Name _____

GPU Programming
LSU EE 4702-1
Take-Home Final Examination

Sunday 8 December to Wednesday 11 December 2019 16:30 CST

Work on this exam alone. Regular class resources, such as notes, papers, solutions, documentation, and code, can be used to find solutions. In addition outside OpenGL references and tutorials, other programming resources, and mathematical references can be consulted. Do not try to seek out references that specifically answer any question here. **Do not discuss this exam with classmates or anyone else**, except questions or concerns about problems should be directed to Dr. Koppelman.

Problem 1	_____	(25 pts)
Problem 2	_____	(20 pts)
Problem 3	_____	(20 pts)
Problem 4	_____	(10 pts)
Problem 5	_____	(25 pts)
Exam Total	_____	(100 pts)

Alias _____

Good Luck!

Problem 1: [25 pts] Appearing below is code based on the solution to Homework 4, in which the Strip-Plus shader is modified so that the start of a triangle strip is indicated by setting the w component of the first two vertices to zero.

```
glBegin(GL_TRIANGLE_STRIP);

// Render the diamond top.
glColor3fv(color_tan);
for ( auto& e: hw04.bump_info ) {   // Note: n iterations.
    pCoor vtop(e.ctop), vl(e.cl);
    vtop.w = vl.w = 0;
    glVertex4fv(vtop);    glVertex4fv(vl);
    glVertex4fv(e.cr);    glVertex4fv(e.cbot);  }

// Render the sides.
glColor3fv(color_olive_drab);
for ( auto& e: hw04.bump_info ) {   // Note: n iterations.

    pCoor pts[2][4] =                    // Construct an array of coordinates.
      {  { e.ctop, e.cr, e.cbot, e.cl }, // Original Diamond
           // Points "below" diamond on cylinder surface.
         { e.ctop+e.nc*dr, e.cr+e.nr*dr, e.cbot+e.nc*dr, e.cl+e.nr*dr } };

    // Render Block's Sides
    for ( int i=0; i<5; i++ ) {
        const int i0 = i & 0x3;
        pCoor v1 = pts[1][i0], v2 = pts[0][i0];
        if ( !i ) v1.w = v2.w = 0;
        glVertex4fv( v1 ); glVertex4fv( v2 );  }
  }
glEnd();
```

(a) Appearing above is the CPU code performing a rendering pass for the protrusions. Compute the amount of data sent from the CPU to the GPU for one rendering pass. The amount of data should be in units of bytes and in terms of n, the number of protrusions. (The `hw04.bump_info` container holds n elements.)

☐ Amount of data per pass in units of bytes in terms of n:

(b) For a rendering pass using the code above rendering n protrusions, what is the number of vertex shader and geometry shader invocations.

☐ Number of vertex shader invocation in terms of n:

☐ Number of geometry shader invocation in terms of n:

(*c*) Appearing below is the interface block for the input to the fragment shaders used by both the Strip-Plus and points shaders (Homework 3 and 4). Notice that both `normal_e` and `color` have interpolation qualifier `flat`. For one of the two removing the `flat` qualifier would hurt performance but would not change the appearance of the protrusions. For one of the two removing the `flat` qualifier would hurt performance and change the appearance of the protrusions. Identify which and explain.

```
in Data_to_FS
{
  flat vec3 normal_e;
  vec4 vertex_e;
  flat vec4 color;
};
```

☐ Removing `flat` (from either or both) hurts performance because:

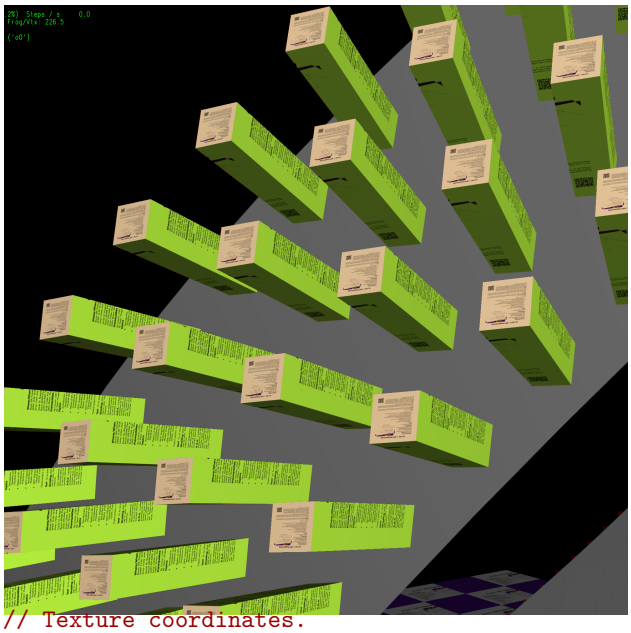
☐ Removing `flat` from `normal_e` ◯ *will* or ◯ *will not* change appearance.

☐ Explain.

☐ Removing `flat` from `color` ◯ *will* or ◯ *will not* change appearance.

☐ Explain.

Problem 2: [20 pts] The screenshot to the right shows textures applied to the protrusions rendered using a modified version of the Homework 4 solution. Appearing below is the points geometry shader with code to apply the texture to the diamond (top), but lacking code to apply the texture to the sides.



(*a*) Add code so that the texture is applied to the sides as shown. In particular, each side of the protrusion should show exactly $\frac{1}{4}$ of the texture. Note that `texcoord` is a `vec2` type, and `texcoord.x` and `texcoord.y` are the x and y coordinates of the texture.

```
// Emit the diamond-shaped top.
color = color_diamond;
normal_e = gl_NormalMatrix * nl;

int ord[4] = { 1, 0, 2, 3 }; // Vertex order.
vec2 tc[4] = { {0,0}, {1,0}, {1,1}, {0,1} }; // Texture coordinates.
for ( int i=0; i<4; i++ )
  {
    texcoord = tc[ord[i]];     // Set texture coordinates.
    vertex_e = pts_e[ord[i]];  // Retrieve vertices in the correct order.
    gl_Position = gl_ProjectionMatrix * vertex_e;
    EmitVertex();
  }
EndPrimitive();

// Emit the sides.
color = color_edge;
for ( int i=0; i<5; i++ ) {
    int i0 = i & 0x3;          // The current edge.
    int i1 = ( i + 1 ) & 0x3;  // The next edge. (Used to compute the normal.)



    vertex_e = pts_e[i0+4];
    gl_Position = gl_ProjectionMatrix * vertex_e;
    EmitVertex();



    vertex_e = pts_e[i0];
    gl_Position = gl_ProjectionMatrix * vertex_e;
    EmitVertex();

    // This normal will be used for the next two vertices.
    normal_e = cross( pts_e[i1].xyz   - pts_e[i0].xyz, pts_e[i0+4].xyz - pts_e[i0].xyz );
  }
EndPrimitive();
```

☐ Add code to apply texture to sides as described above.

(*b*) Notice that the image on the diamonds appears upside down. Modify the code below so that the image on the diamond always appears right side up in the global coordinate space (but is still aligned with the diamond so it will be crooked). This means that the texture coordinates assigned on one side of the ring will be different than the other. For example, in the part of the ring shown in the screen shot the images are upside down. But on the other side of the ring those images would be right-side up.

```
// Emit the diamond-shaped protrusion top.
color = color_diamond;
normal_e = gl_NormalMatrix * nl;

int ord[4] = { 1, 0, 2, 3 }; // Order in which to emit vertices.
vec2 tc[4] = { {0,0}, {1,0}, {1,1}, {0,1} }; // Texture coordinates.
```

```
for ( int i=0; i<4; i++ )
  {
    texcoord = tc[ord[i]];     // Set texture coordinates.
    vertex_e = pts_e[ord[i]];  // Retrieve vertices in the correct order.
    gl_Position = gl_ProjectionMatrix * vertex_e;
    EmitVertex();
  }
EndPrimitive();
```

☐ Modify code above so that texture on each diamond is rightside up.

