` is a 2×2 array of 4-component vectors. Each component is a float which is 4 bytes. The total size is $2 \times 2 \times 4 \times 4\,\mathrm{B} = 64\,\mathrm{B}$. The member `position` is also $64\,\mathrm{B}$. Member `texCoord` is $16\,\mathrm{B}$, `normal_e` is $24\,\mathrm{B}$, and `radial_e` is $12\,\mathrm{B}$. The total size of the output is $180\,\mathrm{B}$. That's for one invocation of the vertex shader. The vertex shader is run n times so the total size of the output data is $\boxed{180n\,\mathrm{B}}$.

(c) For an entire rendering pass (execution of `glDrawArrays`) how much data is read by the geometry shader (not shown) per frame? It's important to remember that the input to the vertex shader is line strips and the input to the geometry shader is lines. The answer should be in bytes and in terms of c, s, and n.

☑ Amount of data input to geometry shader:

The vertex shader is run once for each vertex sent (or specified) by the CPU. The number of times the geometry shader is run depends on the input primitives. The CPU sends vertices as line strips. The geometry shader reads them as lines. To read a line the geometry shader needs to read two primitives, for a size of $360\,\mathrm{B}$. The geometry shader will be run once per line. Because line strips are being used the geometry shader is run $n - 1$ times. (If the CPU had send the data as individual lines the geometry shader would have been run $n/2$ times.) The total amount of data read by the geometry shader is therefore $\boxed{360(n-1)\,\mathrm{B}}$.

Problem 1, continued: Recall that `balls_pos` is an array of ball positions, something that changes each frame. Note that the vertex shader is reading the positions from a buffer object.

It would be possible, though wasteful, to use a vertex shader input for the ball positions.

(*d*) Show the vertex shader input declarations that can be used for ball positions. *Hint: This part is easy, look at the code on the previous page.*

☑ Declarations:

The solution appears below. Two `vec4`'s are declared, corresponding to the two ball positions read by the vertex shader. This looks harmless, until you compare the amount of data that would have to be sent for these inputs in comparison to the amount of data when using buffer objects. See the next subproblem.

```
// SOLUTION
in ivec2 gl_Vertex; // Vertex Shader Inputs
in vec4 ball_pos_i;
in vec4 ball_pos_im1;
```

(*e*) Compute the amount of data sent from the CPU to the GPU needed when the ball positions are read as vertex shader inputs. The answer should be in bytes and in terms of c, s, and n.

☑ Amount of data for ball positions as vertex shader inputs:

The two ball positions together consume 32 bytes per vertex shader invocation, and so the total amount of data is $\boxed{32n\,\mathrm{B}}$. Recalling that $n = cs$, we can see that this is much larger than the amount of data sent when buffer objects are used. That's because when vertex inputs are used the position of each ball must be sent from the CPU to the GPU $2s$ times, with buffer objects it is sent just once.

Problem 2: [15 pts] The fragment shader used in many class demonstrations and homework assignments performs lighting calculations. This yields the best results, but is computationally expensive. The historic location for computing lighting is the vertex shader.

(*a*) With the help of a diagram describe how fragment-shader lighting computation yields a more realistic image than vertex-shader lighting computation.

☑ Fragment shader lighting more realistic because:

Because the lighted color of a fragment is based upon the position of the light in relation to that fragment. In contrast, when the vertex shader is used to compute lighting the lighting is computed for the primitive's vertices and these values are interpolated to get the lighted color of a fragment. Suppose the light were close to the center of a triangle. We would expect there to be a brightly-lit pool near the light and the vertices to be darker. We get just that with fragment-shader computed lighting. With vertex shader lighting the triangle could have uniform coloring (if the light were equidistant from the vertices).

(*b*) How many times more work does it take to compute lighting in the fragment shader? **State any assumptions** used for your answer.

☑ Using the fragment shader for lighting requires $x\times$ more computation assuming:

Assuming that there are $3x$ fragments. Regardless of the projected size we would compute lighting 3 times if using the vertex shader on a triangle, so if the fragment shader requires $x\times$ more computation it must have $3x$ fragments.

(*c*) Doing lighting calculations in the fragment shader does not always make a big difference in appearance. Consider a hybrid scheme in which a fragment-shader input determines whether the fragment shader should do the lighting calculations or just use the `color` input. Which stage is best suited to determine if the fragment shader should do lighting calculations?

☑ Shader stage best suited to determine where to compute lighting?

☑ Explain.

The geometry shader, because it has access to all the vertices in a primitive and so has the information it needs to determine if the light is much closer to the interior of a primitive than to any vertex. (The geometry shader, and all programmable stages have access to light source coordinates and other information.) The vertex shader can't do this because it can only see one vertex. The fragment shader does not normally have access to all the vertices, but even if it did it would not be suitable because just determining whether or not to perform lighting calculations would use a substantial amount of computing power (because the fragment shader is often executed many more times than the vertex shader, as those who read the solution to the previous subproblem know).

Problem 3: [20 pts] The CUDA kernel below computes an operation on all pairs of elements of array `d_pos`, the number of array elements is `chain_length`. Let B denote the block size of the kernel launch. The grid size (number of blocks) is (and must be) `chain_length`.

```
__global__ void force_pairs()
{
  const int a_idx = blockIdx.x;
  const float4 pos_a = d_pos[a_idx];                    // Global Access A

  float3 force = make_float3(0,0,0);

  for ( int b_idx = threadIdx.x;  b_idx < chain_length;  b_idx += blockDim.x)
    {
      const float4 pos_b = d_pos[b_idx];
      force += force_compute(pos_a,pos_b);
    }

  d_vel[a_idx] += delta_t * d_balls[a_idx].mass_inv * force; // Error here.
}
```

(*a*) Compute the amount of data read for the entire kernel due to the line Global Access A. The size of a `float4` is 16 bytes and the minimum request size is 32 bytes. Compute this number in terms of B (block size) and c (`chain_length`). *Note: In the original exam n was used for chain length.*

☑ Amount of data read due to Access A.

Short answer: $\boxed{\frac{32cB}{32}\,\mathrm{B} = cB\,\mathrm{B}}$. (Don't get confused between B which is the block size and B which is the unit *byte*.)

The total number of threads is the product of the block size and the number of blocks, Bc in this case. Global Access A accesses element at index `a_idx`, which is set to `blockIdx.x`. Recall that all threads in a block share the same value of `blockIdx.x` and so only c different elements of `d_pos` are being read. However, the code above can access the same element multiple times. In class we assumed that requests are formed based on the combined accesses of all threads in a warp. For the code above all threads in a warp are accessing the same element, so a single request is formed for a warp. The size of that request is 32 bytes, the minimum size request (which is wasteful for us since we need only 16 bytes of data). To determine the total amount of data accessed we need to compute the number of warps. That's the number of threads divided by 32, or $cB/32$ in our case. (We will conveniently assume that B is a multiple of 32.) That gives us the total data size of $\frac{32cB}{32}\,\mathrm{B} = cB\,\mathrm{B}$.

(*b*) Modify the code so that it uses shared memory to reduce the amount of data read due to Global Access A.

☑ Modify to use shared memory to reduce data read for A.

The modified code appears below. Variable `pos_a` is now declared shared. Only `threadIdx.x` 0 initializes it, so there is just one global access per block. The `__syncthreads` prevents the other threads from accessing `pos_a` until thread 0 writes it.

```
__global__ void force_pairs() {
  const int a_idx = blockIdx.x;
  // SOLUTION below
  __shared__ float4 pos_a;  // Declare pos_a shared.
  if ( threadIdx.x == 0 ) pos_a = d_pos[a_idx]; // Just 1 thread writes it.
  __syncthreads();   // All threads must wait until thread 0 writes it.
  // SOLUTION above.

  float3 force = make_float3(0,0,0);
```

Problem 3, continued: The CUDA code below is the same as the previous page. The line marked "Error here" has two problems.

```
__global__ void force_pairs()
{
  const int a_idx = blockIdx.x;
  const float4 pos_a = d_pos[a_idx];                  // Global Access A

  float3 force = make_float3(0,0,0);

  for ( int b_idx = threadIdx.x;  b_idx < chain_length;  b_idx += blockDim.x)
    {
      const float4 pos_b = d_pos[b_idx];
      force += force_compute(pos_a,pos_b);
    }

  d_vel[a_idx] += delta_t * d_balls[a_idx].mass_inv * force; // Error here.
}
```

(*c*) Due to one problem the wrong value will be written into `d_vel`. Explain why.

☑ Wrong value written to `d_vel` because ....

The `+=` operator reads memory, adds something to the value, then writes the sum. The problem is that multiple threads can be executing that statement at the same time, for example, all the threads in a warp. Suppose we wanted each thread in a warp to add 1 and that the original value were 10, we would want the final value to be equal to 42 (the original value plus 32). But, if each thread reads a 10, adds one to it, then writes the result, we'd have 32 threads writing 11. So we'd get 11 instead of 42.

(*d*) The line marked "Error here" is also inefficient. Explain why and fix the problem. *Hint: Use shared memory.*

☑ Reason for inefficiency.

☑ Fix with shared memory.

Global memory access is slow, especially when many threads are accessing the same location.

Notice that all threads in a block are accessing the same elements of `d_vel` and `d_balls`. Also notice that the same result would be obtained if we added up the values of `force` for each thread in a block, and used that value to update `d_vel`. The code below does just that. A shared array `forces` is used to hold the force computed by each thread. A block-wide sum of forces is computed in two steps. In the first step, performed by the first warp in a block, the sum of forces in each lane is computed. (The lane is `threadIdx.x` modulo the warp size, 32.) In the second step, performed by just one thread, the force sums for each lane are added together yielding a block-wide sum. That is used to update `d_vel`.

```
// SOLUTION below
__shared__ float3 forces[1024];  // Array for sharing forces.
forces[threadIdx.x] = force;     // Write thread's computed force.
__syncthreads();
if ( threadIdx.x >= 32 ) return;

// Compute force sum for each lane.
for ( int i=threadIdx.x+32; i<blockDim.x; i+=32 )
  force[threadIdx.x] += forces[i];

if ( threadIdx.x > 0 ) return;
```

```
// Compute force sum for entire block.
for ( int i=0; i<32; i++ ) forces[0] += forces[i];

d_vel[a_idx] += delta_t * d_balls[a_idx].mass_inv * forces[0];
```

Problem 4: Answer each question below.

(*a*) [5 pts] A CUDA application launches a kernel with a grid size of 20 blocks. Describe a situation in which that is inefficient and explain why.

☑ Twenty blocks is inefficient when:

The number of multiprocessors does not evenly divide 20. For example, if the number of multiprocessors were 3, 6, 7, 9, 11 to 19, and 21 or higher. Twenty blocks can be efficient when the number of multiprocessors is 1, 2, 4, 5, 10, or 20.

☑ Because:

Because the work cannot be divided evenly and so some multiprocessors would finish early. If our goal were minimum computation time we'd want all the multiprocessors to be working until the computation finished.

(*b*) [5 pts] A CUDA application launches a kernel with a block size of 19 threads. Why is that inefficient?

☑ Nineteen threads is inefficient because:

Lots of reasons. First, there are not enough threads to use the common functional units on Kepler GPUs. (For example, the 192 single-precision FP units.) Even if there were just 19 functional units of the type needed by the code, additional threads are needed to hide latency. The general rule given in class was 256 threads per multiprocessor.

Problem 4, continued:

(*c*) [5 pts] The loop body in the CUDA code below does one of five kinds of operations. Which is the biggest concern for the code below, grid size, block size, or warp size?

```
const int tid = threadIdx.x + blockIdx.x * blockDim.x;
for ( int i = tid; i<size; i += num_threads ) {
  int k = kind[i];
  float4 d = data[i];
  switch ( k ) {
    case 0: operation_x(i,d); break;
    case 1: operation_y(i,d); break;
    case 2: operation_z(i,d); break;
    case 3: operation_a(i,d); break;
    case 4: operation_b(i,d); break;
  }
}
```

☑ Biggest concern for code is grid, block, or [warp]. (Circle one.)

☑ Explain:

The warp size is the biggest concern because one instruction is fetched for all threads in a warp, though only the threads that need the instruction will execute it. If `k` had the same value for all threads in a warp, there would be no problem, every thread would need whatever instruction was fetched. But if some threads in a warp retrieved a value of 0 for `k` and others a value of 1, then execution might first follow the path down `case 0`, with the threads getting `k=1` not executing those instructions. Later the path down `case 1` would be followed with the `k=0` threads ignoring instructions. With five cases, execution time could be five times longer. The smaller the warp size the smaller this problem. For example, if the warp size were just two then at worst execution time would be twice as long. And perhaps it would be easier to cluster `kind` values so that they would be the same for the two members of these hypothetical 2-thread warps.

(*d*) [5 pts] For many classroom CUDA examples a structure is used to hold application related variables, such as the one below. For the GPU, this had been declared in the `__constant__` address space, but it could have been declared in the global (device) address space (shown commented out below). What's the disadvantage of the global space for this?

```
struct App {
  int num_threads, array_size;
  Vector *v_in;
  Vector *d_v_in;
};

App app;  // In host address space.
__constant__ App d_app;  // In device constant address space.
// __device__ App d_app;  // In device global address space.
```

☑ Disadvantage of global address space for the `App` structure?

