Name _____

EE 7700-2

Take-Home Final Examination

Thursday, 8 May 2008 to Friday 9 May 2008

Problem 1	_____	(35 pts)
Problem 2	_____	(40 pts)
Problem 3	_____	(25 pts)
Exam Total	_____	(100 pts)

Alias _____

Good Luck!

Problem 1: [35 pts]The code below draws a figure (say, a person) multiple times to achieve the effect of a crowd (yes, of identical twins).

The figure is drawn at a different locations and at each location can be facing a different direction (but still standing straight up).

```
void world::figs_render
{
  // Number of figures: num_figs.
  // Number of vertices per figure: fig_size.

  glBindBuffer(GL_ARRAY_BUFFER,gpu_vertex_buffer);
  glVertexPointer(3,GL_FLOAT,0,NULL);
  glEnableClientState(GL_VERTEX_ARRAY);

  glBindBuffer(GL_ARRAY_BUFFER,gpu_normal_buffer);
  glNormalPointer(GL_FLOAT,0,NULL);
  glEnableClientState(GL_NORMAL_ARRAY);

  glBindBuffer(GL_ARRAY_BUFFER,gpu_texcoord_buffer);
  glTexCoordPointer(2,GL_FLOAT,0,NULL);
  glEnableClientState(GL_TEXTURE_COORD_ARRAY);

  glBindBuffer(GL_ARRAY_BUFFER,0);

  vs_fig->use(); // Turn on "fig" vertex shader.

  glMatrixMode(GL_MODELVIEW);
  glPushMatrix();

  for ( int i=0; i<num_figs; i++ )
    {
      glLoadMatrixf( figure_info[i].matrix );
      glVertexAttrib4f
        ( vsal_figure_info,
          figure_info[i].a, figure_info[i].b,
          figure_info[i].c, figure_info[i].d );
      glDrawArrays(GL_TRIANGLES,0,fig_size);
    }

  glPopMatrix();

  glDisableClientState(GL_VERTEX_ARRAY);
  glDisableClientState(GL_NORMAL_ARRAY);
  glDisableClientState(GL_TEXTURE_COORD_ARRAY);

}
```

Problem 1, continued:

(*a*) How much data is sent to the GPU for each call of `figs_render`? Your answer should be in terms of the variables used in the code.

(*b*) Consider a version of `figs_render` that does not use array buffers or other buffer objects but draws the same thing as the original `figs_render`. How much data would that version send for each call to `figs_render`?

Problem 1, continued: The code below is the shader code for `figs_render`.

```
// Shader Code
void
vs_ff_vertex(vec4 vtx)
{
  gl_Position = gl_ModelViewProjectionMatrix * vtx;
}

void
vs_lighting(vec4 vtx, vec4 color, vec3 normal)
{
  vec4 vertex_e = gl_ModelViewMatrix * vtx;
  vec3 norm_e = gl_NormalMatrix * normal;
  vec4 light_pos = gl_LightSource[0].position;
  vec4 v_vtx_light = light_pos - vertex_e;
  float phase_light = dot(norm_e, normalize(v_vtx_light).xyz);
  const vec3 ambient = gl_LightSource[0].ambient.rgb;
  const vec3 diffuse = gl_LightSource[0].diffuse.rgb;
  const float dist = length(v_vtx_light);
  const float distsq = dot(v_vtx_light,v_vtx_light);
  const float atten_inv =
    gl_LightSource[0].constantAttenuation +
    gl_LightSource[0].linearAttenuation * dist +
    gl_LightSource[0].quadraticAttenuation * distsq;
  vec4 new_color;
  new_color.rgb = color.rgb * ( phase_light * diffuse / atten_inv + ambient );
  new_color.a = color.a;
  gl_FrontColor = new_color;
}

void vs_main_vig()
{
  vs_ff_vertex(gl_Vertex);
  vs_lighting(gl_Vertex,gl_Color,gl_Normal);
}
```

Problem 1, continued:

Consider again the original code for `figs_render` and the shader code on the previous page. The call to `glLoadMatrixf` sets uniform (constant) variables. Since a vertex shader is being used it is possible to replace `glLoadMatrixf` with a call to `glVertexAttribXXX` or `glUniformXXX`.

(*c*) One advantage of using `glVertexAttribXXX` or `glUniformXXX` is that the amount of data sent can be reduced. Show how that would be done.

☐ Show what data would be sent in calls to `glVertexAttribXXX` or `glUniformXXX`.

☐ Show what changes would be necessary in the vertex shader code.

(*d*) The amount of data sent with `glVertexAttribXXX` and `glUniformXXX` should be the same. Why might there be a performance benefit using `glVertexAttribXXX`?

(*e*) There may be situations where `glVertexAttribXXX` can't be used but `glUniformXXX` can, so one must suffer the lower performance. It would not occur in the code above, but it might occur in more elaborate code. What is the reason?

Problem 2: [40 pts]Consider an OpenGL extension which added the capability to draw disks (filled circles). A sample use of the extension appears below:

```
glVertex3fv(center);
glTexCoord2(texcenter);
glNormal(normal_and_radius);
glDraw(GL_DISK);
```

Note that the normal provides two pieces of information, the orientation of the disk and its radius. (The normal is normal to the disk.)

(*a*) In the example above there is not enough information to apply the texture. (Assume that the texture, mode, and filtering have been correctly set.)

☐ What information is missing?

☐ Show a possible solution (by changing the way the extension works).

Problem 2, continued: Suppose the goal is to implement a driver for the disk extension for three systems, an unmodified GeForce 3 series, an unmodified GeForce 8800 series, and a modified GeForce 3 series. In all cases the goal is to fit the changes as naturally as possible into the existing software structure and to get good image quality with a small or moderate amount of effort.

(*b*) How might the OpenGL driver implement the disk for the GeForce 3 series.

☐ What data would be sent to the GPU?

☐ Which parts of the rendering pipeline (software) would change, and how would they change?

☐ How well would it work?

(*c*) Suppose one wanted to implement the disk on the GeForce 8800 series. An advantage here is that the changes would fit very naturally into the existing software structure (pipeline) imposed by OpenGL. Explain and show how it would be implemented.

Problem 2, continued:

(*d*) Suppose one could modify the primitive setup and interpolation hardware in the GeForce 3 series so that disk could be rendered. (In the previous parts hardware could not be modified.) Do so without adding too much hardware.

- [ ] How would the modified hardware be different?
- [ ] What data would the driver send to the GPU?
- [ ] Show which parts of the rendering pipeline would be changed and how they would change.
- [ ] How well would it work?

Problem 3: [25 pts]Answer the following mulitthreading questions.

(*a*) In a pre-final problem if a thread missed the cache another would take its place. Why is it unlikely that exactly this would happen in the GeForce 8800? What might happen instead? (Try reading the next part before answering this part).

(*b*) Consider a GeForce 8800-style multiprocessor that consisted of 8 stream processors connected into an SIMD array. The multiprocessor could host a maximum of 256 threads. The number of stages from `IF` to `WB` is 10, and so without cache misses there would be no stalls.

The execution of a fragment shader requires 50 instructions. For this problem lets conveniently assume that in every thread one instruction (say, number 41) will miss the texture cache and so will have to wait 500 cycles for the data.

At what rate would the multiprocessor process fragments? The answer should be in fragments per cycle and please show your work.