Name ______________________________

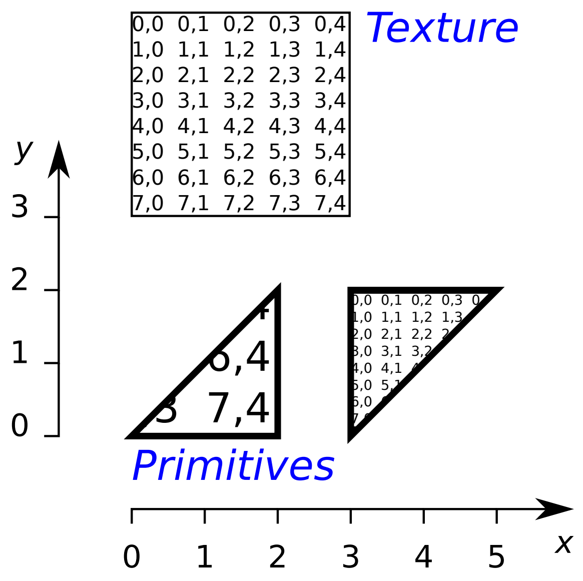
GPU Programming
EE 4702-1
Take-Home Pre-Final Examination

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.



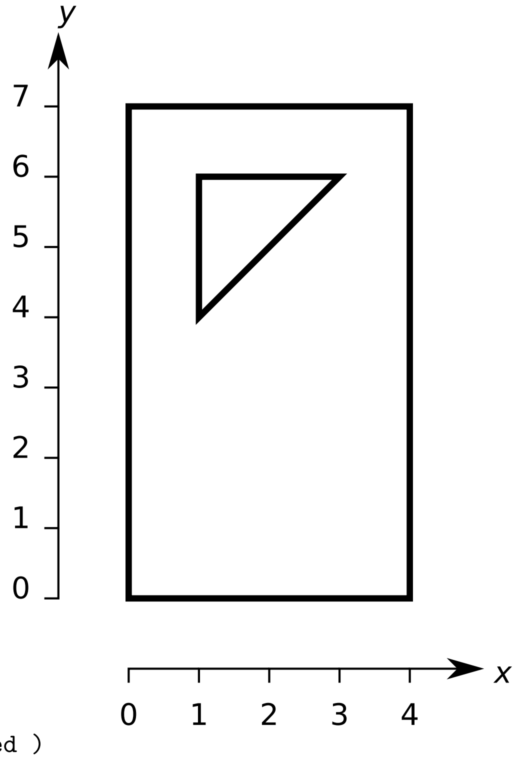
☐ Insert texture coordinates into `bset_triss` so that the texture renders as shown.

☐ 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:

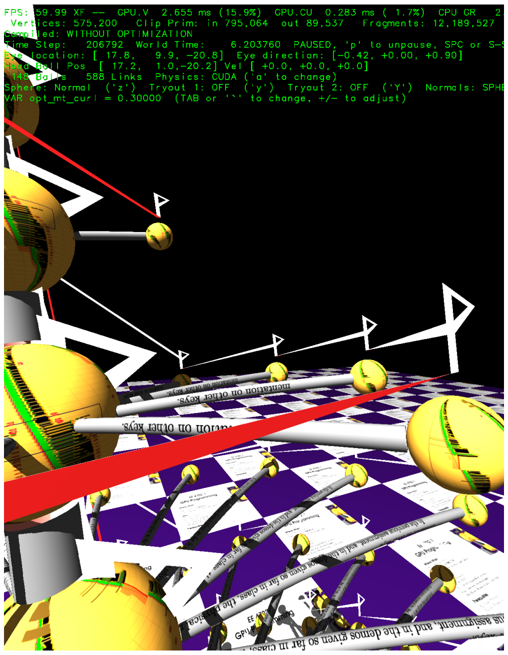
(*b*) Assume that the value of `n_balls` does not change very often, but that the values of `ball->position` do change each time `render_p2` is called. Based on this observation, modify the code to reduce the amount of data sent from the CPU to the GPU. The variable `n_changed` should be useful, it is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

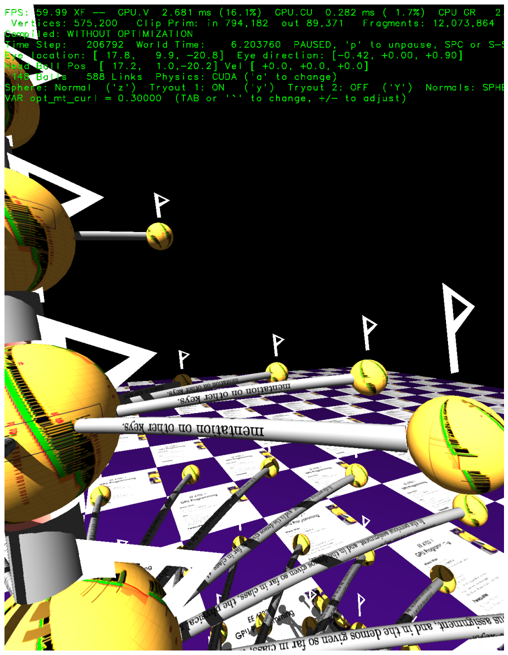
☐ Modify code to reduce CPU to GPU transfer when number of balls does not change.

☐ 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.*



```
FPS: 59.99 XF --  GPU.V  2.655 ms (15.9%)  GPU.CU  0.283 ms ( 1.7%)  CPU GR   2
 Vertices: 575,200   Clip Prim: in 795,064  out 89,537   Fragments: 12,189,527
Compiled: WITHOUT OPTIMIZATION
Time Step:   206792  World Time:     6.203760  PAUSED, 'p' to unpause, SPC or S-S
Eye location: [ 17.8,   9.9, -20.8]  Eye direction: [-0.42, +0.00, +0.90]
Head Ball Pos  [ 17.2,  1.0,-20.2] Vel [ +0.0, +0.0, +0.0]
 148 Balls   588 Links  Physics: CUDA ('a' to change)
Sphere: Normal  ('z')  Tryout 1: OFF  ('y')  Tryout 2: OFF  ('Y')  Normals: SPHE
VAR opt_mt_curl = 0.30000  (TAB or '`' to change, +/- to adjust)
```



```
FPS: 59.99 XF --  GPU.V  2.681 ms (16.1%)  GPU.CU  0.282 ms ( 1.7%)  CPU GR   2
 Vertices: 575,200   Clip Prim: in 794,182  out 89,371   Fragments: 12,073,864
Compiled: WITHOUT OPTIMIZATION
Time Step:   206792  World Time:     6.203760  PAUSED, 'p' to unpause, SPC or S-S
Eye location: [ 17.8,   9.9, -20.8]  Eye direction: [-0.42, +0.00, +0.90]
Head Ball Pos  [ 17.2,  1.0,-20.2] Vel [ +0.0, +0.0, +0.0]
 148 Balls   588 Links  Physics: CUDA ('a' to change)
Sphere: Normal  ('z')  Tryout 1: ON   ('y')  Tryout 2: OFF  ('Y')  Normals: SPHE
VAR opt_mt_curl = 0.30000  (TAB or '`' to change, +/- to adjust)
```

☐ 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.