The global address space is much slower than the constant address space. (The constant address space is much smaller than the global address space and it can't be written by the GPU, but that's not an issue for the `App` structure.)

Problem 5: Answer each question below.

(*a*) [5 pts] The vertex shader below uses the deprecated output `gl_FrontColor` to hold a lighted color. Those setting the OpenGL standard would prefer that you define your own vertex shader output to hold the lighted color. On the other hand, `gl_Position` is not deprecated and it must be written (if not by the vertex shader, then by the geometry shader). Why?

```
void vs_main() {
  gl_Position = gl_ModelViewProjectionMatrix * gl_Vertex;

  vec4 vertex_e = gl_ModelViewMatrix * gl_Vertex;
  vec3 normal_e = normalize(gl_NormalMatrix * gl_Normal);
  gl_FrontColor = generic_lighting(vertex_e,gl_Color,normal_e);
}
```

☑ Cannot substitute `gl_Position` with a user-defined output because:

The value of `gl_Position` is needed by the fixed functionality for clipping and rasterizing. In contrast the fixed functionality never uses the lighted color (except for the deprecated compatibility profile) so there is no need for OpenGL to define a name for it.

(*b*) [5 pts] It's possible to define user-defined outputs for the vertex and geometry shaders, but not for the fragment or compute shaders.

☑ Why can't there be user-defined fragment shader outputs?

A user-defined output of one stage can only be recognized if it's a user-defined input of a subsequent stage. (For example, if you define a vertex shader output named `yadayada`, you need to define a geometry shader input named `yadayada` to use the result.) The fragment shader is the last programmable stage, so there's nothing left to recognize a user-defined output.

☑ Why can't there be user-defined compute shader outputs?

The vertex, geometry, and fragment shaders are part of the rendering pipeline. The term *pipeline* indicates that data flows through these shaders in order, with the output of one stage being sent to the input of another (with possible re-grouping or rasterizing). The compute shader is not part of the rendering pipeline, so it does not have or need inputs and outputs.

(*c*) [5 pts] In the OpenGL shader code below `b` is assigned the value of `a`, but with the elements reversed. It does so the hard way. Do it the easy way (using what we called swizzle operations).

```
vec4 a, b;
b.x = a.w;  b.y = a.z;  b.z = a.y;  b.w = a.x;
```

☑ Assign `b` the easy way.

The easy way is `b=a.wzyx`.

(*d*) [5 pts] The fragment shader input's `smooth` interpolation qualifier (which is the default for floating-point values) indicates that the qualified variable (`color` in the example below) should be linearly interpolated based on object-space coordinates. Linear interpolation is easy to do, so why is `smooth` interpolation expensive?

```
in Data_GF {
  smooth vec4 color;
}
```

☑ `smooth` interpolation is expensive because ...

Because we are working with window-space coordinates, and the interpolation is not linear in window space.

(*e*) [5 pts] The statement below is correct but does not provide the full justification for GPUs. Complete the statement.

> Executing 3D graphics is computationally intensive and so many computers have GPUs to take this computation load off the CPU.

☑ To justify GPUs the statement should go on to say ....

... a significant amount of CPU hardware is not needed for typical 3D graphics computations, so a GPU can be built with more of the hardware that it does need, such as floating-point computation units. (The unneeded hardware includes structures that allow instructions to execute out of order and large caches.)

29 Fall 2013 Solutions

Name Solution

EE 4702

Take-Home Pre-Final Questions

Due: 9 December 2013

Please work on this exam alone. You may use references for OpenGL, CUDA, and similar topics.

Problem 1		(20 pts)
Problem 2		(30 pts)
Problem 3		(15 pts)
Problem 4		(20 pts)
Problem 5		(15 pts)
Exam Total		(100 pts)

Alias 0x501f

Good Luck!

Problem 1: [20 pts]The code below is the `data_unpack` routine from the Homework 3 solution. Recall that the code in Homework 3 requires that the number of threads must be no smaller than the number of balls (the value of `chain_length`).

Re-write the routine so that it works correctly even if the number of balls is greater than the number of threads.

☑ Re-write so number of threads can be fewer than `chain_length`.

☑ Compute the number of threads using CUDA-provided variables such as `blockDim`.

☑ Array `dc.d_pos` must be accessed efficiently.

The solution appears below. The code writing `ball->position` is now in a loop. The loop starts with the same ball index (`bi`) in the pre-solution code, which was set equal to the global thread id (shown as `tid` in the solution). The solution code computes the total number of threads and assigns it to `num_threads`. The increment for the loop is `num_threads`, that will guarantee that we get a value of `bi` that no other thread has.

Access is efficient because consecutive threads will have consecutive values of `bi`. We know that because `bi` was initialized to `threadIdx.x + SOMETHING` (where something is the same for all threads in a warp).

```
// SOLUTION
__global__ void data_unpack_mb() {
  const int tid = threadIdx.x + blockIdx.x * blockDim.x;
  const int num_threads = blockDim.x * gridDim.x;
  for ( int bi = tid;  bi < dc.chain_length; bi += num_threads )
    {
      Ball* const ball = &dc.d_balls[bi];
      ball->position = dc.d_pos[bi];
    }
}
```

Problem 2: [30 pts]Recall that the code for Homework 3 simulated a chain of balls (or string of beads). In this problem we'll extend that code by giving the balls electric charge, either positive or negative. The balls are sealed in a special coating that retains the charge.

The CUDA kernel below applies the force due to these charges. Like the penetration routine it must consider nearly all possible pairs. The code is written to optionally use shared memory when accessing the "b" object.

```
__global__ void time_step_gpu_sol_part_m(float delta_t) {
  const int tid = threadIdx.x + blockIdx.x * blockDim.x;
  const int threads_per_object = num_threads / dc.chain_length;

  const int a_idx = tid % dc.chain_length;
  const int b_idx_start = tid / dc.chain_length;

  __shared__ float4 c_pos[1024];

  float4 pos_a = dc.d_pos[a_idx];                                    // Global Access A
  float3 force = make_float3(0,0,0);

  for ( int j=b_idx_start; j < dc.chain_length; j += threads_per_object )
    {
      if ( use_cache )
        {
          __syncthreads();
          if ( a_idx == 0 ) c_pos[b_idx_start] = dc.d_pos[j];         // Global Access B
          __syncthreads();
        }

      if ( a_idx != j )
        {
          float4 pos_b = use_cache ? c_pos[b_idx_start] : dc.d_pos[j];// Global Access C
          pNorm a_to_b = mn(pos_a,pos_b);
          const bool repel = ( a_idx & 1 ) == ( j & 1 );
          force += ( repel ? -0.15f : 0.15f ) / a_to_b.mag_sq * a_to_b;
        }
    }

  const float3 delta_v = delta_t * dc.d_balls[a_idx].mass_inv * force;
  atomicAdd(&dc.d_vel[a_idx].x,delta_v.x);
  atomicAdd(&dc.d_vel[a_idx].y,delta_v.y);
  atomicAdd(&dc.d_vel[a_idx].z,delta_v.z); }
```

Questions are on next page.

Problem 2, continued:

(*a*) In this part assume that `use_cache` is false, and so shared memory won't be used. Compute the amount of data requested due to accesses to `d_pos` and compute how much of that data is actually used. Do this for the following configuration: `chain_length=256`, a block size of 1024 threads, and a grid size of 16 blocks. The code runs on an NVIDIA Kepler GPU which has memory request sizes of 32, 64, and 128 bytes. For your convenience, comments show where global memory can be accessed. (Array `c_pos` is not in global memory, and is anyway won't be needed until the next subproblem.)

☑ Total size of requests (in bytes) to `d_pos` considering all threads.

☑ Amount of data in those requests that was needed.

First consider the assignment of `pos_a`. Let c be the value of `chain_length` and n be the number of threads.

We'll start with something simple: computing the total amount of data written to `pos_a` (not requested) by all the threads. The type of `pos_a` is a `float4`, which has a size of 16 bytes. The total number of threads is the product of the block size and grid size which is $n = 1024 \times 16 = 2^{10}2^4 = 2^{14} = 16384$ threads. Since each thread accesses 16 bytes the total data written to `pos_a` is $16n = 2^{14}2^4 = 2^{18} = 262144$ bytes.

This would be the total size of all requests if each byte in every request is used exactly once. If some bytes are used zero times the total request size will have to be larger than the amount of data written. If some bytes are used multiple times (because two threads in the same warp want the same data) the total request size can be smaller.

Remember that instructions in NVIDIA GPUs (so far up to CC 3.5) execute in groups called warps. When they need to load data all the threads in a warp (actually a half warp, but we won't consider that here) present an array index (actually a memory address and size) to the multiprocessor hardware. Essentially the 32 indices will be sorted, duplicates will be removed, and the remaining indices will be partitioned into *requests*. A request is a command to read memory and consists of a starting address and a size. The size can be 32, 64, or 128 bytes (for CC 2.X and 3.X devices). The goal of the multiprocessor hardware is to partition the indices so as to minimize the number of requests.

To see how this works, consider the following examples. If each thread accesses 1 byte (which does not apply to our code) and the indices are consecutive then a single 32-byte request will be sufficient for a warp. If the threads each accesses an 8-byte quantity and the indices are consecutive, the total the $32 \times 8 = 256$ bytes can be satisfied by two requests. If all threads access *the same* 8-byte quantity there will be a single request of size 32 bytes and three quarters of the request will go unused.

For Kepler (CC 3.5) devices there is no global L1 memory cache, so all requested memory must be used by the requesting warp or it will be lost. That is, if two warps access the same data, then that data must be requested twice.

Getting back to Global Access A. It should be easy to verify that consecutive threads will have consecutive values of `a_idx` and that with a 16-byte element size there is enough data to fill exactly four 128-byte requests. So each requested data item is used exactly once. Therefore for `pos_a` the total amount requested and needed data is $16n$ or 262144 bytes.

Since `use_cache` is false Global Access B does not occur. The load at Global Access C is to an element at index `j`, which starts at `b_idx_start`. It should be easy to verify that for the first 256 threads `b_idx_start` is zero, for threads 256 to 511 it is one, and so on. Therefore, within a warp all threads will have the same value of `j` and so they will all be requesting the same 16-byte value. A single 32-byte request will be dispatched.

For those following what the code is doing should see that the total number of iterations, counting all threads, will be c^2, since we are doing an all-to-all computation. Even without that leap of insight, by examining the `j` loop bounds we can see that the number of iterations is $c/(n/c) = c^2/n$. Since n threads execute the loop, the total number of executions of the body is c^2.

Each warp (or thread) executes c^2/n iterations of the loop requesting $\frac{c^2}{n}32$ bytes of data but using only half of it. The total number of warps is $n/32$ so the total data requested is $\frac{n}{32}\frac{c^2}{n}32 = c^2$ bytes, the amount of data needed is $\frac{1}{2}c^2$ bytes.

Combining the contributions from the two global accesses (ignoring Global Access B since `use_cache` is off): Total data requested is $16n + c^2 = 2^{18} + 2^{16} = 262144 + 65536 = 327680$ bytes. The amount of data needed is $16n + \frac{1}{2}c^2 = 2^{18} + 2^{15} = 262144 + 32768 = 294912$ bytes.

(*b*) Repeat the problem above for the case when `use_cache` is true. Remember that memory requests are not generated when reading or writing `c_pos` itself.

☑ Total size of requests (in bytes) to `d_pos` considering all threads.

☑ Amount of data in those requests that was needed.

The amount of data accessed by Global Access A will be the same as the `use_cache=false` case, $16n$ bytes. The amount of global data accessed for Global Access C will be zero (since it is now accessing shared memory rather than global memory).

We need to think about Global Access B, in which data from global memory is written to shared memory. As with Global Access C, this one uses index `j`. However because of the `if` statement it is not being executed by every thread. In fact `a_idx==0` will only be true for threads in which `tid` is a multiple of c, which will be for $n/c = 2^{14}/2^8 = 2^6 = 64$ threads. It should be easy to verify that `a_idx==0` for at most one thread in a warp. Therefore the request size will be 32 bytes (of which 16 are needed). The total of all requests will be $32 \times \frac{n}{c} = 32 \times 64 = 2048$ bytes per iteration or $32 \times \frac{n}{c}\frac{c^2}{n} = 32c = 8192$ bytes. The total data needed will be half that, 4096 bytes.

In summary, total size of all requests: $16n + 32c = 262144 + 8192$ bytes, amount needed is $16n + 16c = 262144 + 4096$ bytes.

(*c*) The `d_pos` accesses that are used to fill shared memory, `c_pos[b_idx_start] = dc.d_pos[j];`, are wasteful (though the waste is small in proportion to the total amount of data accessed). Explain why the accesses are wasteful and fix the problem.

☑ Fix problem.

☑ Why are these accesses wasteful?

The accesses are wasteful because at most one thread in a warp is active, and so half the request goes unused. (See the solution to the previous problem.) The code below fixes the problem by having the first few threads load shared memory.

```
// SOLUTION
const int b_idx_start_thd_0 = blockIdx.x * blockDim.x / dc.chain_length;
const int threads_per_object_per_block = blockDim.x / dc.chain_length;
int b_to_shared_idx =
  b_idx_start_thd_0 + threadIdx.x % threads_per_object_per_block;

for ( int j=b_idx_start; j < dc.chain_length; j += threads_per_object )
  {
    if ( use_cache )
      {
        __syncthreads();
        // SOLUTION
        if ( threadIdx.x < threads_per_object_per_block )
          {
             c_pos[b_idx_start_thd_0+threadIdx.x] = dc.d_pos[b_to_shared_idx];
             b_to_shared_idx += threads_per_object;
          }
        __syncthreads();
      } }
```

(*d*) The amount of memory read when assigning `pos_a` can be reduced by using shared memory. Modify the code to do so.

