# LSU EE 4702-1 Homework 2 Solution Due: 10 October 2012

*Follow the* Programming Homework Work Flow *instructions on the procedures page, substituting* hw2 *for* hw1.

The hw2 code, based on demo-8-texture, draws a helix. Due to missing normals, the surface of the helix will not be shaded properly (see Problem 1).

The comments at the top of the file describe controls for the simulation. For this assignment controls have been added to change the amount of detail in the helix (search for "Big Circle" and "Small Circle"), to change how vertices are ordered (look at the i and m options), and to force computation and communication (look at d and r).

For the solution code look for file hw2-sol.cc in the repository.

**Problem 1:** Run the unmodified hw2 code and pay attention to the way the helix is lit. Notice that the shading of the helix is wrong. That's because normals have not been specified. Modify the code so that normals are correctly specified. To do this new arrays and a new buffer object will have to be added.

The normal itself can easily be found by constructing a vector from the helix point (n0) and the wire surface point (s0). That vector is constructed in part of the expression used to compute s0 in the first place, see the code below under Before. In the solution, shown below under Solution, the vector is assigned to a separate variable, p0s0, and that is used both to compute s0 and to compute the normal. Because of the way the helix is constructed the length of p0s0 is small_r, so to compute the normalized normal just divide each element of p0s0 by small_r. The solution actually multiplies by the reciprocal of small_r.

```
// Before: Computation of wire surface point:
pCoor s0 = p0 + cos(theta) * n0 + sin(theta) * b;

// Solution:
pVect p0s0 = cos(theta) * n0 + sin(theta) * b;
pCoor s0 = p0 + p0s0;
pVect norm0 = small_r_inv * p0s0;
```

**Problem 2:** The use of triangle strips reduces vertex redundancy compared to individual triangles. However for the hw2 program there is still redundancy. In particular, point s1 (search for "Construct a Helix" and scroll down to the j loop) is the same as s0 in a prior i iteration.

By using glDrawElements instead of glDrawArrays one can avoid this redundancy. That is, the command sends each point to the GPU once and a separate index array specifies which of those points are used, a point can be used multiple times.

(*a*) Modify the code to use glDrawArrays when variable opt_use_elements is true. (Use key m to change the value of opt_use_elements.) Some code is already in place, including code declaring and initializing the index array and the call to glDrawElements. However, the existing code still adds the same point to the helix_coords list multiple times. The index array (indices) is set to 0, 1, 2, ..., which makes glDrawElements equivalent to glDrawArrays.

Modify the code so that each point is added only once, making appropriate changes to prep_indices (which is used to set indices) so that the helix is drawn correctly. See Section 10.5 of OpenGL version 4.3 for use glDrawArrays.

In the original code the inner loop added two coordinates to prep_coords and the corresponding indices to prep_indices. See the code below under before. In the solution, see below under Solution, only one coordinate, s0, is added to the list. Two indices are added, idx for s0, the other for a coordinate that will be added later, idx + seg_per_small_circle.

```
// Before:
              // Compute point on surface of wire.
              pCoor s0 = p0 + cos(theta) * n0 + sin(theta) * b;
              prep_coords += s0;
              prep_indices += idx; // Index of point just added.

              pCoor s1 = p1 + cos(theta) * n1 + sin(theta) * b;
              prep_coords += s1;
              prep_indices += idx + 1;  // Index of point just added.

// Solution
              pCoor s0 = p0 + p0s0;
              prep_coords += s0;
              prep_indices += idx; // This vertex.
              prep_indices += idx + seg_per_small_circle;
```

The solution initially prepares the arrays assuming that **opt_use_elements** is true. If it's false then it will construct a new list of coordinates using the indices. See the checked in code.

