Staple This Side

Name Solution

Formatted For Two-Sided Printing

GPU Programming
LSU EE 4702-1
Final Examination

Thursday, 12 December 2024 7:30-9:30 CST

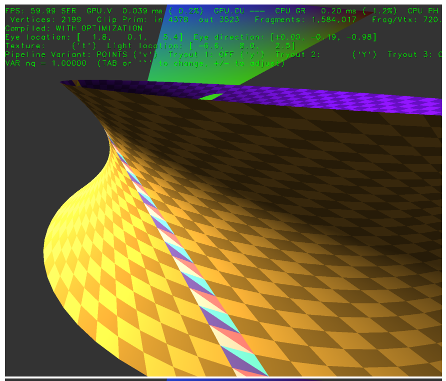
Problem 1		(20 pts)
Problem 2		(15 pts)
Problem 3		(15 pts)
Problem 4		(20 pts)
Problem 5		(10 pts)
Problem 6		(20 pts)
Exam Total		(100 pts)

Alias Maybe

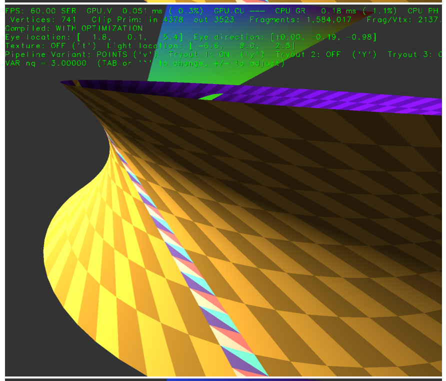
Good Luck!

Staple This Side

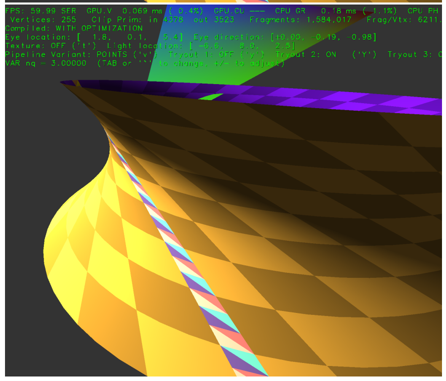
Problem 1: [20 pts] The screenshots to the right show our hyperboloid rendered using code based on the Homework 5 solution. The top screenshot is from code nearly the same as the solution (other than the checkered appearance). Each invocation of the geometry shader (shown on the facing page) draws one *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.



```
FPS: 59.99 SER  GPU.V  0.039 ms ( 0.2%)  GPU.CU ---  CPU GR   0.20 ms ( 1.2%)  CPU PH
 Vertices: 2199   Clip Prim: in 4378  out 3523   Fragments: 1,584,017   Frag/Vtx: 720.
Compiled: WITH OPTIMIZATION
Eye location: [  1.8,   0.1,   5.4]  Eye direction: [+0.00, -0.19, -0.98]
Texture:      ('t')  Light location: [ -6.6,   8.0,   2.5]
Pipeline Variant: POINTS ('v')  Tryout 1: OFF ('y')  Tryout 2:      ('Y')  Tryout 3: C
VAR nq = 1.00000  (TAB or '`' to change, +/- to adjust)
```



```
FPS: 60.00 SER  GPU.V  0.051 ms ( 0.3%)  GPU.CU ---  CPU GR   0.18 ms ( 1.1%)  CPU PH
 Vertices: 741   Clip Prim: in 4378  out 3523   Fragments: 1,584,017   Frag/Vtx: 2137.
Compiled: WITH OPTIMIZATION
Eye location: [  1.8,   0.1,   5.4]  Eye direction: [+0.00, -0.19, -0.98]
Texture: OFF ('t')  Light location: [ -6.6,   8.0,   2.5]
Pipeline Variant: POINTS ('v')  Tryout 1: ON  ('y')  Tryout 2: OFF  ('Y')  Tryout 3: C
VAR nq = 3.00000  (TAB or '`' to change, +/- to adjust)
```



```
FPS: 59.99 SER  GPU.V  0.069 ms ( 0.4%)  GPU.CU ---  CPU GR   0.18 ms ( 1.1%)  CPU PH
 Vertices: 255   Clip Prim: in 4378  out 3523   Fragments: 1,584,017   Frag/Vtx: 6211.
Compiled: WITH OPTIMIZATION
Eye location: [  1.8,   0.1,   5.4]  Eye direction: [+0.00, -0.19, -0.98]
Texture: OFF ('t')  Light location: [ -6.6,   8.0,   2.5]
Pipeline Variant: POINTS ('v')  Tryout 1: OFF ('y')  Tryout 2: ON   ('Y')  Tryout 3: C
VAR nq = 3.00000  (TAB or '`' to change, +/- to adjust)
```

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.

