Name Solution

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

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

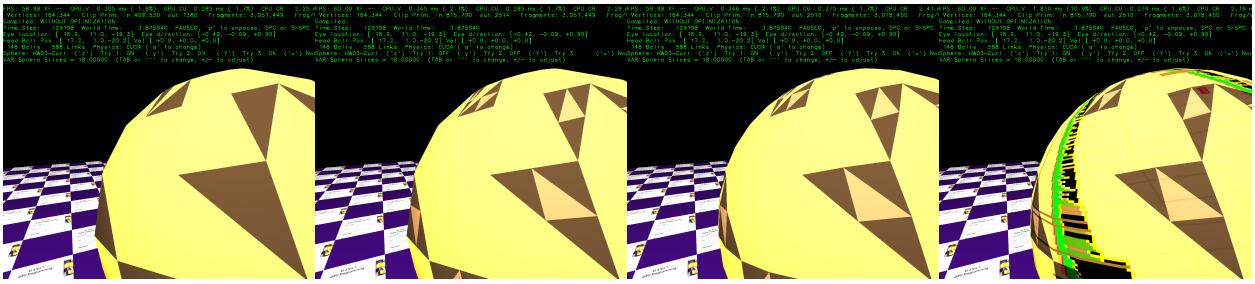
Work on this exam alone. Regular class resources, such as notes, papers, solutions, documentation, and code, can be used to find solutions. In addition outside Vulkan and OpenGL references and tutorials, other programming resources, and mathematical references can be consulted. Do not try to seek out 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.

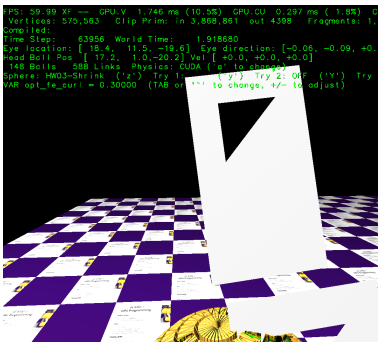
Consider a vector from a triangle vertex to the eye, that vector is called `v_to_eye` in the code below. A triangle with normal `normal_e` is facing away from the user if `dot( v_to_eye, normal_e ) < 0` . If the dot product is zero the triangle can't be seen because it is perfectly edge-on with the eye. (The eye is in the same plane as the triangle.)

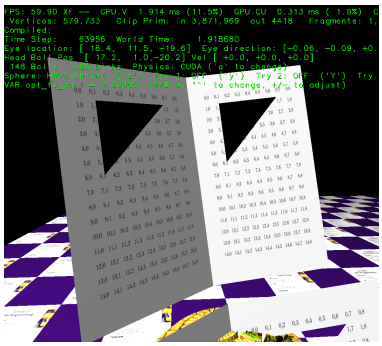
Recall that when rendering a sphere using triangles, the vertex normals are set to the normal of the sphere surface, not to the normal of the triangle. If a triangle crosses the edge of the sphere then at least one of its vertices is on a part of the sphere that is hidden and at least one is on a part 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.

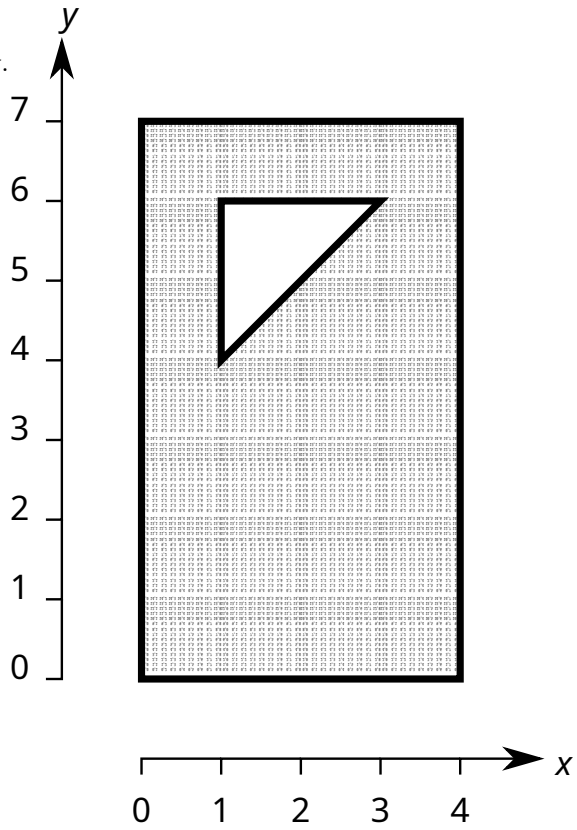




(*a*) First, get the texture right. The code below inserts texture coordinates into the buffer set for the card. Using the diagram show how the texture will appear based on this code. (It won't be blank.) The sampler has been set to repeat the texture.

☑ 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.