This document contains assignments given in LSU GPU Programming over many semesters. It was automatically generated and so some solutions (and even some assignments) are possibly missing. At the top of each page of each assignment is a link to the original assignment. Those who want to print an assignment might follow that link. All assignments and public solutions are available at `https://www.ece.lsu.edu/gpup/prev.html`.

# Contents


**1 Fall 2026** **6**
- 1.1 hw01.pdf . . . 7
- 1.2 hw02.pdf . . . 12
- 1.3 hw03.pdf . . . 17

**2 Fall 2025** **23**
- 2.1 hw01.pdf . . . 24
- 2.2 hw02.pdf . . . 30
- 2.3 hw03.pdf . . . 35
- 2.4 hw04.pdf . . . 42
- 2.5 hw05.pdf . . . 50

**3 Fall 2024** **59**
- 3.1 hw01.pdf . . . 60
- 3.2 hw02.pdf . . . 66
- 3.3 hw03.pdf . . . 72
- 3.4 hw04.pdf . . . 81
- 3.5 hw05.pdf . . . 86

**4 Fall 2023** **92**
- 4.1 hw01.pdf . . . 93
- 4.2 hw02.pdf . . . 97
- 4.3 hw03.pdf . . . 101
- 4.4 hw04.pdf . . . 106
- 4.5 hw05.pdf . . . 111

**5 Fall 2022** **117**
- 5.1 hw01.pdf . . . 118
- 5.2 hw02.pdf . . . 121
- 5.3 hw03.pdf . . . 124
- 5.4 hw04.pdf . . . 130
- 5.5 hw05.pdf . . . 134

**6 Fall 2021** **138**
- 6.1 hw01.pdf . . . 139
- 6.2 hw02.pdf . . . 147
- 6.3 hw03.pdf . . . 151

**7 Fall 2020** **155**
7.1 hw01.pdf . . . 156
7.2 hw02.pdf . . . 159
7.3 hw03.pdf . . . 161

**8 Fall 2019** **164**
8.1 hw01.pdf . . . 165
8.2 hw02.pdf . . . 170
8.3 hw03.pdf . . . 174
8.4 hw04.pdf . . . 177

**9 Fall 2018** **181**
9.1 hw01.pdf . . . 182
9.2 hw02.pdf . . . 184
9.3 hw03.pdf . . . 187
9.4 hw04.pdf . . . 191
9.5 hw05.pdf . . . 195

**10 Fall 2017** **199**
10.1 hw01.pdf . . . 200
10.2 hw02.pdf . . . 202
10.3 hw03.pdf . . . 205
10.4 hw04.pdf . . . 208
10.5 hw05.pdf . . . 211

**11 Fall 2016** **214**
11.1 hw01.pdf . . . 215
11.2 hw02.pdf . . . 217
11.3 hw03.pdf . . . 219
11.4 hw04.pdf . . . 221
11.5 hw05.pdf . . . 223
11.6 hw06.pdf . . . 226
11.7 hw07.pdf . . . 228

**12 Fall 2015** **230**
12.1 hw01.pdf . . . 231
12.2 hw02.pdf . . . 233
12.3 hw03.pdf . . . 235
12.4 hw04.pdf . . . 237

**13 Fall 2014** **239**
13.1 hw01.pdf . . . 240
13.2 hw02.pdf . . . 242
13.3 hw04.pdf . . . 244
13.4 hw05.pdf . . . 246
13.5 hw06.pdf . . . 247

**14 Fall 2013** **249**
14.1 hw01.pdf . . . 250
14.2 hw02.pdf . . . 251
14.3 hw03.pdf . . . 253

**15 Fall 2012** **256**
15.1 hw01.pdf . . . 257
15.2 hw02.pdf . . . 258
15.3 hw03.pdf . . . 259
15.4 hw04.pdf . . . 261
15.5 hw05.pdf . . . 263

**16 Fall 2011** **264**
16.1 hw01.pdf . . . 265

**17 Fall 2026 Solutions** **266**
17.1 hw01-sol.cc.html . . . 267
17.2 hw02-sol.cc.html . . . 276
17.3 hw03-sol.cc.html . . . 328

**18 Fall 2025 Solutions** **345**
18.1 hw01 sol.pdf . . . 346
18.2 hw01-sol.cc.html . . . 354
18.3 hw02-sol.cc.html . . . 363
18.4 hw03-sol.cc.html . . . 425
18.5 hw04-shdr-sol.cc.html . . . 440
18.6 hw05-shdr-sol.cc.html . . . 450
18.7 hw05-sol.cc.html . . . 463

**19 Fall 2024 Solutions** **529**
19.1 hw01-sol.cc.html . . . 530
19.2 hw02-sol.cc.html . . . 541
19.3 hw03-sol.cc.html . . . 608
19.4 hw04-shdr-sol.cc.html . . . 621
19.5 hw04-sol.cc.html . . . 625
19.6 hw05-shdr-sol.cc.html . . . 639
19.7 hw05-sol.cc.html . . . 644

**20 Fall 2023 Solutions** **662**
20.1 hw01-sol.cc.html . . . 663
20.2 hw02-sol.cc.html . . . 670
20.3 hw03-sol.cc.html . . . 726
20.4 hw04 sol.pdf . . . 736
20.5 hw04-shdr-sol.cc.html . . . 745
20.6 hw04-sol.cc.html . . . 749
20.7 hw05 sol.pdf . . . 809

20.8 hw05-shdr-sol.cc.html . . . 828
20.9 hw05-sol.cc.html . . . 834

**21 Fall 2022 Solutions** **892**
21.1 hw01-sol.cc.html . . . 893
21.2 hw02 sol.pdf . . . 898
21.3 hw02-sol.cc.html . . . 903
21.4 hw03 sol.pdf . . . 912
21.5 hw03-shdr-sol.cc.html . . . 925
21.6 hw03-sol.cc.html . . . 928
21.7 hw04-shdr-sol.cc.html . . . 942
21.8 hw04-sol.cc.html . . . 946
21.9 hw05-shdr-sol.cc.html . . . 960
21.10hw05-sol.cc.html . . . 968

**22 Fall 2021 Solutions** **984**
22.1 hw01-sol.cc.html . . . 985
22.2 hw02-shdr-sol.cc.html . . . 997
22.3 hw03-shdr-sol.cc.html . . . 1006

**23 Fall 2020 Solutions** **1015**
23.1 hw01 sol.pdf . . . 1016
23.2 hw01-sol.cc.html . . . 1021

**24 Fall 2018 Solutions** **1034**
24.1 hw01-sol.cc.html . . . 1035
24.2 hw02-sol.cc.html . . . 1050
24.3 hw03-shdr-sol.cc.html . . . 1060
24.4 hw03-sol.cc.html . . . 1065
24.5 hw04-shdr-hw04-sol.cc.html . . . 1076

**25 Fall 2017 Solutions** **1082**
25.1 hw01-sol.cc.html . . . 1083
25.2 hw02-sol.cc.html . . . 1097
25.3 hw03-sol.cc.html . . . 1138
25.4 hw04-shdr-sol.cc.html . . . 1164
25.5 hw05 sol.pdf . . . 1170
25.6 hw05-shdr-sol.cc.html . . . 1177

**26 Fall 2016 Solutions** **1185**
26.1 hw01-sol.cc.html . . . 1186
26.2 hw02-sol.cc.html . . . 1195
26.3 hw03-sol.cc.html . . . 1208
26.4 hw05-shdr-links-sol.cc.html . . . 1240
26.5 hw06 sol.pdf . . . 1246
26.6 hw06-shdr-links-sol.cc.html . . . 1249

26.7 hw06-sol.cc.html . . . 1254
26.8 hw07 sol.pdf . . . 1288
26.9 hw07-cuda-sol.cu.html . . . 1295

**27 Fall 2015 Solutions** **1305**
27.1 hw01 sol.pdf . . . 1306
27.2 hw02 sol.pdf . . . 1312
27.3 hw02-sol.cc.html . . . 1317
27.4 hw03 sol.pdf . . . 1335
27.5 hw03-sol.cc.html . . . 1340
27.6 hw04 sol.pdf . . . 1362
27.7 hw04-sol-shdr-links.cc.html . . . 1368
27.8 hw04-sol-shdr.cc.html . . . 1372
27.9 hw04-sol.cc.html . . . 1378

**28 Fall 2014 Solutions** **1409**
28.1 hw02 sol.pdf . . . 1410
28.2 hw02-sol.cc.html . . . 1413
28.3 hw04 sol.pdf . . . 1426
28.4 hw04-shdr-sol.cc.html . . . 1428
28.5 hw05 sol.pdf . . . 1435
28.6 hw06 sol.pdf . . . 1441
28.7 hw06-shdr-lines-sol.cc.html . . . 1446

**29 Fall 2013 Solutions** **1452**
29.1 hw03 sol.pdf . . . 1453

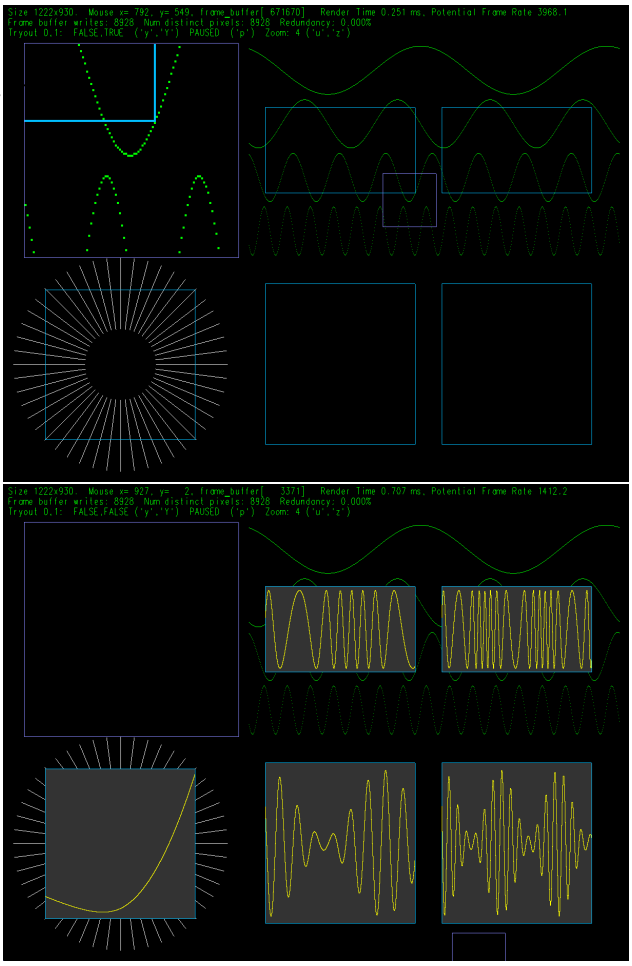
**30 Fall 2012 Solutions** **1461**
30.1 hw01 sol.pdf . . . 1462
30.2 hw02 sol.pdf . . . 1463
30.3 hw03 sol.pdf . . . 1467
30.4 hw04 sol.pdf . . . 1472
30.5 hw05 sol.pdf . . . 1474

# 1 Fall 2026

# LSU EE 4702-1 Homework 1 Due: 9 September 2026

*Formatted 18:10, 1 September 2026*

*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/2026/hw01.cc.html`.



