Name Solution

GPU Programming
EE 4702-1
Midterm Examination

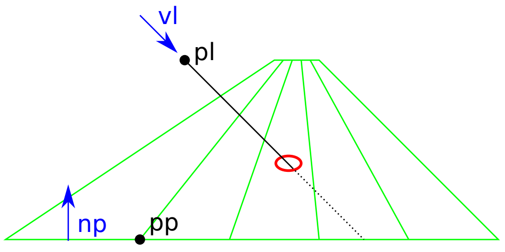
Wednesday, 18 October 2017 11:30–12:20 CDT

Problem 1		(30 pts)
Problem 2		(20 pts)
Problem 3		(25 pts)
Problem 4		(25 pts)
Exam Total		(100 pts)

Alias Think of something, quick!

Good Luck!

Problem 1: [30 pts] The illustration to the right shows a ring surrounding the point at which a line crosses a plane. When completed the code fragment below will render the ring. The line is defined by parametric equation $P(t) = \texttt{pl} + t \times \texttt{vl}$. Point `pp` is on the plane and vector `np` is normal to the plane. The ring is circular and is on the plane.



(a) Modify the code below so that `t` is assigned the value for which `pc` is on the plane.

☑ Compute t such that $P(t)$ is on the plane.

This is of course a line/plane intercept problem. We need to find t such that $\overrightarrow{P(t)P_p} \cdot n_p = 0$, where P_p is `pp` and n_p is `np`. First, separate t,

$$\begin{aligned} 0 =& \overrightarrow{P(t)P_p} \cdot n_p \\ =& (P_p - P(t)) \cdot n_p \\ =& (P_p - P_l - tv_l) \cdot n_p \\ =& \left(\overrightarrow{P_lP_p} - tv_l\right) \cdot n_p \\ =& \overrightarrow{P_lP_p} \cdot n_p - tv_l \cdot n_p \end{aligned}$$

then finish solving for t:

$$t = \frac{\overrightarrow{P_lP_p} \cdot n_p}{v_l \cdot n_p}$$

The code below computes t this way.

(*b*) Modify the code below to render the ring. Use `r1` for the inner radius and `r2` for the outer radius. Render it using `slices` sets of triangles. Use variable `pc` even if you haven't yet solved the previous part.

☑ Include: ☑ Code to set up the rendering pass, ☑ to set the color (red), ☑ to set the correct normal, ☑ and to compute and set vertex coordinates.

Solution appears below.

```
World::render_ring( pCoor pl,  pVect vl,     // Line definition.
                    pCoor pp,  pNorm np,     // Plane definition.
                    float r1,  float r2 ){  // Ring radii.

  pVect vlp(pl,pp);  // SOLUTION


  float t = dot(vlp,np) / dot(vl,np); // SOLUTION

  pCoor pc = pl + t * vl;

  // SOLUTION - Compute axis for drawing ring.
  pNorm ax(pc,pp);
  pNorm ay = cross(np,ax);

  glBegin(GL_TRIANGLE_STRIP);

  glNormal3fv(np);
  glColor3fv( color_red );

  const int slices = 100;
  const float delta_theta = 2 * M_PI / slices;
  for ( int i=0; i<=slices; i++ ) {
      const float theta = i * delta_theta;


      pVect d = ax * cos(theta) + ay * sin(theta);
      glVertex3fv( pc + r1 * d );
      glVertex3fv( pc + r2 * d );

    }
}
```

Problem 2: [20 pts] Recall that in Homework 2 a sphere was rendered using a spiral slice. The coordinates of the slice were placed in a buffer object and a sphere was rendered by performing multiple rendering passes each using the same buffer object but a different transformation. Let c denote the number of coordinates in the buffer object, and s denote the number of slices per sphere.

(a) What is the size of the buffer object, in bytes? *Hint: This is an easy question.*

☑ Size of buffer object, in bytes. ☑ State any assumptions made.

Short answer: $4 \times 4c\,\mathrm{B} = 16c\,\mathrm{B}$, with `pCoor` used for a coordinate, which I assume I remember correctly.

Explanation: The buffer object holds c coordinates, each coordinate is a `pCoor`. A `pCoor` is a structure holding four `float`s, so its size is $4 \times 4\,\mathrm{B}$.