☑ Use shared memory to reduce redundant global accesses to initialize `pos_a`.

Solution appears below. The values of `a_idx` are already consecutive, but they repeat four times (for the sample values given for the problem) within a block. Because of this repetition each position is loaded from global memory four times. In the solution each position is loaded only once, a position is written to shared memory by one thread and then read by all the threads that need it.

```
// SOLUTION
__shared__ float4 c_pos[1024];
if ( threadIdx.x < dc.chain_length ) c_pos[threadIdx.x] = dc.d_pos[a_idx];
__syncthreads();
float4 pos_a = c_pos[a_idx];
```

Problem 3: [15 pts]Consider the following excerpt from the Cone code from the Homework 2 solution. (File `hw2-sol.cc` or visit `http://www.ece.lsu.edu/koppel/gpup/gpup/2013/hw2-sol.cc.html`).

```
void render_probb(pCoor base, float radius, pVect to_apex) {
    if ( opt_lod != bo_lod ) { // If needed lod != stored lod.
        bo_lod = opt_lod;

        /// Code for computing coordinates and normals omitted.
        if ( !buffer_obj_coords ) {
            glGenBuffers(1,&buffer_obj_coords);
            glGenBuffers(1,&buffer_obj_norms);
            glGenBuffers(1,&buffer_obj_colors); }                /// SOLUTION

        glBindBuffer(GL_ARRAY_BUFFER, buffer_obj_coords);
        glBufferData(GL_ARRAY_BUFFER, coords.occ() * sizeof(float),
                     coords.get_storage(), GL_STATIC_DRAW);

        glBindBuffer(GL_ARRAY_BUFFER, buffer_obj_norms);
        glBufferData(GL_ARRAY_BUFFER, norms.occ() * sizeof(float),
                     norms.get_storage(), GL_STATIC_DRAW);

        /// SOLUTION BELOW
        PStack<float> repcolors;
        for ( int i=0; i<num_coords; i++ )
          { repcolors += color.r; repcolors += color.g; repcolors += color.b; }
        glBindBuffer(GL_ARRAY_BUFFER, buffer_obj_colors);
        glBufferData(GL_ARRAY_BUFFER, repcolors.occ() * sizeof(float),
                     repcolors.get_storage(), GL_STATIC_DRAW);
        /// SOLUTION ABOVE
      }
    // Transform computation and modelview update omitted.
    glBindBuffer(GL_ARRAY_BUFFER, buffer_obj_colors);            /// SOLUTION
    glColorPointer( 3, GL_FLOAT, 0, NULL );                      /// SOLUTION
    glEnableClientState(GL_COLOR_ARRAY);                         /// SOLUTION

    // Tell OpenGL to get coordinates and normals from buffer objects.
    glBindBuffer(GL_ARRAY_BUFFER, buffer_obj_coords);
    glVertexPointer( 3, GL_FLOAT, 0, NULL);
    glEnableClientState(GL_VERTEX_ARRAY);

    glBindBuffer(GL_ARRAY_BUFFER, buffer_obj_norms);
    glNormalPointer(GL_FLOAT,0,NULL);
    glEnableClientState(GL_NORMAL_ARRAY);

    // Draw the cones. (There will be minor flaws since 1 strip used.)
    glDrawArrays(GL_QUAD_STRIP,0,num_coords);

    // Cleanup code omitted.
```

Questions are on next page.

Problem 3, continued:

(*a*) The code uses buffer objects for vertex coordinates and normals. Why doesn't it also use a buffer object for color?

☑ Why doesn't color need a buffer object?

Because all vertices are assigned the same color. The way OpenGL works, that one color is sent to the GPU just once (rather than once for each vertex), and so there would be little benefit to using a buffer object.

(*b*) Modify the code so that it uses a buffer object for color, even if that's not a good idea.

☑ Modify code so color *does* use a buffer object for color.

Solution appears on the previous page. First, a buffer object was prepared which contained multiple copies of `color`.

In the rendering code function `glColorPointer` was used to indicate that colors would be specified in an array. The function `glColorPointer` was never used in class, but could be found in the OpenGL 4.3 specification section 10.3.2. That section also gives the constant to use with `glEnableClientState`, `GL_COLOR_ARRAY`.

(*c*) Suppose that each time the routine above was called `opt_lod` had a different value. Also suppose that `if ( !buffer_obj_coords )` were somehow changed to `if ( true )`.

☑ What would eventually happen?

By changing the `if` condition to always be true, a new buffer object name is created each time the code is called. The old buffer objects are not destroyed, they continue to hold their data. So the system would eventually run out of memory. (With the `if` condition as it was, the old buffer object name would be re-used on a second call so the data in the old buffer object would be replaced by the new data.)

Problem 4: [20 pts]Consider another excerpt from the Cone code from the Homework 2 solution. (File `hw2-sol.cc` starting at line 266. or visit `http://www.ece.lsu.edu/koppel/gpup/gpup/2013/hw2-sol.cc.html`).

```
// Compute transform moving and positioning code from local to global
// space.
//
const float to_height = to_apex.mag();
pVect from_apex(0,0,1);
pNorm rn(from_apex,to_apex);
const float rot_angle = pangle(from_apex,to_apex);
pMatrix_Translate trans_transl(base);
pMatrix_Rotation trans_rot(rn,rot_angle);
pMatrix_Scale trans_scale(radius);
trans_scale.rc(2,2) = to_height;
pMatrix xform = trans_transl * trans_rot * trans_scale;

// Specify our transformation to OpenGL.
//
glMatrixMode(GL_MODELVIEW);
glPushMatrix();
glMultTransposeMatrixf(xform.a);

if ( !dont_set_color ) glColor3fv(color);

glBindBuffer(GL_ARRAY_BUFFER, buffer_obj_coords);
glVertexPointer( 3, GL_FLOAT, 0, NULL);
glEnableClientState(GL_VERTEX_ARRAY);

glBindBuffer(GL_ARRAY_BUFFER, buffer_obj_norms);
glNormalPointer(GL_FLOAT,0,NULL);
glEnableClientState(GL_NORMAL_ARRAY);

glDrawArrays(GL_QUAD_STRIP,0,num_coords);
```

The code at the top of the excerpt computes a transform matrix that will position and scale the cone to the desired location. The transform is computed in terms of coordinate `base`, scalar `radius`, and vector `to_apex`. The command `glMultTransposeMatrixf` takes the transform matrix we computed and multiplies it by the existing modelview matrix. The resulting matrix is written as an OpenGL Shading Language uniform with name `gl_ModelViewMatrix`.

In this problem we'll consider computing the transform on the GPU instead. The GPU code will compute the transform in terms of `base`, `radius`, and `to_apex`.

Questions are on next page.

Problem 4, continued:

(*a*) Declare `base`, `radius`, and `to_apex` as uniforms as they would be in an OpenGL SL source file. *Hint: Use the demo-10 (Vertex and Geometry Shaders) code as an example, and look for* `wire_radius`. *The demo-10 code can be accessed from the repo or*
`http://www.ece.lsu.edu/koppel/gpup/code/gpup/demo-10-shader.cc.html` *(CPU code),*
`http://www.ece.lsu.edu/koppel/gpup/code/gpup/demo-10-shdr-simple.cc.html` *(simple shaders), and*
`http://www.ece.lsu.edu/koppel/gpup/code/gpup/demo-10-shdr-geo.cc.html` *(more elaborate shaders).*

Solution appears below. Notice that the names of the vector data types in OpenGL shading language and CUDA differ, so base is `vec4`, not `float4`.

```
// SOLUTION
layout ( location = 1 ) uniform vec4 base;
layout ( location = 2 ) uniform float radius;
layout ( location = 3 ) uniform vec3 to_apex;
```

(*b*) Write code to send the uniforms from the CPU to the GPU. *Hint: See the previous hint.*

The solution appears below. The command `glUniformX` writes a value into a uniform declared in an OpenGL shading language program (as was done for the previous part). The first argument specifies the location, see above. The remaining arguments are the values. Note that the digit near the end of the command specifies the number of arguments and that the last letter specifies the data type, float in this case.

```
glUniform4f(1, base.x, base.y, base.z, base.w );
glUniform1f(2, radius);
glUniform3f(3, to_apex.x, to_apex.y, to_apex.z);
```

(*c*) Suppose that `xform` is computed in a vertex shader making use of the enormous floating-point capability of the mighty GPU. Do you expect execution to be faster or slower than using the CPU to compute xform? Explain.

Slower than the CPU. An individual GPU thread is slower than a CPU thread. The CPU computes the transform **once** for use by all vertices, so the GPU's parallelism is no advantage. Each vertex shader would need to compute the transform and would take more time to do so than the CPU.

(*d*) Suppose that `xform` is computed in a geometry shader. Will execution be faster or slower than using the vertex shader when we are rendering quad strips? (That is, the systems we are comparing both use quad strips, one uses the vertex shader to compute the xform, the other uses the geometry shader.) Explain. *Note: I should have asked about triangle strips.*

Assuming that lighting and other calculations performed by the vertex shader to not take much time, the geometry shader will be faster. With quad strips, most vertices are shared by two quads and so the number of vertices is twice the number of quads. That means the transform would be computed twice as many times if the vertex shader were used. (The answer would be the same number of times if triangle strips were used.)

(*e*) Suppose that `xform` is computed in a geometry shader. Will execution be faster or slower than using the vertex shader when we are rendering individual quads? (That is, the systems we are comparing both use individual quads, one uses the vertex shader to compute the xform, the other uses the geometry shader.) Explain.

With individual quads the geometry shader has a big advantage, since the vertex shader is called four times for each quad and so the transform would be computed four times as often if the vertex shader were used.

Problem 5: [15 pts]Answer each question below.

(*a*) Texture access is performed in the fragment shader. Suppose texture access was performed by the vertex shader and the texel values were interpolated, the same way other attributes such as color were.

☑ What impact would that have on appearance?

Yuck! Suppose the texture image was of a house, and the whole house was supposed to appear on the primitive. Since texels are grabbed at the vertices, only three texels would be retrieved and their colors would be interpolated. Unless the triangle were small the image of the house would be unrecognizable.

☑ What impact would that have on performance?

Performance would be alot higher since texture access and filtering is computationally costly and the fragment shader is called many more times than the vertex shader for typical primitives.

(*b*) Inputs to a fragment shader have interpolation qualifiers, they are `flat`, `noperspective`, and `smooth`. Why are they not needed for the inputs to the geometry shader?

☑ Interpolation qualifiers not needed for geometry shader inputs because:

Interpolation only has meaning for fragments. (The points that fill the area of a primitive.) The geometry shader operates on primitives, say the three vertices of a triangle. So there is nothing to interpolate.

(*c*) The OpenGL command `glMatrixMode(GL_MODELVIEW)`, used in setting the modelview matrix, is deprecated. (Meaning that its use is discouraged and that it may be removed from the language.) Why was the command once essential but now considered unnecessary?

☑ `glMatrixMode` now unnecessary because:

Because parts of the rendering pipeline are now programmable, including the vertex shader. The vertex shader is supposed to transform object-space coordinates to clip space. The fixed-functionality used to do that using, in part, the modelview matrix. Since the fixed functionality no longer computes clip space coordinates there is no need to have a way of specifying a modelview matrix. The use can pass the model view matrix (or the user's equivalent) as a user-defined variable.

Name Solution

GPU Programming
EE 4702-1
Final Examination

12 December 2013, 15:00–17:00 CST

Problem 1		(25 pts)
Problem 2		(25 pts)
Problem 3		(15 pts)
Problem 4		(10 pts)
Problem 5		(15 pts)
Problem 6		(10 pts)
Exam Total		(100 pts)

Alias Click Here

Good Luck!

Problem 1: (25 pts) The code below computes forces between all pairs of a set of balls, similar to the code from the pre-final and Homework 3. Unlike those prior versions, the code below is written for the case where the number of balls (`chain_length`) is a multiple of the total number of threads.

Let c denote the number of balls, n denote the total number of threads (`num_threads`), and g denote the grid size (number of blocks). The size of a `float4` is 16 bytes. The code runs on an NVIDIA Kepler GPU, which supports requests of size 32, 64, and 128 bytes.

```
const int tid = threadIdx.x + blockIdx.x * blockDim.x;
const int num_threads = blockDim.x * gridDim.x;
const int a_idx_start = tid;
const int balls_per_thread = dc.chain_length / num_threads;
for ( int i=0; i<balls_per_thread; i++ )
  {
    const int a_idx = a_idx_start + i * num_threads;
    const float4 pos_a = dc.d_pos[a_idx];                    // Global Access A
    float3 force = make_float3(0,0,0);
    for ( int b_idx = 0;  b_idx < dc.chain_length;  b_idx++ )
      {
        const float4 pos_b = dc.d_pos[b_idx];                // Global Access B
        force += force_compute(pos_a,pos_b);
      }
    dc.d_vel[a_idx] += delta_t * dc.d_balls[a_idx].mass_inv * force;
  }
```

(*a*) In terms of n, c, and g compute the amount of data requested and used for the line marked Global Access A.