**Problem 0:** Follow the instructions on the `https://www.ece.lsu.edu/koppel/gpup/proc.` page for account setup and programming homework work flow. Compile and run the homework code unmodified. It should initially show some sine waves and some radially arranged white rectangles with an inner red circle and an outer green ring (which is just another name for a thick green circle). There should also be blue boxes. The box in the upper left of the screenshot shows a magnified version of the area around the mouse pointer. See the screenshot to the upper right. When the assignment is finished the blue boxes will show function plots, see the screenshot to the lower right. The top row of blue boxes show FM-modulated signals. The lower-left plot shows an *activation function* recently described in the AI literature, and plots to the right of that the lower right show AM-modulated signals.

## Display of Performance-Related Data

The top green text line shows performance-related and other information. `Size` refers to the size of the window. `Mouse` refers to the coordinates of the mouse pointer. Coordinate $(0, 0)$ is at the **lower** left of the window. Text `frame_buffer[N]` shows the index of the frame buffer corresponding to the point under the mouse pointer. (In the assignment file `frame_buffer` is abbreviated to `fb`, for convenience.)

`Render Time` and `Potential Frame Rate` show the CPU time needed to write the frame buffer. They can be ignored for this assignment.

## General 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 `hw01.png` or `hw01-debug.png`.

The value of two Boolean debug-support variables, `tryout0` and `tryout1`, are shown in the green text, pressing `y` toggles `tryout0` (between `true` and `false`) and pressing `Y` toggles `tryout1`. The variables are available in routine `render_hw01` and `plot_func`, use them to try things out.

### Problem-Specific User Interface

The box in the upper left shows a zoomed version of the area around the mouse pointer. Pressing `z` will increase the zoom amount and pressing `u` will decrease it.

Some of the functions to be plotted change with time and so they appear animated. Press `p` to toggle this animation on and off.

### Integer Coordinate and Vector Types

This assignment uses both integer and floating point types for coordinates and vectors. The integer types are `iCoor` for 2D coordinates and `iVect` for 2D vectors. Most other course code uses type `pCoor` for floating-point 3D coordinates (actually 4D, we'll learn in a few weeks) and `pVect` for floating-point 3D vectors.

The `iCoor` and `iVect` classes each have two members, `x` and `y`:

```
class iCoor {
public:
  iCoor( int x, int y ):x(x),y(y){}
  int x, y;
};
class iVect {
public:
  iVect( int x, int y ):x(x),y(y){}
  int x, y;
};
```

The addition, subtraction, and division operators are overloaded with these types. Here are some examples of how to use them:

```
void demo() {
  iCoor c0(3,4);  // Set to x=3, y=4.
  iCoor c1;
  c1.x = 5; c1.y = 7;  // Set to x=5, y=7;

  iCoor c3 = c1;

  // Component-wise subtraction.
  // Equivalent to v1.x=c0.x-c1.x;  v1.y=c0.y-c1.y;
  iVect v1 = c0 - c1;

  // Component-wise addition.
  // Equivalent to c4.x=c1.x+v1.x;  c4.y=c1.y+v1.y;
  iCoor c4 = c1 + v1;

  // Not allowed: Can't add two coordinates.
  iCoor c5 = c4 + c1;
}
```

### Code Generation and Debug Support

The compiler generates an optimized version of the code, `hw01`, and a debug-able version of the code, `hw01-debug`. The `hw01-debug` version is compiled with optimization turned off, which makes it easier to debug. When needed, you are strongly encouraged to run `hw01-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. These are `hw01_data.tryout0` and `hw01_data.tryout1`. You can use these variables in your code (for example, `if ( hw01_data.tryout1 ) { x += 5; }`) to help debug, to help familiarize yourself with how the code works, or to experiment with new ideas. Keys `y` and`Y` to toggle their values, which are shown in the green text at the label `Tryout 0,1:`.

### Resources

A good reference for C++ is `https://en.cppreference.com/w/`. See Homework 1 assigned in the past few semesters for similar problems.

### 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 Claude 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:** Routine `func_plot`, when completed, will plot a given function over a given range into a given rectangle in the frame buffer. The declaration is:

```
template< typename F > void func_plot
( pFrame_Buffer& fb,
  iCoor plot_lower_left, int plot_width, int plot_height,
  float func_x0, float func_x1, F func )
```

Argument `fb` is the frame buffer object used in many frame-buffer assignments. For examples on how it can be used see the code in `line_draw` and `render_hw01`.

Arguments `plot_lower_left`, `plot_width` and `plot_height` specify where in the frame buffer the plot is to be placed. Argument `plot_lower_left` specifies the frame buffer pixel coordinates of the lower-left of the plot to be drawn. (See the discussion of the `iCoor` object covered earlier in this assignment.) Arguments `plot_width` and `plot_height` are the width and height of the plot to be drawn. These variables are already used by code in `func_plot` to draw a box around the plot area.

Argument `func` is the function to be plotted, it can be any function that takes a `float` type as an argument and returns a `float` type. Arguments `func_x0` and `func_x1` specify the range of $x$ values to be plotted.

The code below calls `func_plot` to plot `sinf` (sine to floating-point precision) from $-2\pi$ to $7\pi$. The plot will be drawn in a rectangle of width 400 pixels and height 200 pixels. (See `render_hw01` for other examples of how to call `func_plot`.) Note that for this assignment there is no need to call `func_plot`, that's already done.

```
iCoor plot_ll( 20, 20 );
float p_wd = 400,  p_ht = 200;
float px0 = -2 * numbers::pi,  px1 = numbers::pi * 7;
func_plot( fb,   plot_ll, p_wd, p_ht,   px0, px1,   sinf );
```

In the unmodified code `func_plot` draws a deep-sky-blue box around the plot rectangle:

```
const uint32_t color_deep_sky_blue(0x00bfff);
iCoor p0 = plot_lower_left; // Just an abbreviation.
iVect vx( plot_width, 0 ), vy( 0, plot_height );
iCoor p1 = p0 + vx + vy;
line_draw( fb, p0, p0 + vy, color_deep_sky_blue );
line_draw( fb, p0, p0 + vx, color_deep_sky_blue );
line_draw( fb, p0 + vy, p1, color_deep_sky_blue );
line_draw( fb, p0 + vx, p1, color_deep_sky_blue );
```

It also computes the first and the last values of `func` to be plotted, but does not plot them.

```
float func_y0 [[maybe_unused]] = func(func_x0);
float func_y1 [[maybe_unused]] = func(func_x1);
```

(*a*) Modify `func_plot` so that it sets the plot area to `color_background`, where `color_background` is a variable in `func_plot`. Do this after the lines are drawn. *Note: this is easy.*

(*b*) Modify `func_plot` so that the function is plotted as described below.

Let *plot area* refer to the rectangle specified by `plot_lower_left`, `plot_width`, and `plot_height`. In the discussion below $x_0$ and $x_1$ refer to `func_x0` and `func_x1`, $w$ and $h$ refer to `plot_width` and `plot_height`, $i_0$ and $j_0$ refer to `plot_lower_left.x` and `plot_lower_left.y`; and $i_1 = i_0 + w$ and $j_1 = j_0 + h$. Let $y = f(x)$ be the value returned by `func` for input $x$.

To solve this two mappings are needed, one from pixel space coordinates to function space, and one from function space coordinates to pixel space. Pixel space coordinates are what are used to write the frame buffer. Values $i_0, i_1, j_0, j_1$ are pixel space coordinates. Values $x_0$, $x_1$, $f(x_0)$, and $f(x_1)$ are function space coordinates. The function space $x$ coordinate is used for the argument to $f(x)$ and the function space $y$ coordinate is used for the return value of $y = f(x)$. The mappings should be linear, and so they can be achieve by maybe multiplying and adding something to, say, a pixel space $x$ coordinate to get a function space $x$ coordinate.

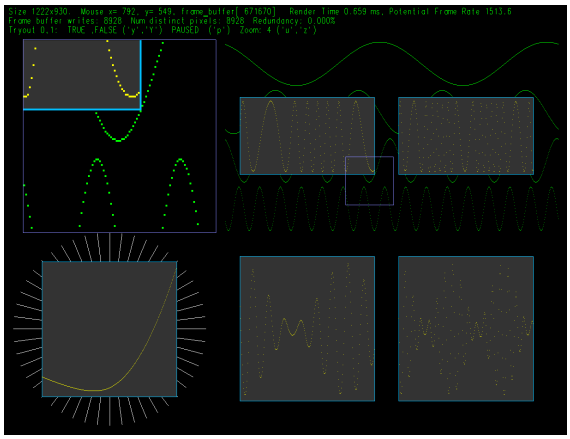
The code needs to iterate from $i_0$ and $i_1$, and for each $i \in [i_0, i_1]$ the code needs to use the mapping to convert $i$, which is in pixel space, to a function-space $x$ value. Using that compute $y = f(x)$ and then convert $y$ to pixel space, call the value $j$, and write the frame buffer at pixel coordinate $i, j$ (and at other locations to avoid gaps).

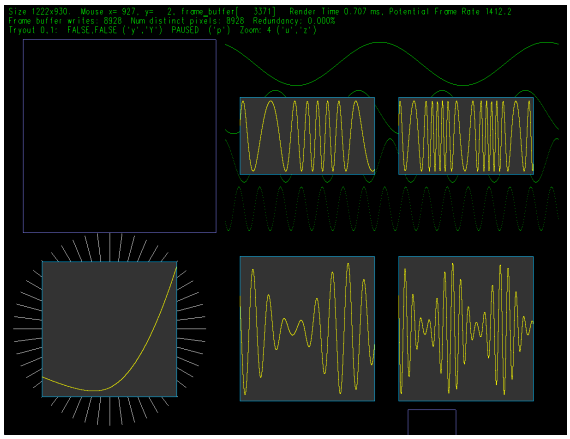
- ☐ Plot the function from $x_0$ to $x_1$, with the value for $x_0$ placed along the left edge (at pixel $x$ coordinate $i_0$) and the value for $x_1$ placed along the right edge.

- ☐ Scale the values returned by `func` so that they fit within the plot area and so that the largest $y = f(x)$ is placed $.025h$ pixels from the top of the plot area and the smallest $y = f(x)$ is $.025h$ pixels from the bottom of the plot area. ☐ Make sure the top and bottom of the plots don't touch the top and bottom of the plot box.