(b) Appearing below is the code rendering a sphere, adapted from the Homework 2 solution. Based on this code, estimate the amount of data send from the CPU to the GPU to render the sphere, **not counting the buffer object**.

```
pMatrix_Rotation rot_xz( pVect(0,1,0), delta_theta ),  rot_yz( pVect(1,0,0), M_PI );
// Code to render one sphere.
glTranslatefv( ball->position );
glScale1f( ball->radius );
glMultTransposeMatrixf( pMatrix_Rotation(ball->orientation) );

for ( int j=0; j<2; j++ ) {
     for ( int i=0; i<s/2; i++ ) {
        glColor3fv( i & 1 ? color_lsu_spirit_purple : ball->color );
        glDrawArrays(GL_TRIANGLE_STRIP,0,c);
        glMultTransposeMatrixf(rot_xz);
      }
    glMultTransposeMatrixf(rot_yz);
  }
```

☑ In terms of s, amount of data sent to GPU to render sphere.

Short Answer: For each slice a new modelview matrix and color are sent to the GPU, for a total of $64 + 12 = 76\,\mathrm{B}$ per slice. For the entire sphere $\boxed{76s\,\mathrm{B}}$.

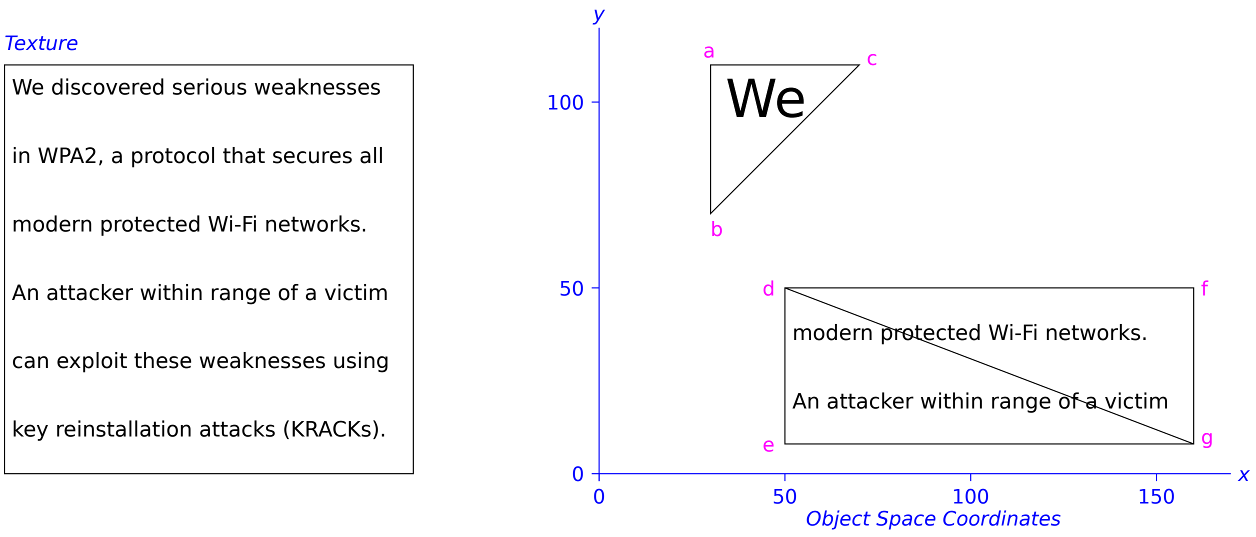
Explanation: The modelview matrix has $4 \times 4 = 16$ elements of size 4 bytes each, for a total size of $4^3 = 64\,\mathrm{B}$. The color size is $3 \times 4 = 12\,\mathrm{B}$. Because `glColor3fv` is called just before the rendering pass is started it should be clear that it is sent once per rendering pass, contributing $12s\,\mathrm{B}$ of data. The number of times the modelview matrix is sent to the GPU depends upon the OpenGL implementation. The commands `glTranslatefv`, `glScale1f`, and `glMultTransposeMatrixf` are all used to modify the default matrix (modelview in this case). An *eager* implementation might send the matrix each time one of these was called. A *lazy* implementation would update the copy of the modelview matrix on the CPU, but only send it to the GPU when a rendering pass is initiated. (In this situation lazy is better.) This solution assumes a lazy implementation. Since the loop body (which contains the command initiating the rendering pass, `glDrawArrays`) is executed $2 \times \frac{s}{2} = s$ times the array is sent s times and so the array contributes $64s\,\mathrm{B}$ of data per sphere. (An eager implementation would send the matrix $3 + 2(1 + \frac{s}{2})$ times, with the commands before the loop contribute the 3 term, and the $1+$ part contributed by the `glMultTransposeMatrixf` at the end of the `j` loop.)

