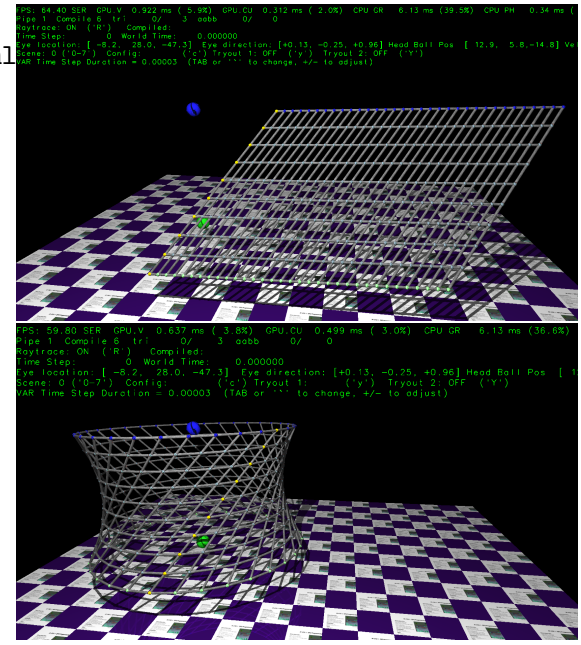
LSU EE 4702-1 **Homework 2** **Due: 25 September 2024**

All of the code for this assignment is in the course repo. HTMLized versions of the assignment file are at `https://www.ece.lsu.edu/koppel/gpup/2024/hw02.cc.html`.

Problem 0: Follow the instructions on the `https://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 grid, see the screenshot to the upper right. The grid is made up of balls connected by links, some are colored to help identify where they are, including yellow balls along the left-hand side. In the lower screen shot, which is from a correctly solved assignment, the balls and links form a shape called a *hyperboloid of one sheet*. Notice that the yellow line of balls in the two screenshots are in exactly the same place. The hyperboloid can be formed by rotating that yellow line of balls around a central axis. Problems 1 and 2 achieve this rotation in two different ways. The large green and blue balls mark the axis of rotation.



User Interface

Press `Ctrl=` to increase the size of the green text and `Ctrl-` to decrease the size. Press `F12` to generate a screenshot. The screenshot will be written to file `hw02.png` or `hw02-debug.png`. Press `F10` to start recording a video, and press `F10` to stop it. The video will be in file `hw02-1.webm` or `hw02-debug-1.webm`.

Initially the arrow keys, `PageUp`, and `PageDown`, can be used to move around the scene. Using the `Shift` modifier when pressing one of these keys increases the amount of motion, using the `Ctrl` modifier reduces the amount of motion. Use `Home` and `End` to rotate the eye up and down, use `Insert` and `Delete` to rotate the eye to the sides.

After pressing `l` the motion keys will move the light instead of the eye, after pressing `b` the motion keys will move the head ball around, and after pressing `e` the motion keys operate on the eye.

The simulation can be paused and resumed by pressing `p` or the space bar. Pressing the space bar while paused will advance the simulation by 1/30 s. Gravity can be toggled on and off by pressing `g`.

The `+` and `-` keys can be used to change the value of certain variables. These variables specify things such as the gravitational acceleration, dynamic friction, and variables that may be needed for this assignment.

The variable currently affected by the `+` and `-` keys is shown in the bottom line of green text. Pressing `Tab` and `Shift-Tab` cycles through the different variables. To locate variables which can be set this way, and to see how they were set, search for `variable_control.insert` in the assignment file.

Assignment-Specific User Interface

The code can display several scenes, numbered 0 through 7. The currently displayed scene is iden-

tified in the penultimate line of green text. For this assignment use scenes 0-2 (the code starts with scene 1). Press `0` to select scene 0, `1` for scene 1, etc. Scene 3 shows a crude corona virus particle, Scene 4 shows a wheel, scene 5 a top (as in a spinning child's toy), scene 6 is a very crude parachute, and scene 7 is empty. The code for scenes 0-2 are in routines `World::ball_setup_prob_0`, `World::ball_setup_prob_1`, and `World::ball_setup_prob_2`. The code for scenes $3 \leq x \leq 7$ is in routine `World::ball_setup_`x.

