LSU EE 4702-1 **Homework 6** Solution **Due: 24 November 2014**

Problem 0: Follow the class account instructions for obtaining the assignment, substitute hw06 where needed. This homework code is based on Homework 4/3, and the physics simulation is identical. Like those assignments it renders a spiral. The spiral can be rendered using two routines, `render_spiral2` and `render_spiral3`; the `m` key switches between them. Routine `render_spiral2` is based on the Homework 4 solution, and initially `render_spiral3` is identical to `render_spiral2`, it will be modified for this assignment.

Routine `render_spiral2`, rather than sending actual vertex coordinates to the GPU, sends indices. The x component of a vertex "coordinate" is the ball number (the value of `i` in `render_spiral1`), the y component is the relative position between ball `i-1` and ball `i`, ranging from 0 to `opt_segments-1`, and the z component indicates whether the vertex is on the inner or outer edge of the spiral. The code in vertex shader `vs_main` in file `hw06-shdr-triangles.cc` uses these values to compute the actual vertex coordinate and normal, and to compute the texture coordinates.

The level of detail used for the spiral is controlled by variable `opt_segments`. Its value can be modified by first pressing the `TAB` key until "Number of segments per spiral" appears, and then pressing `+` and `-` to adjust its value. Check out and compile the code, and make sure that it runs correctly.

For this assignment modify files `hw06.cc` and `hw06-shdr-lines.cc`. Solutions to the non-code questions can be provided in any of the following forms:

- As comments in one of the files above.
- As plain text in a separate file named `hw06.txt` in the `hw06` directory.
- In portable document format in a separate file named `hw06.pdf` in the `hw06` directory.
- Submitted on paper.
- E-mailed. If E-mailing, plain text and PDF are preferred.

Problem 1: The code in `render_spiral2` is based on the solution to Homework 4. As discussed in Homework 5 Problem 3, the vertex shader used for `render_spiral2` in Homework 4 wastes computation because it computes information both for inner and outer vertices but only sends one of these to the geometry shader. The problem suggests using line strips to solve that problem. The solution to Homework 5 presented some details on how this would be done, but it did not show any code. For this problem, modify `render_spiral3` and the shaders so that it avoids the waste.

For this problem you must modify code both in `render_spiral3` in file `hw06.cc` and the code in `hw06-shdr-lines.cc`, both will be collected by the TA-bot. Initially the code in `render_spiral3` is nearly identical to `render_spiral2` and `hw06-shdr-lines.cc`, except that it invokes shaders from file `hw06-shdr-lines.cc` (which you should modify). Routine `render_spiral2` invokes shader code from `hw06-shdr-triangles.cc`, which you should not modify.

Here are some things to watch out for:

- Be sure to adjust the maximum number of vertices specified for the geometry shader output. If this number is too low execution will end with an error.
- When you modify an interface block (such ad `Data_to_GS`) be sure to modify the other interface block with the same name (one is a shader output, the other is a shader input).

Here are some debugging tips:

- Check for shader code compilation errors when your program starts. The errors are sent to `stdout` and should appear in the shell or in gdb, depending on how you started the program.
- If execution ends with an OpenGL error, you can get a more detailed error message by turning on OpenGL debugging. To do this change `false` to `true` in `popengl_helper.ogl_debug_set(false);` near the end of `hw04.cc`.
- Three UI-controlled variables are available for debugging. They are `debug_bool.x`, `debug_bool.y`, and `debug_float`. The Booleans can be toggled with `d` and `D`. The float can be adjusted by pressing `Tab` until "Debug Float" appears and then press `+` and `-` to adjust the value.

The solution has been checked into the repo, look at files `hw06sol.cc` and `hw06-shdr-lines-sol.cc`. Web versions can be found at `https://www.ece.lsu.edu/koppel/gpup/2014/hw06sol.cc.html` and `https://www.ece.lsu.edu/koppel/gpup/2014/hw06-shdr-lines-sol.cc`, the repo versions will be most up to date.

Before reading this discussion be sure to review the solution to Homework 5, `https://www.ece.lsu.edu/koppel/gpup/2014/hw05_sol.pdf`, which describes how the code operates. Highlights are provided here.

