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

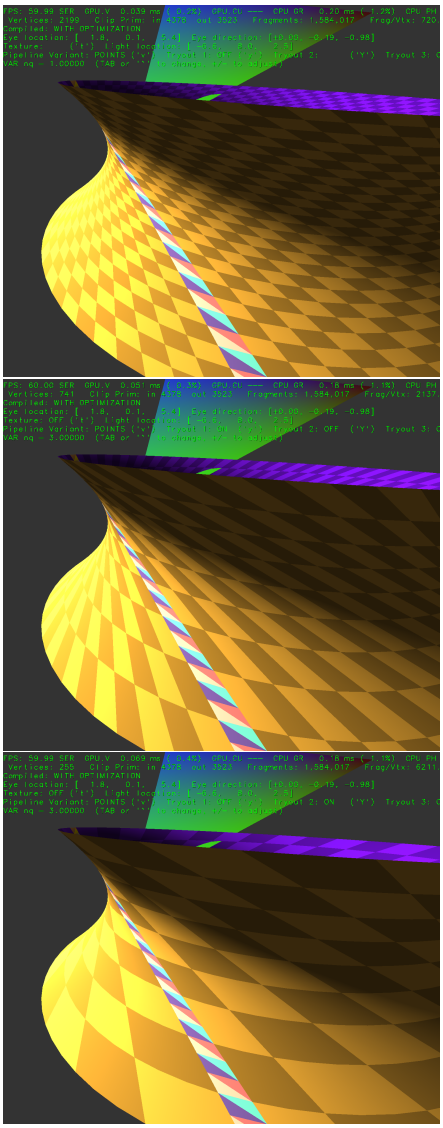
Problem 1	_____	(20 pts)
Problem 2	_____	(15 pts)
Problem 3	_____	(15 pts)
Problem 4	_____	(20 pts)
Problem 5	_____	(10 pts)
Problem 6	_____	(20 pts)
Exam Total	_____	(100 pts)

Alias _____

Good Luck!

Staple This Side

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

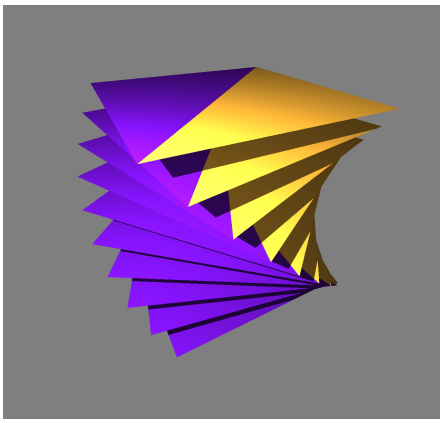
Staple This Side

```
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 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* `distance(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. Future exam question?
  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