- ☐ The plot should be continuous (without gaps) so for each $i$ (pixel $x$ coordinate) it may be necessary to plot multiple points. In the screenshot on the lower left the plots are not continuous (they are dotted). Make sure the plots are continuous as in the screenshot on the lower right.

- ☐ The code should be reasonable efficient. Don't call the line drawing routine to fill the gaps between points because the overhead of setting up a line draw can be avoided here.





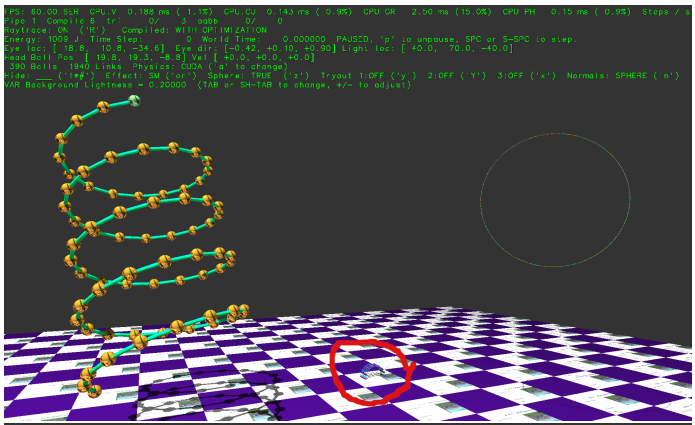
# LSU EE 4702-1 Homework 2 Due: 25 September 2026

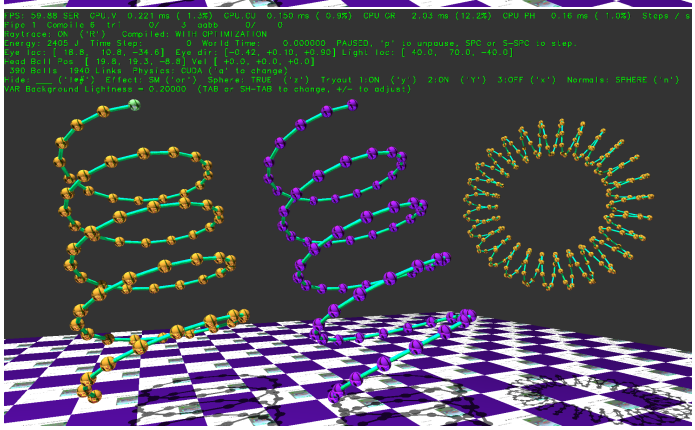
*Formatted 11:43, 22 September 2026*

*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/2026/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 three objects, balls forming a helix on the left, a ring on the right, and a small easy-to-miss helix lying on the platform. See the screenshot to the upper right. When the assignment is correctly solved there will be two similar helices (with gold and purple balls), plus a *toroidal helix* on the right. See the screenshot to the lower right. The purple helix is drawn by routines `hw02_prob1` and `spring_make2`. The toroidal helix is drawn by code in `hw02_prob2`.





## Homework Overview

File `hw02.cc` contains several routines that draw helices. The easiest to understand is `World::spring_make`. It uses the parametric circle drawing approach used in class. Routine `spring_make2` avoids the need for computing trigonometry functions each iteration by instead using a transformation matrix, `global_space_delta_pos` to advance the current ball's coordinate, `pos_i`, to the coordinate of the next ball.

## User Interface

The code can display several scenes, numbered 0 through 7. The currently displayed scene is identified in the penultimate line of green text. For this assignment use scene 7. 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 for this assignment.

Each time [7] is pressed the specifications for the displayed items will change. (That's to prevent students from ignoring call arguments.) Each time [Ctrl]-[7] is pressed the same specifications will be used, to help in debugging.

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.

## 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 three Boolean tryout variables, `opt_tryout1`, `opt_tryout2`, and `opt_tryout3` 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, Y, x 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.

## 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 Claude 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**.

Students are expected to understand their own solutions. If you use something, such as a function, expression, or method of calculation found elsewhere, **make sure that you understand it**, don't just paste it blindly.

### 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 process that takes effort, but one that also provides the satisfaction of progress and of developing skills and understanding.

**Problem 1:** Routine `World::spring_make2( m, n_balls )` constructs a helix consisting of `n_balls` balls connected to their neighbors with links. The ball coordinates are initially computed in a local space and then are transformed to global space using matrix `m`. In the local space the entire helix fits in a box of $x \pm 1$, $y \pm 1$, and $z$ from 0 to 1. In the local space the helix has a radius of 1 and its axis starts at the origin and is along the $z$ axis. The first ball is placed at local coordinate $\begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix}$ and the last ball will be about $\begin{bmatrix} 1 \\ 0 \\ 1 \end{bmatrix}$.

Routine `World::hw02_prob1(axis_top, axis_bottom, r, n_balls)` prepares a transformation matrix, `m`, and then calls `spring_make2` using `m` and `n_balls`. In the unmodified assignment `m` is an identity matrix. Because `m` is identity the second helix is shown in its local space, not in the intended global space. See the helix circled in red on the screenshot.

In `World::hw02_prob1` compute `m` so that the helix axis goes from coordinate `axis_top` to `axis_bottom` and the helix has a radius of `r`. If this is solved correctly the second helix should look exactly like the first one (including size and orientation) except for the ball colors.

☐ Compute `m` so that ☐ the two helices are identical (except for position and color).

☐ **Do not** change routine `spring_make2`.

▷ See routine `spring_make2` for examples of how to use transformation matrices.

▷ Solution will require a scale matrix, or should scale some other way.

☐ As always, code should be clear and reasonably efficient.

*There's another problem on the next page.*

**Problem 2:** In the unmodified code `World::hw02_prob2 (ring_top, ring_center, helix_vx, n_balls)` constructs a ring (circle) of `n_balls` balls around coordinate `ring_center`, passing through `ring_top`, and in a plane with normal `helix_vx`. Because the value of `n_balls` is large the balls and links are small and perhaps easy to overlook.

Modify `hw02_prob2` so that it constructs a *toroidal helix*. A toroidal helix is like a helix except that its axis forms a circle. The toroidal helix axis should be on the circle (ring) described above.

Consider the code from `spring_make` that computes a coordinate (`pos`) on a regular helix:

```
for ( int i=0; i<n_balls; i++ )
  {
    float theta = i * delta_theta;
    pCoor point_on_axis = axis_top + i * vz;
    pVect vector_from_axis_to_helix = cosf(theta) * vx + sinf(theta) * vy;
    pCoor pos = point_on_axis + vector_from_axis_to_helix;
```

Coordinate `pos` is computed by adding the coordinate of a point on the helix axis to a vector from that point to the helix.

For the toroidal helix `point_on_axis` should be a point on the circle, which should be easy to compute since the unmodified code already computes it. For the toroidal helix `vector_from_axis_to_helix` should be computed the same way as the regular one *except* that something must be done with `vx` and/or `vy` to account for rotation around the circle.

- ☐ Modify `hw02_prob2` so that it draws the toroidal helix.
- ☐ There should be about 9 balls per rotation around the helix.
- ☐ The distance between neighboring balls should be the same, including the first and last balls.
- ☐ Set `ball_rad` to $\frac{1}{5}$ the distance between balls or $r\Delta\theta/5$ where $r$ is the helix radius and $\Delta\theta$ is for the helix, not the ring.
- ☐ As always, code should be clear and reasonably efficient.

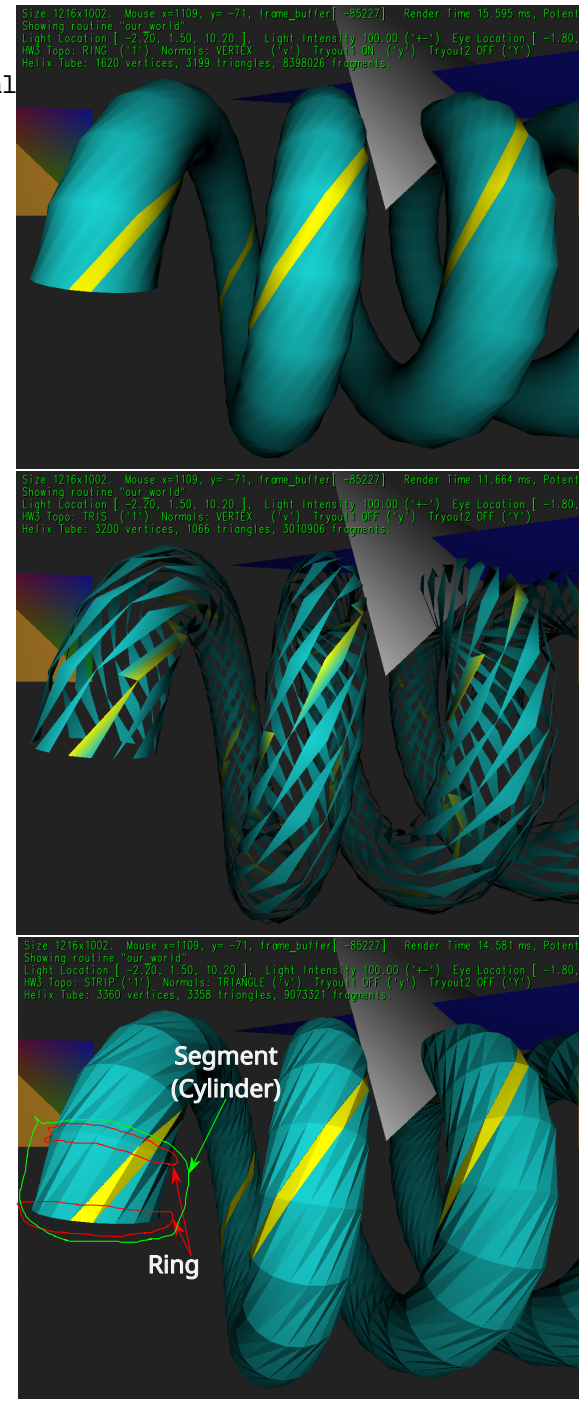
**LSU EE 4702-1** **Homework 3** **Due: 16 October 2026**

*Formatted 14:26, 20 October 2026*

*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/2026/hw03.cc.html`.

**Problem 0:** In the unlikely event that this is still needed, 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 cyan helix with a yellow stripe, plus four triangles. This shows the helix rendered using vertices in triangle-strip order, this order is shown by the green text to the right of `HW3 Topo:`, see the screenshot to the upper right. The upper shot shows the helix the way it's supposed to be rendered. In the middle screenshot, also taken from an unmodified assignment, the vertices are in triangle-strip order but the rendering system, `Our_3D`, expects them to be in individual triangle order. That's why the surface of the helix is not complete. When Problem 1 is correctly solved the helix rendered using individual triangles would look almost exactly like the helix rendered using a strip. There is a third way of ordering vertices, called *rings* for this assignment, that's the subject of Problem 2.




