Name Solution

GPU Programming
LSU EE 4702-1
Final Examination

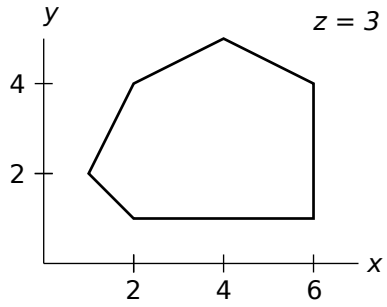
Tuesday, 4 December 2018 12:30–14:30 CST

Problem 1		(15 pts)
Problem 2		(30 pts)
Problem 3		(15 pts)
Problem 4		(40 pts)
Exam Total		(100 pts)

Alias em dash

Good Luck!

Problem 1: [15 pts] Appearing below is a geometric figure.



(*a*) Complete the individual-triangle rendering pass below so that it renders the figure with all triangles facing in the positive z direction and without overlapping triangles. Use the provided abbreviation `glV`. *Note: The part about facing the $+z$ direction was not in the original exam.*

☑ Complete code to render shape using individual triangles.

```
#define glV glVertex3f
 glBegin(GL_TRIANGLES);

 glV(2,1,3); glV(2,4,3); glV(1,2,3);  // SOLUTION
 glV(6,4,3); glV(4,5,3); glV(2,4,3);
 glV(2,1,3); glV(6,4,3); glV(2,4,3);
 glV(2,1,3); glV(6,1,3); glV(6,4,3);

 glEnd();
```

(*b*) Complete the triangle-strip rendering pass below so that it renders the figure.

☑ Complete code to render shape using a triangle strip.

```
#define glV glVertex3f
 glBegin(GL_TRIANGLE_STRIP);

 glV(2,1,3); glV(1,2,3);  // SOLUTION
 glV(6,1,3); glV(2,4,3);
 glV(6,4,3); glV(4,5,3);

 glEnd();
```

Problem 2: [30 pts] Appearing below is shortened host and shader code based on Homework 3, in which text was drawn on the triangular spiral. One drawback of this code is that it uses old-fashioned, deprecated, inefficient `glVertex` calls. On the following pages are routines that will implement a more efficient version of this code in which data such as `ctr` are placed in buffer objects and a rendering pass is performed *for the entire chain*, not just one triangular spiral.

```
for ( int i=2; i<chain_length; i++ ) {  // Host Code
    pCoor p0 = balls[i-2].position, p1 = balls[i-1].position, p2 = balls[i].position;
    pCoor ctr = (p0+p1+p2)/3; // Compute location of triangle center.
    // --- CODE REMOVED FOR BREVITY ---
    pCoor pprev = ctr;
    float delta_a = 0.6 / opt_n_segs;

    if ( opt_shader == SO_HW03 ) {
        glBegin(GL_TRIANGLE_STRIP);  // Render spiral using 1 triangle strip.

        for ( int j=0; j<opt_n_segs; j++ ) {
            pVect v = va[j%3];                // Vector from center to fold.
            pCoor p = ctr + j * delta_a * v;  // Coordinate of fold.
            pNorm n = cross( p - pprev, vz ); // Normal of segment.
            float tex_x = total_len_compute(j,delta_a,distv) * tex_scale;

            glNormal3fv(n);
            glTexCoord2f(tex_x,0);  glVertex3fv(p + vz);
            glTexCoord2f(tex_x,1);  glVertex3fv(p - vz);
            pprev = p;
          }
        glEnd();
      } }
```

```
void vs_main_hw03() { // Vertex Shader Code
  gl_Position  <= O = gl_ModelViewProjectionMatrix  <= U * gl_Vertex <= I;
  vertex_e = gl_ModelViewMatrix  <= U * gl_Vertex <= I;
  normal_e = normalize(gl_NormalMatrix  <= U * gl_Normal <= I);
  tex_coord = gl_MultiTexCoord0.xy <= I;   }
```

(*a*) But first let c denote the value of `chain_length` and n denote the value of `opt_n_segs`. Determine the amount of data, in bytes, sent from the CPU to the GPU for the code shown above. (Do not consider uniforms and other hidden code.)