In the unmodified assignment `World::ball_setup_prob_0` and `World::ball_setup_prob_1` contain the same code and `World::ball_setup_prob_2` is similar to the other two. The solution to Problem 1 should be put in `World::ball_setup_prob_1` and the solution to Problem 2 should be put in `World::ball_setup_prob_2`. Use `World::ball_setup_prob_0` for experimentation.

Initially, each time scenes 0, 1, and 2 are started (by pressing `0`, `1`, or `2`) they will show a different grid, including its location and the number of balls. (The random initialization is done to avoid solutions that only work for one particular configuration.) Pressing `c` will toggle between randomly choosing a scene configuration when the scene is started and leaving the configuration unchanged. The state of this option is shown in the penultimate line of green text by label Config. Its value is either RANDOM or FROZEN.

One strategy to compare the code in the three `prob` routines is to pause the simulation (`p`), freeze the configuration (`c`), and then press `0`, `1`, and `2` to see how the different routines render the same configuration.

Code Generation and Debug Support

The compiler generates an optimized version of the code, `hw02`, and a debug-able version of the code, `hw02-debug`. The `hw02-debug` version is compiled with optimization turned off, which makes it easier to debug. When needed, you are strongly encouraged to run `hw02-debug` under the GNU debugger, `gdb`. See the material under "Running and Debugging the Assignment" on the course procedures page. **You must learn how to debug.** If not, you will be at a severe disadvantage *for the rest of your life, if that's how long you stubbornly resist learning!*

To help you debug your code and experiment in one way or another, the user interface lets you change *tryout* variables. There are two Boolean tryout variables, `opt_tryout1` and `opt_tryout2`, and one floating-point tryout variable `opt_tryoutf`. You can use these variables in your code (for example, `if ( opt_tryout1 ) { x += opt_tryoutf; }`) to help debug, to help familiarize yourself with how the code works, or to experiment with new ideas. Keys `y` and `Y` to toggle the values of the Boolean variables; their values are shown in the green text at the label `Tryout 1:` and `Tryout 2:`. The user interface can also be used to modify host floating-point variable `opt_tryoutf` using the `Tab`, `+`, and `-` keys, see the previous section.

Resources

A good reference for C++ is `https://en.cppreference.com/w/`. A past homework assignment that may be helpful is 2018 Homework 1. In particular look at the alternative solution to this assignment for help with Problem 2.

Homework Code Overview

The unmodified code in routines `World::ball_setup_prob_0()`, `World::ball_setup_prob_1()`, and `World::ball_setup_prob_2()`, all construct a grid of balls connected by links. Initially these three routines are nearly identical. Routine `World::ball_setup_prob_1()` is for the solution to Problem 1 and `World::ball_setup_prob_2()` is for the solution to Problem 2. Routine `World::ball_setup_prob_0()` is for experimentation for those who (rightly) fear messing up whatever they've done so far in `World::ball_setup_prob_1()` and `World::ball_setup_prob_2()`.

Each routine starts by writing the configuration of the hyperboloid to construct into convenience variables (such as `axis`):

```
// The axis used to construct the hyperboloid.
const pNorm axis{h.axis};

// A point on the axis used to construct the hyperboloid.
const pCoor center_pos{h.center_pos};

// The endpoints of the first line used to construct the hyperboloid.
const pCoor line_0_bot_pos{h.line_0_bot_pos};
const pCoor line_0_top_pos{h.line_0_top_pos};

const int n_balls = h.n_balls;
const int n_lines = h.n_lines;
```

Though these describe the hyperboloid to be constructed, the unmodified code just draws a grid. Point `center_pos` is the center of the bottom of the hyperboloid. Its position is marked with a green ball in the screen shot. Normal vector `axis` points in the direction of the axis. In the screenshots `axis` would point from the green ball to the blue ball. Coordinates `line_0_bot_pos` and `line_0_top_pos` are the end points of the first line (line 0) used to construct the hyperboloid. The yellow balls form that first line. The number of balls in each line, including the first one, is specified in `n_balls`, for the configuration shown in the screenshot `n_balls=10`. *Unless I counted wrong or updated the screenshot.* Variable `n_lines` specifies the total number of lines to draw.

