Name Solution

GPU Programming
EE 4702-1
Final Examination

Tuesday, 5 December 2017 12:30–14:30 CST

Problem 1		(15 pts)
Problem 2		(20 pts)
Problem 3		(30 pts)
Problem 4		(35 pts)
Exam Total		(100 pts)

Alias Wait.

Good Luck!

Problem 1: [15 pts] Appearing below is code based on Homework 5 (the tile cloud). First is code that launches the rendering passes, followed by the shaders. The shaders below were written to work with the rendering pass launched by `case 1`. In `case 3`, which is incomplete, the rendering pass input primitives are lines instead of triangles. Complete the `case 3` code and modify the shaders to work with your completed `case 3` code.

☑ Complete the `case 3` code. Use abbreviations such as `glV` (for `glVertex4f`), `glC`, and `glN`.

Solution appears below. Since a line has only two vertices, the five pieces of information, `color`, `pt_00`, `tact_pos`, `ax`, and `ay`, need to be sent using three attributes, `glColor`, `glNormal`, and `glVertex`, rather than the two attributes used in case 1.

```
case 1: {  pShader_Use use(s_hw05_tiles_1);
           glBegin(GL_TRIANGLES);
           for ( Tile* tile: tiles ) {  glColor4fv(tile->color);    glVertex4fv(tile->pt_00);
                                        glColor4fv(tile->tact_pos); glVertex4fv(tile->ax);
                                                                    glVertex4fv(tile->ay); }
           glEnd();  } break;
case 3: {  pShader_Use use(s_hw05_tiles_3);
           glBegin(GL_LINES);   //  <--- DON'T FORGET, LINES
           for ( Tile* tile: tiles ) {

             glColor4fv(tile->color);                         glVertex4fv(tile->pt_00);
             glColor4fv(tile->tact_pos); glNormal(tile->ax); glVertex4fv(tile->ay);

           }
           glEnd();  } break;
```

☑ Modify shader code (below) to work with `case 3` (above). Look at ☑ type of primitive (above), ☑ interface blocks, ☑ shader routines, and ☑ layout declarations.

Solution appears below.

```
out Data_to_GS { vec4 vertex_o;    vec4 color;       };

void vs_main_tiles_3() {  // Vertex shader routine.                 // CHANGE/ADD SOMETHING
  vertex_o = gl_Vertex;
  color = gl_Color;
  normal_o = gl_Normal;  // SOLUTION - Pass along third attribute.
}

// SOLUTION - Put normal_o in interface block.
in Data_to_GS { vec4 vertex_o;      vec4 color;   vec3 normal_o  } In[3];

// SOLUTION - Changed triangles to lines.
layout ( lines ) in;                                              // CHANGE/ADD ONE OF
layout ( triangle_strip, max_vertices = 4 ) out;                      // THE LAYOUT DECLARATIONS

void gs_main_tiles_3() {   // Geometry shader routine.              // CHANGE/ADD SEVERAL THINGS

  vec4 pt_00 = In[0].vertex_o;

  vec4 ax_o = vec4(In[1].normal_o,0);      // SOLUTION. Change to match new location.

  vec4 ay_o = vec4(In[1].vertex_o.xyz,0);  // SOLUTION. Change to new location.

  vec4 tact_pos = In[1].color;

  color = In[0].color;

  vec4 vtx_o[4];
  vtx_o[0] = pt_00;          vtx_o[1] = pt_00 + ax_o;
  vtx_o[2] = pt_00 + ay_o;   vtx_o[3] = pt_00 + ay_o + ax_o;
  // The code below does not need to be changed and so isn't shown.
```

Problem 2: [20 pts] Appearing below is the `render_tiles` routine from Homework 5.

(*a*) Estimate the amount of data sent from the CPU to the GPU for the rendering pass started by the code below. Use the following symbol: n, the number of tiles.

```
  case 1: {
      pShader_Use use(s_hw05_tiles_1);
      glUniform2i(1, opt_tryout1, opt_tryout2);
      glUniform1i(2, light_state_get());
      glUniform1f(3, world_time);

      glBegin(GL_TRIANGLES);
      for ( Tile* tile: tiles ) {
          glColor4fv(tile->color);
          glVertex4fv(tile->pt_00);
          glColor4fv(tile->tact_pos);
          glVertex4fv(tile->ax);
          glVertex4fv(tile->ay);
        }
      glEnd();
    }
    break;
```

✓ Amount of data, in bytes:

Integer and float variables are each 4 bytes. There are a total of four variables sent as uniforms, for a total of $4 \times 4 = 16\,\mathrm{B}$. Note that uniforms are just sent once. Each `glColor` and `glVertex` call sends $4 \times 4 = 16\,\mathrm{B}$ ignoring overhead. For each tile $5 \times 16 = 80\,\mathrm{B}$ are sent. The total amount of data is $16\,\mathrm{B} + 80n\,\mathrm{B}$.