☑ Amount of data sent from CPU to GPU in terms of c and n, in bytes.

The amount of data sent is $\boxed{(c-2)n(3+2(2+3))4\,\mathrm{B}}$, where the $3+$ term gives the number of floats sent by `glNormal3fv`, and the $2(2+3)$ term gives the number of floats sent by the calls to `glTexCoord2f` and `glVertex3fv`, the $4\,\mathrm{B}$ factor accounts for the size of a float, and $(c-2)n$ is the number of executions of the inner loop body.

(*b*) In the vertex shader above, label variables with the appropriate letter as requested below.

☑ Label uniform variables with a **U**, ☑ shader input variables with an **I**, and ☑ fixed-function shader outputs with an **O**.

Solution appears above, look for <= These.

Problem 2, continued: Appearing below is the improved triangular spiral host code and shaders. The host code prepares four buffer objects and then starts an instanced rendering pass with line strips as the input primitive. The rendering pass uses a vertex shader and a geometry shader to complete the primitives. (The fragment shader is not a part of this problem.) The only inputs to the vertex shader are the vertex and instance IDs, which should be used to retrieve or compute information about the spiral segments.

```
// Host Code -- DO NOT MAKE OR ASSUME MODIFICATIONS TO THIS CODE.
      for ( int i=2; i<chain_length; i++ ) {
          // Put ball structures and coordinates into convenient variables.
          pCoor p0 = balls[i-2].position, p1 = balls[i-1].position, p2 = balls[i].position;
          pCoor ctr = (p0+p1+p2)/3;   ctr_a.push_back(ctr);

          // -- CODE PREPARING va_a, vz_a, and distv_a NOT SHOWN --
        }
      glUniform1i(5, opt_n_segs);
      TO_BUFFER_OBJECT(ctr_a,1);      TO_BUFFER_OBJECT(va_a,2);
      TO_BUFFER_OBJECT(distv_a,3);    TO_BUFFER_OBJECT(vz_a,4);
      glBindBuffer(GL_ARRAY_BUFFER,0);

      glDrawArraysInstanced(GL_LINE_STRIP, 0, opt_n_segs, ctr_a.size() );
```

(*c*) Modify the vertex and geometry shaders on the following pages to efficiently render the triangle spirals, make any needed changes to the interface blocks, but **do not modify** or assume modifications to the host code (above). For your convenience each shader contains a copy of the host code. Cross out or modify that code as needed. Use the handy abbreviations at the top of the page.

☑ Update the interface blocks as needed.

```
#ifdef _VERTEX_SHADER_
out Data {  // Out of Vertex Shader to Geometry Shader
 int ins_id, vtx_id;
 //  ☑   Add any needed declarations here.

 // SOLUTION
 vec4 p_e[2], p_c[2]; // Send pre-computed eye- and clip-space coords.
 float tex_x;         // Position of texture at fold.

};
#endif
#ifdef _GEOMETRY_SHADER_
in Data {  // In to Geometry Shader from Vertex Shader
 int ins_id, vtx_id;
 //  ☑   Add any needed declarations here.

 // SOLUTION
 vec4 p_e[2], p_c[2]; // Send pre-computed eye- and clip-space coords.
 float tex_x;         // Position of texture at fold.

} In[2];

// Out of Geometry Shader to Fragment Shader -- No changes needed
out Data {   vec3 normal_e;   vec4 vertex_e;   vec2 tex_coord;  };
#endif
```

Problem 2, continued: Vertex shader code on this page.

☑ Cross out unneeded code. ☑ Cross out data type for shader outputs. ☑ Avoid redundant computation.

