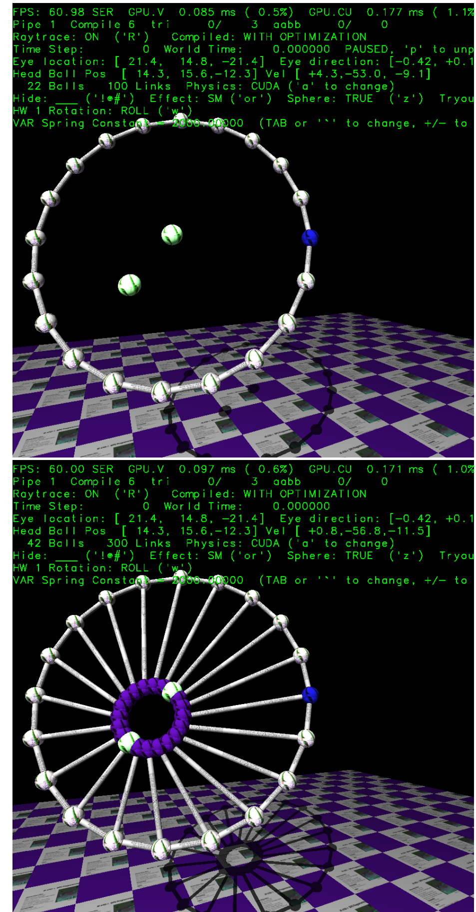
LSU EE 4702-1 Homework 2 Due: 21 September 2023

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/2023/hw02.cc.html`*. The solution code is at* `https://www.ece.lsu.edu/koppel/gpup/2023/hw02-sol.cc.html`*.*



Problem 0: If not already done, follow the instructions on `https://www.ece.lsu.edu/koppel/gpup/proc.html` for account setup and programming homework work flow. Compile and run the homework code unmodified. The code is based on the links demo in directory `proj-base/v-links`. The homework code should initially show a ring of balls rotating around the ring center. Inside the ring are two green balls. (These balls disappear after 3 seconds.) See the screenshot to the upper right. The ring will fall to the platform and start rolling to the right. In Problem 1 an inner ring is to be added and linked to the outer ring, that is shown in the lower screenshot. After the inner ring is added the rotation will no longer work correctly. That will be fixed in Problem 2.

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.ogg` or `hw02-debug-1.ogg`.

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 will be 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. 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 1 through 6. Press `1` to select scene 1 (the

silly tree), `2` for scene 2 (I'm going to ask GPT-4 to name it when it can read images), `3` for scene 3 (a crude corona virus particle), `4` for this homework assignment (the ring), `5` for another copy of this assignment code, and `6` for a top (a spinning child's toy). Scene 1 is initialized by the code in `World::ball_setup_1`, etc. The solution to this assignment should be put in `World::ball_setup_4` but `World::ball_setup_5` can be used to test ideas.

The scenes for this assignment, 4 and 5, initialize randomly each time they are reset. (That is to avoid solutions that accidentally only work for a special case of positioning, etc.) Sometimes it's useful to see the exact same scene a second time. For that press `Ctrl 4` or `Ctrl 5`. The desired size and initial positioning of the rings will be exactly the same. It is even possible to change things like spin axis, `w`, and gravity, `g`, before pressing `Ctrl 4`. One might use this, for example, to see spin and roll applied to the same item.

The line of green text near the bottom shows the desired rotation axis for the rings in Scene 4 and 5. Initially it will show "HW 2 Rotation: ROLL." Pressing `w` will rotate the desired rotation between ROLL, SPIN, and NONE. See Problem 2 for the interpretation of roll, spin, and none.

Display of Performance-Related Data

The top green text line shows performance in various ways. The number to the right of FPS shows the frame rate, the number of frames completed per second. On some displays 60 is the target frame rate and anything significantly lower than that indicates mediocre performance. Next, the green text shows whether frames are being prepared one at a time (serially), SER, or the steps in preparing a frame are being overlapped, OVR. In overlap mode commands for one frame are recorded while commands for a prior frame may be executing. For assignments early in the semester the mode will be kept at SER.

GPU.V shows how long the GPU spends updating the frame buffer (per frame), GPU.CU shows the execution of CUDA code per frame. CUDA code is physics in some assignments, but not this one and so the time should be shown as `---`. On some of 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 recording Vulkan graphics commands (or whatever it does in the callback installed by `vh.cbs_cmd_record.push_back`). CPU PH is the amount of time that the CPU spends on physics or whatever it does in the callback installed by the call to `vh.display_cb_set`.

For this assignment rendering is done by ray tracing. (Other assignments will use rasterization.) For rasterization 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 next line indicates whether the code was compiled with optimization. Use the version without optimization for debugging and the version with optimization for performance measurements.

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.

To help you debug your code and experiment in one way or another, the user interface lets you change variables. In most assignments these include the variables `opt_tryout1`, `opt_tryout2`, `opt_tryout3`, and `opt_tryoutf`. You can use these variables in your code (for example, `if ( opt_tryout1 ) { x += 5; }`) to help debug, to help familiarize yourself with how the code works, or to experiment with new ideas. Keys `y`, `Y`, and `Z` toggle the value of host Boolean variables `opt_tryout1`, `opt_tryout2`, and `opt_tryout3`. The user interface can also be used to modify host floating-point variable `opt_tryoutf` using the `Tab`, `+`, and `-` keys, see the previous section.

Problem 1: The code showing Scene 4 is in routine `World::ball_setup_4`. Search for "Put Homework 2" to quickly find this routine. In the unmodified code this routine draws just one ring and shows two green balls inside the ring. Those balls are at positions `p1` and `p2`. (Objects `p1` and `p2` are provided in the code.)