## Homework Overview

In this assignment a tube helix is to be rendered using three different topologies (vertex orderings). The term *helix* refers to the curve parametricly described by $H(t) = H_0 + ta_z + \cos(\omega t)a_x + \sin(\omega t)a_y$, where $a_x, a_y, a_z$ are local axes, $\omega$ is a rotation rate and $H_0$ is an initial position. That cyan thing in the screenshot is a tube helix, formed by sweeping a ring along a helix. The code uses the term helix to refer to the curve and tube helix to the 3D object.

The code places the coordinates, normals, and colors for the ring in containers `coors_ring`, `norms_ring`, and `colors_ring`. The first color in `colors_ring` is yellow, the others are cyan. Transformation matrices map the ring to

locations along the helix and also rotate slightly, which is why the yellow stripe wraps around.

The topologies used to render the tube helix can be chosen by pressing 1. The topology in use is shown on the green text to the right of `HW3 Topo:`. The topologies are *triangle strip* (`STRIP` in green text, `Topology_Strip` enumeration constant), *individual triangles* (`I-TRIS` in green text, `Topology_Individual` enumeration constant), and *rings* (`RING` in green text, `Topology_Ring` enumeration constant). The triangle strip and individual triangle orderings work the same way as the like-named orderings in OpenGL and Vulkan. The rings ordering was made up for this assignment.

These orderings are described below using the tube helix as an example. To better see the triangles used to form the tube helix press v (which toggles whether to use tube-helix normals or triangle normals).

Let $v_0, v_1, \ldots, v_i, \ldots v_{n-1}$ denote the $n$ vertices provided to our rendering system. Using the variable names used in `hw03.cc:World::render_scene()`, the vertices provided to our rendering system are collected in containers `coors_os`, `normals_os`, and `colors`. Quantity $n$ would be the number of elements in `coors_os`, and $v_i$ would denote `coors_os[i]`, `normals_os[i]`, and `colors[i]`. (That is, $v_i$, represents all attributes provided for a vertex, in this case coordinate, normal, and color.)

Recall that in an individual triangle topology three vertices are provided for each triangle. Vertices $v_0, v_1, v_2$ form the first triangle, $v_3, v_4, v_5$ form the second triangle, etc. To render $T$ triangles $3T$ vertices are needed.

If the set of triangles to be rendered can be ordered so that adjacent triangles share an edge then triangle strip ordering can be used. Vertices $v_0, v_1, v_2$ form the first triangle, vertices $v_1, v_2, v_3$ form the second triangle, $v_2, v_3, v_4$ form the third triangle, etc. The first and second triangles share edge $v_1, v_2$, the second and third share $v_2, v_3$, etc. So to render $T$ triangles only $n = T + 2$ vertices are needed. That's almost $\frac{1}{3}$ the number of vertices!

The ring topology, which was made up for this assignment, is even better. Consider the problem of rendering a tube (the tube helix is an example). If triangle strips were used each vertex would need to be specified at least twice. The helix in this assignment is rendered by concatenating what are almost cylinders, each almost cylinder is called a segment in the code. A segment is specified by a ring of vertices of its top and a ring at its bottom. (The term *ring* is used in the code, the vertices in a ring form a circle.) (In the code the ring coordinates are in containers `tr_co_curr` and `tr_co_prev`.) Notice that the vertices of the top ring of one segment are identical to the vertices of the bottom ring of the next segment. Let $n_r$ denote the number of vertices in a ring (in the code this is variable `n_ring_vtx`). Just for one segment, using individual triangles we'd need to specify $6n_r$ vertices per segment for the $2n_r$ triangles. A helix of $n_s$ segments would need $6n_r n_s$ vertices, and most vertices are duplicated six times (there are six triangles that meet at a point, corresponding to the duplicated vertex).

Using triangle strips each segment would require $2(n_r + 1)$ vertices: $n_r + 1$ for the bottom ring, and another $n_r + 1$ for the top. The $+1$ is there to complete the last two triangles, which touch the first two. The entire tube helix requires $2n_s(n_r + 1)$ vertices to specify the $T = 2n_s n_r$ triangles, which is an improvement but still most vertices appear twice (since a top and bottom ring are shared inside the helix).

In the ring topology each ring is to be sent just once. Code in `World::render_scene()` already sends vertices in the correct order for a ring topology. Search for `case Topology_Ring:` to find this

code. Continuing to use $n_r$ to denote the number of vertices in a ring, vertices $v_0, \ldots, v_{n_r-1}$ form the first ring, vertices $v_{n_r}, v_{n_r+1}, \ldots, v_{2n_r-1}$ form the second ring, and so on. Just given $v_0, \ldots, v_{2n_r-1}$ `Our_3D` should render a cylinder when set for a ring topology. Suppose we append a third ring, $v_{2n_r}, \ldots, v_{3n_r-1}$. Then two adjacent cylinders would be rendered. Vertices $v_{n_r}, v_{n_r+1}, \ldots, v_{2n_r-1}$ would be the top of the first cylinder and the bottom of the second cylinder (assuming each succeeding ring is higher than the one before it).

In a triangle strip topology a triangle is rendered for each vertex beyond the first two. So for $n$ vertices there would be $n-2$ triangles. In a triangle ring two triangles are rendered for each vertex beyond the first $n_r$.

A convenient thing about individual triangles and triangle strips, is that adjacent vertices form a triangle, such as $v_i, v_{i+1}, v_{i+2}$. That's not the case with a ring topology. This is the logical place to explain precisely which vertices are grouped together to form triangles in a ring topology. Say, $v_i, v_\alpha, v_\beta$ where $\alpha =$. But, determining $\alpha$ and $\beta$ are part of Problem 2.

### Display of Performance-Related Data

The top green text line shows performance-related and other information. `Size` refers to the size of the window. `Mouse` refers to the coordinates of the mouse pointer. Coordinate $(0, 0)$ is at the **lower** left of the window. Text `frame_buffer[N]` shows the index of the frame buffer corresponding to the point under the mouse pointer. (In the assignment file `frame_buffer` is abbreviated to `fb`, for convenience.)

`Render Time` and `Potential Frame Rate` show the CPU time needed to write the frame buffer. They can be ignored for this assignment.

Much more important for this assignment (2026 Homework 3) are the amount of data and operations used to render objects. Data for the tube helix is shown on the last line of green text to the right of `Helix Tube:`. The number of vertices, triangles, and fragments are shown. In general, the amount of data sent from CPU to GPU is proportional to the number of vertices (all things being equal). The computation load on the GPU is proportional to the sum of the number of vertices and the number of fragments, usually the number of fragments is larger and dominates computation time. Pay attention to these numbers when solving the problems in this assignment.

### General User Interface

Press Ctrl = to increase the size of the green text and Ctrl - to decrease the size. Pressing + (or =) increases light intensity, pressing - reduces light intensity. Press v to toggle between which normal is used to compute lighting. Initially lighting uses the normals sent to `Our_3D`. Pressing v toggles between using triangle normals and the provided normals (if any).

Initially the arrow keys, PageUp, and PageDown, can be used to move around the scene. After pressing l (lower case ell) the motion keys will move the light instead of the eye, pressing e the motion keys operate on the eye again. Press F12 to generate a screenshot. The screenshot will be written to file `hw03.png` or `hw03-debug.png`.

The value of two Boolean debug-support variables, `opt_tryout0` and `opt_tryout1`, are shown in the green text, pressing y toggles `opt_tryout0` (between `true` and `false`) and pressing Y toggles `opt_tryout1`. The variables are available in routine `World::render_scene()` and `Our_3D::draw_rasterization()`, use them to try things out.

### Assignment-Specific User Interface

Pressing 1 (digit one) toggles the topology of the vertices expected when rendering the tube

helix. The three possible topologies are *Individual Triangles* shown as `I-TRIS` in the green text to the right of `HW3 Topo:`, a *Triangle Strip* shown as `STRIP`, and *Rings*, shown as `RING`.

### Code Generation and Debug Support

The compiler generates an optimized version of the code, `hw03`, and a debug-able version of the code, `hw03-debug`. The `hw03-debug` version is compiled with optimization turned off, which makes it easier to debug. When needed, you are strongly encouraged to run `hw03-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 three Boolean tryout variables, `opt_tryout1`, `opt_tryout2`, and `opt_tryout3` 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], [Y], [x] 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 2025 Homework 3.

### 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 Claude 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**.

Students are expected to understand their own solutions. If you use something, such as a function, expression, or method of calculation found elsewhere, **make sure that you understand it**, don't just paste it blindly.

## 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 process that takes effort, but one that also provides the satisfaction of progress and of developing skills and understanding.

**Problem 1:** The code in `render_scene` near the case for `Topology_Individual` (search for "Problem 1") inserts vertices in an order appropriate for a triangle strip, which of course is not the expected topology. Modify this code so that vertices are inserted in individual triangle order. Be sure to insert the same triangles as the triangle strip code (except for the redundant ones). It is easier to see where triangles are when using triangle-normal lighting, this can be toggled on and off by pressing v.

- ☐ Modify code in `render_scene` so that vertices are inserted correctly for the individual triangle topology.
- ☐ If solved correctly, the number of vertices will be 3× that of triangle strip ordering, while the number of triangles and fragments will be slightly lower than triangle strip ordering.
- ☐ Code must be clear and reasonably efficient.
- ☐ Do not reformat existing code (unless that code is modified as part of solution).

**Problem 2:** Modify the code in `render_scene` and `draw_rasterization` so that the draw with the `Topology_Ring` ordering is shown correctly. Note that the code in `World::render_scene()` for `case Topology_Ring` is already correct. That code can be studied but should not be modified. The only change in `World::render_scene()` is to pass the number of vertices in a ring using whatever mechanism you implement in `Our_3D::draw_rasterization()`. (See 2025 Homework 3 for a similar issue with repeated fans.)

Most work for this problem is in `Our_3D::draw_rasterization()`. The code there works correctly for strips and individual triangles. Modify the code so that it works for rings. Part of this is changing the number of iterations of the main rendering loop, and changing the way `i0`, `i1`, and `i2` in that loop are computed.