☑ Update the interface blocks as needed.

```
const mat4 mv = gl_ModelViewMatrix, mvp = gl_ModelViewProjectionMatrix;
const mat3 nm = gl_NormalMatrix;

void vs_main_lines() {  // Vertex Shader Main Routine
  int ins_id = gl_InstanceID, vtx_id = gl_VertexID;

  int j = gl_VertexID;
  float delta_a = 0.6 / opt_n_segs;
  vec4 distv = distv_a[ins_id];
  vec3 vz = vz_a[ins_id].xyz;
  float tex_scale = 0.2;

  // SOLUTION
  vec3 v = va_a[ins_id][j%3].xyz;
  vec3 p = ctr_a[ins_id].xyz + j * delta_a * v;

  for ( int i=0; i<2; i++ ) // Compute +vz and -vz coords.
    {
      vec4 po = vec4( p + ( i == 0 ? vz : -vz ), 1 );
      p_e[i] = mv * po;
      p_c[i] = mvp * po;
    }

  tex_x = total_len_compute(j,delta_a,distv) * tex_scale;

}
```

Solution appears above. The vertex shader is computing the eye- and clip-space coordinates, saving the geometry shader the need to do so. Note that because the geometry shader primitive is a line strip, each vertex is used for two primitives. So, by doing coordinate space conversions in the vertex shader rather than the geometry shader cuts in half the number of coordinate space conversions needed. Whether this improves performance is a more complex question. However a requirement for this problem was to avoid redundant computation.

Problem 2, continued: Geometry shader code on this page.

☑ Cross out unneeded code. ☑ Cross out data type for shader outputs. ☑ Avoid redundant computation.

☑ Update the interface blocks as needed.

```
layout ( lines ) in;
layout ( triangle_strip, max_vertices = 4 ) out;

const mat4 mv = gl_ModelViewMatrix, mvp = gl_ModelViewProjectionMatrix;
const mat3 nm = gl_NormalMatrix;

void gs_main_lines() {  // Geometry Shader Main Routine

  // SOLUTION
  int ins_id = In[0].ins_id;

  vec3 vz_e = In[0].p_e[0].xyz - In[0].p_e[1].xyz;
  normal_e = normalize(cross( In[1].p_e[0].xyz-In[0].p_e[0].xyz, vz_e ));

  for ( int i=0; i<2; i++ )
    {
      for ( int j=0; j<2; j++ )
        {
          tex_coord = vec2(In[i].tex_x,j);
          gl_Position = In[i].p_c[j];
          vertex_e = In[i].p_e[j];
          EmitVertex();
        }
      }
  EndPrimitive();
}
```

Solution appears above. Notice that the geometry shader input consists of two vertices, `In[0]` and `In[1]`. The `In[0]` vertex corresponds to the vertex used to compute `pprev` on the host code. Notice how the loop nest avoids the need to duplicate code.

Problem 3: [15 pts] Answer each CUDA question below.

(*a*) Both CUDA kernels below do the same thing, but one will execute much less efficiently. Explain why in terms of the minimum request size.

```
__global__ void kmain_simple(float4 *d_in, float *d_out) {
  const int tid = threadIdx.x + blockIdx.x * blockDim.x;
  const int elt_per_thread = ( d_app.array_size + d_app.num_threads - 1 ) / d_app.num_threads;
  const int start = elt_per_thread * tid;
  const int stop = start + elt_per_thread;

  for ( int h=start; h<stop; h++ ) {
      float4 p = d_in[h];
      float sos = p.x * p.x + p.y * p.y + p.z * p.z + p.w * p.w;
      d_out[h] = sos;
    }
}

__global__ void kmain_efficient(float4 *d_in, float *d_out) {
  const int tid = threadIdx.x + blockIdx.x * blockDim.x;
  for ( int h=tid; h<d_app.array_size; h += d_app.num_threads ) {
      float4 p = d_in[h];
      float sos = p.x * p.x + p.y * p.y + p.z * p.z + p.w * p.w;
      d_out[h] = sos;
    }
}
```

☑ Problem with inefficient kernel due to request size.