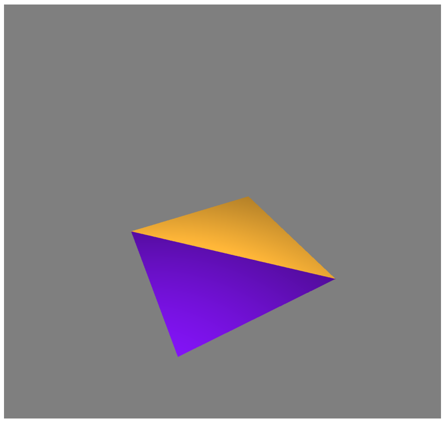
☑ Don't assume that n_q is a factor of either n_s or n_l. That is, don't assume $\lceil \frac{n_l}{n_q} \rceil n_q = n_l$ nor $\lceil \frac{n_s}{n_q} \rceil n_q = n_s$.

☑ Be sure that `EndPrimitive` is called at appropriate times.

Solution appears on the facing page. A new loop, `ss`, was added to iterate around the sides in a quad. The number of iterations in the existing `dl` loop was increased from 2 to `nq+1`. Each iteration of the `ss` loop draws on triangle strip. That strip is ended by a call to `EndPrimitive` at the end of the `ss` loop body.

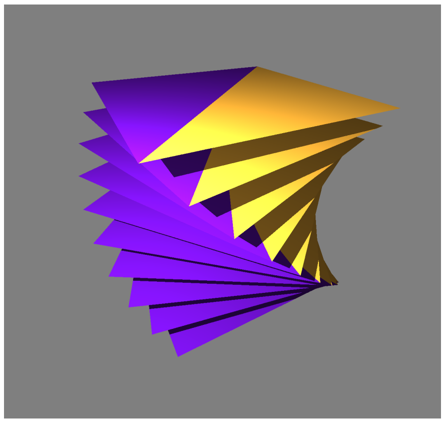
```
 // SOLUTION
layout ( triangle_strip, max_vertices = (2 + 2 * nq) * nq) out;

void gs_points() {
  // SOLUTION - Compute number of quads per line (vertical) and per ring.
  int q_per_line = ( uc.n_levels + nq - 1 ) / nq;  // Note: This rounds up.
  int q_per_ring = ( uc.n_sides + nq - 1 ) / nq;   // Note: This rounds up.

  // Compute index of quad along line ..
  int q_level0 = ( gl_PrimitiveIDIn % q_per_line );
  // .. and use it to find the index of the level along the line.
  int level0 = q_level0 * nq;

  // Compute index of side around ring ..
  int q_side_0 = gl_PrimitiveIDIn / q_per_line;
  // .. and use it to find the index of the side around the ring.
  int side0 = q_side_0 * nq;

  // Return if no work. Not needed if draw uses correct number of vertices.
  if ( side0 >= uc.n_sides ) return;

  for ( int ss = 0; ss < nq; ss++ )   // SOLUTION - Outer loop to iterate over sides in quad.
    {
      if ( side0 + ss >= uc.n_sides ) break;

      // SOLUTION - Inner loop iterates over nq+1 levels, not just 2.
      for ( int dl = 0; dl<=nq; dl++ )
        {
          int level = level0 + dl;
          if ( level > uc.n_levels ) break;

          // SOLUTION - Still need this two-iteration loop for strip.
          for ( int ds = 0; ds < 2; ds ++ )
            {
              int side = side0 + ss + ds;
              int idx = level + side * uc.s_levels;

              // 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();
            }
        }
      // SOLUTION - Call EndPrimitive at the end of the dl loop.
      EndPrimitive();
    }
}
```

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.

Solution appears on the facing page. Note that the normal, `n`, only needs to be computed once since a new square is updated by rotating an existing square around its normal (which so the rotated square has the same normal). Also, translating an object does not change its normals.

Note that transformations are applied from right to left.