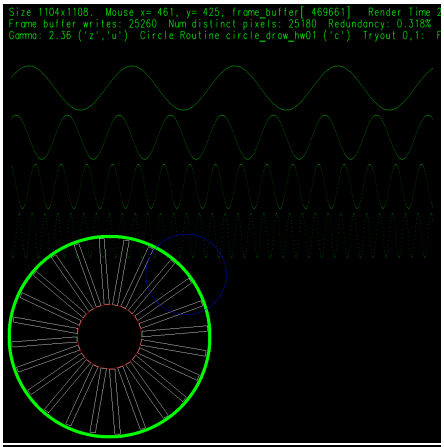
- ☐ Send the number of vertices in a ring to our 3D in a way that matches how other rendering parameters are sent.par
- ☐ Implement ring topology rendering in `draw_rasterization`.
- ☐ Try to avoid adding another level to the big loop in `draw_rasterization`.
- ☐ Make sure that changes for rings do not break the other topologies!
- ☐ Be sure that coordinates, colors, and normals are correct.
- ☐ If solved correctly, the number of vertices will be $\frac{1}{2}$ that of triangle strip ordering, while the number of triangles and fragments will be slightly lower than triangle strip ordering.
- ☐ Code must be clear and reasonably efficient.
- ☐ Do not reformat existing code (unless that code is modified as part of solution).

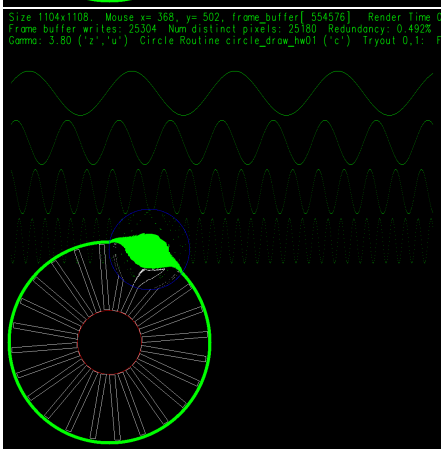
## 2 Fall 2025

## LSU EE 4702-1 Homework 1 Due: 9 September 2025

*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/2025/hw01.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 some sine waves and some radially arranged white rectangles with an inner red circle and an outer green ring (which is just another name for a thick green circle). See the screenshot to the upper right. dots. In the lower image, taken from a correct solution. Though it can't be seen in the screenshots, the render time is much lower. Also, there is a blue circle inside of which the image appears distorted.




### Display of Performance-Related Data

The top green text line shows performance-related and other information. `Size` refers to the size of the window. `Mouse` refers to the coordinates of the mouse pointer. Coordinate $(0, 0)$ is at the **lower** left of the window. Text `frame_buffer[N]` shows the index of the frame buffer corresponding to the point under the mouse pointer. (In the assignment file `frame_buffer` is abbreviated to `fb`, for convenience.)

`Render Time` and `Potential Frame Rate` show the CPU time needed to write the frame buffer. They can be ignored for this assignment.

### General 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 `hw01.png` or `hw01-debug.png`.

The value of two Boolean debug-support variables, `tryout0` and `tryout1`, are shown in the green text, pressing `y` toggles `tryout0` (between `true` and `false`) and pressing `Y` toggles `tryout1`. The variables are available in routine `render_hw01` and `circle_draw_hw01`, use them to try things out.

### Problem-Specific User Interface

The circle can be drawn by two routines `circle_draw_hw01` (where part of the solution is to be placed) and `circle_draw_parametric` (which is there for reference). Pressing `c` will toggle between two routines. The routine currently being used is shown in the last line of green text to the right of `Circle Routine`.

Pressing `z` will increase the distortion factor, $\gamma$, and pressing `u` will decrease it. Distortion, which should appear in the blue circle, won't be seen until Problem 2 is solved correctly.

### Code Generation and Debug Support

The compiler generates an optimized version of the code, `hw01`, and a debug-able version of the

code, `hw01-debug`. The `hw01-debug` version is compiled with optimization turned off, which makes it easier to debug. When needed, you are strongly encouraged to run `hw01-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. These are `tryout0` and `tryout1`. You can use these variables in your code (for example, `if ( tryout1 ) { x += 5; }`) to help debug, to help familiarize yourself with how the code works, or to experiment with new ideas. Keys `y` and`Y` to toggle their values, which are shown in the green text at the label `Tryout 0,1:`.

### Homework Code Overview

For this assignment the code in routines `hw01_render` and `circle_draw` will be modified. The code in `hw01_render` writes the frame buffer with sine waves and a circle of rectangles radiating from a point. The code for the sine waves and circle of rectangles is there for references, but it does not need to be modified for this assignment.

### Resources

A good reference for C++ is `https://en.cppreference.com/w/`. See Homework 1 assigned in the past few semesters for similar problems. In particular, in 2023 Homework 1 Problem 2 part of the frame buffer around the mouse pointer was to be copied to another part of the frame buffer. Something like this copying is also done in Problem 2 in this assignment.

### 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:** Routine `circle_draw_parametric` draws a circle (ring) into the frame buffer:

```
void circle_draw_parametric( pFrame_Buffer& fb, HW01_Data& hw01_data,
  int ctr_x, int ctr_y, int r_outer, int r_inner, uint32_t co)
{
  int win_width = fb.width_get();
  float delta_theta = asinf(1.0f/r_outer);
  for ( float theta = 0; theta < 2*numbers::pi; theta += delta_theta )
    {
      float cth = cosf(theta),       sth = sinf(theta);
      for ( int r = r_inner; r<=r_outer; r++ )
        {
          int x = ctr_x + r * cth,   y = ctr_y + r * sth;
          fb[ y * win_width + x ] = co;
        }}}
```

The center of the circle is at pixel coordinate `ctr_x`, `ctr_y`, the inner radius is `r_inner` and the outer radius is `r_outer`. Pixels of the frame buffer from the inner and outer radius are set to color `co`, or at least they should, but some pixels are skipped. The routine is based on a parametric description of a circle: $P(\theta) = C + \begin{bmatrix} r\cos\theta \\ r\sin\theta \end{bmatrix}$.

The routine above is not the best way of drawing a circle in pixel space (which is what we are using) because it calls trig functions (sine and cosine) many times (and that's computationally costly), because it doesn't write every pixel, and because it writes some pixels multiple times. (If `delta_theta` is made small enough it would write every pixel at least once, but it would be writing many more pixels multiple times.)

The fundamental problem with the routine above is that we are iterating over $\theta$ and $r$, a better approach is to iterate over pixel coordinates. In the unmodified assignment the code in `circle_draw_hw01` does iterate over pixel coordinates:

```
void circle_draw_hw01 ( pFrame_Buffer& fb, HW01_Data& hw01_data,
  int ctr_x, int ctr_y, int r_outer, int r_inner, uint32_t co)
{
  const int win_width [[maybe_unused]] = fb.width_get();
  const int win_height [[maybe_unused]] = fb.height_get();
  /// WARNING: Ridiculously inefficient.
  for ( int x = 0; x < win_width; x++ )
    for ( int y = 0; y < win_height; y++ )
      {
        int dx = x - ctr_x,   dy = y - ctr_y;
        int dsq = dx * dx + dy * dy;
        if ( dsq > r_outer * r_outer ) continue;
        if ( dsq < r_inner * r_inner ) continue;
        fb[ y * win_width + x ] = co;
      }
```

The code iterates over every pixel *in the frame buffer*, and checks whether that pixel is on the circle (between the inner and outer radii). On the plus side this code won't skip any pixels, it won't write the same pixel twice, it doesn't call any trigonometry functions, and it even avoids computing a square root. But all of those positives are outweighed by the fact that it checks every single pixel **in the frame buffer**!

Modify the code so that it iterates over fewer pixels. Consider solutions that keep the two-level loop nest, but in which the loop bounds are chosen so that the body is executed fewer times.

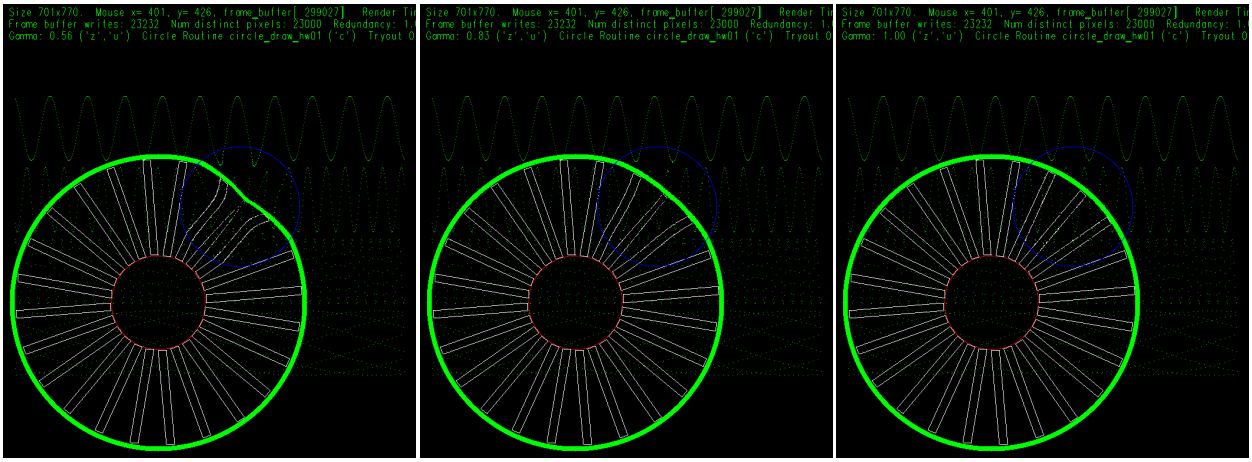
Consider the following ideas:

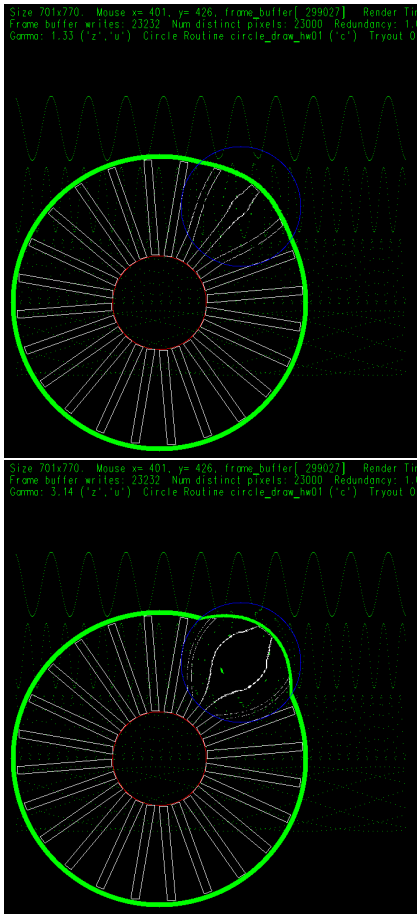
*Idea 1:* Modify the loop bounds so that `x` and `y` iterate over a *bounding box* that contains the circle. (A bounding box of a circle is the smallest possible box containing the circle. Any smaller and a part of the circle would be outside the box.)

*Idea 2:* Compute the limits on the `y` loop as a function of `x`. (Note that Idea 2 replaces or modifies the Idea 1 code.)