(*b*) The code has a variable `opt_use_strips`. Modify the code so that when it's true the helix will be rendered using triangle strips and when it's false it will be rendered using individual triangles. *Note: This part did not appear in the original assignment.*

When **opt_use_strips** is false the helix will be rendered using individual triangles. For these there must be three indices per triangle. So rather than inserting two indices per iteration, six indices are inserted. See the excerpt below:

```
// Solution
              if ( opt_use_strips ) {
                  prep_indices += idx; // This vertex.
                  prep_indices += idx + seg_per_small_circle;
                } else {
                  prep_indices += idx; // This vertex.
                  prep_indices += idx + seg_per_small_circle;
                  prep_indices += idx + seg_per_small_circle + 1;

                  prep_indices += idx; // This vertex.
                  prep_indices += idx + seg_per_small_circle + 1;
                  prep_indices += idx + 1;
                }
```

**Problem 3:** In this problem try to measure the impact of using `glDrawElements` to reduce the redundancy.

(*a*) Compute the amount of data sent from the CPU to the GPU in terms of the number of triangles used in the helix with and without `glDrawElements`. Use $T$ to denote the number of triangles.

Each index (element of **indices**) is 4 bytes, each vertex coordinate (three floats) is 12 bytes, and each normal is 12 bytes. (The color and other attributes are the same for all vertices, so they do not significantly affect the data size.) In the solution below *vertex* refers to both the vertex coordinate and the normal.

For **glDrawArrays** with triangle strips: A total of $T$ indices and $T$ vertices are sent, the data size is therefore $4T + 24T = 28T$ bytes.

For **glDrawElements** with triangle strips: A total of $T$ indices are sent, but only $T/2$ vertices are sent. The total amount of data is $4T + 24T/2 = 16T$ bytes.

For **glDrawArrays** with individual triangles: A total of $3T$ indices and $3T$ vertices are sent, the data size is therefore $3 \times 4T + 3 \times 24T = 84T$ bytes.

For `glDrawElements` with individual triangles: A total of $3T$ indices and $T/2$ vertices are sent, the data size is therefore $3 \times 4T + 24T/2 = 24T$ bytes.

($b$) Measure the impact on performance of this difference in data size. Use the various program options to make this performance difference large. Explain which options were used and show the results.

*Background:* There are two impacts for data size, communication between the CPU and GPU, which has already been discussed in class, and GPU memory and the GPU itself, which has not been discussed much. The solution will discuss both impacts, however the assignments were graded only on consideration of CPU/GPU communication.

For the homework code, communication between the CPU and GPU is controlled by the "Always Send" option. When this option is "no" indices and vertices are only sent when some helix parameter, such as "Small Circle" changes. This data will be sent once per change and so its impact on performance will be very brief. In other words, when "Always Send" is "no" data communication between the CPU and GPU is small, consisting of commands. When "Always Send" is "yes" the impact of data size on performance will be proportional to the array sizes (see previous problem).

In contrast, the sending of the indices and vertices from GPU memory to the GPU must occur every frame. The rendering pipeline is implemented in the GPU itself, the data that it reads (vertices, etc) initially is in GPU memory. The *vertex puller* will read the appropriate vertices from memory and send them to the first stage of the rendering pipeline, the vertex shader. The `glDrawArrays` is used the vertex puller reads vertices sequentially. When `glDrawElements` is used the puller will read the indices sequentially but fetch vertices corresponding to the indices, which can be in any order, and the same vertex can be pulled multiple times.

For `glDrawArrays` the amount of data read is equal to the size of the index and vertex arrays. (The `glDrawArrays` command does not use the indices, but our vertex shader uses them.) For `glDrawElements` the amount of data read from memory depends upon how clever the system is. Consider two cases, *simple* and *"efficient"*. In the simple case, one vertex will be read for each index. The total amount of data read will be the number of elements in the index array times the sum of the sizes of an index and a vertex. For the code in this problem using triangle strips, there will be $T$ index elements, an index is 4 bytes and a vertex coordinate/normal pair is 24 bytes, so the data read from memory will be $28T$ bytes (this is the number under the $M->S1$ column in the table below). Notice that this is more than the $16T$ bytes send from CPU to GPU, that's because each vertex is read twice in the simple case for triangle strips. For individual triangles the difference between CPU to GPU transfer and GPU memory to shader transfer is even greater. In this case there are $3 \times 4T$ bytes read from GPU memory for indices and $3 \times 24T$ bytes for vertices, for a total of $84T$ bytes. This difference in the amount of data read should be reflected in GPU execution time.