(*a*) Modify the code so that it draws a second ring in the same plane as the first ring. Points `p1` and `p2` (see below) should be on opposite sides of the second ring. (See the lower screenshot on the first page of the assignment.) Each ball in the inner ring should connect to its neighbors and to a ball in the outer ring, as shown in the screenshot.

(*b*) If the second ring passes outside of the first ring show the balls of the inner ring in red, otherwise show them in purple.

When working on this problem it might be helpful to turn off gravity, `g`, or pause the simulation `p`, to see the position of the balls before they start to fall.

Information on the routine, `si`, provides information about the rings to be constructed. (Here ring and circle are being used interchangeably.) The code starts by setting some default values for balls. (The defaults are copied into the balls that are constructed and overwritten.)

```
Ball ball_default;
ball_default.specularity = 0.5;
ball_default.velocity = pVect(0,0,0);
ball_default.radius = si.ball_radius;
ball_default.color = color_light_slate_gray;
```

In the unmodified code all that's drawn is the outer ring. The center of the outer ring is in `si.center_pos`, the normal is in `si.ring_normal`, and the radius is in `si.ring_outer_radius`.

The code finds points along the ring in the usual way. It starts by computing local axes a_x and a_y and scales them to the desired radius:

```
// Compute axes ax, ay, that are orthogonal to az.
pVect az = si.ring_normal;
pNorm ax = fabs(az.x) > fabs(az.y)
  ? pVect( az.z, 0, -az.x ) : pVect( 0, az.z, -az.y );
pNorm ay = cross(az,ax);

// Scale axes by radius so that they can be used to find points on circle.
pVect vx = si.ring_outer_radius * ax;
pVect vy = si.ring_outer_radius * ay;
```

The number of balls is `si.n_balls`. A $\Delta\theta$ is computed and a loop constructs and places each ball:

```
const float d_theta = 2 * M_PI / si.n_balls;
for ( int i=0; i<si.n_balls; i++ ) {
    const float theta = d_theta * i;
    pCoor pos = si.center_pos + vx * cosf(theta) + vy * sinf(theta);

    // Construct new ball and add it to the ball list.
    Ball* ball = new Ball(pos,ball_default);
    balls += ball;

    // Set color of first ball to blue, others to white.
```

```
    ball->color = i == 0 ? color_blue : color_white;
  }
```

Next, a link is placed between adjacent balls. A link connecting two balls is constructed by calling `link_new`. The first two arguments are the balls, the third is the link stiffness.

```
// Connect each ball to its neighbor with a link.
for ( int i=0; i<si.n_balls; i++ )
  links += link_new( balls[i], balls[(i+1)%si.n_balls], link_stiffness );
```

The coordinates of `p1` and `p2` are in variables of those names:

```
pCoor p1 [[maybe_unused]] = si.p1;
pCoor p2 [[maybe_unused]] = si.p2;
```

Note that when the second ring is constructed the array `balls` will have `2*n_balls` elements. Be sure to choose the correct indices when connecting inner ring balls to each other and each inner ring ball to an outer ring ball.

Problem 2: Member `si.rotation` shows the desired initial ring rotation. It has three possible values, `Rot_None`, `Rot_Roll`, and `Rot_Spin`. These values are changed by pressing `w`. The value of `si.rotation` is shown by the green text near the bottom starting "HW 2 Rotation:". Let *wheel* refer to the inner and outer rings and their links. When the value is `Rot_None` ball velocities should be set to zero. (That should already work.)

(*a*) When the value of `si.rotation` is `Rot_Roll` set velocities so that the wheel rotates around its center of gravity with the spin axis normal to the wheel plane (which is in object `az` in the code). This is what is done in the unmodified assignment. But, with Problem 1 solved the center of gravity will be different and the distance of each ball from the center of gravity will vary. See `https://www.ece.lsu.edu/koppel/gpup/2023/hw02-video-0g-roll.ogg` for a video of the correct motion, in zero gravity (which makes it easier to see the spin). For a video of roll with gravity on visit `https://www.ece.lsu.edu/koppel/gpup/2023/hw02-video-roll.ogg`.

The mass of a ball is determined by its density and radius. If the defaults aren't changed each ball will have the same mass. Try to compute the center of gravity **without using a loop**. That is, don't just add up the position of each ball and divide by the number of balls.

When testing your code turn off gravity, `g` and reset the scene `4`. The wheel should not drift away and its rotation axis should not change. Also, there should not be wobbling when it starts due to the wrong velocities being set.

(*b*) When the value of `si.rotation` is `Rot_Spin`, set the velocity of the balls so that the wheel rotates around a *stable* axis in the ring's plane. That is, the rotation axis must be orthogonal to the ring's plane, `az`. The axis must be chosen to be stable. (It must be an eigenvector of the wheel's inertia tensor.) An axis passing through the center of the inner and outer ring will be stable. (No need to compute an inertia tensor.) See `https://www.ece.lsu.edu/koppel/gpup/2023/hw02-video-0g-spin.ogg` for a video of the correct spin motion, in zero-gravity (which makes it easier to see the spin). For a video of spin with gravity see `https://www.ece.lsu.edu/koppel/gpup/2023/hw02-video-spin.ogg`.

To determine the velocity of a ball one must find its minimum distance from the axis. (The `Rot_Roll` case is easier because it is only necessary to find the distance between the ball and the center of gravity, which is a point.)

For an example of code that rotates objects look at `World::balls_twirl()` in the assignment file. The routine rotates objects around an axis determined by the head and tail ball. Calling this routine won't work for this problem because the head and tail balls don't define the correct axis for this problem. In fact, the correct axis might not bass through any balls' centers.