*Idea 3:* Have `x` and `y` iterate over only one quadrant of the circle, but have the loop body write four pixels (one write for the quadrant corresponding to `x` and `y`, and the other three writes for the other three quadrants.)

*There's another problem on the next page.*





**Problem 2:** A blue circle is drawn near the end of routine `render_hw01` in a call to `circle_draw`. Modify the code after this point so that the portion of the frame buffer inside the blue circle (but not the blue circle itself) is distorted as follows:

Let $C$ denote the center of the circle (in the code the center is in variable `ctr`) and $R$ its radius (`mag_r` in the code). Let $P = \begin{bmatrix} P_x \\ P_y \end{bmatrix}$ denote the coordinates of a pixel inside the circle.

Let $v = C - P = \begin{bmatrix} C_x - P_x \\ C_y - P_y \end{bmatrix}$ denote the vector from $C$ to $P$. In vector notation $P = C + v$. Let $d$ denote the length of $v$, which in this case is the distance from $C$ to $P$. (If you don't know how to compute the distance, look it up.)

For this problem define *distorted distance* $d' = R\left(\frac{d}{R}\right)^{\gamma}$, where $\gamma$ is a distortion factor. Note that for $\gamma = 1$ we just have $d' = d$. Given a distorted distance we can define an alternative pixel coordinate $P' = C + \frac{d'}{d}v$.

Modify the code in `hw01_render` so that every pixel $P$ in the blue circle is written with the frame buffer contents at $P'$ as defined above. In the code variable `ctr` is the circle center (see how it's used in the call to `circle_draw`). Variable `mag_r` is $R$, and `gamma` is $\gamma$. Variable `P` is $P$; your code can just use `x` and `y`. In the code variable `v` is $v$; your code can just use `dx` and `dy`. The code computes $d^2$, but not $d$. Using the user interface, the value of `gamma` can be modified by pressing `z` and `u`, and its value is shown on the third line of green text.

Some things to watch out for: most of the variables are declared `int`. To work correctly $d'$ must be computed using floating point operands. Let `xp` and `yp` denote the coordinates of $P'$. If these are used to compute a frame buffer index, `fb[ yp * win_width + xp]`, make sure that `yp` is an integer, or cast it to an integer otherwise `yp * win_width` won't give the correct result.

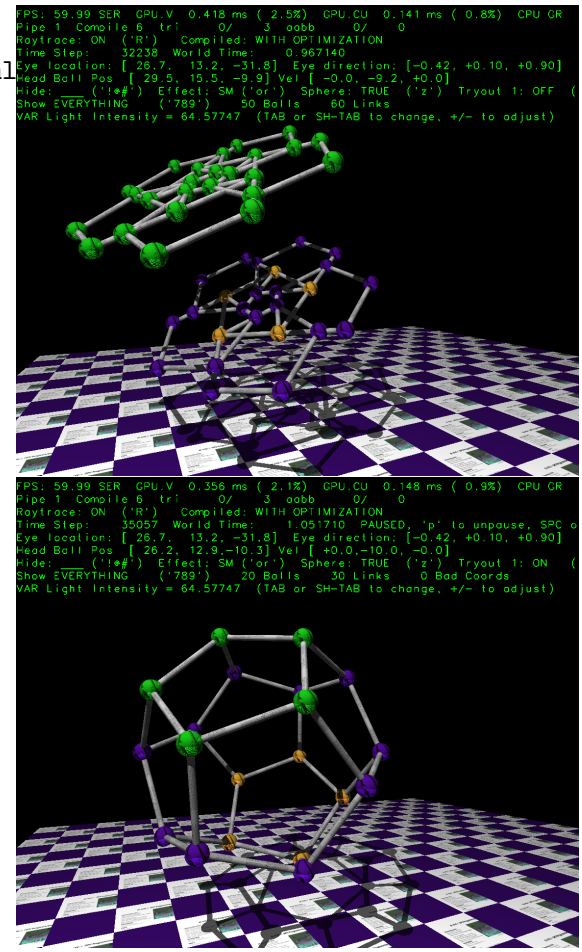
Finally, and importantly remember that the pixel being read (at coordinate $P'$) must be from

the frame buffer *before* it was modified by the code that fills the blue circle. So code like `fb[ y * win_width + x] = fb[ int(yp) * win_width + xp];` won't work correctly because the pixel being read, `fb[ int(yp) * win_width + xp];`, might have already been written. So, the area inside the blue circle first needs to be copied to a buffer area and that buffer area needs to be used when writing inside the blue circle. Before writing any pixels in the blue circle first copy them to array `fb_dup`. Array `fb_dup` is of size $(2(R+1))^2$, and so it has enough space for the blue pixels. It's okay of a bounding box around the blue circle is copied into `fb_dup`.

Once the blue circle are is copied to `fb_dup` one can get the distortion effect by writing `fb[ int(yp) * win_width + xp] = fb_dup[ b_y * b_width + b_x]`, where `b_width` is the width of the area copied **(which can't be larger than** $2(R+1)$**)** and `b_y` and `b_x` are relative to a corner of the copied area. See 2023 Homework 1 Problem 2 and its solution for more on how to use such a buffer area (called `fb_area_dup`) in that problem.

# LSU EE 4702-1 Homework 2 Due: 7 October 2025

*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/2025/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 jumble of balls connected by links, see the screenshot to the upper right. The jumble is constructed by the code in routine `balls_setup_7` in the unmodified code. The routine is supposed to use transformation matrices to compute the coordinates of a regular dodecahedron, but those transformation matrices are just placeholders. In this assignment those placeholders are to be replaced by correct transformations. The screenshot to the lower right is from a correctly solved assignment.

## User Interface

The code can display several scenes, numbered 0 through 9. The currently displayed scene is identified in the penultimate line of green text. For this assignment use scenes 7-9. 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 7-9 are for this assignment. The code for scenes $0 \leq x \leq 7$ is in routine `World::ball_setup_`$x$. The code for scenes 8 and 9 are also in `World::ball_setup_7`.

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

For this assignment use scenes 7-9. Press 7 to show scene 7, etc. Scene 7 shows the entire dodecahedron (or what's supposed to be a dodecahedron). Scene 8 shows only the initial pentagon from which the remainder of the dodecahedron is to be constructed. Scene 9 shows half of the dodecahedron. The penultimate line of green text shows which of the assignment scenes is active.

The penultimate line also shows the number of Balls, Links, and Bad Coords. For scenes 7-9, the number of balls is the number of vertices. In a correct solution Scene 7 should show 20 balls (vertices). Links shows the number of edges, which should be 30 in a correct solution. Bad Coords show the number of coordinates in array `coors` which are not at the expected distance from the dodecahedron center when the dodecahedron is constructed (but not later). Because `coors` contains duplicated coordinates, the number near Bad Coords can be larger than Balls.

The code for scenes 7, 8, and 9 are all in `World::ball_setup_7`. Each time scenes 7, 8, and 9, are started the position, size, and orientation of the dodecahedron will be slightly different.

### 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.

### Regular Dodecahedron

A *regular dodecahedron* is constructed from 12 regular pentagons of edge length $a$ by joining the pentagons at their edges so that each vertex is on the surface of a sphere, called the *circumscribed sphere*, of radius $r_c = a\phi\sqrt{3}/2$. There are a total of 20 vertices and 30 edges. The angle between two adjacent pentagons, called the *dihedral angle*, is $2\arctan(\phi)$ radians, where $\phi$ is the Golden Ratio, $\phi = \frac{1+\sqrt{5}}{2}$. The largest sphere that can fit inside a dodecahedron is called the *inscribed sphere*, its radius is $r_i = a\sqrt{\frac{\sqrt{5}\phi^5}{20}}$.

The code in `ball_setup_7` computes a random location, size, and orientation for a dodecahedron and computes useful information about it. Variable `center_pentagon_0` is the coordinates of the center of the first pentagon. Unit vectors `ax`, `ay`, and `az` are used to compute the pentagon coordinates. Comments in the code describe the other variables.

For this assignment a dodecahedron will be constructed by placing coordinates in vector `coors`. The first five elements of `coors` are the coordinates of a pentagon. Balls corresponding to these coordinates are gold. In the unsolved assignment these coordinate values are correct, but they are computed parametrically (shown below), in Problem 1 they are to be computed using transformations.

```
const float d_theta = 2 * numbers::pi / n_sides;
for ( int i=0; i<5; i++ )
  {
    const float theta = d_theta * i;
    pVect dir = cosf(theta) * ax + sinf(theta) * ay;
    pCoor pos = center_pentagon_0 + rc_pentagon * dir;
    coors << pos;
  }
```

Notice that coordinates are added to the `coors` vector using the `<<` operator. This is equivalent to `coors.push_back(pos)`.

After the first pentagon, the code adds five more pentagons, one for each edge of the original pentagon. The coordinates of the new pentagon are found by applying transformation `m` to the coordinates of the first pentagon:

```
for ( int i=0; i<5; i++ )
  {
    pCoor p0 [[maybe_unused]] = coors[i];
    pCoor p1 [[maybe_unused]] = coors[(i+1)%5];

    pMatrix_Translate tr( pVect(p0,p1) ); // Wrong.
    pMatrix m = tr;

    for ( int j=0; j<5; j++ ) coors << m * coors[j];
  }
```

Balls for these new coordinates are shown in purple (unless they are coordinates of the first pentagon). Each new pentagon shares edge `p0`,`p1` with the first pentagon but is rotated at angle $\theta$ around the axis formed by `p0` and `p1`. The transformation `m` above is supposed to do that, but it doesn't, it just translates the coordinates. This is to be fixed in Problem 2.

At this point in the code `coors` describes six pentagons, the first pentagon and five more. To complete the dodecahedron six more pentagons need to be added. That can be done by rotating the existing points in `coors` around the center of the dodecahedron, in variable `center`. That is done by the code:

```
pMatrix_Translate trd( 5 * az );
pMatrix ms = trd;
for ( int i=0; i<n_coord_demi; i++ ) coors << ms * coors[i];
```

Of course, `ms` is not the correct transformation. In Problem 3 a correct transformation is to be computed.

The coordinates in `coors` are then used to initialize positions of balls and links. This code does not need to be modified for this assignment. The first step in doing so is to identify duplicated coordinates. In a correctly constructed dodecahedron each coordinate will appear three times. The code considers two coordinates to be identical if their distance is less than $10^{-4}$ units. A ball is constructed for each unique coordinate. A link is constructed for each unique edge. Ball ids are

used to construct a key for a link, and a stdlib map is used to check whether a link has already been constructed.

### 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.

### 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 process that takes effort, but one that also provides the satisfaction of progress and of developing skills and understanding.

**Problem 1:** The existing code in `World::ball_setup_7` computes the vertices of the first pentagon parameterically. Modify it so the coordinates are computed using a transformation matrix. Starter code is provided. Please don't waste computation by computing the transformation matrix in the loop.