Staple This Side

```
/// SOLUTION
//
// Construct vectors along sides of the square ..
//
pVect v01( p0, p1 ),  v02( p0, p2 );
//
// .. and use them to compute the normal, side length, and square center.
//
pNorm n = cross( v02, v01 );                             // ☑ Compute n.
float side_len = v01.mag();
pCoor square_center = p0 + 0.5 * ( v01 + v02 );

// Compute rotation angle and square separation specified in the problem.
//
float delta_theta = 0.5 * numbers::pi / max( 1, n_squares - 1 );
float delta_z = side_len / max( 1, n_squares - 1 );

// The transformation, m, consists of the following steps:
//
// - Translate so that the center of the square is moved to the origin.
// - Rotate by delta_theta degrees around the normal, n.
// - Translate the square back to its original location.
// - Translate the square up by delta_z.
//
// Construct the respective transformation matrices:
//
pMatrix_Translate m_to_origin( -square_center );
pMatrix_Rotation  m_rot( n, delta_theta );
pMatrix_Translate m_to_square( square_center );
pMatrix_Translate m_up( delta_z * n );

pMatrix m = m_up * m_to_square * m_rot * m_to_origin; // ☑ 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.

By looking at the data inserted into buffer set `bset_hyperb` one can see that four kinds of data are inserted, a `pCoor`, such as `p00`, a `pTCoor`, such as `pTCoor(0,0)`, a `pNorm`, such as `n00`, and an `int`, such a `c1`. These four items are the attributes associated with a vertex. Their number of elements are 4, 2, 3, and 1, respectively. The total data per vertex is then: $\boxed{(4+2+3+1)4\,\mathrm{B} = 40\,\mathrm{B}}$.

[✓] For the individual triangle grouping how many times will the vertex shader be invoked per draw (frame)?

First, count the number of vertices inserted for one loop body. To do that count any type of data, such as `pCoor`. Notice that six vertices are inserted in the loop body, the first group of lines inserts three, the next group inserts another three. Use n_l for `n_levels` and n_s for `n_sides`. The inner loop iterates n_l times, the outer loop n_s times, so the loop body executes $n_l n_s$ times, and the vertex shader $\boxed{\text{will be invoked } 6n_l n_s \text{ times}}$.

[✓] For the individual triangle (triangle list) 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 an individual triangle (triangle list) grouping the geometry shader is invoked $3n_l n_s/3 = n_l n_s$ times. Each geometry shader invocation uses three vertices, and each vertex is used for exactly one geometry shader invocation. (When number of vertices is a multiple of 3, which is the case here.)

Assuming that clipping is done by the rasterizer, and not before it, the number of triangles sent to the rasterizer is also $n_l n_s$ because our geometry shader always semits one triangle when the color index is positive.

[✓] For the triangle strip grouping how many times will the vertex shader be invoked per draw (frame)? [✓] Pay attention to the loop bounds.

The number of vertices inserted in the loop body is two. The number of iterations is $n_s(n_l+1)$, and so $2n_s(n_l+1)$ vertices are inserted.

[✓] 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)?

Assuming the geometry shader always emitted exactly one triangle (which is not the case here), a draw with V vertices would result in $V-2$ triangles. In our case the number of times the $\boxed{\text{geometry shader is invoked is } 2n_s(n_l+1)-2}$.

Notice that the given geometry shader does not emit a triangle if the last vertex has a color index of zero, and so one needs to look more closely to determine the number of triangles sent to the rasterizer (emitted by the geometry shader). (For this code if the color index is zero a new strip is being started and that some vertices sent to the geometry shader are from the prior strip and we don't want to form a triangle using vertices from the prior and current strip.) So for each strip two invocations are ignored, those are the invocations due to the vertices inserted when `lev=0`. So the total number of triangles $\boxed{\text{sent to the rasterizer is just } 2n_s n_l}$.

[✓] 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.

Roughly one third as many vertices are sent using a triangle strip compared to the individual triangles topology. By ANSI standard 2011-04-01 for performance class descriptors, triangle strips are a lot better. Even with triangle strips a particular vertex is inserted multiple times. Consider two iterations, `side=0;lev=0;` and `side=1;lev=1`. Vertex `p11` from the `side=0;lev=0;` iteration is the same as vertex `p00` from the `side=1;lev=1` iteration, so there is still redundancy. An indexed draw would remove all redundancy of this type.

