Name Solution

GPU Programming
EE 4702-1
Midterm Examination

Monday, 26 October 2015 13:30–14:20 CDT

Problem 1		(21 pts)
Problem 2		(25 pts)
Problem 3		(12 pts)
Problem 4		(18 pts)
Problem 5		(12 pts)
Problem 6		(12 pts)
Exam Total		(100 pts)

Alias Lets go Mets!

Good Luck!

Problem 1: [21 pts] In the code below three coordinates are defined, `ph` (P_h, head), `pt` (P_t, tail), and `ps` (P_s, surface).

(a) Complete the code below so that it renders a cylinder with axis $\overrightarrow{P_h P_t}$, with P_h at the top (flat part) of the cylinder, P_t at the bottom (the other flat part), and with P_s on the surface. Render only the curved parts, not the flat parts. The cylinder surface should be approximated by `slices` flat sides.

☑ Show code before the loop, such as for `ax` and `ay`.

☑ Show code in the loop that emits vertex ☑ coordinates and ☑ normals.

```
pCoor ph( 1.5, 0, -3.2 );
pCoor pt( 0,   5, -5 );
pCoor ps( 9,   6, -9 );
const int slices = 100;
const float delta_theta = 2 * M_PI / slices;
pNorm axis = pt - ph;
// Some of the solution goes here.

// SOLUTION BELOW
pVect hs = ps - ph;
float hl = dot(axis,hs);    // Distance from ph to
pCoor pa = ph + hl * axis;

pNorm ax = ps - pa;
pVect ay = cross(ax,axis);
// SOLUTION ABOVE




glBegin(GL_TRIANGLE_STRIP);
glColor3ub( 256, 89, 16 );
for ( int i=0; i<=slices; i++ )
  {
    const float theta = i * delta_theta;
    pVect n = cos(theta) * ax + sin(theta) * ay;

    // SOLUTION BELOW
    glNormal3fv(n);      // n is the normal.
    pVect rn = r * n;    // Scale n by the cylinder radius.
    pCoor p1 = ph + rn;
    glVertex3fv(p1);
    pCoor p2 = pt + rn;
    // SOLUTION ABOVE
    glVertex3fv(p2);




  }
glEnd();
```

Problem 2: [25 pts] The code below, taken from the solution to Homework 2, renders the surface of a truss. Recall that the truss might be a part of a spinning gyroscope.

```
glEnableClientState(GL_VERTEX_ARRAY);

// glVertexPointer(3, GL_FLOAT, sizeof(pCoor), coord_buffer);
glVertexPointer(3, GL_FLOAT, sizeof(pCoor), NULL);  // SOLUTION

glEnableClientState(GL_NORMAL_ARRAY);

// glNormalPointer(GL_FLOAT, 0, norm_buffer);
glNormalPointer(GL_FLOAT, 0, NULL);                 // SOLUTION

glColor3fv(strip->color);
glDrawArrays(GL_TRIANGLE_STRIP, 0, elts);
glDisableClientState(GL_NORMAL_ARRAY);
glDisableClientState(GL_VERTEX_ARRAY);
```

(*a*) How much data is sent from the CPU to the GPU to render a frame based on the code above. Your answer should be in terms of the variables above.

☑ The amount of data sent per frame is ... ☑ Indicate unit.

The color is sent once. That's 3 floats. The rendering pass consists of `elts` vertices. Each vertex consists of a coordinate (specified by `glVertexPointer`) which is 3 floats and a normal which is also 3 floats. Note that the number of components of a vertex coordinate is explicitly given as 3 in the call to `glVertexPointer`, whereas for normals it must be 3. It's possible to render with 2-, 3-, or 4-component vertex coordinates.

The total amount of data is $3 + (3 + 3)e = 3 + 6e$ floats or $(3 + 6e)4\,\mathrm{B} = (12 + 24e)\,\mathrm{B}$, where e is the same as `elts`.

(*b*) How could someone who is not familiar with Homework 2 be reasonably sure that the code above uses client arrays and not buffer objects? What would be different *in the parts shown* if buffer objects were used?

☑ We would guess code doesn't use buffer objects because ....

If buffer objects were used the last argument to `glVertexPointer` and `glNormalPointer` would likely be a zero or NULL (meaning start at the beginning of the buffer object).

☑ Show changes to parts shown if buffer objects were used.

Changes appear above. The original lines of code are shown commented out.