(c) Buffer object BO2 holds coordinates to render an entire sphere, not just a slice, $s \times c$ coordinates total. Buffer object BO1 holds the slice discussed above. Suppose that initially all data is on the CPU. Let n be the number of sphere to be rendered. In terms of s and c, what is the smallest value of n for which using BO2 will require less data than BO1?

☑ Break-even value for n:

Using BO1 the total amount of data for n spheres is $16c + 76ns$. Using BO2 the total amount of data is $16sc + 76n$. Equating and solving for n yields $n = \frac{16c(s-1)}{76(s-1)} = \frac{16}{76}c$.

Problem 3: [25 pts] The diagram below shows a texture on the left-hand side and a scene to be rendered on the right. The scene consists of three triangles with the texture applied. The text in the texture, written by Mathy Vanhoef, is from the description of a widespread vulnerability in WiFi WPA2 implementations. See `https://www.krackattacks.com`, and please, Fall 2017 people, update your OS and wireless access point firmware. *Grading Note: The description of where the text came from was not in the original exam. The KRACK vulnerability became public two days before the midterm exam.*


Texture
We discovered serious weaknesses
in WPA2, a protocol that secures all
modern protected Wi-Fi networks.
An attacker within range of a victim
can exploit these weaknesses using
key reinstallation attacks (KRACKs).
y
x
100
50
0
0
50
100
150
a
b
c
We
d
e
f
g
modern protected Wi-Fi networks.
An attacker within range of a victim
Object Space Coordinates

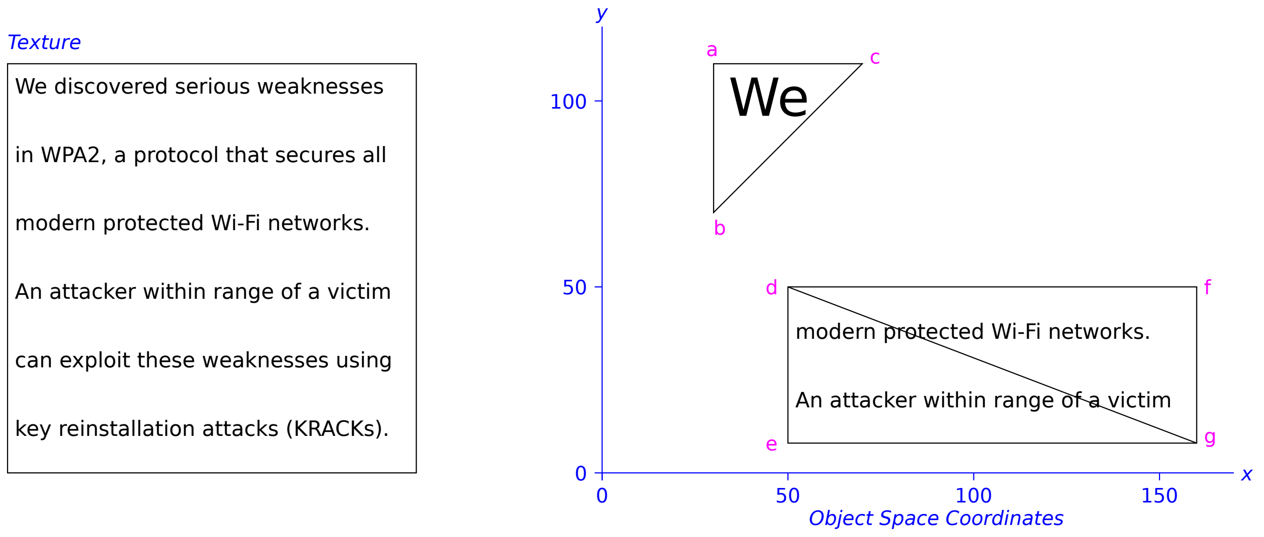
(*a*) Complete the code below so that it renders the triangles with the texture applied as shown. The color and normal have already been set, just specify vertex and texture coordinates. Use abbreviation `glV` for `glVertex2f` and `glT` for `glTexCoord2f`. Only specify x and y components for the vertex coordinates.

☑ Specify ☑ texture and ☑ vertex coordinates so that the scene is rendered as shown.

