EE 7700-2
Take-Home Pre-Final Examination
Friday, 2 May 2008 to Monday 5 May 2008

Problem 1 _________ (25 pts)
Problem 2 _________ (25 pts)
Problem 3 _________ (25 pts)
Problem 4 _________ (25 pts)

Alias __________________________
Exam Total _________ (100 pts)

Good Luck!
Problem 1: [25 pts] The code below computes lighting in opengl/demo-5-shader, it is run either as a vertex shader or a fragment shader. (For this problem it could be either.)

Modify the code so that if the light is behind the primitive (from the viewer’s perspective), the primitive appears red, otherwise the color is based on the material color (variable color); in both cases distance and angle are taken into account when computing intensity.

The code does not have to be run, just write the changes below.

```cpp
vec4 generic_lighting(vec4 vertex_e, vec4 color, vec3 normal_e)
{
    // Perform lighting calculations VTX, using COLOR and NORMAL.
    //
    // Supposedly a tweak, but this routine doesn’t do anything special.
    vec4 light_pos = gl_LightSource[0].position;
    vec4 v_vtx_light = light_pos - vertex_e;
    float phase_light = dot(normal_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;
    return new_color;
}

void vs_main_lighting()
{
    // Use custom lighting model.
    //
    vs_ff_vertex(gl_Vertex);
    vec4 vertex_e = gl_ModelViewMatrix * gl_Vertex;
    vec3 normal_e = normalize(gl_NormalMatrix * gl_Normal);
    gl_FrontColor = generic_lighting(vertex_e,gl_Color,normal_e);
}
```
Problem 2: [25 pts] The C++ code below, taken from the demo-4-lighting routine, applies a transformation to a vertex and then homogenizes the result. Following the C++ code is simplified assembler code, showing its execution (this was taken from PSE).

The C++ code was written for a CPU, but consider the code’s counterpart running on the vertex processor of a GPU. That is, the vertex processor is running a shader which applies a transformation and homogenizes the result (no lighting calculations are done).

The assembly code has been simplified and some comments have been added to help you understand the code.

(a) Suppose the vertex processor uses quad (vector) data types. Show which instructions can be combined into a single vector instruction. (Don’t pick, say, 3 multiplies at random.)

(b) Some of the assembly instructions below do something that shader assembly code does not have to do (and couldn’t do). Circle those instructions and mark them “Not Needed.”

(c) Explain why those instructions aren’t needed and how the GPU gets by without them.

(d) The code below uses registers and memory locations for storage. Consider a vertex processor that can just access input attribute registers, constant registers, temporary registers, and vertex result registers (in output buffer).

For each of these four types of registers find two places in the assembly code below where those registers would be used. For example, one might answer “f99 → constant register” or “address %g9 + 123 → constant register.”

For this part eight items below should be marked.

```c++
///
/// Transform Coordinates from Eye Space to Window Space
///
for ( pVertex_Iterator ci = vtx_list.begin(); ci < vtx_list.end(); ci++ )
{
    pVertex& v = **ci; // Get reference to current vertex
    v *= transform_to_viewport;
    v.homogenize();
}
```
Problem 2, continued:

! Note: Code simplified.

.LLM970  render_light+1603 stl_deque.h:145
000122a8 add %g2, 4, %g2 ! ci++
000122ac cmp %g4, %g2 ! ci < vtx_list.end()
000122b0 bpe, pn %icc, +532i -> {0x12b00 stl_deque.h:148} ! v.x = (from last iteration.)
000122b4 stf %f10, [ %g1 ] {0x1d6208}