☑ Data requested by Global Access A. ☑ Data used by A.

Important thing to note: Each value of `a_idx` is used exactly once. That is, no two threads will have the same value of `a_idx_start` and the same is true for `a_idx` because it is `a_idx_start` plus a multiple of the number of threads.

Short answer: Efficient access, so amount of data requested and used is $16c$ bytes.

Long answer: First consider the first `i` iteration, where `i=0`. For that first iteration `a_idx = tid`, which means we obviously have consecutive threads accessing consecutive elements. Each element is 16 bytes, so for four 128-bytes are needed for each warp, and all 128-bytes are used. This of course, is the most efficient type of access.

It should also be obvious that together all n threads are accessing elements 0 to $n-1$, and that each of these elements is read by exactly one thread. Because `a_idx` is `threadIdx.x` plus something, (where something is `blockIdx.x*blockDim.x+i*num_threads`), we know that consecutive threads will be accessing consecutive elements in all `i` iterations. In the second `i` iteration we'll be accessing elements n to $2n-1$, so again no element has been accessed more than once. This holds for all iterations, so in total $\frac{c}{n}n = c$ elements have been accessed, each just once. The size of an element is 16 bytes, so the total data requested is $\boxed{16c \text{ bytes}}$.

(*b*) In terms of n, c, and g compute the amount of data requested and used for the line marked Global Access B.

☑ Data requested by Global Access B. ☑ Data used by B.

Important thing to notice: Every thread in a warp uses the same value of `b_idx`. A multiprocessor will generate a single memory request for `dc.d_pos[b_idx]` on behalf of all of the threads.

Each `b_idx` iteration requests 32 bytes per warp (the minimum request size), but uses only 16. There are a total of $\frac{n}{32}$ warps so the total request is n bytes per `b_idx` iteration. There are c `b_idx` iterations and $\frac{c}{n}$ `i` iterations, for a total request of $\boxed{\frac{c}{n}cn = c^2 \text{ bytes requested}}$ and $\boxed{\frac{1}{2}c^2 \text{ bytes used}}$.

Problem 2: (25 pts) The code below is similar to the code from the previous problem except that shared memory is used to hold values needed for `pos_b`. The value of `csize` is not well chosen given the way the code is written. As before let c denote the number of balls, n denote the total number of threads, and g denote the grid size (number of blocks). The size of a `float4` is 16 bytes. The code runs on an NVIDIA Kepler GPU, which supports requests of size 32, 64, and 128 bytes.

```
int num_threads = blockDim.x * gridDim.x,  tid = threadIdx.x + blockIdx.x * blockDim.x,
int a_idx_start = tid,                     balls_per_thread = dc.chain_length / num_threads;
const int csize = 8;
__shared__ float4 c_pos[csize];
for ( int i=0; i<balls_per_thread; i++ ) {
    const int a_idx = a_idx_start + i * num_threads;
    const float4 pos_a = dc.d_pos[a_idx];                  // Global Access A
    float3 force = make_float3(0,0,0);
    for ( int b_idx = 0;  b_idx < dc.chain_length;  b_idx++ ) {
        const int c_idx = b_idx % csize;
        if ( c_idx == 0 ) {
            const int loc = threadIdx.x % csize;

            __syncthreads();

            c_pos[loc] = dc.d_pos[b_idx + loc];            // Global Access B

           //  In table:  SH = loc;   GB = b_idx + loc;

            __syncthreads();

          }
        const float4 pos_b = c_pos[c_idx];                // SA = c_idx;
        force += force_compute(pos_a,pos_b);
      }
    dc.d_vel[a_idx] += delta_t * dc.d_balls[a_idx].mass_inv * force;
  }
```

(*a*) In terms of n, c, and g compute the amount of data requested and used for the line marked Global Access B.

☑ Data requested by Global Access B. ☑ Data used by B.

Relatively short answer: The line executes every 8 `b_idx` iterations, when it does 8 consecutive locations are read (these same 8 locations repeat). When they do execute each warp reads 8 distinct locations or $8 \times 16\,\mathrm{B} = 128\,\mathrm{B}$. The total number of executions per thread is $\frac{c}{n}c\frac{1}{8} = \frac{c^2}{8n}$. The total number of warps is $\frac{n}{32}$. The total number of executions considering all warps is $\frac{n}{32}\frac{c^2}{8n} = \frac{c^2}{256}$. As stated above, each execution of Global Access B by a warp accesses $128\,\mathrm{B}$ and so the total data accessed is $\frac{c^2}{256}128\,\mathrm{B} = \frac{c^2}{2}\,\mathrm{B}$. Note that the method used to compute data access changes when `csize` is 32 or larger.

Here's what the code is doing. Instead of performing Global Access B *every* iteration the code does the access once every 8 (or `csize`) iterations and the loaded values are put in shared memory for use in the current iteration and the next seven iterations. The table below shows what is being accessed by threads of `tid` from 0 to 9. Each row shows information on one thread. The numbers at the top of the columns show the values of `b_idx`. The numbers in the `GB` column show which global locations are being accessed (look for `GB` in the code above), and the `SH` column shows to which shared memory location they are being written. The `SA` column shows which shared memory location is being read. Notice that under the `GB` column access is consecutive with `tid`, which is good, but that the same array index repeats (starting in the row for `tid` 8).

b_idx ->		0		0	1	2	3	4	5	6	7	8		8	9
i	tid	SH<-GB		SA	SA	SA	SA	SA	SA	SA	SA	SH<-GB		SA	SA
0	0	0	<- 0	0	1	2	3	4	5	6	7	0	<- 8	0	1
0	1	1	<- 1	0	1	2	3	4	5	6	7	1	<- 9	0	1
0	2	2	<- 2	0	1	2	3	4	5	6	7	2	<-10	0	1
0	3	3	<- 3	0	1	2	3	4	5	6	7	3	<-11	0	1
0	4	4	<- 4	0	1	2	3	4	5	6	7	4	<-12	0	1
0	5	5	<- 5	0	1	2	3	4	5	6	7	5	<-13	0	1
0	6	6	<- 6	0	1	2	3	4	5	6	7	6	<-14	0	1
0	7	7	<- 7	0	1	2	3	4	5	6	7	7	<-15	0	1
0	8	0	<- 0	0	1	2	3	4	5	6	7	0	<- 8	0	1
0	9	1	<- 1	0	1	2	3	4	5	6	7	1	<- 9	0	1

(*b*) Choose `csize` to minimize memory access by Global Access B.

☑ Show new value of `csize`. ☑ Explain.

Short answer: We want to avoid multiple threads accessing the same value of `d_pos`, we can do that by setting `csize` to the block size. If we do that values for `GB` won't repeat in the table above on a particular block. The amount of data accessed per `i` iteration will be $16gc\,\mathrm{B}$ and in total $\frac{c}{n}16gc\,\mathrm{B} = 16\frac{gc^2}{n}\,\mathrm{B}$.

Let s denote the value of `csize`. The amount of data accessed for $s \geq 32$ can be derived as follows. Considering all threads, each time Global Access B executes the amount of data read is $n16\,\mathrm{B}$ (even though only $16s\,\mathrm{B}$ worth of distinct data is read). The number of times it executes is $\frac{c}{n}\frac{c}{s} = \frac{c^2}{ns}$. The amount of data read is $\frac{c^2}{ns}n16\,\mathrm{B} = 16\frac{c^2}{s}\,\mathrm{B}$ for valid values of s. The largest we can make s is the block size, n/g, for that value the amount of data read is $16\frac{c^2}{n/g}\,\mathrm{B} = 16\frac{gc^2}{n}\,\mathrm{B}$. Re-writing this in terms of the block size $b = n/g$ we get $16\frac{n/bc^2}{n}\,\mathrm{B} = 16\frac{c^2}{b}\,\mathrm{B}$.

(*c*) Modify the code so that the amount of data accessed due to Global Access B matches the previous part when `csize` is set to the original value, 8. The change should be within the `if ( c_idx==0 )` block and it is only one line of code.

☑ Modify code to reduce waste when `csize=8`.

Looking the first `GB` column in the visible part of the table from the solution above we see that two threads access element 0, and both write it into the same location in shared memory. That's wasteful, only one thread needs to do that. The solution is to only allow the first `csize` threads to load shared memory, but it's important that all threads call `__syncthreads`:

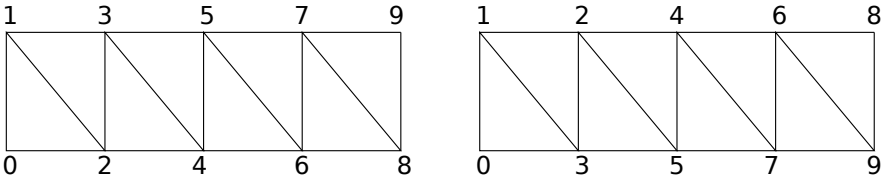
```
for ( int b_idx = 0;  b_idx < dc.chain_length;  b_idx++ ) {
    const int c_idx = b_idx % csize;
    if ( c_idx == 0 ) {
        const int loc = threadIdx.x;
        __syncthreads();
        if ( threadIdx.x < csize )    // SOLUTION: This line inserted.
           c_pos[loc] = dc.d_pos[b_idx + loc];        // Global Access B
```

Problem 3: (15 pts) Answer the following questions about triangles.

(*a*) The diagrams below show two possible numberings for vertices for rendering a rectangle as a triangle strip.

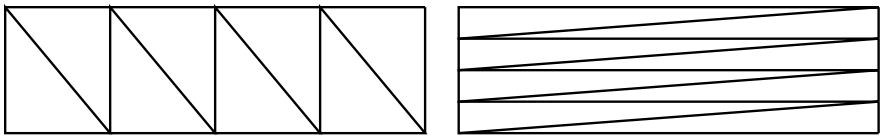
☑ Which one is correct?

☑ Show how the triangles would be rendered for the incorrect case.



The figure on the left is correct. If the numbering on the right were used the diagonals would run from lower-left to upper-right and the line between 0 and 3 would be missing.

(*b*) Several possible strategies are being considered to render a rectangle.



Suppose the left figure above were rendered as a triangle strip (with the vertices correctly numbered).

☑ How many times would the vertex shader be executed?

Ten times, once per vertex.

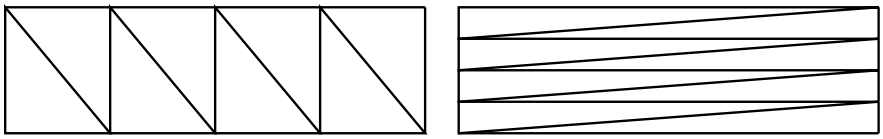
Suppose the left figure above was rendered as individual triangles.

☑ How many times would the vertex shader be executed?

There are eight triangles and so 24 vertices. So the shader would be run 24 times.

Problem 3, continued:

(*c*) Consider the following two ways of tessellating a rectangle.



Suppose that lighting is computed in the vertex shader.

☑ Which tessellation above is better? ☑ Explain.

The tessellation on the right is worse because vertices of each triangle are far away from each other. Lighting is computed at the vertices and interpolated. Suppose a light source were at the center of the rectangle. For the tessellation on the left the center triangles would be more brightly lit than those at the ends. But for the tessellation on the right they would all have about the same brightness.

(*d*) Suppose that lighting were being computed in the fragment shader.

☑ Why would either tessellation above be wasteful?

Because there would be no need for eight triangles. The entire area could be covered by just two triangles. The waste is in running the vertex shader 10 times (for eight triangles) instead of just 4 times (for two triangles).

Problem 4: (10 pts) In class we saw how we could obtain the mirror image of a vertex easily if the mirror were on the xz plane at $y = 0$. For such a mirror, the mirror image of a vertex at (x, y, z) would be at $(x, -y, z)$.

Appearing below are a vertex shader and geometry shader that do nothing special: The vertex shader computes the clip-space coordinates and lighting, and the geometry shader emits one triangle.

Modify the shader(s) so that for each original triangle two are emitted: one in the original position, and the other in the reflected position. (Don't worry about stenciling or anything like that.)

Hint: Don't forget that the mirror is in object-space (world-space) coordinates.

☑ Modify to emit reflected triangle, in addition to regular.

☑ Think about coordinate spaces.

Solution shown below. The modified lines have been identified with `SOLUTION` to the right, or above and below. In the vertex shader first the reflected coordinate is computed in object space (just by flipping the sign of the y component). That is converted to clip space for use by the geometry shader. Note that flipping the sign of the y component of the clip-space coordinate would not work.