Routine `render_spiral3` must be changed to send the vertices as a line strip instead of a triangle strip. Rather than sending an inner and an outer vertex, only one vertex will be sent. The z component of a vertex was used to indicate whether it was an inner our outer vertex. That's no longer needed, so the vertex structure now only has two elements. Notice in the code below that the `struct` has two members, rather than 3.

```
struct ivec2 { int i, j; ivec2(){}; ivec2(int ip, int jp):i(ip),j(jp){}};
ivec2* const indices = new ivec2[ num_elts ];
ivec2 *ip = indices;
for ( int i=1; i<chain_length; i++ )
  for ( int t=0; t<opt_segments; t++ ) *ip++ = ivec2(i,t);
glBindBuffer(GL_ARRAY_BUFFER, indices_bo);
glBufferData
  (GL_ARRAY_BUFFER, num_elts * sizeof(indices[0]), indices, GL_STATIC_DRAW);
delete indices;
```

Other changed code in `render_spiral3`:

```
/// SOLUTION
//  Change number of components from 3 to 2.
glVertexPointer(2,GL_INT,2*sizeof(int),0);
glEnableClientState(GL_VERTEX_ARRAY);
glBindBuffer(GL_ARRAY_BUFFER, 0);

/// SOLUTION
//  Change primitive from triangle strip to line strip.
glDrawArrays(GL_LINE_STRIP, 0, num_elts );
```

The vertex shader was modified so that it sends four rather than two sets of coordinates to the geometry shader. These were declared using 2D arrays to simplify the code in the geometry shader. Here is the interface block between the two shaders:

```
in Data_to_GS {
  vec4 vertex_e[2][2];  // Vertex coordinates in eye space.
```

```
  vec4 position[2][2];  // Vertex coordinates in clip space.
  vec2 texCoord[2];
  vec3 normal_e[2];
  vec3 radial_e;  // Normal for edge primitives.
} In[];
```

Only two sets of texture coordinates and two sets of normals are needed (not four) because the upper and lower spiral use the same texture coordinates and normals. The variable `radial_e` stores the normal used for the edge (wall), that is the same in all four locations (except for a sign flip).

The vertex shader code assigns these elements, taking advantage of the 2D array to avoid code duplication when computing eye- and clip-space coordinates:

```
  // Compute the four vertex coordinates in object space.
  //
  vec3 pos_o[2][2];
  pos_o[0][0] = p_inner;
  pos_o[0][1] = p_outer;
  pos_o[1][0] = p_inner + depth_vector;
  pos_o[1][1] = p_outer + depth_vector;

  // Transform the object-space coordinates to clip and eye space.
  //
  for ( int l=0; l<2; l++ )
    for ( int r=0; r<2; r++ )
      {
        vec4 position_o = vec4( pos_o[l][r], 1 );
        position[l][r] = gl_ModelViewProjectionMatrix * position_o;
        vertex_e[l][r] = gl_ModelViewMatrix * position_o;
      }
```

Because of the way the coordinates are organized the geometry shader can emit the primitives using just two loop nests:

```
  /// Emit the spiral triangles.
  //
  for ( int level=0; level<2; level++ )          // Upper / Lower
    {
      for ( int theta=0; theta<2; theta++ )
        {
          for ( int r=0; r<2; r++ )              // Inner / outer
            {
              normal_e       = In[theta].normal_e[r];
              vertex_e       = In[theta].vertex_e[level][r];
              gl_Position    = In[theta].position[level][r];
              gl_TexCoord[0] = In[theta].texCoord[r];
              is_edge = false;
              EmitVertex();
            }
        }
      EndPrimitive();           // This completes a strip of two triangles.
    }

  /// Emit the edge (wall) triangles.
```

```
//
for ( int r=0; r<2; r++ )                       // Inner / outer
  {
    for ( int theta=0; theta<2; theta++ )
      {
        for ( int level=0; level<2; level++ ) // Upper / Lower
          {
            normal_e    = r==0 ? -In[theta].radial_e : In[theta].radial_e;
            vertex_e    = In[theta].vertex_e[level][r];
            gl_Position = In[theta].position[level][r];
            is_edge = true;
            EmitVertex();
          }
      }
    EndPrimitive();           // This completes a strip of two triangles.
  }
```

Only the highlights are shown above.

Note: Everything done in the vertex shader could have been done in the geometry shader. But doing all the work in the geometry shader is inefficient because each "vertex" emitted by the vertex shader (which actually stores four vertices) is used by two invocations of the geometry shader. If the geometry shader computed things like `vertex_e` then it would be doing twice as much work.

Problem 2: Compare the performance of `render_spiral3` to `render_spiral2`. To see any difference at all the value of `opt_segments` will need to be increased, so do this if necessary.

