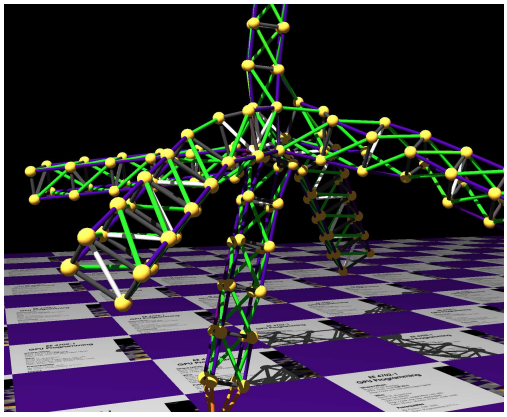
LSU EE 4702-1 **Homework 2** Solution **Due: 22 September 2015**



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 show a swinging string of beads. Press digits `1` through `3` to initialize different scenes, the program starts with scene 1. The illustration to the right is from scene 3 generated with a solution to Problem 1. *Promptly report any problems.*

Unlike the code used for Homework 1, in this version when you manually move the head ball motion is smooth. That is, it doesn't jump to it's new location in one time step, rather it moves smoothly over 250 ms. This will help with Problem 1. Another change is in the collision resolution with the platform. In the Homework 1 code and some of the classroom demos, a ball colliding with the platform would have the y component of its velocity flipped and its position snapped to $y = 0$. In the unmodified Homework 2 code virtual springs are used to compute separation forces for balls dropping beneath the platform.

Use the arrow keys, PageUp, and PageDown to move around the scene. Use key `h` to toggle between the first (head) ball being locked in place and free. Use key `t` to do the same for the last (tail) ball. 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.

Look at the comments in the file `hw02.cc` for documentation on other keys. One fun thing to do is to lock both the first and last ball, move the head ball until the spring is stretched tight, then release one of the balls. Press `p` to pause, then the space bar to single step. *Note: There is nothing to turn in for this first problem.*

The solution code for this assignment has been checked into the repository in file `hw02-sol.cc`. For an HTML version visit `http://www.ece.lsu.edu/koppel/gpup/2015/hw02-sol.cc.html`. The repository version will always be most up to date.

Problem 1: The balls and links code used in Homework 1 and classroom demos simulated indestructible links. In this problem modify the Homework 2 code so that links can snap. The illustration at the top of this assignment shows a screenshot from a solution.

First, define two variables in `My_Piece_Of_The_World`, one indicating a stressed threshold and one indicating a snap threshold. These might refer to a ratio of current length to relaxed length. If a link exceeds the snap threshold set the `snapped` member of `Link` to `true`. Modify the time step routine so that forces for snapped links are ignored. (The snapped links are not removed from the links list.)

If the link length is between the stressed threshold and snap threshold then proportionally change the color of the link to red. `Link` has two color members `natural_color` and `color`. `Link::natural_color` is set when a link is constructed and should not be changed. `Link::color` should be set based upon link stress. If the link is not stressed set `color` to `natural_color`. If it

is stressed set `color` to `(1-s) * natural_color + s * red`, where $0 \leq s \leq 1$ is the stress level. (Determining a value for `s` is part of the problem.)

You can test your code by freezing both the head and tail balls and then using the keyboard to move the head ball, stretching the object.

The solution bases the stress and snap threshold of a link on the ratio of its current length to the relaxed length. This is only used for links with non-zero relaxed lengths, links with a relaxed length of zero are considered indestructible.

For each non-snapped link, the code first determines if its current length, `distance_between_balls`, is above the stress threshold:

```
// First, check if link is at least stressed.
//
if ( link->distance_relaxed > 0.001 // Don't bother with very short links.
     && distance_between_balls
     > link->distance_relaxed * mp.link_stressed_thd )
```

If so, it checks whether it is snapped, and if so it sets the snapped member to `true`. It also sets the color based on the stress.

Physically simulating a snapped link is simple, just skip the code that simulates the link behavior, that's done below with a simple continue statement. (Another solution would be to remove snapped links from the `links` list.)

```
for ( LIter link(links); link; )
  {
    /// SOLUTION -- Problem 1
    //
    if ( link->snapped ) continue;

    // Spring Force from Neighbor Balls
    //
    Ball* const ball1 = link->ball1;
    Ball* const ball2 = link->ball2;
```

Grading Note: A common mistake was to use the global value of `distance_relaxed` rather than the link's value.

Problem 2: Pressing `v` will cycle through three views: Showing only the balls and links (the SKEL), showing only triangles (the SKIN), and showing both. In the unmodified code the triangles will be scattered around the trusses. Modify the code so that triangles are rendered on the surface of the trusses, and optionally other appropriate places, such as the caps of the top.

The code for actually telling OpenGL to render the triangles should be placed in `render_my_piece`, which is near the bottom of the file. It currently contains placeholder code which results in the scattered triangles.

A solution to this problem will require new code in `World::make_truss` to identify which balls form the surface and adds them to arrays, perhaps in `My_Piece_Of_The_Wolrd`. For example, you might add new arrays and fill them with Ball pointers that are in the correct order for rendering.

Modify `render_my_piece` and other routines so that the triangles are placed on the surface, and so that normals are set correctly. This code might use the arrays that you prepared in `make_truss`.

Note that the arrays need to save pointers to Balls, not coordinates, because the balls move and so any coordinates copied during truss construction would result in a skin that doesn't move.

Try to render the skin using strips, rather than individual triangles or other individual primitives (hint).