(*c*) Explain why using buffer objects in the code above would not make a big difference, whereas using buffer objects to render the balls (used to show nodes in the truss) in the same simulation does make a big difference?

☑ Buffer objects would not make a big difference for truss surface because ...

Buffer objects help when the same data needs to be used multiple times. Since the truss can be in constant motion the coordinates would change every frame, and so a buffer object would need to be sent every frame. The amount of coordinate and normal data sent would be the same whether or not buffer objects were used.

☑ Buffer objects do make a big difference for balls (part of truss) because ...

For rendering the ball a buffer object is used that contains the surface coordinates of a sphere with radius one and with a center at the origin. To render a ball at a particular location, in a particular orientation (if a texture is used), and of a particular size a

transformation matrix is constructed that moves the ball specified in the buffer object to the desired location, size, and orientation. When rendering a ball only the transformation matrix needs to be sent, which is 16 floats, rather than a whole set of coordinates. (The transformation matrix is used to update the modelview matrix.)

Problem 2, continued: The Homework 2 code from the previous page is repeated below.

```
glEnableClientState(GL_VERTEX_ARRAY);
glVertexPointer(3, GL_FLOAT, sizeof(pCoor), coord_buffer);
glEnableClientState(GL_NORMAL_ARRAY);
glNormalPointer(GL_FLOAT, 0, norm_buffer);
glColor3fv(strip->color);
glDrawArrays(GL_TRIANGLE_STRIP, 0, elts);
glDisableClientState(GL_NORMAL_ARRAY);
glDisableClientState(GL_VERTEX_ARRAY);
```

(*d*) Re-write the code so that it does not use client arrays, but instead uses calls like `glVertex`. *Hint: Only a few lines of code needed.*

☑ Re-write code so that calls like `glVertex` is used.

The solution appears below. The call to `glDrawArrays` is replaced by a `glBegin`/`glEnd` pair that encloses a loop which sends one vertex at a time to OpenGL.

```
// SOLUTION
glBegin(GL_TRIANGLE_STRIP);
glColor3fv(strip->color);
for ( int i=0; i<elts; i++ )
  {
    glNormal3fv(&norm_buffer[i]);
    glVertex3fv(&coord_buffer[i]);
}
glEnd();
```

Problem 3: [12 pts] The code fragment below is from a demo-10 vertex shader.

```
void vs_main_basic() {
  gl_Position = gl_ModelViewProjectionMatrix * gl_Vertex;
  // Other code removed.
}
```

(*a*) Identity the variable that uses the storage types given below.

☑ Which variable is a vertex shader input?

`gl_Vertex`

.

☑ Which variable is a uniform variable?

`gl_ModelViewProjectionMatrix`

.

☑ Which variable is a vertex shader output?

`gl_Position`

.

(*b*) Two of these variables appear in the OpenGL shading language compatibility profile but not in the undeprecated OpenGL SL standard, meaning that they aren't truly needed but are being kept so as not to break old software. Identify the variable that *is* truly needed and explain why.

☑ The undeprecated, truly needed variable is: It is `gl_Position`

☑ It is truly needed because ...

Shader output `gl_Position` is set to the clip-space coordinate of the vertex. That's something the fixed functionality needs to know to do clipping and to do rasterization.

Problem 4: [18 pts] Answer each question below.

(*a*) The interface block below is used as a fragment shader input. Among other things the fragment shader does lighting calculations. Explain what the `flat` qualifier does and why it is needed.

```
in Data_GF
{
  vec3 var_normal_e;
  vec4 var_vertex_e;
  flat vec4 color;
};
```

☑ The `flat` qualifier ...

Tells the fixed functionality not to interpolate the values of `color` at each triangle vertex when determining a value for the fragment shader input. Instead, the value of `color` sent to the fragment shader will be the `color` output produced by the vertex shader or geometry shader (if there is one) of the last vertex in the primitive.

☑ The `flat` qualifier helps this particular shader program by ...

eliminating computation wasted on interpolation (assuming that the color at each vertex is the same). The interpolation makes sense if the vertex shader performed lighting calculations because each vertex of a primitive would likely have a different lighted color.

(*b*) Suppose the depth (z) test was turned off. How would that affect the rendered image?

☑ Rendered image with depth test turned off would appear ...

Fragments of primitives that were written later would cover fragments of primitives that were written earlier. (With the depth test turned on fragments closer to the user would cover fragments further from the user.)

(*c*) Describe what textures might be used for in a graphics program.