(*a*) Show a table comparing the performance of the two methods versus `opt_segments`. In the table show CPU and GPU OpenGL time. Also indicate which GPU you are using.

The table should show that it takes large values of `opt_segments` before `render_spiral3` shows any benefit.

(*b*) Explain how varying `opt_segments` changes the proportion of the work performed by the three shader stages (vertex, geometry, and fragment).

Increasing `opt_segments` increases the number of triangles, but those triangles are smaller. The total area of the triangles does not change.

The amount of work performed by the vertex shader (actually the number of invocations) is proportional to the number of vertices, and that increases with `opt_segments`. For our code the amount of work performed by the geometry shader also increases with `opt_segments`. However, the amount of work performed by the fragment shader is proportional to the number of fragments, and that is determined by area of the triangles when projected on the screen (frame buffer). That does not change with `opt_segments`. When `opt_segments` increases there are more triangles, but each triangle has fewer fragments, and so the amount of work does not change.

(*c*) In a solution to an exam problem, the computation of `sin(theta)` and `cos(theta)` was moved out of the vertex shader. Instead the pre-computed sine and cosine values would be read from a buffer object. Would this increase or decrease the relative benefit of `render_spiral3`?

Short answer: It would decrease the relative benefit of `render_sprial3` because vertex shader execution time itself would be less important.

Long Answer: Moving the sine and cosine operations out of the vertex shader significantly reduces the amount of execution time spent in the vertex shader. Perhaps `render_spiral2` vertex shader execution would drop from 20% of total rendering time to 3%. (Note: those numbers are totally made up.) If we switch from `render_spiral2` to `render_spiral3` with the trig operations, we drop from 20% to 11.1% of total time, a respectable improvement. But if using `render_spiral2` we are only consuming 3% of execution time then switching to `render_spiral3` will only drop that to 1.52% of execution time, a much smaller improvement.

Problem 3: When the Homework 6 code starts it prints information about the available GPUs, using a CUDA API. CUDA isn't used for anything other than printing this information. Here is some sample output:

```
GPU 0: Tesla K20c @ 0.71 GHz WITH 5119 MiB GLOBAL MEM
GPU 0: CC: 3.5  MP: 13  CC/MP: 192  TH/BL: 1024
GPU 0: SHARED: 49152 B  CONST: 65536 B  # REGS: 65536
GPU 0: L2: 1280 kiB   MEM to L2: 208.0 GB/s  SP 1760.9 GFLOPS  OP/ELT 33.86
GPU 1: GeForce GTX 780 @ 1.02 GHz WITH 3071 MiB GLOBAL MEM
GPU 1: CC: 3.5  MP: 12  CC/MP: 192  TH/BL: 1024
GPU 1: SHARED: 49152 B  CONST: 65536 B  # REGS: 65536
GPU 1: L2: 1536 kiB   MEM to L2: 288.4 GB/s  SP 2348.9 GFLOPS  OP/ELT 32.58
```

In the example above information on two GPUs is given. (My machine and all of the graphic lab computers have two GPUs.) The first GPU is a Tesla K20c and has Compute Capability 3.5. The number of streaming multiprocessors (abbreviated `MP` in the output) is 13. The maximum number of threads per block is 1024.

Suppose the time step routine (a version of `time_step_cpu`) were going to be run on the GPU.

(*a*) Based on the program's default parameters and a GPU on one of the lab machines, how many blocks should be launched and how many threads per block? (The number of blocks and number of threads per block are collectively called the *launch configuration*.) *Please* indicate which GPU and its relevant statistics when answering this question.

This answer is for the K20c shown above. WARNING: If you read the answer to this subproblem you MUST read the answer to the next one. If you do not agree to do this, STOP READING NOW.

Looking in the time step routine we see two outer loops iterating over `chain_length` elements. An obvious thing to consider is to assign each iteration to a thread. The default value of `chain_length` is only 18. The number of blocks should be a multiple of the number of multiprocessors. There are 13 MPs so there should be 13 blocks. The 18 threads should be divided between these blocks, so we need two threads per block (in eight blocks only one thread would be active).

(*b*) Do you expect the code to run efficiently with this launch configuration? Explain.

No. No! NO! Efficiency increases with the number of threads. Because of the way threads are scheduled by the hardware, efficiency is better when the number of threads per block is a multiple of 32. Though the number of threads needed to realize efficiency depends on the type of GPU and especially the characteristics of the code, a good ballpark minimum is 256 threads per block.

Therefore two threads per block will yield really bad performance. The CPU would execute the code much faster.