The key to this problem is getting a list of vertices—ball positions—in the correct order for rendering. The solution does so by adding code to the **make_truss** routine. (Yes, you need to read and understand that routine.) A new loop nest is added with the outer loop iterating over sides and the inner loop iterating along the side of the truss. A new class, **Strip**, is defined to hold the balls for a side of the truss. For a truss with four sides (the default), four strips would be used. In addition to ball pointers, **Strip** holds a color. A new class **Strips** is defined to hold the strips, that's added to **Truss_Info** and **My_Piece_Of_The_World**.

```
for ( int j=0; j<num_sides; j++ ) {
    Strip* const strip = new Strip(color_chocolate);
    truss_info->strips += strip;
    for ( int i=0; i<num_units; i++ )
      {
        const int idx = j + num_sides * i;

        // Compute the index of the ball at (i, (j-1) mod num_sides ).
        //
        const int pn_idx = idx + ( j == 0 ? num_sides - 1 : -1 );

        // Add the ball corresponding to (i,j).
        //
        strip->balls += bprep[idx];
        strip->balls += bprep[pn_idx];
      } }
```

After each truss is constructed the contents of **strips** for the truss is added to the **strips** for our piece of the world. (Sure, this step would not be necessary if there was only one **strips**, and the code in **make_truss** just added directly to that.)

The contents of **strips** is used by code in **render_my_piece** to render a skin. A loop iterates over the strips, and the loop body starts by allocating, if necessary, storage for coordinates and normals

```
for ( SIter strip(mp.strips); strip; )
  {
    const int elts = strip->balls.size();
    if ( elts > buffer_elts )
      {
        buffer_elts = elts;
        coord_buffer =
          (pCoor*) realloc(coord_buffer,elts*sizeof(coord_buffer[0]));
        norm_buffer =
          (pVect*) realloc(norm_buffer,elts*sizeof(norm_buffer[0]));
      }
```

It then fills the two arrays: **coord_buffer**, the array of vertex coordinates, and **norm_buffer**, the array of normals. The coordinates are simply copied from the ball object:

```
    for ( int i=0; i<elts; i++ )
      coord_buffer[i] = strip->balls[i]->position;
```

Normals have to be computed. If the trusses didn't bend, it would be easy to compute a normal, just take the cross product of three ball positions and use that for an entire strip. But since the links change length normals can vary. In the solution below normals are computed based on several neighboring triangles.

```
    for ( int i=0; i<elts-2; i+=2 )
```

```
{
  pVect vlateral0(coord_buffer[i],coord_buffer[i+1]);
  pVect vlateral1(coord_buffer[i+2],coord_buffer[i+3]);
  pVect vleft(coord_buffer[i],coord_buffer[i+2]);
  pVect vright(coord_buffer[i+1],coord_buffer[i+3]);
  pNorm nleft0(cross(vlateral0,vleft));
  pNorm nright0(cross(vlateral0,vright));
  pNorm nleft1(cross(vlateral1,vleft));
  pNorm nright1(cross(vlateral1,vright));
  if ( i == 0 )
    {
      norm_buffer[0] = nleft0;
      norm_buffer[1] = nright0;
    }
  norm_buffer[i] += nleft0;
  norm_buffer[i+1] += nright0;
  norm_buffer[i+2] = nleft1;
  norm_buffer[i+3] = nright1;
  if ( i == elts - 3 )
    {
      norm_buffer[i+2] += nleft1;
      norm_buffer[i+3] += nright1;}}
```

Once the arrays are set up it's a simple matter to render them. (Note that buffer objects are not being used because the vertex and normal arrays are each used just once.)

```
glEnableClientState(GL_VERTEX_ARRAY);
glVertexPointer(3, GL_FLOAT, sizeof(pCoor), coord_buffer);
glEnableClientState(GL_NORMAL_ARRAY);
glNormalPointer(GL_FLOAT, 0, norm_buffer);
glColor3fv(strip->color);
glDrawArrays(GL_TRIANGLE_STRIP, 0, elts);
glDisableClientState(GL_NORMAL_ARRAY);
glDisableClientState(GL_VERTEX_ARRAY);
```

Problem 3: The unmodified code does not simulate friction along the platform. Modify it so that it does. The time step routine uses a virtual spring to compute a separation force for any ball moving under the platform. Use this separation force to compute a friction force, and apply the friction force in the opposite direction along the platform.

The existing code computes a quantity, `force_up`, for a ball in contact with the platform. It is the force exerted by the platform on the ball. In this discussion let f_u be `force_up`.

We need to compute a frictional force, f_s, which will slow down sliding along the platform. Because of the way frictional forces work, the magnitude of f_s will be $k\|f_u\|$, where k is a friction coefficient. For no particular reason, the solution sets $k = 0.1$. The friction force magnitude is computed by:

```
const float fric_coefficient = 0.1;
const float fric_force_mag = fric_coefficient * force_up;
```

The direction of the frictional force will be in the direction of the ball's motion along the platform. Since the platform is aligned with the xz plane that velocity is obtained by setting the y component to zero. The code below computes the direction of this velocity:

```
pNorm surface_v(ball->velocity.x,0,ball->velocity.z);
```

The change in velocity due to the frictional force is computed for the time step:

```
const float delta_v_surf = fric_force_mag / mass * delta_t;
```

If this ΔV were small we could then add it on to the `delta_v` computed for the ball based on other forces (gravity and the links), using the direction vector we computed above:

```
    delta_v -= delta_v_surf * surface_v;
```

But, what if it's large? Frictional forces can reduce the velocity to zero, but they can't reverse it. So we check whether friction will reverse velocity, and if so we set `delta_v` so that velocity along the platform goes to zero:

```
if ( delta_v_surf > surface_v.magnitude )
    delta_v = pVect(-ball->velocity.x,delta_v.y,-ball->velocity.z);
else
    delta_v -= delta_v_surf * surface_v;
```