☑ An example of what a texture might be used for is ...

A texture could give a primitive the appearance of being stone or wood (by using an image of stone or wood). It can be used to apply any image to a primitive.

Problem 5: [12 pts] Answer the following questions.

(*a*) The code below renders two triangles. Add the minimum amount of code to make the first one blue and the second one orange (RGB:1,.35,.06). *Hint: This is an easy problem, don't overthink it. Note: The shade of orange is from the New York Mets logo.*

☑ Add minimum amount of code so that the first triangle is blue and the second is orange.

```
glBegin(GL_TRIANGLES);
glColor3f(0, 0, 1);          // SOLUTION -- Blue
glVertex3f( 10,  20,  0 );
glVertex3f( 15,  20,  0 );
glVertex3f( 10,   0,  0 );
glColor3f(1, .35, 06);       // SOLUTION -- Orange
glVertex3f( 30,  20,  0 );
glVertex3f( 30,   0,  0 );
glVertex3f( 25,   0,  0 );

glend();
```

(*b*) The code below renders two triangles. Add texture coordinates so that when the two triangles are put together they show the entire texture. The texture has already been set up, just provide the coordinates.

☑ Add OpenGL calls to provide texture coordinates.

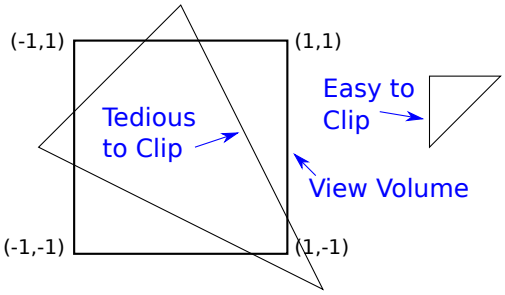
```
glBegin(GL_TRIANGLES);     //  SOLUTION BELOW
                             glTexCoord2f(0,1);
glVertex3f( 10,  20,  0 );   glTexCoord2f(1,1);
glVertex3f( 15,  20,  0 );   glTexCoord2f(0,0);
glVertex3f( 10,   0,  0 );
                             glTexCoord2f(1,1);
glVertex3f( 30,  20,  0 );   glTexCoord2f(1,0);
glVertex3f( 30,   0,  0 );   glTexCoord2f(0,0);
glVertex3f( 25,   0,  0 ); //  SOLUTION ABOVE

glend();
```

Problem 6: [12 pts] Clipping is performed between the output of the geometry shader and the input to the rasterizer.

(*a*) Provide a simple example of a triangle that can be completely clipped. The example should include the triangle's vertex coordinates in clip space.

☑ Easily completely clipable triangle coordinates:

A triangle with vertex clip-space coordinates $(2.3, .7, .5)$, $(3, .7, .5)$, $(2.3, 0, .5)$ is easy to clip because the x component of each vertex coordinate is strictly greater than 1 and therefore no part of it can be in the view volume. (The view volume is a cube with vertex coordinates $(\pm 1, \pm 1, \pm 1)$.) The Easy to Clip triangle appears in the illustration to the right. The illustration shows x and y coordinates only.



(*b*) Provide an example of a primitive for which clipping would be difficult or tedious.

☑ Illustration (not coordinates) of a triangle that is tedious to clip.

The triangle labeled Tedious to Clip in the illustration above.

☑ Briefly explain what makes it tedious.

Clipping results in a seven-sided polygon, which would have to be tessellated into triangles (assuming that the rendering pipeline can't handle polygons other than triangles, which is usually the case). The tedious part is finding the six intersections of the triangle edges with the view-volume faces, plus noticing that an edge of the view volume (point $(1, -1)$) contributes another vertex to the clipped primitive.

(*c*) Suppose due to some error the clipping stage doesn't clip anything. Explain why there would be no difference in the appearance of the rendered image.

☑ Image no different if clip stage doesn't clip because . . .

. . . fragments that are not in the window will be discarded somewhere after rasterization, probably between the rasterization and fragment shader.

(*d*) Again, suppose due to some error the clipping stage doesn't clip anything. Explain why performance would suffer.

☑ Performance suffers because . . .

Consider a triangle that is completely clipped. The clip stage only operates on the three vertices. Suppose that the triangle covers 100 pixels (but outside the window). If it's not clipped by the clip stage the rasterizer will have to generate 100 fragments. Even if they are discarded right after rasterization, work was performed to generate them.