In the interface block (`in Data` and `out Data`) a new declaration was added for the reflected coordinate. In the geometry shader a second triangle is emitted.

```
void vs_main_basic() {
  gl_Position = gl_ModelViewProjectionMatrix * gl_Vertex;

  // Compute eye-space vertex coordinate and normal.
  // These are outputs of the vertex shader and inputs to the frag shader.
  vec4 var_vertex_e = gl_ModelViewMatrix * gl_Vertex;
  vec3 var_normal_e = normalize(gl_NormalMatrix * gl_Normal);
  vec4 lcolor = generic_lighting( var_vertex_e, gl_Color, var_normal_e);

  // SOLUTION below                                                    SOLUTION
  vec4 mir_vertex_o = vec4(gl_Vertex.x, -gl_Vertex.y, gl_Vertex.z, gl_Vertex.w);
  mir_vertex_c = gl_ModelViewProjectionMatrix * mir_vertex_o;
  // SOLUTION above                                                    SOLUTION

  gl_BackColor = gl_FrontColor = lcolor;
}

out Data  {
  int hidx;  // Not used.
  vec4 mir_vertex_c;                                          /// SOLUTION
};
#endif

#ifdef _GEOMETRY_SHADER_
in Data {
  int hidx;  // Not used.
  vec4 mir_vertex_c;                                          /// SOLUTION
} In[3];

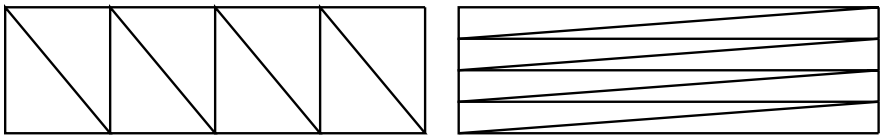
void gs_main_helix() {
  /// Geometry Shader
  //
  // Pre-defined input: gl_PositionIn[] (an array, size determined by prim)
  // Pre-defined output: gl_Position (a scalar, read when EmitVertex called).
```

```
  for ( int j=0; j<2; j++ )                                          /// SOLUTION
  for ( int i=0; i<3; i++ )
    {
      gl_FrontColor = gl_FrontColorIn[i];
      gl_BackColor = gl_BackColorIn[i];
      gl_Position = j == 0 ? gl_PositionIn[i] : mir_vertex_c[i];  /// SOLUTION

      EmitVertex();
    }
  EndPrimitive();
  }                                                                  /// SOLUTION
}
```

Problem 5: (15 pts) Answer each question below.

(*a*) Suppose we have a buffer object for vertices corresponding to the object on the lower left.



☑ How can we use that buffer object to render an object that will look like the one on the right?

Notice that the object on the right can be obtained from the object on the left by rotating and scaling it. So before rendering set the modelview transformation matrix to the following: $M\,T_{\vec{c}}\,R_{\hat{n},90^\circ}\,S_{0.25,4,1}\,T_{-\vec{c}}$, where $T_{-\vec{c}}$ is a translation that moves the center of the object to the origin, $S_{0.25,4,1}$ is a scale transformation that compresses the x axis by 4 and stretches the y axis by 4, and $R_{\hat{n},90^\circ}$ is a rotation matrix that rotates by 90°, and M is the existing modelview transformation.

(*b*) Describe a situation in which using `glBegin` with `glVertex` rather than vertex arrays with (`glDraw`) is a good idea.

☑ Situation in which `glVertex` better:

This would make sense when, for some reason, the number of primitives to render is small, for example, just one triangle. In that case, there is no benefit to `glDraw` and in fact the overhead of setting up the arrays would make it slower.

(*c*) Describe a situation in which using vertex arrays but without buffer objects is a good idea.

☑ Situation in which vertex arrays without buffer objects better:

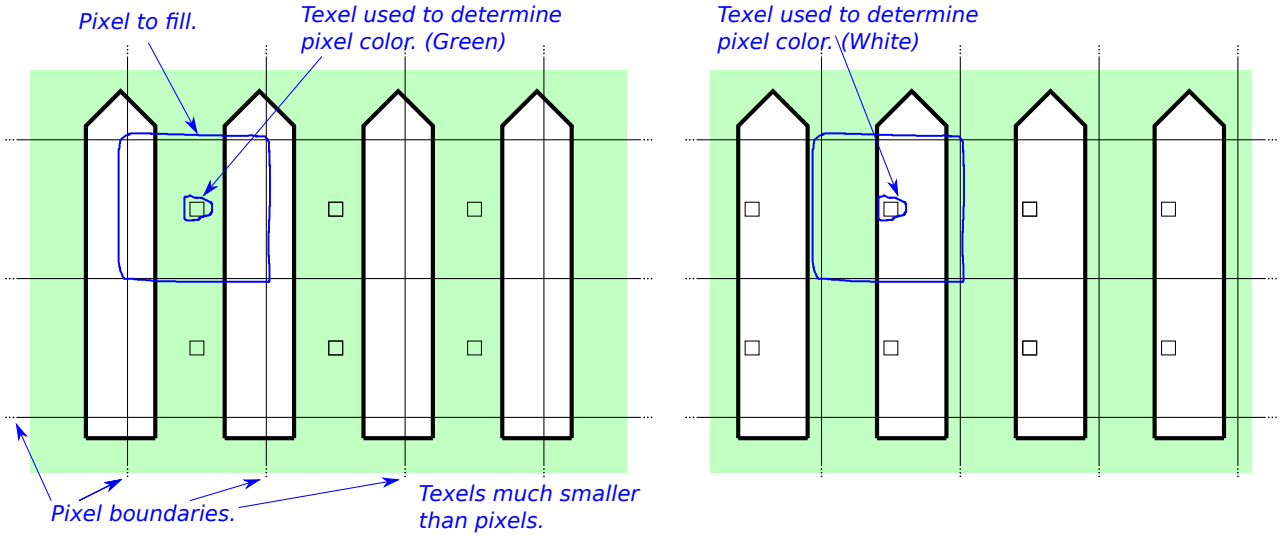
Buffer objects have an advantage over vertex arrays when the same buffer object is used multiple times. In that case the data should be sent to the GPU just once, whereas with vertex arrays the data is sent each time. So vertex arrays without buffer objects are good idea if the vertices would be used just once. This might happen because the set of triangles we are trying to render changes position every frame.

Problem 6: (10 pts) In class the topic of texture filtering was introduced with a description of what would happen if a texture image of a picket fence (white stripes in front of a green background) where rendered without an appropriate MIPMAP level and without filtering. Either the fence or the grass would disappear.

(*a*) Illustrate the problem. On your diagram show the fence, some pixels and some texels. It is important to choose the size of the pixels and texels correctly.

☑ Illustration of problem. ☑ Show pixels. ☑ Show texels.

The illustration below shows the picket fence and pixel boundaries and some texel boundaries, the texels are much smaller than the pixels. The color of a pixel is determined by the color of the texel in the center of the pixel (those are the only texels illustrated). For the left-hand figure the pixels would be green (and so the fence would not be visible). For the right hand figure the pixels would be white (and so the grass would not be visible). The problem is that while each pixel covers both fence and grass, the sampled texel covers *either* fence or grass. Note: This diagram illustrates the OpenGL `NEAREST` texture filtering method when used with just one MIPMAP level.

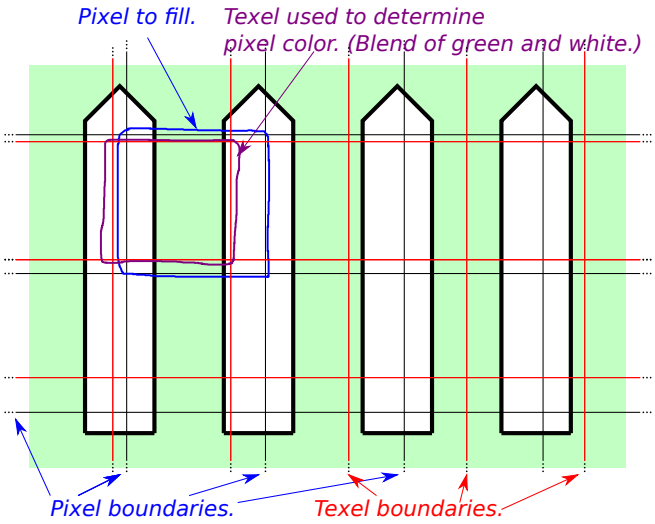


(*b*) Re-draw the diagram showing how the choice of an appropriate MIPMAP level can avoid the problem of the fence disappearing.

☑ Illustration of how MIPMAP level avoids problem.

Re-drawn illustration shown below. In this case a higher MIPMAP level was used, meaning that the texels were larger. (A larger texel is obtained by blending the colors in the smaller texels of a lower MIPMAP level.) In this case the texels are only a little smaller than the pixels. Still, one texel is selected for each pixel. In the illustration the selected texels will be a blend of green and white. Because of the pixel size it will not be possible to discern the pickets, but the color won't suddenly change from white to green as the image moves.

Note: This diagram illustrates the OpenGL `NEAREST` texture filtering method when used with a complete set of MIPMAP levels.



(*c*) Draw a diagram showing an example of how linear filtering can improve the appearance of the fence.

☑ Example of how linear filtering improves appearance.

Refer to the illustration from the previous part. The texel under the word `Blend` will not be used for any pixel because the center of the texels to the left and right are closer to the center of the respective pixels. As a result the image might appear greener than it should. With linear filtering an average of several pixels is used, this avoids the problem of skipping certain texels.

30 Fall 2012 Solutions

Name Solution

EE 4702

Take-Home Pre-Final Questions

Tuesday, 4 December 2012

These are practice questions for the final exam.

Problem 1		(20 pts)
Problem 2		(20 pts)
Problem 3		(20 pts)
Problem 4		(20 pts)
Problem 5		(20 pts)
Exam Total		(100 pts)

Alias 0x501f

Good Luck!

Problem 1: [20 pts]The advantage of an OpenGL triangle strip over individual triangles is that vertices shared by multiple triangles only need to be specified once.

(*a*) Why is the code below not a good example?

```
for ( int i=0; i<1000; i++ )
  {
    glBegin(GL_TRIANGLE_STRIP);
    glNormal3fv(normal[i]);  glVertex3fv(v0[i]);
    glNormal3fv(normal[i]);  glVertex3fv(v1[i]);
    glNormal3fv(normal[i]);  glVertex3fv(v2[i]);
    glEnd();
  }
```

The advantage of a triangle strip is that the work performed by the vertex shader on one vertex can be used for up to three triangles (the triangles that share the vertex). Since the code above only renders one triangle the benefit is not realized. That is, the vertex shader runs three times, once for each vertex. That's the same number of times as if individual triangles were used.

(*b*) Explain the impact of the proper use of triangle strips on each of the items below. For one of the items below the answer should consider two cases, when a buffer object is being used and when it is not being used. Remember, for this part assume that triangle strips are being used properly.

- Total communication bandwidth.

Case 1: Without using a buffer object for vertex data (vertices, normals, etc.). A triangle strip reduces CPU/GPU communication by a factor of 3 for each rendering pass. We can assume there are many rendering passes so the factor of 3 should have a bit impact.

Case 2: Using a buffer object for vertex data. Since a buffer object is used the vertex data will be sent from the CPU to the GPU only once. The communication bandwidth will be reduced by a factor of 3 when it is sent but the overall impact should be much smaller since we are sending the vertices just once rather than once for each rendering pass.

- Total vertex shader execution time.

Reduced by a factor of 3.

- Total fragment shader execution time.

There is no difference between triangle strips and individual triangles in stages after the vertex shader, which includes the fragment shader. So the impact on the fragment shader is none.

(*c*) A triangle strip is well suited for situations in which triangles are approximating a smooth surface (such as the surface of a sphere). It is less well suited for surfaces that really are composed of triangles. Explain why. *Hint: Consider colors and normals.*

Consider a situation in which one would like to render triangles that form a strip but are not on the same plane. In this case the geometric normal for each triangle is different, and we would like to use this normal. However, a normal is specified for a vertex, so all triangles sharing the vertex must get the same normal.

Problem 2: [20 pts]Assume that the lighted color of a triangle depends on conditions at its centroid (a point equidistant from its three vertices), including light and viewer location. Though computed for the center, this same lighted color would be used for all fragments.

(*a*) Which would be the most appropriate shader to compute this color: vertex, geometry, or fragment? Explain.

Geometry. The shader is called once per triangle and has access to all vertices. It will be able to compute the centroid, compute the lighted color, and use that color for each vertex.

(*b*) Regardless of your answer above, explain how a vertex shader could be used to compute the centroid (a first step in computing the color). This will require some extra help from the CPU, but the burden of computation should be on the shader. (*Note: the challenge here is getting the vertex locations.*)

For the vertex shader to compute the centroid it needs to know the other two vertices. The CPU would have to send along that information, by defining additional attributes for the vertex. (Or by borrowing unused attributes such as texture coordinates for texture units that are not in use.) The CPU could also compute the centroid and send that information, but that would not be shifting the burden of computation to the shader.

(*c*) Shown below are three possible declarations for the output of the shader that computes the triangle's lighted color. Each of these will produce the desired result. But one is clearly best, and one is clearly worst. Identify the best and worst to use and explain why for each.

```
flat out vec3 tri_color;
smooth out vec3 tri_color;
noperspective out vec3 tri_color;
```

Since the color is the same over all fragments the lighted color computed by the vertex shader should be delivered unchanged to each fragment shader invocation. That is what the `flat` declaration does. The `noperspective` performs a linear interpolation, and `smooth` performs a perspective-correct interpolation. The `noperspective` is by far the most computationally expensive so it is the worst to use.

Problem 3: [20 pts]Suppose an NVIDIA GPU of compute capability 2.0 has 8 multiprocessors. Each multiprocessor has 32 cores, but a single warp can use only 16 cores (it takes two different warps to keep all 32 cores in a multiprocessor busy).