Each access `d_in[h]` is to 16 bytes of data. Threads in a warp make such an access together. Assume that `elt_per_thread = 200` and consider two consecutive threads in such an access with `tid=10` and `tid=11`. The initial value of `h` for these threads will be 2000 and 2200. That means the starting address of the data access by these two threads differs by $200 \times 16 = 3200$ bytes and so a request will have to be generated for each thread. Since the minimum request size for current NVIDIA devices is 32 B only half of the request will be used.

☑ What is the maximum request size that will avoid this inefficiency? ☑ Explain.

For the access to `d_in[h]` the maximum request size is 16 B, for the write to `d_out[h]` the maximum is 4 B, so overall the maximum is 4 B. This guarantees that every byte of every request will be used in the simple kernel above.

(*b*) A CUDA kernel is to run on a GPU with 8 SMs (MPs). Configuration A consists of 4 blocks of 32 threads each. Configuration B consists of 8 blocks of 16 threads each. Neither is very good. Explain how each one underutilizes the hardware on current NVIDIA GPUs.

☑ Configuration A underutilizes hardware by . . .

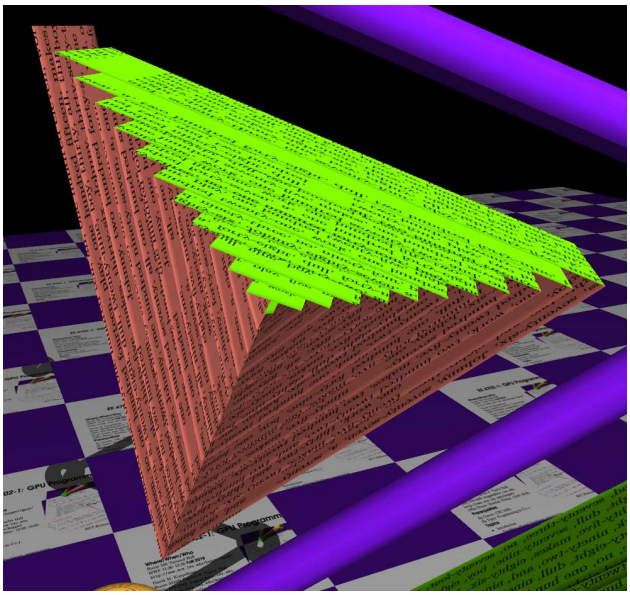
. . . leaving four SMs idle. Since a block can't run on more than on SM at a time with 4 blocks and 8 SMs, at least 4 will be idle.

☑ Configuration B underutilizes hardware by . . .

. . . leaving half of the warp unoccupied. Threads in an NVidia GPU are managed in units called warps, and the warp size up to now is 32. With less than 32 threads some functional units will go unused. The truth is even with 32 threads hardware will go unused since for Kepler (CC 3.x) up to Volta/Turing (CC 7.x) at least two and more often four warps per SM are needed to use all functional units (and often many more than that).

Problem 4: [40 pts] Answer each question below.

(*a*) The screenshot below is the Homework 3 triangular spiral with 37 segments per spiral. Imagine a spiral with even more segments, say 1000.



With a large number of segments there will usually be a large computational load on both the vertex and fragment shaders. In one of these shaders the computational load can be considered wasted, depending on the eye location, **even when the spiral is visible**. In which shader is computation wasted, and why.

☑ Which shader wastes computation? ☑ Why?

The fragment shader wastes computation because in a typical view, such as the one above, most fragments are discarded (since they are blocked from view by segments closer to the viewer).

Sketch two views in which the spiral is visible. In one the computational load is high and mostly wasted. In the other the computational load is lower and not wasted.

☑ View with lower load and little waste. ☑ View with high load and waste.

There will be little waste when spiral normal is aligned with the z axis. In such a view all or most fragments are visible (and there won't be many of them). There will be lots of waste when the spiral normal is orthogonal to z. In that case only one or two segments are visible, fragments for the rest of the segments are discarded.

