Name ______________________________

GPU Programming
EE 4702-1
Final Examination

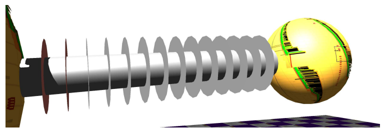
Monday, 5 December 2016   17:30–19:30 CST

Problem 1	________	(20 pts)
Problem 2	________	(20 pts)
Problem 3	________	(15 pts)
Problem 4	________	(20 pts)
Problem 5	________	(25 pts)
Exam Total	________	(100 pts)

Alias ______________________________

Good Luck!

Problem 1: [20 pts] The geometry shader code below is based on the solution to Homework 6 Problem 2 in which a curved link is rendered by having the vertex shader compute points and vectors related to a curve and by having the geometry shader, whose input primitive type is a line strip, emitting a cylinder between the locations described by its two input vertices. Input `In[].t` is the position along the curve, with $0 \leq t \leq 1$.



Add code to the shader so that it draws rings around the cylinder. The inner radius should be the same as the cylinder, the outer radius should be twice the cylinder radius. Do not emit a ring at the link endpoints (the parts that touch the spheres). See the illustration to the upper right.

☐ Complete the code so that it draws the rings. ☐ Can use suggested and other reasonable abbreviations.

☐ Set `normal_e` properly don't forget to set ☐ `vertex_e` and ☐ `gl_Position`.

☐ The code should be reasonably efficient. ☐ Don't overlap primitives.

```
void gs_main_2() {
  const float rad = tex_rad[In[1].iid].z,  sides = sides_rad.x;
  for ( int j=0; j<=sides; j++ ) {
      const float theta = j * ( 2 * M_PI / sides );
      vec3 vect0 = cos(theta) * In[0].norm_e + sin(theta) * In[0].binorm_e;
      vec3 vect1 = cos(theta) * In[1].norm_e + sin(theta) * In[1].binorm_e;

      normal_e = vect1;    vertex_e = In[1].ctr_e + vec4( rad * vect1, 0);
      gl_Position = gl_ProjectionMatrix * vertex_e;  // SUGGESTED ABBREV: glP = glPM * v_e;
      EmitVertex();
      normal_e = vect0;    vertex_e = In[0].ctr_e + vec4(rad * vect0,0);
      gl_Position = gl_ProjectionMatrix * vertex_e;
      EmitVertex();   }
  EndPrimitive();
  float t0 = In[0].t,  t1 = In[1].t;   // Use if needed.




  for ( int j=0; j<=sides; j++ ) {
      const float theta = j * ( 2 * M_PI / sides );
      vec3 v0 = cos(theta) * In[0].norm_e + sin(theta) * In[0].binorm_e; // Use if needed.
      vec3 v1 = cos(theta) * In[1].norm_e + sin(theta) * In[1].binorm_e; // Use if needed.
      vec4 c0 = In[0].ctr_e,  c1 = In[1].ctr_e;   // Use if needed.








    }




}
```

Problem 2: [20 pts] Appearing below is a typical vertex shader.

```
in vec2 gl_MultiTexCoord0;  // Declare size.
void vs_main() {
  gl_Position = gl_ModelViewProjectionMatrix * gl_Vertex;
  vertex_e = gl_ModelViewMatrix * gl_Vertex;
  normal_e = normalize(gl_NormalMatrix * gl_Normal);
  gl_TexCoord[0] = gl_MultiTexCoord0;
  color = gl_Color;
}
```

(*a*) Suppose that the shader above was used in a rendering pass of v vertices. No buffer objects were used and all shader inputs were sourced from client arrays. Compute the amount of data sent from the CPU to the GPU for the rendering pass.

☐ CPU to GPU data for a rendering pass of v vertices counting shader inputs is:

☐ CPU to GPU data for a rendering pass of v vertices counting uniform variables is:

Problem 2, continued: The vertex shader below is used for an instanced draw of spheres. The vertex shader inputs are sourced from buffer objects and other buffer objects are accessed directly using the instance ID.

```
layout ( binding = 1 ) buffer sr { mat4 sphere_transform[]; };
layout ( binding = 3 ) buffer sc { vec4 sphere_color[]; };
void vs_main_instances_sphere() {
  mat4 transform = sphere_transform[gl_InstanceID];
  vec3 ctr = transform[3].xyz;         // Extract coordinate of sphere center.
  float radius = transform[3][3];      // Extract sphere radius.
  mat3 rot = mat3(transform);          // Extract rotation matrix.

  vec3 normal_o = rot * gl_Vertex.xyz;
  vec4 vertex_o = vec4( ctr + radius * normal_o, 1 );

  color = sphere_color[gl_InstanceID];
  gl_Position = gl_ModelViewProjectionMatrix * vertex_o;
  vertex_e = gl_ModelViewMatrix * vertex_o;
  normal_e = normalize(gl_NormalMatrix * normal_o );
  gl_TexCoord[0] = gl_MultiTexCoord0;  }
```

(*b*) In the code above label the vertex shader inputs and uniform variables.