The GPU is to be used to add two arrays element-wise. Suppose that the number of array elements is 2^{24}.

Let t_1 denote the amount of time it takes one thread (yes, just one) to perform the entire calculation on the GPU. The kernel code is shown below:

```
__device__ void prob(int array_size) {
  int tid = threadIdx.x + blockIdx.x * blockDim.x;
  for ( int i=tid; i<array_size; i += num_threads )
    result[i] = a[i] + b[i];
}
```

(*a*) Compute the amount of time it would take to perform the computation for each of the following launch configurations, and explain why they are the same or different (whichever the case applies):

☑ One block of 16 threads.

Execution time is $t_1/16$.

☑ Two blocks of 8 threads each.

Execution time is $t_1/16$.

☑ Explain why same or different.

The total number of threads is 16 in both cases. The one-block configuration would run on one multiprocessor and would be able to use 16 cores, the two-block configuration would run on two multiprocessors and would be able to make use of 8 cores each, for a total of 16.

Since there are enough cores for all threads none of the threads will have to wait for a core (that is occupied by another thread), and so the rate of computation for each thread will be the same as the rate of computation in the one-thread case. Since work is divided evenly between threads, the execution time in both cases will be $t_1/16$.

(*b*) Estimate (an exact answer isn't possible) the amount of time it would take to perform the computation for each of the following launch configurations and explain why they are the same or different (whichever the case applies):

☑ One block of 1024 threads.

Lets assume that 256 threads are enough to keep all of the cores in a multiprocessor busy. Then there would be no difference in performance in configurations from 256 to 1024 threads per block. Then the time is $t_1/256$.

☑ Two blocks of 512 threads each.

Again lets assume that 256 threads are sufficient to keep all cores in a multiprocessor busy. However, since there are two blocks two multiprocessors will be used and so the time will be based on 512 threads, so the execution time is $t_1/512$.

☑ Explain why same or different.

The times are different because with just one block the threads all must run on one multiprocessor, with two blocks they can run on two and so have access to the same number of CUDA cores.

(*c*) Explain how higher memory latency would increase the number of threads needed for good performance.

Memory latency is the amount of time needed to retrieve data from global memory (assuming a cache miss). A thread that need data loaded from global memory must wait (and do nothing) until it arrives. While one thread waits another can execute, so having more threads means that more threads can fill this role (executing while another thread is waiting). The longer the latency the more threads that are needed.

Problem 4: [20 pts]Consider the CUDA code below in which two alternatives are shown: putting a balls array in shared memory or in global memory. Assume the code runs on GPUs of compute capability 2.0 (though the answers could apply to 1.0 through 2.5).

(*a*) Explain why the Option 1 (shared memory) code below will run slowly. Fix the problem making as few changes as possible.

```
struct Ball {
  float3 position;   float3 velocity;
  float radius;      float mass;      };

__shared__ Ball balls[256]; // Option 1 - Shared memory.

__device__ void
update(float deltat)
{
  int start = threadIdx.x * balls_per_thread;
  int stop = start + balls_per_thread;
  for ( int i=start; i<stop; i++ )
    balls[i].position += deltat * balls[i].velocity;
}
```

The code suffers from shared memory bank conflicts.

(*b*) Explain why the Option 2 code below (using global memory) will run slowly. Fix the problems making as few changes as possible. (There are two major things to be fixed.)

```
struct Ball {
  float3 position;   float3 velocity;
  float radius;      float mass;      };

Ball* balls; // Option 2 - Global memory.

__device__ void
update(float deltat)
{
  int start = ( threadIdx.x + blockIdx.x * blockDim.x ) * balls_per_thread;
  int stop = start + balls_per_thread;
  for ( int i=start; i<stop; i++ )
    balls[i].position += deltat * balls[i].velocity;
}
```

Memory access is most efficient when consecutive threads access consecutive elements and when the size of the values accessed are a power of 2. In the code above consecutive threads will access memory addresses that differ by a factor of `balls_per_thread * sizeof(Ball)`, wasting memory requests. Two things are done to fix that. First, the array index will be computed so that it is equal to `threadIdx.x + something`, where `something` is the same for all threads in a block.

Second, the data will be reorganized from an array of structures, to just two arrays, `b_pos` and `b_vel`. (The code for initializing these arrays is not shown). This way an access to `b_pos[0]`, `b_pos[1]`, . . ., `b_pos[31]` can be satisfied by fetching a contiguous block of memory.

Another change was using `float4` rather than `float3` for position and velocity. This ensures that access to the vectors is done by one instruction rather than three.

```
// SOLUTION
float4* b_pos, b_vel;

__device__ void update(float deltat)
{
  const int num_threads = blockDim.x * gridDim.x;
  const int start = threadIdx.x + blockIdx.x * blockDim.x;
  for ( int i=0; i<balls_per_thread; i++ )  {
      const int idx = start + i * num_threads;
      b_pos[idx] += deltat * b_vel[idx];
    }
}
```

Problem 5: [20 pts]Answer each question below.

(*a*) Why is the if/return needed in the CUDA code below?

```
__device__ void prob(int array_size) {
  int tid = threadIdx.x + blockIdx.x * blockDim.x;
  if ( tid >= array_size ) return;
  result[tid] = a[tid] + b[tid];
}
```

The number of threads must be a multiple of the block size. We can assume that `array_size` can be any positive integer.

(*b*) The line of code below checks for a special case to avoid a square root. That might make sense for a CPU, but not for necessarily for CUDA code on a GPU. Describe a situation in which it makes sense for CUDA and a different situation when it makes no sense (meaning it would be faster to do the square root all the time). Assume that 50% of the time d is equal to 1.

```
  if ( d == 1 ) s = 1; else s = sqrt(d);
```

It makes sense if there is no branch divergence, meaning that for all warps the branch is either always taken or always not taken.

If there is branch divergence then the code will run more slowly than code always performing the square root since within a warp the execution time will be the sum of both paths through the if.

(*c*) The loop below is innocent on a CPU, but on a GPU it can execute inefficiently. Identify the problem and fix it.

```
for ( int i=0; i<32; i++ )
  {
    if ( a[i] == t ) { do_stuff(i); break; }
  }
```

Unless the compiler is really clever, threads within a warp computing `do_stuff` will not only compute it in parallel with other threads when the value of `i` matches. The solution is to move `do_stuff` outside of the loop. See the code below.

```
int dsi = -1;
for ( int i=0; i<32; i++ )
  {
    if ( a[i] == t ) { dsi = i; break; }
  }
if ( dsi >= 0 ) do_stuff(dsi);
```

Name Solution

EE 4702

Final Exam

Friday, 7 December 2012, 12:30-14:30 CST

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

Alias 0x501f

Good Luck!

Problem 1: [15 pts]The two code fragments below render the same set of triangles.

```
// Code Fragment A

 glBegin(GL_TRIANGLES);
 for ( int i=0; i<1000; i++ )
   {
     glColor3fv(lsu_spirit_gold); glVertex3fv(v0[i]);
     glColor3fv(lsu_spirit_gold); glVertex3fv(v1[i]);
     glColor3fv(lsu_spirit_gold); glVertex3fv(v2[i]);
   }
 glEnd();

// Code Fragment B
 glColor3fv(lsu_spirit_gold);
 glBegin(GL_TRIANGLE_STRIP);
 for ( int i=0; i< XXX; i++ )
     glVertex3fv(v[i]);
 glEnd();
```

(*a*) Compute the amount of data sent from CPU to GPU for each case. Use an appropriate value of XXX.

For Fragment A:

$$1000 \times 3 \times 2 \times 3 \times 4 = 72000\,\mathrm{B}$$

where the factors are: 1000 triangles, 3 vertices per triangle, two attributes (color and vertex coordinate), three components per attribute, four bytes per component.

For Fragment B, the color is sent once (not per vertex), the size is $3 \times 4 = 12$ bytes. If the same number of triangles are rendered then 1002 vertices need to be sent. The data size is $12 + 1002 \times 3 \times 4 = 12036$ bytes.

Problem 2: [15 pts]Consider the declarations below which come from the vertex shader in the Homework 3 solution.

```
layout ( location = 2 ) uniform float wire_radius;
layout ( location = 3 ) uniform float theta;
layout ( location = 4 ) uniform vec2 texture_scale;
layout ( binding = 1 ) buffer Helix_Coord  { vec4  helix_coord[];  };
layout ( location = 1 ) in ivec2 helix_index;
out vec3 normal_e;
uniform sampler2D tex_unit_0;
```

(*a*) What do the location numbers indicate in the uniform declarations?

They are numbers by which the CPU identifies the respective uniform. The CPU uses these numbers when writing a value to a uniform.

Shown below is CPU code to write a value into the `wire_radius` uniform declared above (in shader code). It is the "2" which tells OpenGL which uniform to deposit the value in. It has nothing to do with the names of the variables, `wire_radius` on the GPU and wireRadius on the CPU.

```
// CPU Code
glUniform1f( 2, wireRadius);
```

(*b*) How is a `uniform` typically used?

It set once before a rendering pass and does not change. One can assume that changing a uniform value is relatively costly.

(*c*) Explain what the `in` and `out` qualifiers used for `helix_index` and `normal_e` indicate.

The `in` indicates that the variable is an input to the vertex shader. Its values are provided by the CPU using the `glVertexAttribX` command. Like colors, normals, and other attributes, a different value can be provided for each vertex. The `out` indicates that the variable is an output from the vertex shader, the vertex shader code must write the variable.

(*d*) Why can't the declarations above be for a geometry shader? (That is, something won't work.)

Because the inputs to a geometry shader must be declared as arrays.

(*e*) Why can't the declarations above be for a fragment shader? (That is, something won't work.)

Because fragment shaders don't have user-declared outputs.

Problem 3: [20 pts]The Homework 5 code simulated a spring, bringing to life the helix we were working with most of the semester. A combination of CPU and GPU code was used to compute the wire surface coordinates. Excerpts of the code appear below.

Grading Note: This problem was based on code used in Homework 5, with which some familiarity was expected.

```
Physics Simulation in CUDA computes helix_position and orientation, read to CPU.
              Wire_Segment* const ws = &wire_segments[i];
              ws->position = helix_position[i];
              ws->orientation = helix_orientation[i];

Computation on CPU:
          helix_coords[i] = ws->position;
          pMatrix_Rotation c_rot(ws->orientation); // Mult: 3 * 9 = 27
          helix_u[i] = c_rot * pVect(0,0,1);
          helix_v[i] = c_rot * pVect(0,1,0);

Computation on Vertex Shader
  vec3 vtx = helix_coord[hidx.x].xyz;
  float pi = 3.14159265;
  float theta = hidx.y * 2 * pi / 20;
  vec3 u = helix_u[hidx.x].xyz;
  vec3 v = helix_v[hidx.x].xyz;
  vec3 normal = normalize( cos(theta) * u + sin(theta) * v );
  // Compute wire surface location by adding normal to helix coordinate.
  //
  vec4 vertex_o;
  vertex_o.xyz = helix_coord[hidx.x].xyz + wire_radius * normal;
  vertex_o.w = 1;
```

Computing the u and v vectors on the CPU requires about 18 multiplications. Consider the method used in the code above and two alternatives: computing these in CUDA or computing them in the vertex shader.

The values of `hidx.x` (position [index of wire segment] along the helix) vary from 0 to $S-1$ and the values of `hidx.y` (position around the cylinder of a wire segment) vary from 0 to $M-1$.

(*a*) Compute the amount of data moving between CPU and GPU (in both directions) for the code shown.

For the solution consider the arrays `helix_position`, `helix_orientation`, `helix_u`, `helix_v`, and `hidx`. Note that `helix_coord` has the same data as `helix_position`.

Let S denote the number of helix segments, this is the size of arrays `helix_position`, `helix_orientation`, `helix_u`, `helix_v`, and `helix_position`. Let W denote the number of points on the wire surface (which surrounds the helix), W will be some multiple of S. Array `hidx` has W elements. The CUDA code is launched with a total of S threads (each computing one helix segment), the vertex shader is run on a rendering pass of W vertices.

Array `helix_orientation` holds quaternions, which are four-element vectors of floats. The size of an element of `helix_orientation` is $4 \times 4 = 16$ bytes. Each element of `hidx` is a two-element integer vector, the total size is 8 bytes. The other arrays hold 3-element FP vectors, of size 12 bytes.

The data moving from GPU to CPU are the arrays `helix_position` and `helix_orientation`, the size is $S(12+16) = 28S$ bytes.

The CPU uses `helix_orientation` to compute the u and v vectors, which it sends to the GPU. The CPU also prepares and sends the `hidx` array. The position array (renamed `helix_coord`) is also sent. Notice that the orientation is not re-sent to the GPU. The total size is $(12 + 12 + 12)S + 8W = 36S + 8W$.

(*b*) Compute the amount of data that would be moved if u and v were computed in CUDA.

Assume that if u and v were computed in CUDA there would be no need to send the orientation to the CPU. In that case the data sent from the CPU to the GPU would be `helix_position` and the vectors: $(12 + 12 + 12)S$ bytes, and increase of $8S$ bytes. Note that the GPU would be able to compute these quantities faster than the CPU, possibly offsetting the increased communication time.

(c) Compute the amount of data that would be moved if u and v were computed in the vertex shader.

