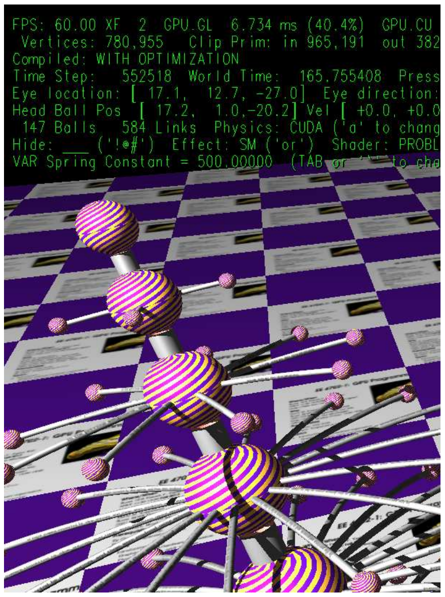
LSU EE 4702-1 Homework 2 Due: 3 October 2017

Problem 0: Follow the instruction on the `http://www.ece.lsu.edu/koppel/gpup/proc.html` page for account setup and programming homework work flow. Compile and run the homework code unmodified. It should initially show a column of beads swaying back and forth. The illustration to the right shows Scene 4 after Problem 2 is completed. Pressing `h` (head) will release one end (to be precise, the ball at one end) and pressing `t` (tail) will release the other end. (Actually, those keys toggle the fixed-in-space status of their respective balls.)



```
FPS: 60.00 XF  2  GPU.GL  6.734 ms (40.4%)  GPU.CU
 Vertices: 780,955   Clip Prim: in 965,191  out 382
Compiled: WITH OPTIMIZATION
Time Step:   552518  World Time:  165.755408  Press
Eye location: [ 17.1,  12.7, -27.0] Eye direction:
Head Ball Pos  [ 17.2,  1.0,-20.2] Vel [ +0.0, +0.0
 147 Balls  584 Links  Physics: CUDA ('a' to chang
Hide: ___ ('!@#')  Effect: SM ('or')  Shader: PROBL
VAR Spring Constant = 500.00000  (TAB or
```

Pressing digits `1` through `4` will initialize different scenes, the program starts with scene 1.

The scenes differ in the number of objects, which include spheres, links, and the platform (which for this assignment we'll consider one object). The rendering of objects by type can be toggled on and off by pressing `!`, `@`, `#`, for spheres, links, and the platform. See the green text line starting with Hide. The scene includes shadow and mirror effects. These can be toggled by pressing `o` and `r`, their state is shown next to Effect. Pressing `n` toggles between using a single normal for each triangle in a sphere (making the location of the triangles obvious), and using the interpolated normals associated with the triangles (making the surface appearance much more sphere-like).

The top green text line shows performance. XF shows the number of display updates per frame buffer update. An ideal number is 1. A 2 means that two display updates were done in the time needed to update the frame buffer, presumably because the code could not update the frame buffer fast enough. GPU.GL shows how long the GPU spends updating the frame buffer (per frame), GPU.CU shows how long the computational accelerator takes per frame. The computational accelerator computes physics in this assignment. On the lab computers the computational accelerator GPU is different than the one performing graphics. CPU GR is the amount of time that the CPU spends on graphics, and CPU PH is the amount of time that the CPU spends on physics.

The second line, the one starting with Vertices, shows the number of items being sent down the rendering pipeline per frame. Clip Prim shows the number of primitives before clipping (in) and after clipping (out).

The program can use three different renderers for the spheres, INSTANCE, PROBLEM 1 (render_p1), and PROBLEM 2 (render_p2). Pressing `v` switches between them.

In this assignment different ways of sending vertices for a sphere will be compared.

Initially the arrow keys, PageUp, and PageDown can be used to move around the scene. Press (lower-case) `b` and then use the arrow and page keys to move the first ball around. Press `l` to move the light around and `e` to move the eye (which is what the arrow keys do when the program starts).

Pressing `Ctrl+` increases the size of the green text and `Ctrl+` decreases it. The `+` and `-` keys can be used to change the value of certain variables to change things like the light intensity, spring constant, and variables needed for this assignment. The variable currently affected by the `+` and `-` keys is shown in the bottom line of green text. Pressing `Tab` cycles through the different variables. Those who want to increase the spring constant to the point that the scene explodes may be disappointed to learn that there is a protection mechanism that increases the mass of balls when the spring constant is high enough to make the system go unstable, such balls turn red.

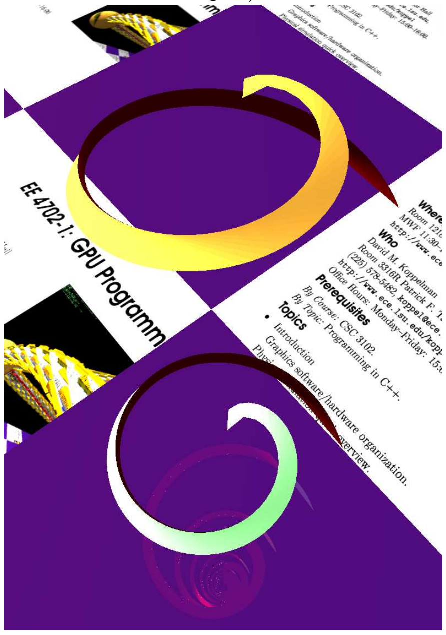
Look at the comments in the file `hw02.cc` for documentation on other keys.