For the "efficient" case each vertex will be read from GPU memory once, even if its index appears multiple times. In that case the amount of data read from GPU memory is the same as the amount of data sent from CPU to GPU. For triangle strips that's $16T$ bytes, compared to $28T$ for the simple method. There are quotes around "efficient" to hint that the reduction in data volume over the simple method might come at some cost. That is the cost of holding on to the fetched vertices (before or after vertex processing) until their last use.

*Solution:* To determine the impact on performance with data size the code was executed for two helix sizes, both have more triangles than the default. The primitive type (triangle strip or individual triangles), draw command (`glDrawArrays` or `glDrawElements`) was varied, and data-resend options were varied. For reference, the data transfer sizes also appear in the table. Column `M->S1` shows the amount of data read from GPU memory to the GPU for the simple case, where the number of vertices read is equal to the number of indices. The data under `M->S2` shows the amount of data read for the "efficient" case, where the number of vertices read is equal to the number of *distinct* indices (which is equal to the number of vertices in the array).

```
Key: Strip, Triangle Strips;  Array, glDrawArrays
     Indiv, Individual Tri;   Elmnt, glDrawElements
     0: Only re-send data for changes.   1: Always re-send data.
     C->G:  Amount of data sent from CPU to GPU.
     M->S1: Amount of data sent from GPU memory to shader, assumption 1.
     M->S2: Amount of data sent from GPU memory to shader, assumption 2.

Small Circle: 40  Big Circle: 70 - Only recompute on change
 Config             GPU  CPU  Data  Data  Data
                              C->G  M->S1 M->S2
 Strip Array 1      2.6  0.9  28T   28T   28T
 Indiv Array 1      4.5  1.9  84T   84T   84T
 Strip Elmnt 1      2.3  0.7  16T   28T   16T
 Indiv Elmnt 1      2.5  0.8  24T   84T   24T
 Strip Array 0      1.9  0.4  0     28T   28T
 Indiv Array 0      2.5  0.4  0     84T   84T
 Strip Elmnt 0      1.9  0.4  0     28T   16T
 Indiv Elmnt 0      1.9  0.4  0     84T   24T

Small Circle: 100  Big Circle: 100 - Only recompute on change.
 Config             GPU  CPU  Data  Data  Data    # of Vtx
                              C->G  M->S1 M->S2   Shader Runs
 Strip Array 0      2.8  0.4  0     28T   28T     T      T
 Indiv Array 0      5.0  0.4  0     84T   84T     3T     3T
 Strip Elmnt 0      2.8  0.4  0     28T   16T     T      0.5T
 Indiv Elmnt 0      2.9  0.4  0     84T   24T     3T     0.5T
```

Look first at the "Always Send" entries, those with a 1 in the config column. Both the CPU and GPU times are roughly proportional to the amount of data sent. The worst time is for individual triangles with `glDrawArrays`, the time for the other options are much better.

When "Always Send" is "no" CPU time does not vary with the other options, as expected. What's interesting is the GPU time. It's only large for individual triangles with arrays. The effect is also seen in the second table, which has data on a helix with a larger number of triangles. The GPU time is consistent with the "efficient" case.

(*c*) Try to determine if there is any difference in performance due to computation. There would be less computation using `glDrawElements` if the vertex shader were run on each vertex only once. But does that actually happen? Explain which options were used and show the results.

The second table above also includes columns showing the number of times the vertex shader is run per triangle, based on different assumptions. The first `Vtx` column shows the number of vertex shader runs assuming that the vertex shader is run once per index, the second column is the number of runs under the assumption that it is run once per vertex. The last two rows provide the best basis for comparison. The slightly lower performance of the individual triangle row might be due to the $3T$ shader runs (versus $T$ for the strips) but it also might just be due to the amount of memory read.