Problem 3: [20 pts] Appearing below are kernels based on the `demo-cuda-02-basics.cu` classroom demonstration file along with a routine that launches the kernels, `launch_kernels`.

```
void launch_kernels() {
  const int thd_per_block = 512;
  const int number_of_blocks = 16;

  app.array_size = 1 << 20;
  cudaMemcpyToSymbol( d_app, &app, sizeof(app), 0, cudaMemcpyHostToDevice );

  kmain_simple <<< number_of_blocks, thd_per_block >>> ();
  kmain_efficient <<< number_of_blocks, thd_per_block >>> ();
  kmain_tuned <<< number_of_blocks, thd_per_block >>> ();
}

__global__ void kmain_simple() {
  const int tid = threadIdx.x + blockIdx.x * blockDim.x;
  const int num_threads = blockDim.x * gridDim.x;

  const int elt_per_thread = ( d_app.array_size + num_threads - 1 ) / num_threads;
  const int start = elt_per_thread * tid;  // Bad: Non-consecutive access.
  const int stop = start + elt_per_thread;

  for ( int h=start; h<stop; h++ )
    {
      int idx = h;
      float4 p = d_app.d_in[idx];  // Bad: Non-consecutive access.
      float sos = p.x * p.x + p.y * p.y + p.z * p.z + p.w * p.w;
      d_app.d_out[idx] = sos;
    }
}

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

  for ( int h=tid; h<d_app.array_size; h += num_threads )
    {
      const int idx = h;
      float4 p = d_app.d_in[idx];  // Good: Consecutive access.
      float sos = p.x * p.x + p.y * p.y + p.z * p.z + p.w * p.w;
      d_app.d_out[idx] = sos;
    }
}
```

```
__global__ void kmain_tuned() {
  const int tid = threadIdx.x + blockIdx.x * blockDim.x;
  const int num_threads = blockDim.x * gridDim.x;
  constexpr int strip_len = 4;
  // Data "strip" is 32 threads wide and strip_len threads long.

  const int wp_sz = 32;         // Warp size.
  const int wp = tid / wp_sz;   // This thd's warp number within kernel. (0-)
  const int ln = tid % wp_sz;   // This thd's lane number within warp.  (0-31)
  const int start = wp * wp_sz * strip_len + ln;

  for ( int h=start; h<d_app.array_size; h += strip_len * num_threads )
    {
      float soses[strip_len];
      for ( int i=0; i<strip_len; i++ )
        {
          int idx = h + i * wp_sz;
          float4 p = d_app.d_in[ idx ];
          soses[i] = p.x * p.x + p.y * p.y + p.z * p.z + p.w * p.w;
        }
      for ( int i=0; i<strip_len; i++ )
        d_app.d_out[ h + i * wp_sz ] = soses[i];
    }
}
```

(*a*) The tables below are to be filled with the value of variable `idx` from the three kernels, a total of eight values per table. Each column shows a different thread (two threads from each of two blocks), and each row shows a different loop iteration. Fill the tables based on the code above. *Hint: The upper left entry of each table will be zero.*

☐ Show the values of `idx` for the `kmain_simple` kernel.

```
blockIdx.x:     0        0        1        1
threadIdx.x:    0        1        0        1
-------------------------------------------

1st h iter


2nd h iter
```

☐ Show the values of `idx` for the `kmain_efficient` kernel.

```
blockIdx.x:     0        0        1        1
threadIdx.x:    0        1        0        1
-------------------------------------------

1st h iter


2nd h iter
```

☐ Show the values of `idx` for the `kmain_tuned` kernel. ☐ Note that the rows are for the first `i` loop, **not the `h` loop**.

```
blockIdx.x:     0        0        1        1
threadIdx.x:    0        1        0        1
-------------------------------------------

1st i iter


2nd i iter
```

(*b*) The routine `launch_kernels` always launches a configuration with 16 blocks.

Suppose a GPU had s SMs (also called MPs in class).

☐ Explain why launching s blocks is a good idea.

☐ Which is worse, launching ◯ $s-1$ *blocks* or ◯ $s+1$ *blocks* ? ☐ Explain.