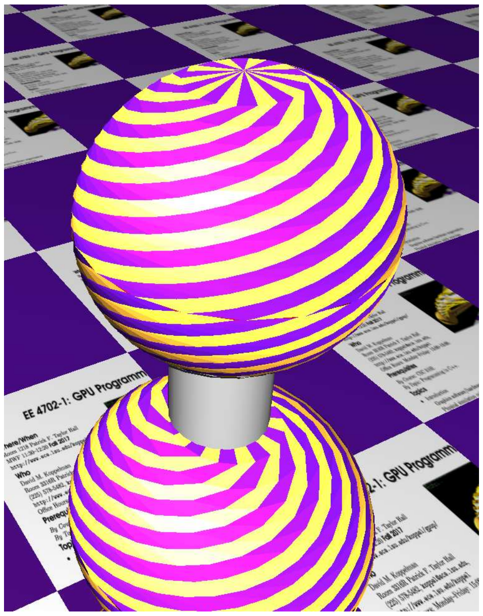
Problem 1: The code in `render_p1` contains code adapted from the `demo-7-vtx-arrays.cc` classroom demonstration for computing sphere coordinates and putting it in a buffer object. That code is followed by a loop that finds information about each ball to render. Unmodified, `render_p1` will render a single ball at the origin. Modify `render_p1` so that it efficiently renders each ball at the appropriate location and orientation and with the appropriate color, and does so using the prepared coordinates in the buffer object.

Assume that the number of balls to render can be large, and so try to avoid doing things multiple times that only need to be done once.

- Remember to set the color.
- Use the radius, position (center), and orientation (rot) appropriately.

Problem 2: The code in `render_p2` is similar to the code in `render_p1` except that the buffer object is initialized with just a single triangle. Modify the code so that the buffer object holds the coordinates of a spiral slice of a sphere and use these slices to render an entire sphere with every other slice colored purple. Individual spiral slices are illustrated below to the left. (The spiral starts with a visible triangle rather than a smooth curve because it is constructed from triangles and the first triangle is relatively large.) The screenshot to the right shows sphere rendered using these slices (though with slightly different parameters so the individual spirals to the left aren't exactly the same shape as the ones to the right).





The shape of a spiral slice is defined as follows. Let

$$P(\Delta_\theta, t) = \begin{bmatrix} \sin(\eta(t))\cos(\theta(t)+\Delta_\theta) \\ \cos(\eta(t)) \\ \sin(\eta(t))\sin(\theta(t)+\Delta_\theta) \end{bmatrix},$$

where $\eta(t) = \frac{\pi}{2}\sqrt{t}$, $\theta(t) = \omega\sqrt{t}$, and ω is the value of variable `opt_omega`. The line $P(\Delta_\theta, t)$ for $0 \leq t \leq 1$ follows the surface of a sphere from the top to the equator making a spiral on its way down. In the illustration above the two long spiral edges follow $P(0,t)$ and $P(\frac{2\pi}{p}, t)$ for $0 \leq t \leq 1$, where p is the value of variable `opt_spirals`. (The third, short, edge is the part of the sphere equator between the two lines.)

(*a*) Modify `render_p2` so that the buffer object is filled with the coordinates of one spiral in the order needed for a triangle strip rendering pass. Generate coordinates for `opt_slices` values of t from 0 to 1 (inclusive). The code in `render_p2` already has a loop that computes the correct values of t and $\sqrt{t}$.

The back color of triangles has been set to red. The front color will usually be gold, and for the next part sometimes purple. If your sphere is red that means it's inside out.

(*b*) Render a sphere by rotating and reflecting this spiral. A hemisphere can be rendered by rendering the spiral p times and rotating the spiral by $\frac{2\pi}{p}i$ on the i'th rendering, where p is the value of `opt_spirals`. When i is odd use the ball's own color, when i is even use purple for the color. The whole sphere is rendered by rendering a hemisphere and then rotating 180 along the x or z axis. So to render one sphere $2p$ rendering passes are used, each rendering passes uses the same buffer object for coordinates, but has the modelview matrix set for the rotation.

- As in Problem 1, render all of the spheres in the `balls` list.
- Make sure that the balls rotate appropriately. In scenes other than 1 it is easy to see whether ball rotation is correct.
- Don't forget to alternate color between purple and the ball's color.
- As with the first problem, avoid doing unnecessary work. For example, don't copy data into a buffer object that already contains the data it needs.

The three different rendering methods differ in the number of rendering passes needed per frame to render the spheres. In Homework 3, the impact of the rendering passes on CPU and GPU time will be examined, and this will be put in the context of other tasks performed by the CPU and GPU.