Name Solution

GPU Programming
EE 4702-1
Midterm Examination

Friday, 6 November 2009, 9:40–10:30 CST

Problem 1		(40 pts)
Problem 2		(20 pts)
Problem 3		(20 pts)
Problem 4		(20 pts)
Exam Total		(100 pts)

Alias Transformed

Good Luck!

Problem 1: [40 pts] Write OpenGL code to render a filled circle (a disc) of radius `r`, centered at the origin and with normal $(0, 1, 0)$.

- The distance between all vertices in a primitive should be approximately `vert_dist`.
- Use triangle strips. Multiple strips are okay, but there should be no easy way to make the strips longer.
- Use `glVertex3f` calls, don't try to construct arrays.
- Don't specify colors, normals, or other attributes.
- Assume transformations, lighting, etc, have all been set up. Start with `glBegin`.

☑ Code rendering a circle of radius `r`, center at origin.

☑ Vertex distance about `vert_dist`.

☑ Good use of strips.

☑ Code reasonably efficient.

Solution appears below. The code renders a series of concentric rings, each ring is rendered using a triangle strip. The innermost ring would probably be better rendered using a triangle fan, but the problem says to use a triangle strip.

```
// SOLUTION
for ( float r_outer = r;  r_outer > 0; r_outer -= vert_dist )
  {
    const float delta_theta = vert_dist / r_outer;
    const float r_inner = max(0.0,r_outer-vert_dist);

    glBegin(GL_TRIANGLE_STRIP);
    glNormalf(0,1,0);
    for ( float theta = 0; theta <= two_pi; theta += delta_theta )
      {
        glVertex3f( r_outer * cosf(theta), 0, r_outer * sinf(theta) );
        glVertex3f( r_inner * cosf(theta), 0, r_inner * sinf(theta) );
      }
    glEnd();
  }
```

Problem 2: [20 pts] Answer the normal questions below.

(*a*) Show an expression for the normal to triangle ABC, where A, B, and C are the vertex coordinates.

☑ Normal to ABC

The triangle defines a plane, and the normal to a plane is just the cross product of any two non-parallel vectors in the plane. Two convenient vectors are $\vec{AB}$ and $\vec{AC}$. So the normal is $\vec{AB} \times \vec{AC}$.

(*b*) In the code sample below the pair of triangles is rendered using two different methods, identified as Method 1 and Method 2.

Describe the difference in appearance of the triangles rendered using Method 1 and Method 2 when diffuse lighting is used and `norm_ABC != norm_CBD`.

```
pVect norm_ABC = find_normal(A,B,C);    pVect norm_CBD = find_normal(C,B,D);

// Method 1
glBegin(GL_TRIANGLES);
glNormalfv(norm_ABC);
glVertex3fv(A);  glVertex3fv(B);  glVertex3fv(C);

glNormalfv(norm_CBD);
glVertex3fv(C);  glVertex3fv(B);  glVertex3fv(D);
glEnd();

// Method 2
pNorm norm_X = norm_ABC + normCBD; // Sum of two vectors normalized.
glBegin(GL_TRIANGLES);
glNormalfv(norm_ABC);  glVertex3fv(A);
glNormalfv(norm_X);    glVertex3fv(B);  glVertex3fv(C);

glVertex3fv(C);        glVertex3fv(B);
glNormalfv(norm_CBD);  glVertex3fv(D);
glEnd();
```

☑ Difference in appearance between Method 1 triangles and Method 2 triangles.

Since a diffuse light model is used the lighted color of a vertex is a function of both its distance from the light source and its normal.

In Method 1, two different normals are used for vertex B, one normal when it's part of triangle ABC and a different normal when it's part of CBD. As a result there are two different lighted colors for B and so there could be a sudden change in shading between points on ABC and CBD that are near B (the same is true for C). The appearance will be two triangles touching along an edge.

In Method 2, the same normal is used for vertex B in both triangles and so the lighted color of vertex B will be the same for both triangles. As a result points close to B on the two triangles will be nearly the same color. The same is true for points near edge BC on the two triangles, they will be nearly the same color. However, vertex A and D will have different lighted colors. The appearance might be of a curved quadrilateral, something like a 2D diamond put on a curved surface.