☐ Label vertex shader inputs with a `I` and uniform variables with a `U`.

(*c*) Compute the amount of data sent from the CPU to the GPU for an instanced draw rendering pass in which n instances (spheres) are rendered, each consisting of v vertices. Assume that initially all buffer objects are on the CPU. Note that `mat4` is a 4×4 matrix of floats.

☐ Amount of CPU to GPU data for n instances of v vertices, accounting for ☐ uniforms, ☐ buffer objects, ☐ and anything else that really needs to be moved.

Problem 3: [15 pts] Consider our well written CUDA example (slightly simplified):

```
__global__ void kmain_efficient() {
  const int tid = threadIdx.x + blockIdx.x * blockDim.x;
  const int n_threads = blockDim.x * gridDim.x;

  // Note: There's no need to understand this code. Just remember
  // that each thread does about the same amount of work.

  for ( int h=tid; h<d_app.array_size; h += n_threads ) {
      float4 p = d_app.d_in[h];
      d_app.d_out[h] = dot(p,p);
    }
}
```

Let n denote the value of `array_size`, and let S denote the number of streaming multiprocessors (abbreviated SMs and sometimes MPs). Assume that n is large so that there's plenty of work to do for each thread.

(a) Determine a launch configuration in terms of n and S that uses the minimum number of blocks needed to maximize warp (or thread) occupancy on the SMs. Do this for a device with a block size limit of 1024 threads, an SM limit of 2048 threads, and an SM limit of 16 blocks. *Hint: There's no need to use both n and S.* Use B for the block size and G for the number of blocks (grid size).

☐ Block size $B =$

☐ Explain.

☐ Grid size (number of blocks) $G =$

☐ Explain.

(b) Let B and G be a solution to the problem above (meaning a good choice for B and G for S SMs) and let t_o be the execution time in a kernel launch of that configuration. Estimate the execution time for the configurations below. *Hint: Remember that G was chosen to maximize the number of warps per SM.*

☐ Estimate the execution time for a launch of G blocks of size $B - 1$. ☐ Explain.

☐ Estimate the execution time for a launch of $G - 1$ blocks of size B. ☐ Explain.

☐ Estimate the execution time for a launch of $G + 1$ blocks of size B. ☐ Explain.

Problem 4: [20 pts] Consider the CUDA kernel below. Let n denote the value of `array_size`, B the block size, and G the number of blocks.

```
__global__ void prob_x(float2 *d_in, float *d_out) {
  const int tid = threadIdx.x + blockIdx.x * blockDim.x;
  const int n_threads = blockDim.x * gridDim.x;

  for ( int h=tid; h<d_app.array_size-1; h += n_threads ) {
      float2 p = d_in[h];
      float2 q = d_in[h+1];
      d_out[h] = dot(p,q);
    }
}
```

(*a*) Compute how much data is read from global memory during the execution of the kernel. Note that each element `d_in[i]` is accessed by two threads.

☐ Amount of data read:

☐ Explain.

(*b*) The kernel below includes a declaration for shared memory intended to fix the inefficiency in the kernel above. Complete the kernel.

☐ Use shared memory to avoid waste.

```
__global__ void prob_x2(float2 *d_in, float *d_out) {
  const int tid = threadIdx.x + blockIdx.x * blockDim.x;
  const int n_threads = blockDim.x * gridDim.x;

  __shared__ float2 sm[1025];

  for ( int h=tid; h<d_app.array_size; h += n_threads ) {
      float2 p = d_in[h];










      float2 q =
      d_out[h] = dot(p,q);
    }
}
```

Problem 5: [25 pts] Answer each question below.

(*a*) Show a diagram illustrating the relationship between vertices and primitives in a triangle strip. Number both the vertices and the triangles in the correct order. Show at least three triangles.

☐ Diagram with number vertices and triangles illustrating a triangle strip.

(*b*) Describe what is tested in the stencil test and what the stencil test might be used for.

☐ A stencil test checks:

☐ Typical use for a stencil test:

(*c*) Interpolation qualifiers, `flat`, `noperspective`, and `smooth`, are used for the inputs of which rendering pipeline stage? What do they do?

☐ They appear at the input to:

☐ They affect the qualified variable by:

(*d*) Between which coordinate spaces does the OpenGL projection matrix transform.

☐ The projection matrix transforms coordinates ☐ from ☐ to

(*e*) What is the difference between a material property (including attributes emitted with `glColor3f`) and a lighted color?

☐ Difference between material property and lighted color?