Name _____

EE 4702

Final Exam

Friday, 7 December 2012, 12:30-14:30 CST

Problem 1	_____	(15 pts)
Problem 2	_____	(15 pts)
Problem 3	_____	(20 pts)
Problem 4	_____	(15 pts)
Problem 5	_____	(15 pts)
Problem 6	_____	(20 pts)
Exam Total	_____	(100 pts)

Alias _____

Good Luck!

Problem 1: [15 pts]The two code fragments below render the same set of triangles.

```
// Code Fragment A

 glBegin(GL_TRIANGLES);
 for ( int i=0; i<1000; i++ )
   {
     glColor3fv(lsu_spirit_gold); glVertex3fv(v0[i]);
     glColor3fv(lsu_spirit_gold); glVertex3fv(v1[i]);
     glColor3fv(lsu_spirit_gold); glVertex3fv(v2[i]);
   }
 glEnd();

// Code Fragment B
 glColor3fv(lsu_spirit_gold);
 glBegin(GL_TRIANGLE_STRIP);
 for ( int i=0; i< XXX; i++ )
     glVertex3fv(v[i]);
 glEnd();
```

(*a*) Compute the amount of data sent from CPU to GPU for each case. Use an appropriate value of XXX.

Problem 2: [15 pts]Consider the declarations below which come from the vertex shader in the Homework 3 solution.

```
layout ( location = 2 ) uniform float wire_radius;
layout ( location = 3 ) uniform float theta;
layout ( location = 4 ) uniform vec2 texture_scale;
layout ( binding = 1 ) buffer Helix_Coord  { vec4  helix_coord[];  };
layout ( location = 1 ) in ivec2 helix_index;
out vec3 normal_e;
uniform sampler2D tex_unit_0;
```

(*a*) What do the location numbers indicate in the uniform declarations?

(*b*) How is a `uniform` typically used?

(*c*) Explain what the `in` and `out` qualifiers used for `helix_index` and `normal_e` indicate.

(*d*) Why can't the declarations above be for a geometry shader? (That is, something won't work.)

(*e*) Why can't the declarations above be for a fragment shader? (That is, something won't work.)

Problem 3: [20 pts]The Homework 5 code simulated a spring, bringing to life the helix we were working with most of the semester. A combination of CPU and GPU code was used to compute the wire surface coordinates. Excerpts of the code appear below.

Grading Note: This problem was based on code used in Homework 5, with which some familiarity was expected.

```
Physics Simulation in CUDA computes helix_position and orientation, read to CPU.
              Wire_Segment* const ws = &wire_segments[i];
              ws->position = helix_position[i];
              ws->orientation = helix_orientation[i];

Computation on CPU:
          helix_coords[i] = ws->position;
          pMatrix_Rotation c_rot(ws->orientation); // Mult: 3 * 9 = 27
          helix_u[i] = c_rot * pVect(0,0,1);
          helix_v[i] = c_rot * pVect(0,1,0);

Computation on Vertex Shader
  vec3 vtx = helix_coord[hidx.x].xyz;
  float pi = 3.14159265;
  float theta = hidx.y * 2 * pi / 20;
  vec3 u = helix_u[hidx.x].xyz;
  vec3 v = helix_v[hidx.x].xyz;
  vec3 normal = normalize( cos(theta) * u + sin(theta) * v );
  // Compute wire surface location by adding normal to helix coordinate.
  //
  vec4 vertex_o;
  vertex_o.xyz = helix_coord[hidx.x].xyz + wire_radius * normal;
  vertex_o.w = 1;
```

Computing the u and v vectors on the CPU requires about 18 multiplications. Consider the method used in the code above and two alternatives: computing these in CUDA or computing them in the vertex shader.

The values of `hidx.x` (position [index of wire segment] along the helix) vary from 0 to $S-1$ and the values of `hidx.y` (position around the cylinder of a wire segment) vary from 0 to $M-1$.

(*a*) Compute the amount of data moving between CPU and GPU (in both directions) for the code shown.

(*b*) Compute the amount of data that would be moved if u and v were computed in CUDA.

(*c*) Compute the amount of data that would be moved if u and v were computed in the vertex shader.

(*d*) Which option computes the same thing multiple times? Is there any benefit to this redundancy?

Problem 4: [15 pts]A goal of Homework 2 was to use `glDrawElements` in place of `glDrawArray`. For our task of rendering a helix, `glDrawElements` reduces the number of vertices that need to be sent from CPU to GPU by a factor of 2.

(*a*) Explain why half the number of vertices are needed.

(*b*) When using `glDrawElements` what needs to be sent from CPU to GPU that was not needed with `glDrawArray`? How large is it in comparison to the vertex arrays?

Problem 5: [15 pts]Answer the following questions abut coordinate spaces. For the transformations use $T_{\vec{v}}$ to indicate a translation transformation and $R_{\vec{u}\to\vec{v}}$ indicate a rotation transform that rotates $\vec{u}$ to $\vec{v}$.

(*a*) Let P_o be a coordinate in object space coordinates, let E be the location of the eye and $\vec{v}$ be a normal to the projection surface. Show how to transform P_o into an eye-space coordinate.

(*b*) Let P be a point in clip space. How can we determine if P is in the view volume?

(*c*) Suppose we are rendering to a window of size 1000×1000 pixels. Let $P_c = (0.5, 0.5, 0.5)$ be a coordinate in clip space.

☐ Provide an example of clip space coordinate $P_d \neq P_c$ that is likely to be in the same pixel as P_c.

☐ Provide an example of clip space coordinate that is in a pixel close to P_c.

Problem 6: [20 pts]Consider a multiprocessor in an NVIDIA CC 2.0 GPU, the type frequently discussed in class. Such a multiprocessor had 32 CUDA cores, two schedulers, and contexts for 48 warps. For all questions below assume a kernel is launched with one block per multiprocessor.

(*a*) Explain why a block size of one warp is guaranteed to leave half the cores unused.

(*b*) Explain why a block size of two warps will likely result in cores being idle most of the time.

For the next two parts consider Kernel A and Kernel B, below. Notice that in Kernel A I1 must wait for I0 and I2 must wait for I1. In Kernel B I3 must wait for I0, but I1 does not have to wait for I0. Inspect the code for additional dependencies. Suppose that this pattern of dependencies continues in each kernel. That is, in Kernel A each instruction depends on its immediate predecessor, while in Kernel B each depends on the third previous instruction.

```
# Kernel A
I0:  add  r1, r2, r3
I1:  mul  r4, r1, r5
I2:  sub  r6, r4, r7
I3:  or   r8, r6, r9
...

# Kernel B
I0:  add  r1, r2, r3
I1:  add  r10, r12, r13
I2:  add  r20, r22, r23
I3:  mul  r4, r1, r5
I4:  mul  r14, r11, r15
I5:  mul  r24, r21, r25
```

(*c*) Suppose that each kernel is launched with two warps per block (and one block per MP). Assume that the instruction latency is 24 cycles. Draw a diagram showing the time that each instruction executes, until the pattern is obvious.

(*d*) Compute the CUDA core utilization (the fraction of time a core will be used) for each kernel.