Solution appears below in blue. The solution uses the full function names, such as `glTexCoord2f`. On the exam it would be okay to use the abbreviations `glT` and `glV`.

```
glBegin(GL_TRIANGLES);
glNormal3f(0,0,1);
glColor3f(0.5,0.5,0.5);

glTexCoord2f(0,1);       // SOLUTION: Texture coord for Vtx a.
glVertex2f(30,110);      // Vertex a
// SOLUTION (below): Specify the remaining texture and vertex coords.
glTexCoord2f(0,.85);    glVertex2f(30,70);    // Vertex b
glTexCoord2f(0.1, 1);   glVertex2f(70,110);   // Vertex c

glTexCoord2f(0, .66);   glVertex2f(50,50);    // Vertex d
glTexCoord2f(0, .33);   glVertex2f(50,10);    // Vertex e
glTexCoord2f(1, .33);   glVertex2f(160,10);   // Vertex g

glTexCoord2f(0, .66);   glVertex2f(50,50);    // Vertex d
glTexCoord2f(1, .33);   glVertex2f(160,10);   // Vertex g
glTexCoord2f(1, .66);   glVertex2f(160,50);   // Vertex f


glEnd();
```

Problem 3, continued:



We discovered serious weaknesses

in WPA2, a protocol that secures all

modern protected Wi-Fi networks.

An attacker within range of a victim

can exploit these weaknesses using

key reinstallation attacks (KRACKs).

modern protected Wi-Fi networks.

An attacker within range of a victim

(b) Suppose that defg in the diagram above is four pixels across and two pixels down. Suppose that texturing were set up to use only a single mipmap level and nearest filtering. Explain how defg could appear all white in that case.

✓ At 4×2 pixels defg all white because:

Because the center of each pixel just happens to be over a part of the texture that's white. That's not so unlikely because most of the texture is white.

(c) Explain how the 4×2 rectangle would appear with linear filtering and mipmap levels. Explain the key points of linear texture filtering.

✓ Appearance with mipap levels and linear filtering.

The rectangle would appear gray.

✓ Key points of filtering.

Given a texture of width w, the width of the texture at mipmap level i is $w2^{-i}$. (Mipmap level 0 is the original image.) The color of a texel at mipmap level i might be the average of the four texels that it covers at mipmap level $i-1$. (This method of determining the color of a texel at a particular mipmap level is similar to the method used for determining the color of a filtered texel for a fragment.)

With linear-mipmap-linear filtering enabled, the filtered texel corresponding to a pixel is computed using the two closest mipmap levels, call them levels i and $i+1$. Level i is chosen so that the texels are smaller than the pixels and at level $i+1$ the texels are larger than the pixels. The filtered texel is based on a weighted average of up to four pixels at each of the two levels. The weight for a texel is based on the fraction of the pixel covered by the texel.

Problem 4: [25 pts] Answer each question below.

(*a*) Show a transformation matrix M that will translate a homogeneous coordinate P to $P + \begin{bmatrix} 7 \\ 6 \\ 5 \end{bmatrix}$. (That is, find M such that $MP = P + \begin{bmatrix} 7 \\ 6 \\ 5 \end{bmatrix}$.) Only show nonzero array elements.

☑ $M =$

The transformation matrix is:

$$M = \begin{pmatrix} 1 & 0 & 0 & 7 \\ 0 & 1 & 0 & 6 \\ 0 & 0 & 1 & 5 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

(*b*) Describe the "from" and "to" coordinate spaces for the following OpenGL matrices:

☑ The ModelView matrix maps a coordinate from ☑ object space to ☑ eye space.

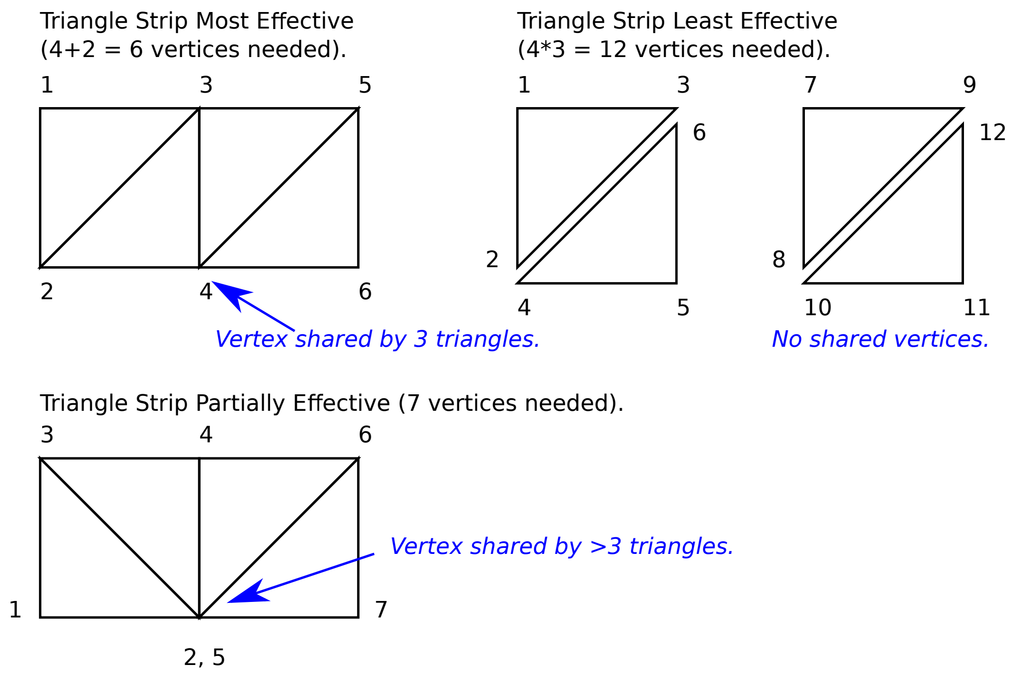
☑ The Projection matrix maps a coordinate from ☑ eye space to ☑ clip space.