In the grid constructed by the unmodified code the number of balls per line, and the number of lines is correct. Furthermore, that first line is in the correct position. But the remaining lines are not. In this code overview, the code in its current state will be described. The problems describe how it should be modified to draw the hyperboloid.

The first thing the code does is it places a marker ball at `center_pos`:

```
// Drop a marker ball at the center position.
make_marker( center_pos, color_green, ball_radius*5 );
```

This was done to make the problem easier to describe. Please feel free to drop additional marker balls to help familiarize yourself with what's going on and to debug your code.

Ultimately the balls are linked together to form a gird. A 2D container is declared to hold the balls for the linking code:

```
vector<vector<Ball*>> balls_lines;
```

Container `balls_lines` can be indexed like a 2D array, such as `balls_lines[0][1]`. Next comes the loop nest constructing the grid. The `j` loop iterates over lines and the `i` loop iterates over balls:

```
pVect grid_pitch( 5 * ball_radius, 0, 0 );

for ( int j=0; j<n_lines; j++ ) {
    // Compute the endpoints of the current line to draw.
    pCoor line_bot_pos = line_0_bot_pos + j * grid_pitch;
    pCoor line_top_pos = line_0_top_pos + j * grid_pitch;


    // Compute a vector that will advance from the bottom to the top.
    pVect v_line = (1.0/(n_balls-1)) * ( line_top_pos - line_bot_pos );

    // Construct a line consisting of n_balls balls.
```

```
    for ( int i=0; i<n_balls; i++ ) {
        pCoor pos = line_bot_pos + i * v_line;
        Ball* const ball = new Ball(pos);
```

The code in the `i` (inner) loop places balls on a straight line from `line_bot_pos` to `line_top_pos`. That code is correct (though an alternative way of computing these positions is the subject of Problem 2). The `j` (outer) loop places `line_bot_pos` and `line_top_pos` on straight lines starting at the end points of line 0, and separated by vector `grid_pitch`. Vector `grid_pitch` should not be used in the solutions to Problems 1 and 2. Balls on the line starting from `line_bot_pos` are green (except the first one) and those starting at `line_top_pos` are blue (except the first one).

Next, code in the `i` loop body puts the ball in our 2D container and also in the simulator's list, `balls`. The ball's members are set, including the color. Feel free to modify the color if that helps with debugging.

```
        Ball* const ball = new Ball(pos);
        balls_line.push_back( ball );   // Our list of balls, for linking.
        balls.push_back( ball );        // List of simulated balls.

        ball->velocity = pVect(0,0,0);
        ball->radius = ball_radius;
        ball->color =
          j == 0 ? color_gold :
          i == 0 ? color_pale_green :
          i == n_balls - 1 ? color_blue :
          color_light_slate_gray;
    }
```

Finally, after the `j` loop exits, a new loop nest adds the links:

```
for ( int j=0; j<balls_lines.size(); j++ ) {
    auto &bl0 = balls_lines[j], &bl1 = balls_lines[(j+1)%n_lines];
    for ( int i = 0; i<bl0.size(); i++ ) {
        if ( i ) links += link_new( bl0[i-1], bl0[i], largs );
        links += link_new( bl0[i], bl1[i], largs );
     }
}
```

The routine `link_new` constructs a link between the two balls in its arguments. The link actually consists of several ideal springs, placed not at the center of the ball, but on several places on its surface so that the link will resist twisting and when relaxed the balls will be in their original orientation. The spring constants are set so that the initial position and orientation of the balls are the relaxed configuration. For this assignment it won't be necessary to change how the balls are linked together, but feel free to play.

