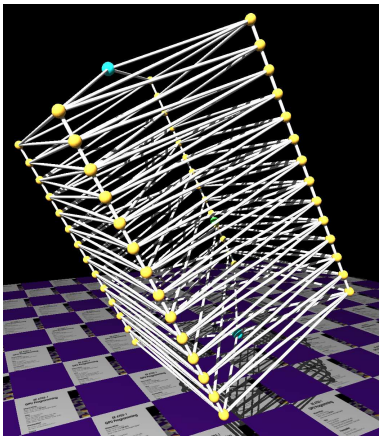
LSU EE 4702-1 **Homework 1** Solution **Due: 14 September 2015**

Some of the effort for this homework assignment is in learning to use the various pieces of software, such as a text editor. Those less familiar with Linux software development procedures might seek out a more knowledgeable classmate to minimize frustration and wasted time.

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 `4` to initialize different scenes, the program starts with scene 1. The illustration to the right is from scene 3. *Promptly report any problems.*

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 `hw01.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.*



Problem 1: Pressing `2` shows scene 2, a truss and three marker balls. The truss should fall but the marker balls should stay in place. Modify the code in `balls_setup_2` so that the ends of the truss (the peak of the pyramid at each end) are initially in the same position as the cyan marker balls and so that a longitudinal (long direction) link passes through the green marker ball, see the screen shot to the right.

The locations of the marker balls are declared at the top of `balls_setup_2`, and a little further below there is a place for a solution. The problem can be solved by initializing variables `spacing`, `delta_unit`, `loc_x`, and `loc_z` correctly.

Of course, the truss must be between the head and tail balls and should have `chain_length` units. The modified code should make good use of the coordinate classes. For example, DON'T set `x`, `y`, and `y` members individually.

The mathematics for this problem is fairly simple: you need to find the vector pointing from `head_pos` to `tail_pos`, call that vector the *axis*. The vector `delta_unit` should be some fraction of the axis vector. Find the closest point on the axis to `surface_point`, call it S. Use S to find `loc_x`, `loc_y` and `spacing`.

To test out a solution it might be helpful to pause simulation before switching to scene 2. Then switch to scene 2 and check whether the cyan balls are at the head and tail locations and whether the green ball is on a longitudinal link.

To solve this problem properly, one must understand how the truss is constructed by `make_truss`. The structure `truss_info` passed to `make_truss` has a member holding a set of positions, `base_coors`. In the first iteration inside `make_truss` a set of balls will be placed at those positions. In the second iteration a second set of balls will be constructed and placed at those positions plus `truss_info.unit_length`. The number of iterations is determined by `truss_info.num_units`.

Let n denote the value of `num_units` and let δ denote the length of `unit_length`. The number of layers of balls will be n, and the length will be $(n-1)\delta$. (If $n = 2$ there would be two layers of balls and the distance would be just δ.)

As one can see by examining the code, the head ball is placed δ above the start of the truss and the tail ball is placed δ below the last layer of the truss. Let c be the value of `chain_length`. In order for there to be c layers $\delta = \frac{1}{c+1}\overrightarrow{P_H P_t}$, where P_H and P_T are the coordinates `head_pos` and `tail_pos`, respectively. The lines of code below compute δ, and also the position of the start of the truss, `top_pos`.

```
pNorm axis(head_pos,tail_pos);
float unit_length = axis.magnitude / ( chain_length + 1 );
pVect delta_unit = unit_length * axis;
truss_info.num_units = chain_length;
truss_info.unit_length = delta_unit;
pVect top_pos = head_pos + delta_unit;
```

The `j` loop is used to find the coordinates of the first layer of balls:

```
for ( int j=0; j<sides; j++ ) {
    const double angle = double(j)/sides*2*M_PI;
    pCoor chain_first_pos =
      top_pos
      + spacing * cos(angle) * loc_x
      + spacing * sin(angle) * loc_z;
    truss_info.base_coors += chain_first_pos; }
```