Problem 4: [10 pts] The code fragments below are based on the `time_step_intersect_1` kernel from classroom demo-cuda-04. In the first fragment global memory (`helix_position`) is copied to shared memory (`pos_cache`) before use. In the second fragment global memory is accessed directly.

```
/// First Fragment – With Shared Memory
__shared__ float4 pos_cache[1024];
for ( int b_idx = b_idx_start; b_idx < hi.phys_helix_segments; b_idx += thd_per_a )
  {
    __syncthreads(); // Line A
    if ( threadIdx.x < thd_per_a )
      pos_cache[threadIdx.x] =
        helix_position[ b_idx - b_idx_start + threadIdx.x ];
    __syncthreads(); // Line B

    float4 b_position = pos_cache[b_idx_start];

    pVect ab = mv(a_position,b_position);
    // [snip]
```

```
/// Second Fragment – Without Shared Memory
for ( int b_idx = b_idx_start; b_idx < hi.phys_helix_segments; b_idx += thd_per_a )
  {
    float4 b_position = helix_position[b_idx];

    pVect ab = mv(a_position,b_position);
    // [snip]
```

(*a*) Why are the `syncthreads` needed in the first fragment. In particular:

☐ What would happen if the `syncthreads` on `Line B` were removed?

☐ What would happen if the `syncthreads` on `Line A` were removed?

(*b*) Since both fragments access `helix_position`, why is the fragment using shared memory potentially better?

☐ Ignoring request size, reason that first (shared) fragment might be faster than second?

☐ Accounting for request size, reason that first (shared) fragment might be faster than second?

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

(*a*) The code fragment below, taken from the classroom demo-7 code, updates the buffer object when `gpu_buffer_stale` is `true`. Variable `gpu_buffer_stale` is set to `true` initially, and again only when something about the sphere changes (for example, the number of triangles used to approximate it). The line commented `DISASTER` was added for this problem. Explain what will go wrong. *Hint: The problem occurs when* `gpu_buffer_stale` *is frequently true.*

```
if ( gpu_buffer_stale ) {
    gpu_buffer_stale = false;

    // Generate buffer id (name), if necessary.
    if ( !gpu_buffer ) glGenBuffers(1,&gpu_buffer); // ORIGINAL
    glGenBuffers(1,&gpu_buffer);                    // DISASTER

    // Tell GL that subsequent array pointers refer to this buffer.
    glBindBuffer(GL_ARRAY_BUFFER, gpu_buffer);

    // Copy data into buffer.
    glBufferData
      (GL_ARRAY_BUFFER,           // Kind of buffer object.
       coords_size*sizeof(pCoor), // Amount of data (bytes) to copy.
       sphere_coords.data(),      // Pointer to data to copy.
       GL_STATIC_DRAW);           // Hint about who, when, how accessed.

    // Tell GL that subsequent array pointers refer to host storage.
    glBindBuffer(GL_ARRAY_BUFFER, 0);
  }
```

☐ Explain the disaster that the `DISASTER` line causes.

(*b*) Explain why the true sphere shader has higher performance than the tessellated sphere shader when there are lots of small (in window space) spheres.

☐ True sphere faster than tessellated for small spheres because:

(*c*) When rendering shadow volumes lighting calculations and texture lookups are not done, both of which can be time consuming. Then what is it about rendering shadow volumes that makes it potentially take more time than rendering the objects that cast the shadows?

☐ Rendering shadow volumes takes longer than objects because:

(*d*) How can a surface normal be used by the lighting routine to compute the lighted color? What are the advantages and disadvantages of doing the calculation in the fragment shader?

☐ How is a surface normal used to compute lighted color?

☐ Advantage of computing lighted color in fragment shader over doing the calculation in the vertex shader.

☐ Disadvantage of computing lighted color in fragment shader.

(*e*) Which rendering pipeline stages would be affected by a switch from an individual triangle (`GL_TRIANGLES`) rendering pass to a triangle strip rendering pass.

☐ The affected stages are (check all that apply): ◯ *vertex* , ◯ *geometry* , ◯ *fragment* . ☐ Explain.