If u and v were computed in the vertex shader we would no longer need to send them from the CPU, but we would have to send the orientation. The data from CPU to GPU would be $(12 + 16)S + 8W = 28S + 8W$, a reduction of $8S$ bytes.

(d) Which option computes the same thing multiple times? Is there any benefit to this redundancy?

Because the vertex shader is run W times each value of u and v will be computed W/S times, rather than once in the other variations. There will be a benefit if execution time were limited by the time needed to send data.

Problem 4: [15 pts]A goal of Homework 2 was to use `glDrawElements` in place of `glDrawArray`. For our task of rendering a helix, `glDrawElements` reduces the number of vertices that need to be sent from CPU to GPU by a factor of 2.

(*a*) Explain why half the number of vertices are needed.

The helix constructed in Homework 2 consists of multiple triangle strips. There are vertices in one triangle strips that are in the same position as vertices in an adjacent triangle strip. When using `glDrawArrays` such vertices need to be sent twice, one for each strip. The command `glDrawElements` avoids that by using vertex indices (position in an array). (See answer to next part.) This way a vertex can be specified any number of times. For the helix this reduces by half (over a triangle strip) the number of vertices sent.

(*b*) When using `glDrawElements` what needs to be sent from CPU to GPU that was not needed with `glDrawArray`? How large is it in comparison to the vertex arrays?

With `glDrawElements` one must send an array of vertices and an array of indices. It is the array of indices which is not needed using the simpler `glDrawArray` command.

A vertex consists of a coordinate and its attributes (such as the normal, a color, etc.). A coordinate typically consists of three floats, for a size of 12 bytes, a normal is another 12 bytes. Stopping there we have 24 bytes per vertex. An index is an integer, which might be 4 bytes, one sixth the size of a vertex.

Based on the way we used `glDrawElements` in Homework 2, two indices are used per unique vertex, so the index sizes would be one third the vertex size.

Problem 5: [15 pts]Answer the following questions abut coordinate spaces. For the transformations use $T_{\vec{v}}$ to indicate a translation transformation and $R_{\vec{u}\to\vec{v}}$ indicate a rotation transform that rotates $\vec{u}$ to $\vec{v}$.

(*a*) Let P_o be a coordinate in object space coordinates, let E be the location of the eye and $\vec{v}$ be a normal to the projection surface. Show how to transform P_o into an eye-space coordinate.

In eye space the eye is at the origin and the projection surface normal points in the $-z$ direction. (The projection surface can be thought of as the computer monitor that the eye is looking at [or the plane in which the monitor's screen resides].)

First translate (move) E to the origin, that is done by $T_{\vec{E0}}$, where $\vec{E0}$ is a vector from E to the origin. Next rotate the space so the projection surface normal, use $R_{\vec{v}\to(0,0,-1)}$. The transformation matrix is the product $R_{\vec{v}\to(0,0,-1)}T_{\vec{E0}}$, note that translation is on the right (and so is performed first). The transformed coordinate is $R_{\vec{v}\to(0,0,-1)}T_{\vec{E0}}P_0$.

(*b*) Let P be a point in clip space. How can we determine if P is in the view volume?

In clip space the view volume is a cube centered on the origin, aligned with the axes, and with edges of length 2. The cube vertices are therefore at $(\pm1, \pm1, \pm1)$. Let $P = (x, y, z)$. It is in the view volume if $|x| \leq 1$, $|y| \leq 1$, and $|z| \leq 1$. If P is homogeneous there is no need to homogenize. Let $P = (x, y, z, w)$; it is in the clip space if $|x| \leq w$, $|y| \leq w$, and $|z| \leq w$.

(*c*) Suppose we are rendering to a window of size 1000×1000 pixels. Let $P_c = (0.5, 0.5, 0.5)$ be a coordinate in clip space.

☑ Provide an example of clip space coordinate $P_d \neq P_c$ that is likely to be in the same pixel as P_c.

In clip space x ranges from -1 to +1, and that spans 1000 pixels. So in clip space the distance between adjacent pixels in the x direction is $2/1000 = .002$. A coordinate likely in the same pixel can be obtained by moving less than 0.002: $P_d = (0.501, 0.5, 0.5)$.

☑ Provide an example of clip space coordinate that is in a pixel close to P_c.

Move a distance .002 in clip space: $P_d = (0.502, 0.5, 0.5)$.

Problem 6: [20 pts]Consider a multiprocessor in an NVIDIA CC 2.0 GPU, the type frequently discussed in class. Such a multiprocessor had 32 CUDA cores, two schedulers, and contexts for 48 warps. For all questions below assume a kernel is launched with one block per multiprocessor.

(*a*) Explain why a block size of one warp is guaranteed to leave half the cores unused.

Because one scheduler is dedicated to even-numbered warps and can only issue to the first 16 CUDA cores, the other scheduler is dedicated to odd-numbered warps and can only issue to the last 16 CUDA cores. If a block only has one warp it is even and so can only use half the cores.

(*b*) Explain why a block size of two warps will likely result in cores being idle most of the time.

With two warps it will be possible to issue to all of the cores. However instruction latency is high in NVIDIA GPUs, meaning that many cycles must pass from the time an instruction executes until its result is ready for another instruction. If there are many warps that's not a problem because while one warp is waiting for an instruction result other warps can issue and so keep the cores busy.

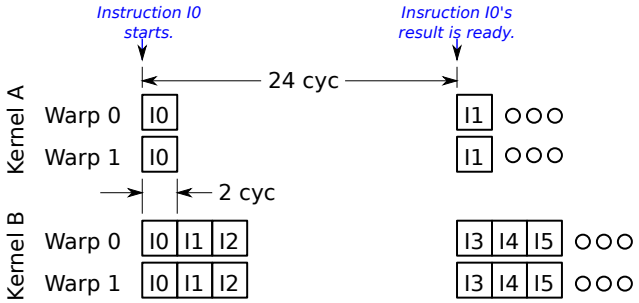
For the next two parts consider Kernel A and Kernel B, below. Notice that in Kernel A I1 must wait for I0 and I2 must wait for I1. In Kernel B I3 must wait for I0, but I1 does not have to wait for I0. Inspect the code for additional dependencies. Suppose that this pattern of dependencies continues in each kernel. That is, in Kernel A each instruction depends on its immediate predecessor, while in Kernel B each depends on the third previous instruction.

```
# Kernel A
I0:  add  r1, r2, r3
I1:  mul  r4, r1, r5
I2:  sub  r6, r4, r7
I3:  or   r8, r6, r9
...

# Kernel B
I0:  add  r1, r2, r3
I1:  add  r10, r12, r13
I2:  add  r20, r22, r23
I3:  mul  r4, r1, r5
I4:  mul  r14, r11, r15
I5:  mul  r24, r21, r25
```

(*c*) Suppose that each kernel is launched with two warps per block (and one block per MP). Assume that the instruction latency is 24 cycles. Draw a diagram showing the time that each instruction executes, until the pattern is obvious.

Diagram appears below.



(*d*) Compute the CUDA core utilization (the fraction of time a core will be used) for each kernel.

For Kernel A during the 24-cycle latency each thread can execute one instruction, for a total of 64 instruction counting all threads in a block. During 24 cycles the hardware can issue $24 \times 32 = 768$ instructions, so the utilization for A is $\frac{64}{768} = \frac{1}{12} = 8.3\%$. In Kernel B during the 24-cycle latency waiting for results from `I0` a thread can execute `I1` and `I2`. Counting all 64 threads, that's 192 instructions. The utilization for B is $\frac{192}{768} = \frac{1}{4} = 25\%$.

31 Fall 2011 Solutions

Name Solution

EE 4702

Take-Home Pre-Final Examination

Tuesday, 29 November 2011 to Friday, 2 December 2011

Work on this exam alone. Regular class resources, such as notes, papers, documentation, and code, can be used to find solutions. Do not try to seek out references that specifically answer any question here. Do not discuss this exam with classmates or anyone else. Any questions or concerns about problems should be directed to Dr. Koppelman.

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

Alias Compatibility Profile Proud

Good Luck!

Problem 1: [20 pts]The advantage of an OpenGL triangle strip over individual triangles is that vertices shared by multiple triangles only need to be specified once.

(*a*) Why is the code below not a good example?

```
for ( int i=0; i<1000; i++ )
  {
    glBegin(GL_TRIANGLE_STRIP);
    glNormal3fv(normal[i]);  glVertex3fv(v0[i]);
    glNormal3fv(normal[i]);  glVertex3fv(v1[i]);
    glNormal3fv(normal[i]);  glVertex3fv(v2[i]);
    glEnd();
  }
```

The example renders just one triangle, and so the benefit of re-using the vertex shader output does not apply.

(*b*) Explain the impact of the proper use of triangle strips on each of the items below. For one of the items below the answer should consider two cases, when a buffer object is being used and when it is not being used. Remember, for this part assume that triangle strips are being used properly.

- Total communication bandwidth.

When no buffer object is used a triangle strip reduces CPU/GPU communication by a factor of 3 for each rendering pass. When a buffer object is used the reduction by a factor of 3 is only realized when the buffer object is first sent. If the object is re-used (which one would assume otherwise why use a buffer object) then there is no change in CPU/GPU communication. The impact on GPU device memory to GPU communication depends upon how code for primitive assembly is written (something which was not speculated in class).

- Total vertex shader execution time.

Reduced by a factor of 3.

- Total fragment shader execution time.

No change.

(*c*) A triangle strip is well suited for situations in which triangles are approximating a smooth surface (such as the surface of a sphere). It is less well suited for surfaces that really are composed of triangles. Explain why. *Hint: Consider colors and normals.*

Unless all the triangles in a strip are in the same plane, the geometric normal for each triangle can be different, even those sharing vertices. The triangle normal is specified for a vertex, so all triangles sharing the vertex will get the same normal.

Problem 2: [20 pts]Assume that the lighted color of a triangle depends on conditions at its centroid (a point equidistant from its three vertices), including light and viewer location. Though computed for the centroid, this same lighted color would be used for all fragments.

(*a*) Which would be the most appropriate shader to compute this color: vertex, geometry, or fragment? Explain.

Geometry. The shader is called once per triangle and has access to all vertices.

(*b*) Regardless of your answer above, explain how a vertex shader could be used to compute the centroid (a first step in computing the color). This will require some extra help from the CPU, but the burden of computation should be on the shader. (*Note: the challenge here is getting the vertex locations.*)

For the vertex shader to compute the centroid it needs to know the other two vertices. The CPU would have to send along that information, by defining additional attributes for the vertex. (Or by borrowing unused attributes such as texture coordinates for texture units that are not in use.) The CPU could also compute the centroid and send that information, but that would not be shifting the burden of computation to the shader.

(*c*) Shown below are three possible declarations for the output of the shader that computes the triangle's lighted color. Each of these will produce the desired result. But one is clearly best, and one is clearly worst. Identify the best and worst to use and explain why for each.

```
flat out vec3 tri_color;
smooth out vec3 tri_color;
noperspective out vec3 tri_color;
```

Since the color is the same over all fragments the lighted color computed by the vertex shader should be delivered unchanged to each fragment shader invocation. That is what the `flat` declaration does. The `noperspective` performs a linear interpolation, and `smooth` performs a perspective-correct interpolation. The `noperspective` is by far the most computationally expensive so it is the worst to use.

Problem 3: [20 pts]Suppose an NVIDIA GPU of compute capability 2.0 has 8 multiprocessors. Each multiprocessor has 32 cores, but a single warp can use only 16 cores (it takes two different warps to keep all 32 cores in a multiprocessor busy).

The GPU is to be used to add two arrays element-wise. Suppose that the number of array elements is 2^{24}.

Let t_1 denote the amount of time it takes one thread (yes, just one) to perform the entire calculation on the GPU. The kernel code is shown below:

```
__device__ void prob(int array_size) {
  int tid = threadIdx.x + blockIdx.x * blockDim.x;
  for ( int i=tid; i<array_size; i += num_threads )
    result[tid] = a[tid] + b[tid];
}
```

(*a*) What would be the time to perform the calculation for a single block of just two threads (in terms of t_1)?

With two threads the computation will finish in half the time, $\frac{t_1}{2}$, since the resources on the system are very far from being saturated with two threads.

(*b*) Consider a launch of eight blocks of 64 threads each. Explain why reducing the number of blocks by one will leave some CUDA cores totally unused.

A block can't be assigned to more than one multiprocessor so with fewer than eight blocks one MP, and its 32 cores, will go unused.

(*c*) Consider a launch of eight blocks of 64 threads each. Explain why reducing the block size by 17 or more will leave some CUDA cores totally unused.

In a CC 2.0 MP 16 cores are used by odd warps, 16 by even warps. At a block size of 64 there are just two warps, one odd and one even. If there are fewer than 16 threads then some cores will never be used.

(*d*) Explain why a launch of 16 blocks of 512 threads would take about the same amount of time to run as a launch of 32 blocks of 256 threads. *Note: Up to 8 blocks or 48 warps can run on a multiprocessor.*

In either configuration there are $\frac{16}{8}\frac{512}{32} = 32$ warps per multiprocessor.

(*e*) Explain how memory latency would increase the number of threads needed for good performance.

Would need to cover latency.

Problem 4: [20 pts]Consider the CUDA code below in which two alternatives are shown: putting a balls array in shared memory or in global memory. Assume the code runs on GPUs of compute capability 2.0 (though the answers could apply to 1.0 through 2.2).