Problem 3: [20 pts] Consider the three methods of specifying vertices shown below.

```
switch ( opt_method ) {

case VM_Individual: {   /// Use Individual Vertices
    glBegin(GL_TRIANGLE_STRIP);
    for ( int i=0; i<coords_size; i+=3 ) {
        glNormal3f(coords[i],coords[i+1],coords[i+2]);
        glVertex3f(coords[i],coords[i+1],coords[i+2]);  }
    glEnd();
    break;  }

case VM_Array: {        /// Use Vertex Arrays
    glNormalPointer(GL_FLOAT,0,coords);
    glEnableClientState(GL_NORMAL_ARRAY);
    glVertexPointer(3,GL_FLOAT,3*sizeof(float),coords);
    glEnableClientState(GL_VERTEX_ARRAY);
    glDrawArrays(GL_TRIANGLE_STRIP,0,coords_size/3);
    glDisableClientState(GL_NORMAL_ARRAY);
    glDisableClientState(GL_VERTEX_ARRAY);
    break;  }

case VM_Buffer: {       /// Use Buffer Objects
    glBindBuffer(GL_ARRAY_BUFFER,gpu_buffer);
    glVertexPointer(3,GL_FLOAT,3*sizeof(float),NULL);
    glEnableClientState(GL_VERTEX_ARRAY);
    glNormalPointer(GL_FLOAT,0,NULL);
    glEnableClientState(GL_NORMAL_ARRAY);
    glDrawArrays(GL_TRIANGLE_STRIP,0,coords_size/3);
    glBindBuffer(GL_ARRAY_BUFFER,0);
    glDisableClientState(GL_NORMAL_ARRAY);
    glDisableClientState(GL_VERTEX_ARRAY);
  break;  } }
```

(*a*) Why is the individual vertex method slower than the others?

☑ Reason that individual vertex method slower than the others.

Each time `glVertex3f` is called the GL driver code must package the coordinates and send them to the GPU. The time to transmit the data is dwarfed by the overhead of signaling the GPU that data is available, plus function call overhead on the CPU side and similar overheads on the GPU side.

In the `VM_Array` code, which uses vertex arrays, data transmission time is much longer (assuming the number of vertices is larger) but the amount of overhead for the `glDrawArrays`, which is called once, is roughly the same as the overhead for `glVertex3f` in the `VM_Individual` code, and so the overhead per vertex is much smaller.

(*b*) When used the right way the method using buffer objects is much faster than the others.

☑ Why are buffer objects faster than vertex arrays, when used the right way?

When using vertex arrays the entire vertex array is sent from CPU to GPU each time, even if it doesn't change. If vertices don't change, they can be placed in a buffer object and sent to the GPU once, as was presumably done for the `VM_Buffer` case. Then the time for `glDrawArrays` will not include data transfer time, and so will be much faster.

☑ Describe a situation in which the buffer object and vertex array method would have about the same performance.

If the vertices are different each time then the buffer object data would be sent to the GPU and could just be used once, there would be no advantage over using a vertex array. The performance would be similar.

Performance would also be similar if the number of vertices were small, because then the transmission time saved would be insignificant.

Problem 4: [20 pts] Answer each question below.

(*a*) OpenGL allows different material property colors for ambient, diffuse, emissive, and specular lighting. However only a few of these can be changed from vertex to vertex. Why?

☑ Why can't all material properties be changed each vertex?

Data associated with a vertex are called attributes. The OpenGL implementation must prepare storage for a full set of attributes for each vertex it is processing. This can be cumbersome if the number of attributes is too large and so there is a limit. If each vertex could have a full set of four colors for both front and back faces, that would consume $4 \times 2 \times 4 = 32$ floating-point numbers worth of attribute storage, too much. (This argument applies to fixed functionality, shaders can use their attributes however they like. The number of attributes is still limited though.)

(*b*) OpenGL lets you specify any transformation matrix for the projection, it doesn't have to be a frustum.

☑ Describe the appearance of a scene in which the projection matrix were identity. What parts of world space would be visible?

After the projection matrix is applied coordinates are in clip space. In clip space the view volume (the visible part of the scene) is fixed to a cube centered on the origin with $x \in [-1, 1]$, $y \in [-1, 1]$, and $z \in [0, 1]$. Anything outside of this box won't be visible. So the visible parts of the scene are those objects with ± 1 of the user's eye. There will be no perspective foreshortening, so equal-sized objects at distance 0 from the eye will be the same size as those at distance 1.

(*c*) Textures are provided or used to generate multiple MIPMAP levels. Explain what a MIPMAP level is and why it is necessary.

☑ What is a MIPMAP level?

It is a resolution level. Level 0 is the original image. Level 1 is an image with half the number of pixels along each axis (total number of pixels is $\frac{1}{4}$), and so on.

☑ Why is it necessary?

To avoid problems when a level-0 texel is much larger or smaller than a pixel. If the level-0 texel is much smaller (the texture image will appear tiny) we would like to use some kind of average of all the texels that cover the pixel (otherwise details, such as a picket fence, can disappear or not appear properly). The pixels in a higher MIPMAP level are averages of several pixels in a lower level. Consider a texture with a white picket fence and a black background and a situation where pixels are larger than texels. If we used just the level-0 image we might choose the texels that included only the black background between the white fence, so the fence would be invisible. If we used a prepared MIPMAP, say at level l, we would retrieve a texel that was a blend of background and the white fence, appearing gray.