Name Solution

GPU Programming
LSU EE 4702-1
Final Examination

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

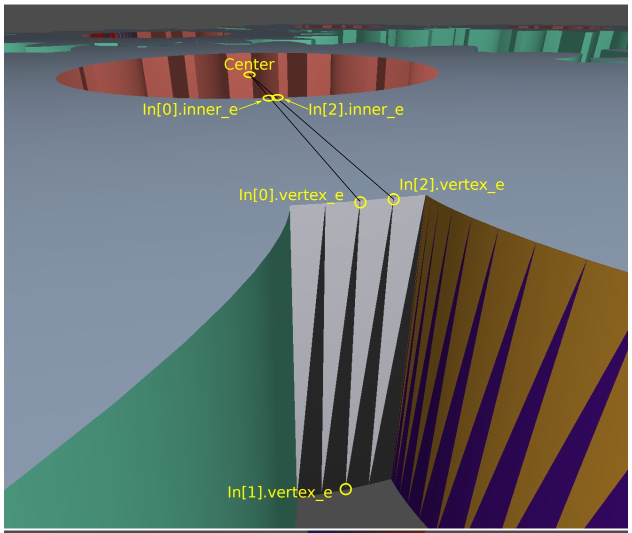
Problem 1		(20 pts)
Problem 2		(20 pts)
Problem 3		(15 pts)
Problem 4		(30 pts)
Problem 5		(15 pts)
Exam Total		(100 pts)

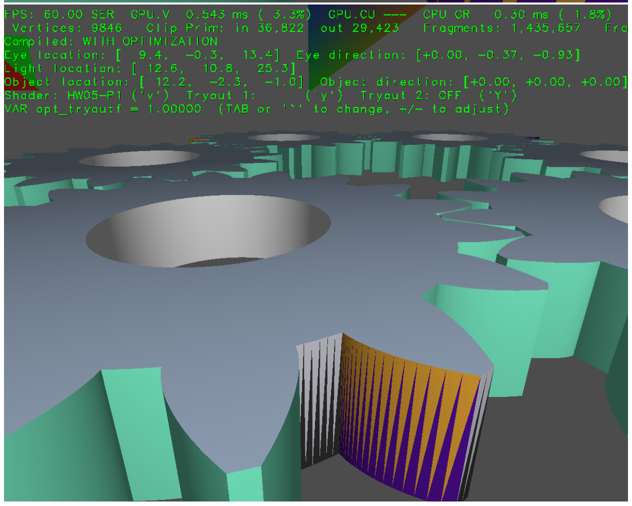
Alias Just say no to GPU scalpers.

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.*





```
FPS: 60.00 SER  GPU.V  0.543 ms ( 3.3%)  GPU.CU ---   CPU GR   0.30 ms ( 1.8%)
 Vertices: 9846   Clip Prim: in 36,822  out 29,423   Fragments: 1,435,657   Fra
Compiled: WITH OPTIMIZATION
Eye location: [  9.4,  -0.3,  13.4]  Eye direction: [+0.00, -0.37, -0.93]
Light location: [ 12.6,  10.8,  25.3]
Object location: [ 12.2,  -2.3,  -1.0]  Object direction: [+0.00, +0.00, +0.00]
Shader: HW05-P1 ('v')  Tryout 1:      ('y')  Tryout 2: OFF  ('Y')
VAR opt_tryoutf = 1.00000  (TAB or '`' to change, +/- to adjust)
```

☑ 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.

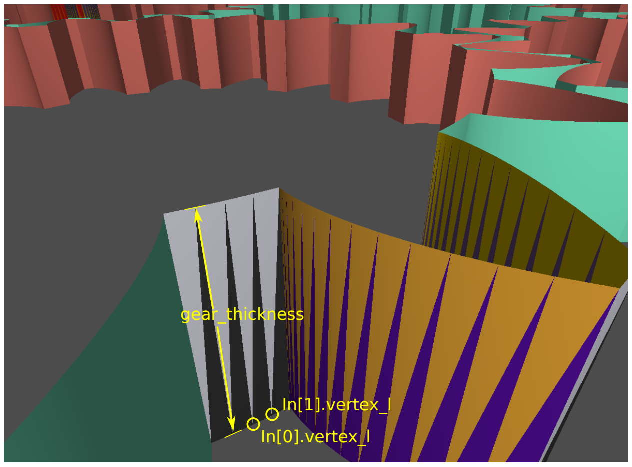
```
layout ( triangles ) in;
layout ( triangle_strip, max_vertices=8 ) out;
layout (location = 0) in Data_VG {
  vec4 vertex_c, vertex_e, inner_c, inner_e;
  vec3 normal_e;
  int color_idx;
} In[];

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();
}
```

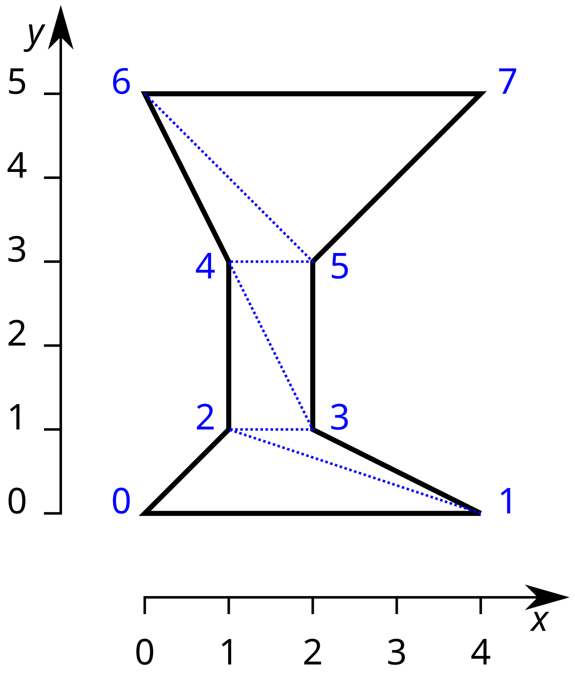
Problem 2: [20 pts] Answer the questions below about rendering an hourglass figure.

(*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.

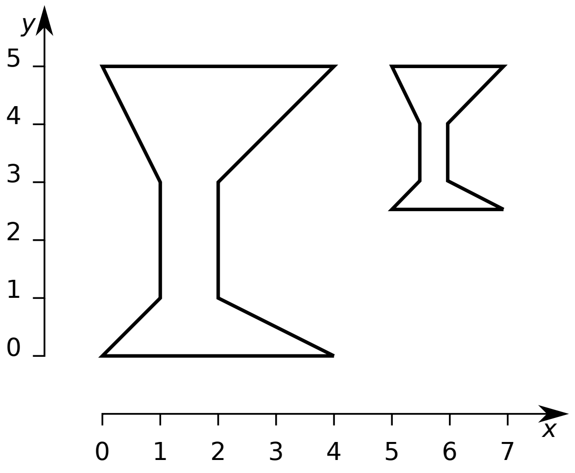


The solution appears below. In the code below vertex numbers are provided in the comments. In the diagram those same vertex numbers are shown, along with the resulting triangles (shown with blue dotted lines). A common mistake with a triangle strip is inserting coordinates in the wrong order, such as 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.