.LLM947  render_light+1544 demo-4-lighting.cc:480
000121c8 lduw [ %g2 ], %g1 {0x1d4840}
000121cc ldf [ %fp - 500 ], %f8 {0x7ffffb3c}
000121d0 ldf [ %g1 + 4 ], %f16 {0x1d620c} ! f16 = v.y;
000121d4 fmuls %f8, %f16, %f8
000121d8 ldf [ %g1 ], %f10 {0x1d6208}
000121dc ldf [ %fp - 504 ], %f12 {0x7ffffb38}
000121e0 fmuls %f12, %f10, %f12
000121e4 fadds %f12, %f8, %f12
000121e8 ldf [ %fp - 516 ], %f8 {0x7fffffb2c}
000121ec fmuls %f8, %f16, %f8
000121f0 ldf [ %fp - 520 ], %f11 {0x7fffffb28}
000121f4 ldf [ %g1 + 8 ], %f15 {0x1d6210}
000121f8 fmuls %f15, %f10, %f11
000121fc fadds %f15, %f8, %f11
00012200 ldf [ %fp - 496 ], %f8 {0x7ffffffb40}
00012204 fmuls %f8, %f15, %f8
00012208 fadds %f12, %f8, %f12
0001220c ldf [ %fp - 512 ], %f8 {0x7ffffffb30}
00012210 fmuls %f8, %f15, %f8
00012214 ldf [ %g1 + 12 ], %f14 {0x1d6214}
00012218 fadds %f11, %f8, %f11
0001221c ldf [ %fp - 492 ], %f8 {0x7ffffffb44}
00012220 fmuls %f8, %f14, %f8
00012224 ldf [ %fp - 532 ], %f9 {0x7ffffffb1c}
00012228 fadds %f12, %f8, %f12
0001222c fmuls %f9, %f16, %f9
00012230 ldf [ %fp - 508 ], %f8 {0x7ffffffb34}
00012234 fmuls %f8, %f14, %f8
00012238 ldf [ %fp - 536 ], %f13 {0x7ffffffb18}
0001223c fadds %f11, %f8, %f11
00012240 fmuls %f13, %f10, %f13
00012244 ldf [ %fp - 552 ], %f8 {0x7ffffffb08}
00012248 fadds %f13, %f9, %f13
0001224c ldf [ %fp - 528 ], %f9 {0x7ffffffb20}
00012250 fmuls %f9, %f15, %f9
00012254 fadds %f10, %f8, %f10
00012258 fadds %f13, %f9, %f13
0001225c ldf [ %fp - 544 ], %f8 {0x7ffffffb10}
00012260 ldf [ %fp - 524 ], %f9 {0x7ffffffb24}
00012264 fmuls %f9, %f14, %f9
00012268 fdivs %f17, %f12, %f12
0001226c fadds %f13, %f9, %f13
00012270 fmuls %f15, %f8, %f15
00012274 ldf [ %fp - 548 ], %f9 {0x7ffffffb0c}
Second iteration starts below.

.LLM970 render_light+1603 stl_deque.h:145
000122a8 add %g2, 4, %g2
000122ac cmp %g4, %g2
000122b0 bpe, pn %icc, +532i -> {0x12b00 stl_deque.h:148}
000122b4 stf %f10, [ %g1 ] {0x1d6208}
000122bc lduw [ %fp - 20 ], %l1 {0x7ffffd1c}
000122c0 cmp %g3, %l1
000122c4 bnp, pt %icc, -66i -> {0x121bc demo-4-lighting.cc:480}
000122c8 addc %g0, 0, %g1
Problem 3: [25 pts] Answer the following questions about multithreading.

Suppose a fragment processor provides a fragment shader with 16 quad input attributes, 32 quad temporaries, and 128 quad constants. The fragment processor does not provide bypass paths.

(a) The fragment processor has the following stages:

IF ID SZ E1 E2 E3 E4 WB

How many threads and how much storage would the fragment processor need to avoid dependence stalls? *Hint: This is easy.*

Suppose an L1 texture cache miss /L2 texture cache hit adds 10 cycles to the execution above.

Estimate the number of threads and the storage needed for each alternative below:

(b) Add dummy stages so that most instructions will use the same number of stages as one that suffers an L1 cache miss, L2 hit.

(c) Have an extra thread ready that can run in place of one that misses L1 and hits L2. *Hint: This would be trivial were it not for the requirement that operations occur in the order in which they were issued to OpenGL. That is, the missing thread can’t write after those from preceding vertices.*
Problem 4: [25 pts] Answer each question below.

(a) The stream processors in the GeForce 8800 are able to read and write memory, yet for frame buffer updates they send an operation to a ROP (raster op processor). Frame buffer operations are seemingly simple operations: read $z$ and stencil buffer, if $z$ and stencil test pass either write new fragment or blend new fragment with existing pixel. A stream processor can do this using a small number of instructions. So why is a ROP needed?

(b) In bump mapping a texel specifies not a color, but an amount by which the surface normal is bent (along the shape of a bump). If the surface normal is bent away from the light the fragment would appear darker than otherwise, if towards the light, brighter. Why would it be computationally expensive (significant for older-generation GPUs) to use the bump map in this way (at least taking a straightforward approach)?

(c) It is possible to implement a geometry shader on a stream processor of a GeForce 8800. Why couldn't a GeForce 6800 run a geometry shader using either the vertex processor or the fragment processor? What feature allows the 8800 to do so?