(*b*) The vertex shader below is used with the `case 1` code above. In terms of n, how much data is read by this vertex shader for a rendering pass?

```
void vs_main_tiles_1() {
  vertex_o = gl_Vertex;
  color = gl_Color;
}
```

✓ Total data read by shader above for a rendering pass:

An invocation of the vertex shader reads $2 \times 16 = 32\,\mathrm{B}$. Three vertices are sent down the rendering pipeline for each tile, and so the total amount of data read is $n \times 3 \times 32 = 96n\,\mathrm{B}$.

✓ Explain why the amount of data read by the vertex shader might be different than the amount of data sent from CPU to GPU when executing the `case 1` code.

The same vertex shader code is used for every invocation and it always reads two items. We know that the color is not set for the third vertex of each triangle, but the vertex shader code does not check to see if it is third (and it might not be possible) and so it reads two items every time. The host code however, does avoid a `glColor` call for one out of three vertices and that might reduce the amount of data sent.

Problem 2, continued: The rendering pass below provides the same data to the shaders as the one in the previous part, but there are significant differences.

```
#define TO_BO(name,num,update)                                                    \
  glBindBuffer(GL_ARRAY_BUFFER,bos_tiles[num]);                                   \
  if ( update ) glBufferData                                                      \
    (GL_ARRAY_BUFFER, name.size()*sizeof(name[0]), name.data(), GL_STREAM_DRAW); \
  glBindBufferBase(GL_SHADER_STORAGE_BUFFER,num,bos_tiles[num]);

      pShader_Use use(s_hw05_tiles_2);
      glUniform2i(1, opt_tryout1, opt_tryout2);
      glUniform1i(2, light_state_get());
      glUniform1f(3, world_time);

      TO_BO(pt_00,     1, pt_00_data_stale);
      TO_BO(ax,        2, axes_data_stale);
      TO_BO(ay,        3, axes_data_stale);
      TO_BO(color,     4, color_data_stale);
      TO_BO(tact_pos,  5, tact_data_stale);
      pt_00_data_stale = axes_data_stale = color_data_stale = tact_data_stale = false;

      glDrawArrays(GL_POINTS,0,tiles.size());
```

(*c*) Explain why the code above would be more efficient than the `case 1` code when rendering the very first frame. Also explain why the code above might be more efficient on the second frame than on the first, depending on how the tiles are used. *Note: The original exam did not clearly state that the second comparison was of two executions of the code above, the first on the first frame and the second on some later frame.*

☑ More efficient than `case 1` for the very first frame because:

It is inefficient to use the `glVertex` and related calls because they handle a small amount of data. Sending over all of the data as large buffer objects avoids this inefficiency.

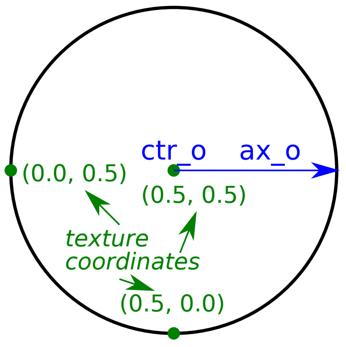
☑ Code above more efficient rendering second frame than first frame because . . .

Only data that is changed needs to be re-sent from the CPU to the GPU.

☑ This efficiency can be assumed by use of variable such as . . . because . . .

Because of the variable names containing `stale`. That implies that only some data needs to be re-sent, the other data on the GPU is still fresh and so doesn't need to be replaced.

Problem 3: [30 pts] The code in this problem is similar to the Homework 5 tile code except that it will be used to render discs.



(*a*) Complete the geometry shader below so that it emits primitives for a disc (a filled-in circle) writing the geometry shader outputs shown. The provided shader code reads the disc center, `ctr_o`, disc normal, `nz_o`, and a vector to a point on the circumference, `ax_o` (see diagram), all in object space. The shader output includes `tcoord`, a texture coordinate. Assign this consistent with the texture coordinates shown on the diagram. Assume that the shader output primitive, a triangle fan, will work correctly even though it is not included in OpenGL Shading Language 4.5. A triangle fan makes the solution simpler. *Note: This triangle fan discussion was intentionally omitted from the original exam.*

✓ Emit primitives for the disc, make sure it's filled in.