```
    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();  }
```

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.)

Solution appears on the facing page. The ray payload has been initialized to tan, but the alpha component still set to zero. After `traceNV` returns write the frame buffer with the ray payload. If there was a miss it will still be tan.

(*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.

The specify the range of distances to consider for intersections. The distances are lengths of the ray argument to `traceNV`, which is `eye_to_pixel_g` in the code shown. An intersection at a distance less than `tmin` or more than `tmax` will be ignored.

☑ 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).

Our ray tracing code looks at every triangle, so it does not matter what `tmax` is. In a practical ray tracing implementation the triangles are organized to reduce the number of intersection tests. One way is to sort triangles into boxes (hierarchically). If the closest face of one such box is already at distance greater than `tmax` from the eye then there is no need to test the triangles in the box, saving 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* `length(MYVECTOR)` *to find the length of a vector.*

The solution appears on the facing page.

```
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 tmin = 1;

  // SOLUTION (b) -- Compute global space coordinates of pixel on far plane ..
  //
  vec4 pixel_cf = vec4( pixel_c2d.x, pixel_c2d.y,  1,  1 );
  vec4 pixel_gf = homogenize( ut.object_from_clip * pixel_cf );
  //
  // .. and compute vector from eye to far pixel.
  //
  vec3 eye_to_pixel_gf = pixel_gf.xyz - eye_g.xyz;
  //
  // Now set tmax to length of eye_to_pixel_gf scaled to eye_to_pixel_g:
  //
  tmax = length( eye_to_pixel_gf ) / length( eye_to_pixel_g );

  // SOLUTION (a) -- Initialize payload to tan.
  //
  rp_color = color_tan;   rp_color.a = 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; // Removed for part (b) solution.

  imageStore(fb_image, ivec2(gl_LaunchIDNV.xy), vec4(rp_color.rgb, 0.0));
```

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.

Solution appears below. Vector `vec_ve` is a vector from the eye to the vertex (fragment) (remember that in eye space the eye is at the origin). The dot product of this vector and the vector from the vertex to the light indicates how much they point in the same direction. Raising it to a power enhances this, meaning that `align` will only be close to one if the two vectors nearly point in the same direction. The value of `align` is used to increase the amount of light, `attenuation`.

```
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];



  /// SOLUTION
  vec3 vec_ve = vertex_e.xyz;
  float align = pow( dot( normalize(vec_ve), normalize( vec_vl ) ), 11 );
  if ( retro && align > 0 ) attenuation += 10 * align * attenuation;



  // Compute lighted color.
  frag_color = texel * mat_color * uni_light.color * attenuation;
}
```

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?

A material property is what is commonly called color, it describes how the primitive interacts with light. A lighted color is how the primitive appears taking into account the material property and also the color of the light and the positioning of the primitive relative to the light. A material property might be one of the vertex attributes, or it could be common to all primitives in a draw. The lighted color is computed by the vertex or fragment shader and is written to the frame buffer (perhaps after being mixed with textures).

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.

A vertex color attribute works best when each vertex or triangle has its own unique color. The same effect could be obtained using a color index, but that would require more data since for each vertex one must move (perhaps from CPU to GPU) a color index and the color itself. It's not just the increase in data size, there would be the added delay from reading the color from a storage buffer.

☑ Describe a situation in which a color index attribute is the best of the three options. ☑ Explain the advantage.

A color index attribute works best when there is more than one color, but the number of different colors is small. Because a color index is $\frac{1}{4}$ the size of a color, the amount of data moved from CPU to GPU is smaller. (The amount of data moved from GPU memory to GPU cores would also be smaller due to caches, some thing not covered in this course but which is covered in LSU EE 7722.).

☑ Describe a situation in which a color in a uniform variable is the best of the three options. ☑ Explain the advantage.

This works best when all primitives in a draw are the same color. This uses the least amount of data, and has the added benefit that data in uniform variables are read quickly.

(*b*) Answer the following questions on homogeneous coordinates.

☑ Provide an example of a homogeneous coordinate ☑ and its homogenized value.

An example of a homogeneous coordinate is $P = \begin{bmatrix} 6 \\ 8 \\ 10 \\ 2 \end{bmatrix}$. The homogenized value of P is $\begin{bmatrix} 3 \\ 4 \\ 5 \\ 1 \end{bmatrix}$.

☑ Describe something that can be done with a homogeneous coordinate needed for rendering that can't be done with ordinary coordinates.

It is possible to apply a translation to a homogeneous coordinate by multiplying by a 4×4 matrix. It is also possible to apply a projection transformation by a 4×4 matrix (a frustum transform).