Note that the balls of the first pentagon are shown in gold. It may help to use Scene 8 when working on this problem.

**Problem 2:** The code in `World::ball_setup_7` near `HW 2 Problem 2` uses transformation matrix `m` to move the coordinates of the first pentagon (`coors[0]` through `coors[4]`, accessed in the `j` loop) to a new position for each edge of the first pentagon, and it inserts these coordinates in `coors`. The edge of the first pentagon is in `p0` and `p1`. A placeholder is shown for `m`. Correctly compute `m` so that it rotates the first pentagon coordinates by angle `theta_dihedral` (a variable in the code) around the axis formed by `p0` and `p1`.

If this problem is solved correctly half a dodecahedron will be constructed. Call that a *demi-dodecahedron*. The balls of the demi-dodecahedron are shown in gold and purple.

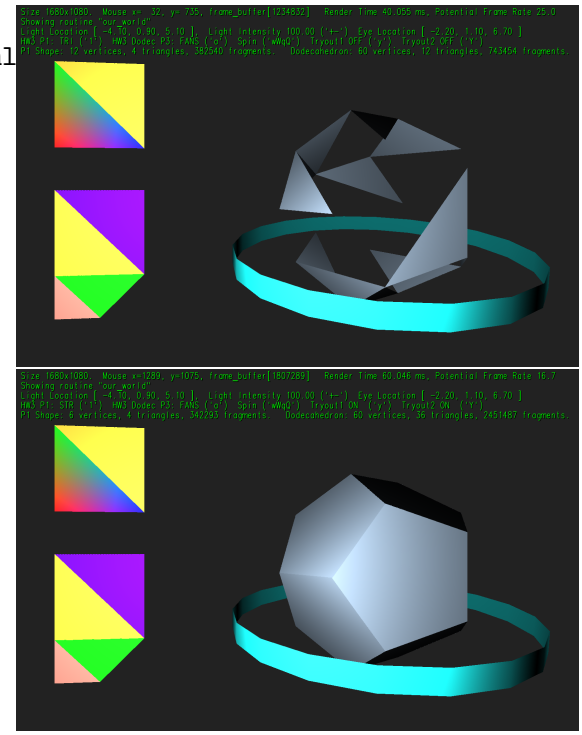
**Problem 3:** For this problem compute the remaining coordinates of the dodecahedron by applying a transformation to the demi-dodecahedron coordinates. Starter code is shown near `HW 2 Problem 3`. Compute transformation matrix `ms` so that it places the vertices in the correct location. The transformation should rotate the coordinates around the center of the dodecahedron (the center of the circumscribed sphere). More than one rotation will be necessary. The vertices added here are shown in green.

**LSU EE 4702-1** **Homework 3** **Due: 21 October 2025**

*Formatted 10:42, 14 October 2025*

*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/2025/hw03.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 scene with a cyan ring, colorful triangles, and a jumble of slate gray triangles. See the screenshot to the upper right. When the assignment is solved the jumble is replaced by a regular dodecahedron and the group of colorful triangles is rendered more efficiently. See the screenshot to the lower right, which is from a correctly solved assignment.



## General User Interface

Press `Ctrl=` to increase the size of the green text and `Ctrl-` to decrease the size. Pressing `+` (or `=`) increases light intensity, pressing `-` reduces light intensity. Press `v` to toggle between which normal is used to compute lighting. Initially lighting uses the normals sent to `Our_3D`. Pressing `v` toggles between using triangle normals and the provided normals (if any).

Initially the arrow keys, `PageUp`, and `PageDown`, can be used to move around the scene. After pressing `l` the motion keys will move the light instead of the eye, pressing `e` the motion keys operate on the eye again. Press `F12` to generate a screenshot. The screenshot will be written to file `hw03.png` or `hw03-debug.png`.

The value of two Boolean debug-support variables, `opt_tryout0` and `opt_tryout1`, are shown in the green text, pressing `y` toggles `opt_tryout0` (between `true` and `false`) and pressing `Y` toggles `opt_tryout1`. The variables are available in routine `World::render_scene()` and `Our_3D::draw_rasterization()`, use them to try things out.

## Assignment-Specific User Interface

Pressing `q`, `Q`, `w`, and `W` rotates the dodecahedron (or what will be the dodecahedron). Each keypress rotates by $\pm\pi/30$ radians around one of two axes. Rotating the dodecahedron may help in figuring out a bug when solving Problems 2 or 3, and it's also fun.

Pressing `1` toggles the topology of the vertices expected when rendering the group of triangles shown on the lower left. The two possible topologies are *Individual Triangles* shown as `TRI` in the green text to the right of `HW3 P1:`, and *Triangle Strips* shown as `STR` in the green text to the right of `HW3 P1:`. See Problem 1.

Pressing `a` toggles the topology of the vertices expected when rendering the dodecahedron. The two possible topologies are *Individual Triangles* (for Problem 2) and *Triangle Fans* (for Problem 3). The expected topology is shown to the right of `HW3 Dodec P2:` in the penultimate line of green text.

## Display of Performance-Related Data

The top green text line shows performance-related and other information. `Size` refers to the size of the window. `Mouse` refers to the coordinates of the mouse pointer. Coordinate $(0, 0)$ is at the **lower** left of the window. Text `frame_buffer[N]` shows the index of the frame buffer corresponding to the point under the mouse pointer. (In the assignment file `frame_buffer` is abbreviated to `fb`, for convenience.) `Render Time` and `Potential Frame Rate` show the CPU time needed to write the frame buffer. They can be ignored for this assignment.

Much more important for this assignment (2025 Homework 3) are the amount of data and operations used to render objects. Data is shown for the group of triangles in the lower right on the last line of green text to the right of `P1 Shape:`, and for the dodecahedron, to the right of `Dodecahedron:`. For each, the number of vertices, triangles, and fragments are shown. In general, the amount of data sent from CPU to GPU is proportional to the number of vertices (all things being equal). The computation load on the GPU is proportional to the sum of the number of vertices and the number of fragments, usually the number of fragments is larger and dominates computation time. Pay attention to these numbers when solving the problems in this assignment.

## Code Generation and Debug Support

The compiler generates an optimized version of the code, `hw03`, and a debug-able version of the code, `hw03-debug`. The `hw03-debug` version is compiled with optimization turned off, which makes it easier to debug. When needed, you are strongly encouraged to run `hw03-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`. 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:`.

## Regular Dodecahedron

A *regular dodecahedron* is constructed from 12 regular pentagons of edge length $a$ by joining the pentagons at their edges so that each vertex is on the surface of a sphere, called the *circumscribed sphere*, of radius $r_c = a\phi\sqrt{3}/2$. There are a total of 20 vertices and 30 edges. The angle between two adjacent pentagons, called the *dihedral angle*, is $2\arctan(\phi)$ radians, where $\phi$ is the Golden Ratio, $\phi = \frac{1+\sqrt{5}}{2}$. The largest sphere that can fit inside a dodecahedron is called the *inscribed sphere*, its radius is $r_i = a\sqrt{\frac{\sqrt{5}\phi^5}{20}}$.

The code in `World::render_scene()` computes the coordinates of the 12 pentagons forming a dodecahedron and places them into container `coors`. Unlike Homework 2, the initial position is fixed (not randomly chosen). The orientation is specified by variables `dod_angle1` and `dod_angle2` which can be changed using keys `q` and `w`. For this assignment it will only be necessary to work with the coordinates in `coors`. There will be no need to transform them, find a set of unique values, nor to find interesting normals.

It is important to understand that there are $5 \times 12$ coordinates in `coors`. The first five are coordinates of the first dodecahedron face, etc. In Problem 2 it will be necessary to examine the coordinates for each face and determine the coordinates of triangles needed to render the face.

## Code Overview

The code in this assignment is based on `cpu-only/demo-06-rend-pipe.cc`, which shows how code for rendering 3D objects might be separated into two parts, one for someone writing graphics

applications (such as a game) and one for writing a graphics driver for a particular computing device (such as a GPU). In `demo-06` and this assignment the graphics application is in class `World`, and the graphics driver is in object `Our_3D`.

Class `Our_3D` exposes an API that users can use to draw 3D graphics. The API consists of calls (member functions) such as `vtx_coors_set` to specify coordinates, `transform_eye_from_object_set`, to specify a transformation to eye space, and so on. The API in `Our_3D` is based on the APIs provided by an OpenGL or Vulkan implementation, though much simpler.

Code in `World` uses this API (as described in class). To render a scene `World` must provide transformation matrices, vertex coordinates, colors, and lighting information. That is done in `World::render_scene()`.

Variable `gc` points to an `Our_3D` object. First, transformation matrices are provided to `Our_3D`:

```
// Compute transformation from Object Space to Eye Space ..
pMatrix_Translate center_eye(-eye_location);
pMatrix_Rotation rotate_eye(eye_direction,pVect(0,0,-1));
pMatrix eye_from_object = rotate_eye * center_eye;
// .. and give the transformation to Our3D:
gc.transform_eye_from_object_set(eye_from_object);

// Compute transformation from Eye Space to Clip Space ..
pMatrix_Frustum clip_from_eye
  ( -width_m/2, width_m/2,  -height_m/2, height_m/2,  qn, 5000 );
// .. and give the transformation to Our3D:
gc.transform_clip_from_eye_set(clip_from_eye);
```

We also need to specify the light location and color:

```
light_location = pCoor(-4.1,.9,5.1);
light_color = color_white * 100;
gc.light_location_set( light_location ).light_color_set( light_color );
```

To render something, coordinates and colors (and optionally normals) of each vertex must be sent. The code below populates containers `coors_os` and `colors` with coordinates and colors and sends them to `Our_3D`:

```
vector<pCoor> coors_os;
vector<pColor> colors;

// Yellow and Multicolored Triangles
coors_os << pCoor(-7,2,-2) << pCoor(-7,4,-2) << pCoor(-5,2,-2);
colors   << color_red      << color_green    << color_blue;

coors_os << pCoor(-7,4,-2) << pCoor(-5,4,-2) << pCoor(-5,2,-2);
colors << color_lsu_spirit_gold << color_lsu_spirit_gold
       << color_lsu_spirit_gold;

gc.vtx_coors_set(coors_os);      // Coordinates to render.
gc.vtx_colors_set(colors);       // Colors to render.
```

After execution of the code above `Our_3D` knows about the coordinates and colors. Remember (from class) that *vertex* refers to a coordinate, color and other information. The code above sent data on six vertices, for each a color and coordinate. Next we tell `Our_3D` how to group our vertices into triangles:

```
gc.topology_set(Topology_Individual);  // Use individual triangle grouping.
```

The three possible options are `Topology_Individual`, `Topology_Strip`, and `Topology_Fan`. For the excerpt above individual triangles is correct because it matches how the vertices in `coors_os` are ordered. (Other topologies are considered in Problem 1 and 3.)

