Name Solution

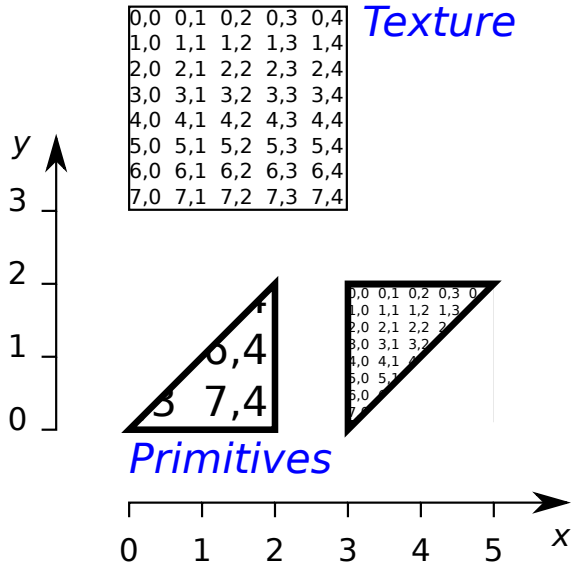
GPU Programming
EE 4702-1
Take-Home Pre-Final Examination

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.



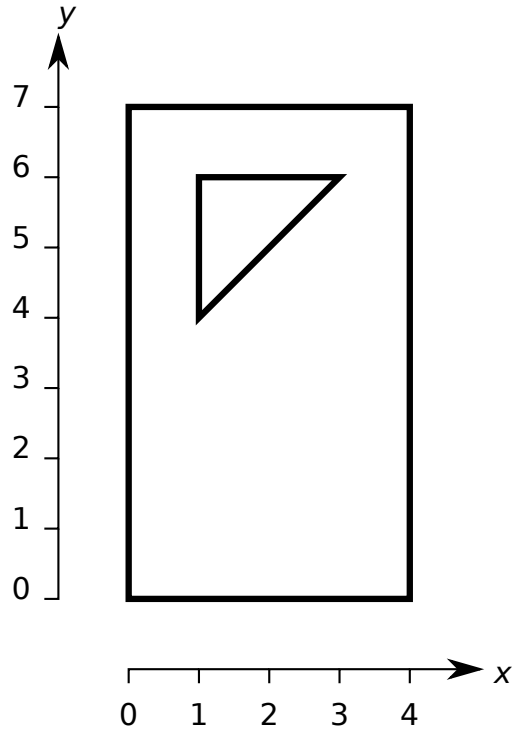
☑ Insert texture coordinates into `bset_triss` so that the texture renders as shown.

☑ 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 $16 \times 4\,\mathrm{B}$. Each `i` iteration inserts one matrix, so the total size is $64n\,\mathrm{B}$.

Each `pCoor`, a 4-element vector, is $4 \times 4\,\mathrm{B}$. 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 $16np\,\mathrm{B}$ for data in `bset_p2`.

The total amount of data is $n(64 + 16p)\,\mathrm{B}$.

(*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.

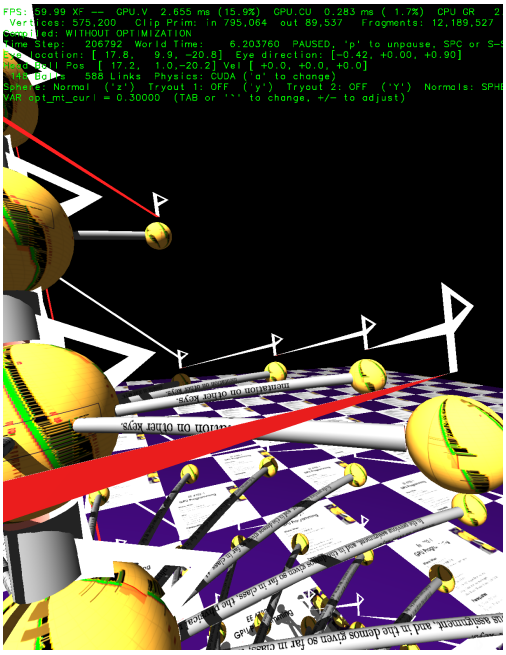
☑ Modify code to reduce CPU to GPU transfer when number of balls does not change.

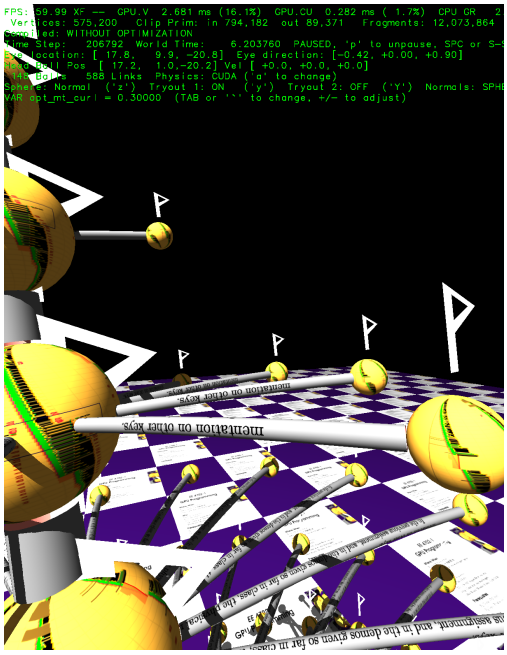
The calls to methods of `bset_p2` need to be guarded with `if ( n_changed )`. These changes are shown in the code in large blue type on the code several pages back.

☑ 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 $\frac{64}{64+16p}\,\mathrm{B}$.

(*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.