(*a*) Explain why the Option 1 (shared memory) code below will run slowly. Fix the problem making as few changes as possible.

```
struct Ball {
  float3 position;   float3 velocity;
  float radius;      float mass;      };

__shared__ Ball balls[256]; // Option 1 - Shared memory.

__device__ void
update(float deltat)
{
  int start = threadIdx.x * balls_per_thread;
  int stop = start + balls_per_thread;
  for ( int i=start; i<stop; i++ )
    balls[i].position += deltat * balls[i].velocity;
}
```

There are bank conflicts because of the size of the Ball structures, 32 bytes, and the indexing. Changing the size from 32 to 36 will ensure that accesses to consecutive elements (such as `balls[i].position.x` and `balls[i+1].position.x` will be to different bank.

The bank number is bits 2 to 6 of the memory address. Let $a[i]$ denote the address of element i. If the structure size is $32 = 2^5$ then $a[i+4] = a[i] + 4 \times 2^5 = a[i] + 2^7$. Those two addresses must be identical in bit positions 2 to 6, and so there will be a bank conflict, which will slow access. If the structure size were 36 then $a[i+4] = a[i] + 4 \times (2^5 + 2^2) = a[i] + 2^7 + 2^4$ which differ in position 2 to 6 so there is no bank conflict. No two threads in a half-warp can conflict and so the padded structure avoids conflicts.

(*b*) Explain why the Option 2 code below (using global memory) will run slowly. Fix the problems making as few changes as possible. (There are two major things to be fixed.)

```
struct Ball {
  float3 position;   float3 velocity;
  float radius;      float mass;      };

Ball* balls; // Option 2 - Global memory.

__device__ void
update(float deltat)
{
  int start = threadIdx.x * balls_per_thread;
  int stop = start + balls_per_thread;
  for ( int i=start; i<stop; i++ )
    balls[i].position += deltat * balls[i].velocity;
}
```

Cache use is more efficient when memory addresses are accessed consecutively. This requires changing the iteration order and braking the structure into individual arrays. See the solution below.

```
// SOLUTION
__constant__ float3 *balls_position;
__constant__ float3 *balls_velocity;
__device__ void update_sol(float deltat)
{
  int start = threadIdx.x;
  int stop = blockDim.x * balls_per_thread;
  for ( int i=start; i<stop; i+= blockDim.x )
    balls_position[i] += deltat * balls_velocity[i];
}
```

Problem 5: [20 pts]Answer each question below.

(*a*) Why is the if/return needed in the CUDA code below?

```
__device__ void prob(int array_size) {
  int tid = threadIdx.x + blockIdx.x * blockDim.x;
  if ( tid >= array_size ) return;
  result[tid] = a[tid] + b[tid];
}
```

Number of threads must be a multiple of the block size, but the array size does not have to be. Therefore we need to check if a `tid` is out of range.

(*b*) The line of code below checks for a special case to avoid a square root. That might make sense for a CPU, but not for necessarily for CUDA code on a GPU. Describe a situation in which it makes sense for CUDA and a different situation when it makes no sense (meaning it would be faster to do the square root all the time). Assume that 50% of the time d is equal to 1.

```
  if ( d == 1 ) s = 1; else s = sqrt(d);
```

It makes sense if there is no branch divergence, meaning that for all warps the branch is either always taken or always not taken.

If there is branch divergence then the code will run more slowly than code always performing the square root since within a warp the execution time will be the sum of both paths through the if.

(*c*) The loop below is innocent on a CPU, but on a GPU it can execute inefficiently. Identify the problem and fix it.

```
for ( int i=0; i<32; i++ )
  {
    if ( a[i] == t ) { do_stuff(i); break; }
  }
```

Unless the compiler is really clever, threads within a warp computing `do_stuff` will not only compute it in parallel with other threads when the value of `i` matches. The solution is to move `do_stuff` outside of the loop.

32 Fall 2009 Solutions

Name Solution

GPU Programming
EE 4702-1
Midterm Examination

Friday, 6 November 2009, 9:40–10:30 CST

Problem 1		(40 pts)
Problem 2		(20 pts)
Problem 3		(20 pts)
Problem 4		(20 pts)
Exam Total		(100 pts)

Alias Transformed

Good Luck!

Problem 1: [40 pts] Write OpenGL code to render a filled circle (a disc) of radius `r`, centered at the origin and with normal $(0, 1, 0)$.

- The distance between all vertices in a primitive should be approximately `vert_dist`.
- Use triangle strips. Multiple strips are okay, but there should be no easy way to make the strips longer.
- Use `glVertex3f` calls, don't try to construct arrays.
- Don't specify colors, normals, or other attributes.
- Assume transformations, lighting, etc, have all been set up. Start with `glBegin`.

☑ Code rendering a circle of radius `r`, center at origin.

☑ Vertex distance about `vert_dist`.

☑ Good use of strips.

☑ Code reasonably efficient.

Solution appears below. The code renders a series of concentric rings, each ring is rendered using a triangle strip. The innermost ring would probably be better rendered using a triangle fan, but the problem says to use a triangle strip.

```
// SOLUTION
for ( float r_outer = r;  r_outer > 0; r_outer -= vert_dist )
  {
    const float delta_theta = vert_dist / r_outer;
    const float r_inner = max(0.0,r_outer-vert_dist);

    glBegin(GL_TRIANGLE_STRIP);
    glNormalf(0,1,0);
    for ( float theta = 0; theta <= two_pi; theta += delta_theta )
      {
        glVertex3f( r_outer * cosf(theta), 0, r_outer * sinf(theta) );
        glVertex3f( r_inner * cosf(theta), 0, r_inner * sinf(theta) );
      }
    glEnd();
  }
```

Problem 2: [20 pts] Answer the normal questions below.

(*a*) Show an expression for the normal to triangle ABC, where A, B, and C are the vertex coordinates.

☑ Normal to ABC

The triangle defines a plane, and the normal to a plane is just the cross product of any two non-parallel vectors in the plane. Two convenient vectors are $\vec{AB}$ and $\vec{AC}$. So the normal is $\vec{AB} \times \vec{AC}$.

(*b*) In the code sample below the pair of triangles is rendered using two different methods, identified as Method 1 and Method 2.

Describe the difference in appearance of the triangles rendered using Method 1 and Method 2 when diffuse lighting is used and `norm_ABC != norm_CBD`.

```
pVect norm_ABC = find_normal(A,B,C);    pVect norm_CBD = find_normal(C,B,D);

// Method 1
glBegin(GL_TRIANGLES);
glNormalfv(norm_ABC);
glVertex3fv(A);  glVertex3fv(B);  glVertex3fv(C);

glNormalfv(norm_CBD);
glVertex3fv(C);  glVertex3fv(B);  glVertex3fv(D);
glEnd();

// Method 2
pNorm norm_X = norm_ABC + normCBD; // Sum of two vectors normalized.
glBegin(GL_TRIANGLES);
glNormalfv(norm_ABC);  glVertex3fv(A);
glNormalfv(norm_X);    glVertex3fv(B);  glVertex3fv(C);

glVertex3fv(C);        glVertex3fv(B);
glNormalfv(norm_CBD);  glVertex3fv(D);
glEnd();
```

☑ Difference in appearance between Method 1 triangles and Method 2 triangles.

Since a diffuse light model is used the lighted color of a vertex is a function of both its distance from the light source and its normal.

In Method 1, two different normals are used for vertex B, one normal when it's part of triangle ABC and a different normal when it's part of CBD. As a result there are two different lighted colors for B and so there could be a sudden change in shading between points on ABC and CBD that are near B (the same is true for C). The appearance will be two triangles touching along an edge.

In Method 2, the same normal is used for vertex B in both triangles and so the lighted color of vertex B will be the same for both triangles. As a result points close to B on the two triangles will be nearly the same color. The same is true for points near edge BC on the two triangles, they will be nearly the same color. However, vertex A and D will have different lighted colors. The appearance might be of a curved quadrilateral, something like a 2D diamond put on a curved surface.

Problem 3: [20 pts] Consider the three methods of specifying vertices shown below.

```
switch ( opt_method ) {

case VM_Individual: {   /// Use Individual Vertices
    glBegin(GL_TRIANGLE_STRIP);
    for ( int i=0; i<coords_size; i+=3 ) {
        glNormal3f(coords[i],coords[i+1],coords[i+2]);
        glVertex3f(coords[i],coords[i+1],coords[i+2]);  }
    glEnd();
    break;  }

case VM_Array: {        /// Use Vertex Arrays
    glNormalPointer(GL_FLOAT,0,coords);
    glEnableClientState(GL_NORMAL_ARRAY);
    glVertexPointer(3,GL_FLOAT,3*sizeof(float),coords);
    glEnableClientState(GL_VERTEX_ARRAY);
    glDrawArrays(GL_TRIANGLE_STRIP,0,coords_size/3);
    glDisableClientState(GL_NORMAL_ARRAY);
    glDisableClientState(GL_VERTEX_ARRAY);
    break;  }

case VM_Buffer: {       /// Use Buffer Objects
    glBindBuffer(GL_ARRAY_BUFFER,gpu_buffer);
    glVertexPointer(3,GL_FLOAT,3*sizeof(float),NULL);
    glEnableClientState(GL_VERTEX_ARRAY);
    glNormalPointer(GL_FLOAT,0,NULL);
    glEnableClientState(GL_NORMAL_ARRAY);
    glDrawArrays(GL_TRIANGLE_STRIP,0,coords_size/3);
    glBindBuffer(GL_ARRAY_BUFFER,0);
    glDisableClientState(GL_NORMAL_ARRAY);
    glDisableClientState(GL_VERTEX_ARRAY);
  break;  } }
```

(*a*) Why is the individual vertex method slower than the others?

☑ Reason that individual vertex method slower than the others.

Each time `glVertex3f` is called the GL driver code must package the coordinates and send them to the GPU. The time to transmit the data is dwarfed by the overhead of signaling the GPU that data is available, plus function call overhead on the CPU side and similar overheads on the GPU side.

In the `VM_Array` code, which uses vertex arrays, data transmission time is much longer (assuming the number of vertices is larger) but the amount of overhead for the `glDrawArrays`, which is called once, is roughly the same as the overhead for `glVertex3f` in the `VM_Individual` code, and so the overhead per vertex is much smaller.

(*b*) When used the right way the method using buffer objects is much faster than the others.

☑ Why are buffer objects faster than vertex arrays, when used the right way?

When using vertex arrays the entire vertex array is sent from CPU to GPU each time, even if it doesn't change. If vertices don't change, they can be placed in a buffer object and sent to the GPU once, as was presumably done for the `VM_Buffer` case. Then the time for `glDrawArrays` will not include data transfer time, and so will be much faster.

☑ Describe a situation in which the buffer object and vertex array method would have about the same performance.

If the vertices are different each time then the buffer object data would be sent to the GPU and could just be used once, there would be no advantage over using a vertex array. The performance would be similar.

Performance would also be similar if the number of vertices were small, because then the transmission time saved would be insignificant.

Problem 4: [20 pts] Answer each question below.

(*a*) OpenGL allows different material property colors for ambient, diffuse, emissive, and specular lighting. However only a few of these can be changed from vertex to vertex. Why?

☑ Why can't all material properties be changed each vertex?

Data associated with a vertex are called attributes. The OpenGL implementation must prepare storage for a full set of attributes for each vertex it is processing. This can be cumbersome if the number of attributes is too large and so there is a limit. If each vertex could have a full set of four colors for both front and back faces, that would consume $4 \times 2 \times 4 = 32$ floating-point numbers worth of attribute storage, too much. (This argument applies to fixed functionality, shaders can use their attributes however they like. The number of attributes is still limited though.)

(*b*) OpenGL lets you specify any transformation matrix for the projection, it doesn't have to be a frustum.

☑ Describe the appearance of a scene in which the projection matrix were identity. What parts of world space would be visible?

After the projection matrix is applied coordinates are in clip space. In clip space the view volume (the visible part of the scene) is fixed to a cube centered on the origin with $x \in [-1, 1]$, $y \in [-1, 1]$, and $z \in [0, 1]$. Anything outside of this box won't be visible. So the visible parts of the scene are those objects with ± 1 of the user's eye. There will be no perspective foreshortening, so equal-sized objects at distance 0 from the eye will be the same size as those at distance 1.

(*c*) Textures are provided or used to generate multiple MIPMAP levels. Explain what a MIPMAP level is and why it is necessary.

☑ What is a MIPMAP level?

It is a resolution level. Level 0 is the original image. Level 1 is an image with half the number of pixels along each axis (total number of pixels is $\frac{1}{4}$), and so on.

☑ Why is it necessary?

To avoid problems when a level-0 texel is much larger or smaller than a pixel. If the level-0 texel is much smaller (the texture image will appear tiny) we would like to use some kind of average of all the texels that cover the pixel (otherwise details, such as a picket fence, can disappear or not appear properly). The pixels in a higher MIPMAP level are averages of several pixels in a lower level. Consider a texture with a white picket fence and a black background and a situation where pixels are larger than texels. If we used just the level-0 image we might choose the texels that included only the black background between the white fence, so the fence would be invisible. If we used a prepared MIPMAP, say at level l, we would retrieve a texel that was a blend of background and the white fence, appearing gray.