This loop computes coordinates on a circle with center `top_pos`, radius `spacing`, and on a plane defined by `loc_x` and `loc_z`, which you can think of as the x and z axes. Let P_s denote the coordinate `surface_pos`, and let A_S be the point on the line $P_H P_T$ closest to P_s. For one of the longitudinal links to pass through P_s we can set `loc_x` to $\widehat{A_S P_s}$ and `spacing` to the magnitude of $\overrightarrow{A_s P_s}$. The first coordinate computed by the `j` loop, call it P_0, will be directly "above" P_s. That is, $P_s = P_0 + t\overrightarrow{P_H P_T}$ for some t. The code below assigns `axis_s` to the point on the axis closest to P_s, and then computes `loc_x` and `loc_z`.

```
pVect s_to_head(surface_pos,head_pos);
const float axis_s_to_h_dist = dot(s_to_head,axis);
pCoor axis_s = head_pos - axis_s_to_h_dist * axis;
```

Problem 2: When `w` is pressed the simulator will call routine `balls_twirl`, which currently does nothing. Modify the routine `balls_twirl` so that it adds velocity to balls in a direction around the axis defined by the position of `head_ball` and `tail_ball` (if these are defined). The amount of velocity should be based on the distance from the axis, so that collection of balls rotates as a unit.

The mathematics for solving this problem are similar to the mathematics for the first problem. Write an equation for the line connecting the head and tail balls (the axis), say $S = B_0 + tv$, where B_0 is the first ball and v is a vector pointing towards the last ball. If we are choosing a velocity change for ball B, we need to find the point on the axis closest to B. For such a point $\overrightarrow{SB}$ will be orthogonal to v. One can use a property of the dot product to solve for S and another operation to find the force direction, a vector orthogonal to both v and $\overrightarrow{SB}$.

Solution appears below. Using the same technique used in Problem 1, find a point on the line from the head to tail ball (the axis) closest to the ball. Use that point to find a vector to the ball from the axis, and using a cross product find a vector along which to apply the force. Notice that the vector `axis_to_b` is not normalized. The change in velocity should be proportional to its length.

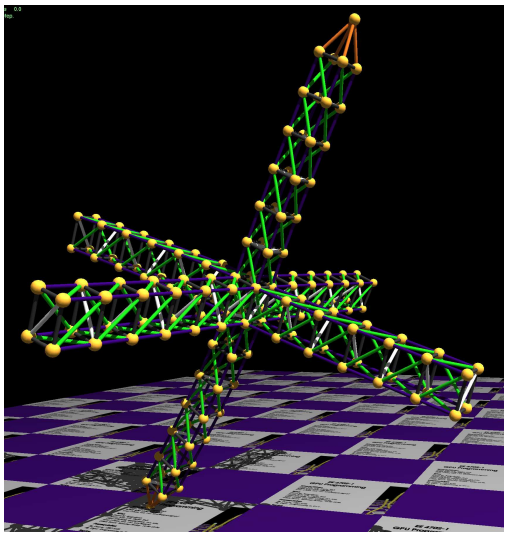
Grading Note: In many solutions `axis` *was computed inside the loop. This is wasteful because the head and tail ball don't change inside the loop and so there is no need to recompute it. (The compiler may or may not figure that out.)*

A common error was to make the change in velocity proportional to the distance from the head ball to the ball (rather than the distance of the ball to the axis).

```
pNorm axis(head_ball->position, tail_ball->position);
for ( BIter ball(balls); ball; ) {
    // Find point on axis closest to ball.
    //
    pVect b_to_top(ball->position,head_ball->position);
    const float dist_along_axis = dot(b_to_top,axis);
    pCoor axis_b = head_ball->position - dist_along_axis * axis;

    // Compute a vector to the ball, and use it to find direction
    // to apply force.
    pVect axis_to_b(axis_b,ball->position);
    pVect rot_dir = cross(axis_to_b,axis);

    // Apply change in velocity. The 10 is arbitrary.
    ball->velocity += 10 * rot_dir;
  }
```



Problem 3: Modify `ball_setup_4` so that it adds two or four more trusses to the existing truss, these should be neatly attached to the center of the original truss and should fall on a plane normal to the original truss. See the illustration to the right.

The truss should be well balanced, so that when it is twirled (using `w` from the previous problem) it balances like a gyroscope.

To solve this problem examine the code in routine `make_truss`, and also pay attention to the comments that describe how balls and links are added to the lists `balls` and `links`.

