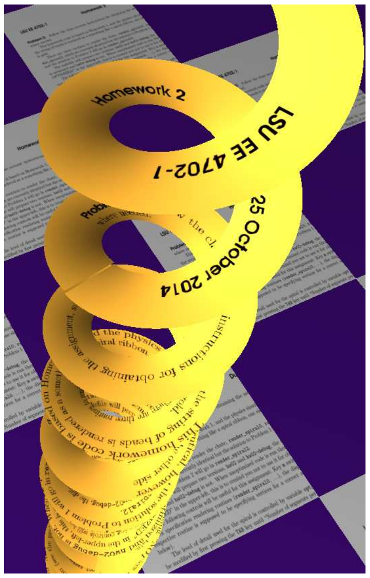

Problem 0: Follow the class account instructions for obtaining the assignment, substitute hw02 where needed.

This homework code is based on Homework 1, and the physics simulation is identical, however the string of beads is rendered as a something like a spiral ribbon, one side is purple, the other side is gold.

There are three routines to render the chute, `render_spiral0`, `render_spiral1`, and `render_spiral2`, which are currently identical but the solution to Problem 1 will go in `render_spiral1` and the solution to Problem 2 will go in `render_spiral2`.

The makefile will prepare two versions, `hw02` and `hw02-debug`, the difference being that `hw02` is optimized and `hw02-debug` is not. When unoptimized code is run there will be a flashing "NOT OPTIMIZED" in the upper-left, this is to remind you not to use it for obtaining performance data.

The following controls will be useful for this assignment: Key `m` switches between the different vertex specification rendering routines (`render_spiral0`, ...), the display will indicate how the respective routine is *supposed to be* specifying vertices for a correct solution (see the problems below).

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. The image on the upper right shows the appearance of the spiral after a correct solution to Problem 2.

The source code for the solution is in the repo under the name `hw02-sol.cc` and at `http://www.ece.lsu.edu/koppel/gpup/2014/hw02-sol.cc.html`

Problem 1: Examine the code in `render_spiral1`. There you will find that individual triangles are used to render the chute.

(*a*) Modify the code so that it uses triangle strips. Be sure to do this correctly, the ribbon should be gold on one side and purple on the other and fewer vertices should be sent to the GPU.

In the original version of the code each iteration of the `t` loop rendered two triangles by sending six vertices. In the solution a triangle strip is used for the whole `t` loop, with each iteration providing two vertices, the new inner and outer vertices computed in the iteration.

A common error was to put the `glBegin/glEnd` inside the `t` loop (not around it). The original code put the `glBegin/glEnd` pair around just two triangles. The solution puts these around the entire segment (`t`) loop. Thus a triangle strip has $2s$ triangles specified using $2s+2$ vertices. Putting `glBegin/glEnd` inside the `t` loop wastes most of the potential of the triangle strip. If that were done the $2s$ triangles would be rendered using $4s$ vertices, almost twice as much.

Another common error was to retain the `if ( t > 0 )` which prevented vertices being emitted on the first iteration. Since we are using triangle strips we need those first two vertices.

(*b*) Measure how much of a performance difference triangle strips make. (Do this by comparing to `render_spiral0` using the `m` key.)

The measurement below is based on the optimized code (the one without **-debug** in the file name. It ran on an NVIDIA Quadro K2100M. The number of segments was increased to 100 to make it easier to measure the graphics load. For individual triangles the reported CPU and GPU times were $5.4\,\mathrm{ms}$ and $1.35\,\mathrm{ms}$. For strips they were $2.5\,\mathrm{ms}$ and $0.77\,\mathrm{ms}$. So, by switching to triangle strips the GPU performance was $2.16\times$ better and the CPU time was $1.75\times$ better.

(*c*) Estimate the amount of data sent from CPU to GPU for the two versions of the code.

Let s denote the number of segments (value of **opt_segments**) and let n denote the number of balls (the value of **chain_length**). Each spiral segment (the part that's computed by an iteration of the **t** loop) consists of two triangles. The total number of triangles is therefore $2sn$.

For each vertex the following information is sent: a texture coordinate, a normal, and the vertex coordinate. The vertex coordinate and normal each have three components, the texture coordinate has two components, for a total of 8 components, each of which is a four-byte float. So for each vertex $8 \times 4\,\mathrm{B} = 32\,\mathrm{B}$ are sent.

When specifying vertices using individual triangles, three vertices are sent for each triangle. The total data is $3 \times 2sn \times 32\,\mathrm{B} = 192sn\,\mathrm{B}$ per frame. For triangle strips, the number of vertices sent is two plus the number of triangles, or $2 + 2sn$. The total amount of data is $(2 + 2sn)32\,\mathrm{B} \approx 64sn\,\mathrm{B}$ per frame, a bit more than a third of the amount sent when using individual triangles.

(*d*) Performance should be better with the triangle strips. How much was the improvement from `render_spiral0` to `render_spiral1` affected by:

- reduction in computation by the CPU and GPU? *Note: The original question did not include the phrase "by the CPU and GPU".*

A lot for the GPU. That is, the vertex shader (the fixed function vertex shader for this homework) is run three times more frequently when using individual triangles. The CPU does about the same amount of computation in both cases. For example, it computes **norm_inner** the same number of times in **render_spiral0** and **render_spiral1**.

- reduction in data sent to the GPU?

A lot. One third as much data is sent.

- reduction in work performed by the fragment shader (or fixed function equivalent)?

None. The number of triangles hasn't changed, and therefore the number of fragments has not changed either, and so the fragment shader does the same amount of work in both cases.

Problem 2: The code in `render_spiral0` and `render_spiral1` apply a texture to the spiral. The texture is a draft of this homework assignment. The code applies the texture so that the top and bottom of the page is mapped to the inner- and outer- part of the spiral. Modify `render_spiral1` so that the width of the page is one over the chain length and as one goes down the spiral one goes down the page. (As though the page were cut into horizontal strips and those strips were connected to form a spiral.) See the illustration above.

Based on the description above, the width of the page should span the **t** loop. That is, for **t=0** the texture coordinate for the left-hand side of the page should be used, for **t=0.5** the texture coordinate for the middle of the page should be used, etc. This matches the coordinate space for textures, so the s component of the texture (akin to the x axis) can be set to **t**. The t component (akin to the y axis) of the texture (which has nothing to do with variable **t**) should be based on variable **i**. For **i=1** we want to be at the top of the page, meaning t is zero, for **i=chain_length** we want t to be 1.

The problem below was originally assigned as Homework 2 Problem 3 but was later reassigned as Homework 3 Problem 1.

Problem 1: In this problem use buffer objects to store vertices for one spiral section and transformation matrices to render all of them.

(*a*) Modify `render_spiral2` so that it uses buffer objects. (See the code in `demo-7-vtx-arrays.cc` for example of how to do that.)

A spiral section is a spiral going from one ball to the next, it is what is constructed inside of the `t` loop. Use buffer objects to store just one spiral section.

Then use transformation matrices to transform the spiral into the buffer object into the one that you need. For your convenience a routine `make_transform(f1,f2,t1,t2)` is available which will transform a spiral between points `f1` and `f2` to a spiral between points `t1` and `t2`.

(*b*) Measure the performance improvement over the two methods used in Homework 2.