Collaboration Rules

Each student is expected to complete his or her own assignment. It is okay to work with other students and to ask questions in order to get ideas on how to solve the problems or how to overcome some obstacle (be it a question of C++ syntax, interpreting error messages, how a part of the problem might be solved, etc.) It is also acceptable to seek out programming and graphics resources. It is okay to make use of AI LLM tools such as ChatGPT and Copilot to generate sample code. (Do not assume LLM output is correct. Treat LLM output the same way one might treat legal

advice given by a lawyer character in a movie: it may sound impressive, but it can range from sage advice to utter nonsense.)

After availing oneself to these resources **each student is expected to be able to complete the assignment alone**. Test questions will be based on homework questions and **the assumed time needed to complete the question will be for a student who had solved the homework assignment on which it was based**.

Student Expectations

To solve this assignment students are expected to avail themselves of references provided in class and on the Web site, such as for C++ programming and to seek out any additional help and resources that might be needed. (Of course this doesn't mean asking someone else to solve it for you.) Students are expected to experiment to learn how graphics work, and how to code C++ sequences. Experimentation might be done on past homework assignments. Students are also expected to learn what error messages mean by consulting documentation and by asking others (including Dr. Koppelman), and also to develop debugging skills. It is the students' responsibility to resolve frustrations and roadblocks quickly. (Just ask for help!)

This assignment cannot be solved by blindly pasting together code fragments found in class notes or past assignments. Solving the assignment is a multi-step learning processes that takes effort, but one that also provides the satisfaction of progress and of developing skills and understanding.

Problem 1: Modify the code in `World::ball_setup_prob_1` so that it draws a hyperboloid of one sheet by re-writing the code that initializes `line_bot_pos` and `line_top_pos` so that it computes coordinates `line_bot_pos` and the coordinates `line_top_pos` so that they each form (or lie on) circles, as described below. The circles have the same normal **but their radii are not necessarily the same**. (In the unmodified code the coordinates `line_bot_pos` form the line along which the green balls are placed and `line_top_pos` form the line along which the blue balls are placed.)

The *bottom circle* has center `center_pos` and axis `axis`. Coordinate `line_0_bot_pos` is on the bottom circle. Values assigned to `line_bot_pos` should be on the bottom circle, and for `j=0` the value of `line_bot_pos` should be `line_0_bot_pos`. The `n_lines` balls should be separated by $2\pi/n_l$ radians, where n_l is the value of `n_lines`.

The *top circle* has axis `axis` and point `line_0_top_pos` lies on the circle (and should be computed for `j=0`). The center is not explicitly given. It is at the point where the line starting at `center_pos` in the direction `axis` intercepts the top plane. The center can be found using expression `center_pos + height * axis` with the height correctly chosen. Note that the radius of the top circle can be different than the radius of the bottom circle.

There have been many past code examples for drawing circles. Most of the information to draw the bottom circle has been given. The top circle is only a little trickier.

Problem 2: Modify the code in `World::ball_setup_prob_2` so that it draws a hyperboloid of one sheet using a transformation matrix as described below. The code in `World::ball_setup_prob_2` constructs the grid differently than the other two routines. In the first iteration of the `j` loop it computes the coordinates of a ball in a method similar to the other routines:

```
for ( int j=0; j<n_lines; j++ ) {
    for ( int i=0; i<n_balls; i++ )
      {
        pCoor pos;
        if ( j == 0 ) {
            // Use the line_0 end point to find the position on the
            // first line.
            pos = line_0_bot_pos + i * line_pitch;
          } else {
            pCoor pos_prev = balls_lines[j-1][i]->position;
            pos = transform * pos_prev;
          }
```

But in subsequent iterations it uses transformation matrix `transform` to compute the coordinates of ball `i` on line `j` using the coordinates of ball `i` on line `j-1`. In the unmodified code `transform` is set to a translation, and so the balls form the grid:

```
// Compute a vector that will advance from one vector to another.
pVect grid_pitch( 5 * ball_radius, 0, 0 );

// Translate so that coordinate P is moved to P + grid_pitch.
pMatrix_Translate transl( grid_pitch );
pMatrix transform = transl;
```

Modify `transform` so that it transforms the coordinate of `j-1` ball `i` to the correct coordinate for line `j` ball `i`. See the alternative solution of 2018 Homework 1 for an example of how to construct a similar transformation matrix.