✓ Don't forget to: ✓ Set `max_vertices`, ✓ set `normal_e`, ✓ set `tcoord`, and of course ✓ set `gl_Position`.

Solution appears on the next page.

Solution appears below.

```
out Data_to_FS { flat vec3 normal_e;  vec3 vertex_e;  vec2 tcoord; };
layout ( points ) in;

layout ( triangle_fan, max_vertices = slices + 1       ) out;  // SOLUTION

void gs_main_disc() {
  int vertex_id = In[0].vertex_id;
  vec4 ctr_o = ctrs[vertex_id];
  vec3 ax_o = axs[vertex_id].xyz;
  vec3 nz_o = nzs[vertex_id].xyz;

  const int slices = 10;
  const float pi = 3.1415926536;
  const float delta_theta = 2 * pi / slices;
// Abbrevs: glmvp, gl_ModelViewProjectionMatrix; glmv, gl_ModelViewMatrix; gln, gl_NormalMatrix

  // SOLUTION

  vec3 ay_o = cross(normalize(nz_o),ax_o);  // Compute y vector.


  // Compute the normal. Used for the entire disc.
  normal_e = gl_NormalMatrix * nz_o;

   // SOLUTION - Compute and emit attributes for the center coordinate.
  tcoord = vec2(0.5,0.5);
  gl_Position = gl_ModelViewProjectionMatrix * ctr_o;
  vertex_e = gl_ModelViewMatrix * ctr_o;
  EmitVertex();

  for ( int i=0; i<=slices; i++ )
    {
      float theta = i * delta_theta;    float costh = cos(theta), sinth = sin(theta);

      // SOLUTION - Compute attributes for point on circle and emit.
      vec4 pt_o = ctr_o + vec4( costh * ax_o + sinth * ay_o, 0 );
      tcoord = vec2( 0.5 + 0.5 * costh, 0.5 - 0.5 * sinth );
      vertex_e = gl_ModelViewMatrix * pt_o;
      gl_Position = gl_ModelViewProjectionMatrix * pt_o;
      EmitVertex();

    }
}
```

Problem 3, continued:

(*b*) Appearing below is the fragment shader for the code above. If variable `nevermind` were `true` the fragment shader would not write a fragment, but it's set to `false` in the code. (The `discard` keyword returns from the fragment shader without writing a fragment.)

The primitives emitted by the geometry shader (if solved correctly) will render a 10-sided polygon, which is not exactly a disc (circle). Modify the code so that it emits a perfect disc based on **the largest circle that can fit inside the polygon**. (For partial credit, a circle of radius 0.4 in texture coordinate units.) Use texture coordinates to determine whether a fragment is in the circle.

☑ Assign `radius` the correct value in terms of `slices`.

☑ Set `nevermind` so that a fragment is discarded if it's outside a radius-`radius` circle based on `tcoord`.

Solution appears below. Using texture coordinates makes it very easy to find the distance from the fragment to the disc center.

```
void fs_main_disc() {
  const int slices = 10;
  const float pi = 3.1415926535;
  const float delta_theta = 2 * pi / slices;

  // SOLUTION -- Set radius based on texture coordinates in diagram.
  float radius = 0.5 * cos(pi/slices);
  vec2 disc_center_t = vec2(0.5,0.5);

  // SOLUTION -- Find distance to center using texture coordinates.
  const bool nevermind = distance(tcoord,disc_center_t) > radius;


  vec4 texel = texture(tex_unit_0,tcoord);   // NO NEED TO CHANGE THIS CODE.
  vec4 color = colors[vertex_id];
  gl_FragColor = texel * generic_lighting(vertex_e, color, normal_e);
  gl_FragDepth = gl_FragCoord.z;
}
```

(*c*) The inputs to fragment shader `fs_main_disc` appear below. Explain the implications of removing the `flat` qualifier as described below.

```
in Data_to_FS  {  flat vec3 normal_e;   flat int vertex_id;
                  vec3 vertex_e;        vec2 tcoord;              };
```

☑ Explain impact on ☑ correctness and ☑ efficiency if `flat` were removed from `normal_e` and `vertex_id`.

Removing `flat` from `normal_e` would be wasteful but the code would still work correctly. It would be wasteful because the normal will be the same on every point on the disc and so the effort to interpolate it (between values at the three vertices) is wasted. On the other hand, `vertex_id` is an integer and OpenGL Shading Language won't interpolate integers, it's an error to even try

☑ Explain impact on ☑ correctness and ☑ efficiency if `flat` were added to `vertex_e` and `tcoord`.