(*c*) Show two drawings, each consisting of four triangles. The one should be well-suited for rendering using a triangle strip, for the other a triangle strip should make no difference.

☑ Show a 4-triangle example well-suited to triangle strips and one in which a ☑ triangle strip doesn't help at all.

The well-suited triangle strip appears on the upper left. What makes this effective is that except for the starting (1,2) and ending (5,6) vertices, each vertex is shared by three triangles. Triangle strips don't help at all when there aren't any shared vertices, as in the drawing on the upper right. In the bottom drawing, labeled Partially Effective, triangle strips are better than individual triangles, but since 7 rather than 6 vertices are needed, they are not most effective. A triangle fan would be the best choice for this drawing.

Grading Note: Many answers to the doesn't-help part seemed to be based on the assumption that the triangles had to be touching in some way. If that were the case the question could be answered by having a vertex of one triangle touching an edge (but not a vertex) of another triangle.


Triangle Strip Most Effective
(4+2 = 6 vertices needed).
1
3
5
2
4
6
Vertex shared by 3 triangles.
Triangle Strip Least Effective
(4*3 = 12 vertices needed).
1
3
6
2
4
5
7
9
12
8
10
11
No shared vertices.
Triangle Strip Partially Effective (7 vertices needed).
3
4
6
1
7
2, 5
Vertex shared by >3 triangles.

(*d*) Suppose that buffer object o30 has 30 sphere coordinates and buffer object o1k has 1000 sphere coordinates. In both buffer objects the sphere coordinates are well chosen. They will be used to render a sphere which is fully visible. *Note: In the original exam the sizes were 1000 and 10,000.*

☑ Describe a situation in which the sphere would look better using o1k than using o30 and ☑ describe a situation in which the sphere would look the same with o1k as with o30. ☑ Explain.

Short Answer: o1k looks better when the projected sphere is large (covers alot of the window [frame buffer]), they look about the same when the projected sphere size is small. For the large sphere o30 can look bad because there are many pixels between triangle vertices so there will be straight lines where the modeled object is curved.

Long Answer: The sphere would look better using o1k than with o30 if the distance between triangle vertices using o30 were many pixels. A triangle edge would appear as a straight line and this would be most apparent along the limb (edge) of the sphere. With o1k these lines would be shorter, and so the curve would be better approximated. In contrast, if the entire sphere covered a small fraction of the window, then even using o30 the distance between vertices in a triangle would be only a pixel or two and so the edge of the sphere would appear properly curved either way.

Grading Note: Many answered that the situations differed in the use of surface normals for computing lighting. This is only partly correct. If triangle normals are used then the change in lighting from one triangle to another would be sudden making the triangle boundaries obvious. For that reason, o1k might look better since the triangles are smaller. In contrast, if sphere surface normals are used then the lighted color at points that are nearby but on different triangles would be almost the same (because being nearby there normals would be almost identical). So in this situation fewer vertices are needed for good appearance away from the sphere limb. However, the limb (edge) of the sphere would still look bad for o30 since it is constructed of relatively long straight lines.

Suppose that it takes much more time to render a sphere using o1k than with o30. The vertex, geometry, and fragment shader stages are running custom written code which could use some tuning.

☑ Which shaders should be tuned to potentially fix the performance problem? ☑ Which shaders be ignored? ☑ Explain.

The vertex shader is run once for each vertex. Since o1k has many more vertices the vertex shader could be responsible for the low performance. The same argument can be made for the geometry shader. In contrast, the fragment shader is run once per fragment and the number of fragments will be the same using o1k and o30 since the projected size of the sphere would be the same. For that reason there is no need to tune the fragment shader.