NEED TO ADD SKETCH. If you are preparing for the 2019 final exam, ask for the sketch to be put in this solution. (Not the one you may have printed, the one on the Web site.)

(*b*) The OpenGL call `glColor` is used to specify a color, say purple `glColor3f(1,0,1);`. In typical use is that the color that will be written to the frame buffer? Explain.

☑ Are the arguments to `glColor` the exact color to be written to the frame buffer? ☑ Explain.

No. In typical use the value sent by calling `glColor` is used by the vertex shader to compute a lighted color. The fragment shader mixes the lighted color, actually a value interpolated from the lighted colors provided by all of the vertices making the primitive, with texture values. The result of that will be written to the frame buffer (if tests, such as the z test, pass).

(*c*) Describe the difference between the `flat`, `noperspective`, and `smooth` interpolation qualifiers.

☑ The `flat` qualifier's feature, ☑ the `noperspective` qualifier's, feature, ☑ the `smooth` qualifier's, feature.

Short Answer: `flat`, value is taken from the provoking vertex; `noperspective`, value is linearly interpolated between the vertices based on coordinates in clip space; `smooth`, value is linearly interpolated between the vertices based on coordinates in object space.

Explanation: The values provided to the fragment shader from variables declared using the `flat` qualifier will be the values written by the provoking (last) vertex of the primitive. For example, suppose color were declared `flat vec4 color;`, and the colors sent for the three vertices of a triangle were red, green, and blue. The triangle would be blue.

For both `noperspective` and `smooth` the value is interpolated linearly from the values provided for each vertex. With `smooth` the interpolation is based on object space coordinates, which is realistic but computationally expensive, while with `noperspective` the interpolation uses window space coordinates, which is easier to do but won't look right. Consider a line primitive with an attribute `tex_x`. Suppose the value for first vertex is `tex_x=0` and for the second vertexi is `tex_x=1`. For the declaration `noperspective float tex_x`, a value of 0.5 would be find at the point halfway between the two vertices based on a distance measured from the monitor. (Don't do this at home, you might scratch your monitor.) For the declaration `smooth float tex_x` the 0.5 value is halfway between the two vertices measured object space.

(*d*) Consider a rendering pass using triangles as the primitive. OpenGL (compatibily profile) allows one to specify a normal for each triangle vertex, but as we all know a triangle, geometrically, has just one normal. Why would one specify different normals for each vertex? Explain how such normals are chosen.

☑ Why might one choose different normals for each vertex of a triangle?

☑ Give an example and describe how those normals are chosen.

Because the triangle is approximating a surface, so by providing the normal of that surface, rather than that of the triangle, lighting will be more realistic. For example, if the triangles are approximating a cylinder use the cylinder surface normal.

(*e*) Describe what the inputs to the rasterization stage are, what the rasterization stage does, and what its outputs are.

☑ Rasterization stage input, ☑ rasterization stage job, ☑ rasterization stage output:

The input to rasterization is a primitive, which is the set of vertices corresponding to a primitive (say, three vertices for a triangle). The rasterization stage emits one fragment for each pixel that the primitive covers. The fragment consists of the attributes interpolated based on the fragment's position within the primitive.

(*f*) The true-sphere shader used in class rendered spheres perfectly. Would it make sense to use a similar approach to write a true-cube shader that can perfectly render a cube?

☑ Does a true cube shader make sense? ☑ Explain.

It does not make sense because a cube can be perfectly rendered using triangles. That was not possible for a sphere because, of course, the surface of a sphere is round and the flat surface of a triangle would only approximate the sphere surface.

(*g*) The unlabeled diagram below shows how shadow volumes can be used to render shadows. On the diagram show the location of the eye and light source, fragment(s) found to be in the shade, and fragment(s) found to be illuminated.

☑ Show: ☑ eye, ☑ light source, ☑ shaded fragment(s), ☑ illuminated fragment(s).

Solution appears below.