Adding `flat` would reduce the amount of computation, however that doesn't mean anything if the results are wrong. Adding `flat` to `vertex_e` would affect the lighted color of the fragment, which would be noticeable if the disc were near a light source. Adding `flat` to `tcoord` would result in the texture not being applied and if the part-b code were used, would result in a polygon rather than a disc.

Problem 4: [35 pts] Answer each question below.

(a) Explain how the depth (z-buffer) test is used. Provide a diagram of an example in which sorting primitives by eye distance makes the depth test unnecessary, and another example diagram in which even with sorting a depth test is necessary for proper rendering.

✓ Explain how depth test used.

The depth test is applied just before a fragment is written to the frame buffer. The fragment's z value is compared to the z value of the corresponding pixel. Usually, if the fragment's z value is closer to the eye it gets written, otherwise it is abandoned.

☐ Diagram illustrating example in which sorting makes depth test unnecessary. *Hint: Example can have two primitives.*

☐ Diagram illustrating Example in which depth test necessary even with sorting. *Hint: Example can have three primitives.*

(b) Appearing below are sample uses of two procedures related to the stencil buffer. Explain what each one does in general (not necessarily in the example).

```
glStencilFunc(GL_EQUAL,4,-1);
glStencilOp(GL_REPLACE,GL_KEEP,GL_KEEP);
```

✓ The `glStencilFunc` procedure is used to . . ..

. . . specify the type of stencil test to perform. The value in the stencil buffer location corresponding to a fragment is compared to the reference value (the second parameter, 4 above). The type of comparison is specified by the first argument. In the example above the test passes if the stencil value equals 4.

✓ The `glStencilOp` procedure is used to . . ..

. . . specify under what conditions to update the stencil buffer and how to update it. The argument GL_REPLACE indicates that a new value should be written to the stencil buffer. The value to be written is the second argument to the most recent `glStencilFunc` call. Argument GL_KEEP indicates that the stencil buffer should not be changed. The first argument is used if the stencil test fails. The second if the depth test fails, and the third if the depth test passes.

(*c*) In general, why doesn't it make sense to access a texture in a vertex shader?

☑ Bad idea to access a texture in a vertex shader because . . .

Accessing a texture returns a texel. In typical use a texel is associated with a frame buffer location (a pixel). Each fragment shader invocation is associated with a frame buffer location, but a vertex shader is not. A vertex shader invocation typically is associated with a coordinate that maps to a frame buffer location. But that vertex usually is part of a primitive which covers many locations, and so the texel would only apply to one of those locations.

(*d*) How is eye space defined? Describe the transformations that need to be applied to map from object space to eye space. Illustrate your answer using a sample scene.

☑ Defining features of eye space are . . ..

In eye space the viewer's eye is at the origin and facing in the $-z$ direction.

☑ In the following scene to transform from object to eye space the modelview matrix is used, which . . .

. . . the product of a rotation and translation. The translation moves the eye to the origin, and the rotation moves the users monitor to face the $-z$ direction.

(*e*) An NVIDIA GPU has 10 SMs. Consider a kernel which evenly divides work among its threads, the usual assumption made in class. Further assume that there is a large amount of work. Let $t(G)$ denote the execution time when the kernel is launched with G blocks. In all cases the block size is 1024 threads.

Let $a = t(10)$, the time when launched with 10 blocks on the 10-SM GPU.

☑ Find an expression for $t(5)$ in terms of a.

$t(5) = 2a$. It takes twice as long because each block has twice as much work to do.

☑ Find an expression for $t(15)$ in terms of a.

$t(15) = 2\frac{10}{15}a$. With 15 blocks each block performs $\frac{10}{15}$ times as much work as for the $a = t(10)$ case. Since there are 10 SMs some SMs will have to compute two blocks. Since the block sizes are large the times for each of the two blocks are added. (If the block sizes were small, say 32 threads, then the time for two blocks on an SM would be about the same as the time for one.)

☑ Find an expression for $t(20)$ in terms of a.

$t(20) = 2\frac{10}{20}a = a$. The reasoning is the same as the previous case, but this time *all* SMs compute two blocks.

(*f*) Draw a sketch showing the shadow volume corresponding to a triangle. Include the triangle, the light source, some object in the shadow and some object seen through the shadow.

☐ Show: ☐ the light, ☐ triangle, ☐ shadow volume for the triangle, ☐ shadowed object, ☐ object seen through shadow.