The basic idea is to create four new trusses (or `sides` new trusses) and attach them to the existing truss. The code added for the solution starts by finding the index of the middle layer of balls in the original truss (the one constructed in `balls_setup_4` before any new code is added), that's assigned to `idx_center`. This will be used to retrieve balls from `truss_info.balls` in order to find attachment points for the new truss. Each iteration of the `i` loop creates a new truss and attaches it to the original one. To start, the balls in the center of the original truss on side `i` are found

and set to `base_coors`. These balls are also used to compute `delta_dir`. Links are added to connect the original and new truss.

```
void World::ball_setup_4() {
  pCoor first_pos(13.4,17.8,-9.2);
  const float spacing = distance_relaxed;
  pVect delta_pos = pVect(spacing*0.05,-spacing,0);
  pNorm delta_dir = delta_pos;
  pNorm tan_dir = pVect(0,0,1);
  pNorm um_dir = cross(tan_dir,delta_dir);

  // Erase the existing balls and links.
  //
  balls.erase(); links.erase();

  Truss_Info truss_info;

  truss_info.num_units = chain_length;
  truss_info.unit_length = delta_pos;

  const int sides = 4;

  for ( int j=0; j<sides; j++ )
    {
      const double angle = double(j)/sides*2*M_PI;
      pCoor chain_first_pos =
        first_pos
        + 0.5 * spacing * cos(angle) * tan_dir
        + 0.5 * spacing * sin(angle) * um_dir;

      truss_info.base_coors += chain_first_pos;

    }

  make_truss(&truss_info);

  // Find the array index of the center of the original truss.
  // This will be used to attach the new trusses to the original truss.
  //
  const int idx_center = chain_length / 2 * sides;

  // Add a new truss to each side of the original truss.
  //
  for ( int i=0; i<sides; i++ )
    {
      Truss_Info ti;
      ti.num_units = chain_length / 2;

      // Find the attachment point indices corresponding to this side.
      //
      const int idx_1 = idx_center + ( i == 0 ? sides - 1 : i - 1 );
      const int idx_2 = idx_center + i;
```

```
    // Get pointers to the balls on the original truss that we are
    // going to attach the new truss to.
    //
    Ball* const b0 = truss_info.balls[idx_1];
    Ball* const b1 = truss_info.balls[idx_1 - sides];
    Ball* const b2 = truss_info.balls[idx_2 - sides];
    Ball* const b3 = truss_info.balls[idx_2];

    // Construct a new truss to its base balls exactly overlap
    // our chosen attachment points.
    //
    ti.base_coors += b0->position;
    ti.base_coors += b1->position;
    ti.base_coors += b2->position;
    ti.base_coors += b3->position;

    // Determine the unit_length for our new truss based on
    // the attachment points.
    //
    pNorm v_head = cross(b1->position,b2->position,b3->position);
    ti.unit_length = delta_dir.magnitude * v_head;

    make_truss(&ti);

    // Connect the new truss to the original truss.
    //
    links += new Link(b0,ti.balls[0],color_red);
    links += new Link(b1,ti.balls[1],color_red);
    links += new Link(b2,ti.balls[2],color_red);
    links += new Link(b3,ti.balls[3],color_red);

    // Add the links and balls from the new truss to the simulator's
    // links and balls lists.
    //
    links += ti.links;
    balls += ti.balls;
  }

// Insert links to balls at either end.
//
head_ball = balls += new Ball;
head_ball->position = first_pos - delta_pos;
for ( int j=0; j<sides; j++ )
  links += new Link( head_ball, truss_info.balls[j], color_chocolate );

tail_ball = balls += new Ball;
tail_ball->position = first_pos + chain_length * delta_pos;

const int bsize = truss_info.balls.size();
```

```
  for ( int j=0; j<sides; j++ )
    links += new Link( tail_ball, truss_info.balls[bsize-sides+j],
                       color_chocolate );

  for ( BIter ball(balls); ball; )
    {
      ball->locked = false;
      ball->velocity = pVect(0,0,0);
      ball->radius = 0.15;
      ball->mass = 4/3.0 * M_PI * pow(ball->radius,3);
      ball->contact = false;
    }

  balls += truss_info.balls;
  links += truss_info.links;

  opt_tail_lock = false;
  opt_head_lock = false;
}
```