So far we've only provided information on what we want to do, but nothing has been drawn yet. To actually draw (write the frame buffer) the draw command must be issued:

```
gc.draw_rasterization();          // Tell Our3D to render now.
```

If we want to render objects with different topologies, the need to be done with separate draw commands. See the code in `World::render_scene()`.

### Resources

A good reference for C++ is `https://en.cppreference.com/w/`. For an examples on how to insert coordinates into vertex lists (*e.g.*, `coors_os`, `colors`) see 2024 Midterm Exam Problem 1. For an example of modifications to `Our_3D`, the subject of Problems 1 and 3, see 2024 Midterm Exam Problem 2. Also see the 2024 Midterm Exam solution code in the repository in directory `hw/gpup/2024/mt/mt.cc`.

### 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 process that takes effort, but one that also provides the satisfaction of progress and of developing skills and understanding.

**Problem 1:** The code in `World::render_scene()` near `Problem 1 solution goes` (search for it) inserts coordinates into `coors_os` in an order suitable for individual triangles, but when the green text shows `HW3 P1: STR` (change using `1`) the draw command expects a triangle strip:

```
if ( opt_hw3_p1_strips ) {
    // Render shape using a triangle strip. (When Problem 1 solved.)
    gc.topology_set( Topology_Strip );

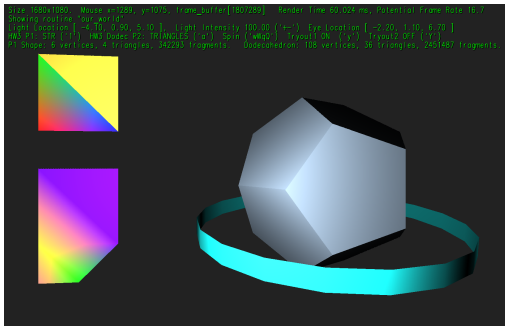
    //  Problem 1 solution goes here.

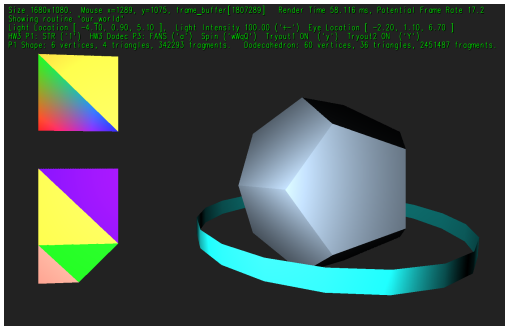
    coors_os << pCoor(-6,-2,-2) << pCoor(-7,-2,-2) << pCoor(-7,-1,-2);
    colors   << color_salmon    << color_salmon    << color_salmon;

    coors_os << pCoor(-7,1,-2) << pCoor(-5,1,-2) << pCoor(-5,-1,-2);
    colors << color_lsu_spirit_purple << color_lsu_spirit_purple
           << color_lsu_spirit_purple;

    coors_os << pCoor(-7,-1,-2) << pCoor(-7,1,-2) << pCoor(-5,-1,-2);
    colors << color_lsu_spirit_gold << color_lsu_spirit_gold
           << color_lsu_spirit_gold;

    coors_os << pCoor(-7,-1,-2) << pCoor(-6,-2,-2) << pCoor(-5,-1,-2);
    colors   << color_green     << color_green     << color_green;
```





(*a*) Modify the code so that vertices are in an order suitable for a triangle strip. When solving this make sure the green text shows `STR` to the right of `HW3 P1:` (press `1` if necessary). In a correct solution the shape should be rendered using six vertices forming 4 triangles. (In an unsolved file there will be 12 vertices forming 10 triangles.) Before the next subproblem is finished the triangle colors will blur into each other, see the screenshot to the upper left.

(*b*) When rendering individual triangles we sometimes want each vertex to have its own color, perhaps for colorful gradient effects such as the red/green/blue triangle in the upper left. But, more often we want one color for a triangle. In the unmodified code `Our_3D` uses a separate color for each vertex. Modify `Our_3D::draw_rasterization()` so that when triangle strips are rendered the color of a triangle is based on one vertex, not a blend. If necessary modify the solution to the previous subpart so that the colors are correct. When solved correctly the Problem 1 shape triangle colors will appear as in the screenshot to the upper right.

**Problem 2:** The code in `World::render_scene()` renders the dodecahedron two ways, using individual triangles (this problem) or triangle fans (next problem). For individual triangles the code takes dodecahedron coordinates from `coors` and places them in `coors_os`:

```
switch ( opt_hw3_variation ){
case HW3_Triangles:
  // Homework 3 Problem 2

  for ( size_t i=0; i<coors.size(); i+=5 )
    for ( int j=0; j<5; j++ )
      coors_os << coors[ i + j ];

  gc.topology_set(Topology_Individual);
  break;
```

The code above won't render the dodecahedron correctly because the order of vertices (and their number) in `coors` does not specify individual triangles.

Modify the code so that it emits individual triangles. To do so vertices in `coors` must be inserted into `coors_os` multiple times. Do not worry about colors or normals, they are inserted further below in the code.

To see the effect of these changes make sure that the green text shows `TRIANGLES` to the right of `HW3 Dodec P3:`.

**Problem 3:** The code in `World::render_scene()` for rendering the dodecahedron using triangle fans is correct (the code in `Our_3D` is not):

```
case HW3_Fans:
  // Problem 3
  // Don't change this code, modify the code in Our_3D.
  gc.topology_set(Topology_Fan,5); // Use a fan grouping.
  coors_os = coors;
```

The `topology_set` command above tells `Our_3D` to expect triangle fans **and** that each triangle fan will be described by five vertices. It then just copies the dodecahedron vertex coordinates, `coors`, into the container we will send to `Our_3D`, `coors_os`. The vertices in `coors_os` are grouped into pentagons. These work as triangle fans.

The problem is that the code in `Our_3D` works with individual triangles and triangle strips, but not triangle fans:

```
const bool topology_strip = topology == Topology_Strip;
const bool topology_fan [[maybe_unused]] = topology == Topology_Fan;
size_t inc = topology_strip ? 1 : 3;
const ssize_t stop = coors_os.size() - 2;

for ( ssize_t i=0; i<stop; i += inc )
  {
    ssize_t idx_0 = i;
    ssize_t idx_1 = i + 1;
    ssize_t idx_2 = i + 2;

    pCoor o0 = coors_os[idx_0],  o1 = coors_os[idx_1],  o2 = coors_os[idx_2];
```

Modify this code so that it renders $n$-vertex triangle fans correctly, where $n$ is the value of variable `topology_n_vertices`. (To solve this problem one needs to understand how triangle fans work.) For the dodecahedron we set $n = 5$, but the code should work for any $n$.

Code for implementing a triangle fan appeared on the 2024 midterm exam. However, in that midterm exam there was just one fan per draw, here there are 12 fans for the dodecahedron, each consisting of `topology_n_vertices` vertices. The midterm exam code is in the repo in the same directory as the 2024 homework assignments.

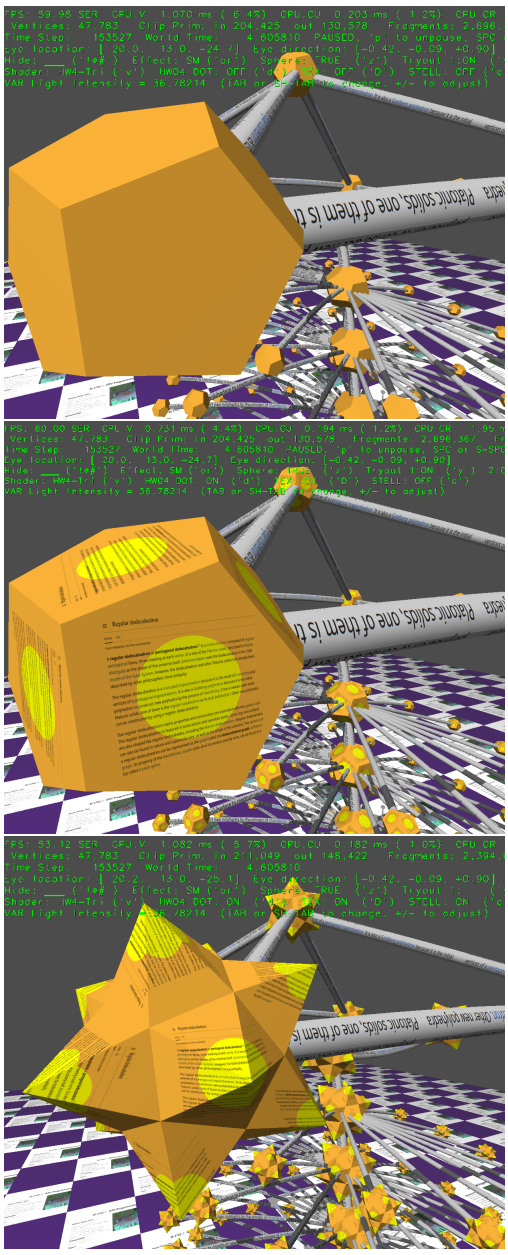
To see the effect of these changes make sure that the green text shows `FANS` to the right of `HW3 Dodec P3:`. When this is solved correctly the green text should show $5 \times 12 = 60$ vertices and $3 \times 12 = 36$ triangles for the dodecahedron.

Of course, `Our_3D` should continue to show other geometry correctly.

**LSU EE 4702-1** **Homework 4** **Due: 21 November 2025**

*Formatted 10:00, 8 December 2025*

*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/2025/hw04-shdr.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 canopy constructed from dodecahedra and links. (This is scene 2 from the links demo code, but with the balls replaced by dodecahedra.) See the screenshot to the upper right. When Problem 1 is correctly solved each dodecahedron face has a yellow dot and shows text from the Wikipedia article on dodecahedra. See the middle screenshot to the right. When Problem 2 is correctly solved the dodecahedra are *stellated*, meaning that the pentagon on each face is replaced by a five-sided pyramid. See the screenshot to the lower right.

## User Interface

The code can display several scenes, numbered 0 through 7. The currently displayed scene is identified in the penultimate line of green text. For this assignment any scene can be used, Scene 1 is the default because the eye is positioned close to a large dodecahedron. 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. The code for scenes $0 \leq x \leq 7$ are in routine `World::ball_setup_`$x$.

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 `hw04.png` or `hw04-debug.png`. Press `F10` to start recording a video, and press `F10` to stop it. The video will be in file `hw04-1.webm` or `hw04-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. Normally, these variables specify things such as the gravitational acceleration, and 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

Variables and the UI for setting them have been provided for features to be coded in the problems. Pressing `d` will toggle the value of Boolean `opt_hw04_dot`. When correctly solved the gold dot will be shown only when `opt_hw04_dot` is `true`. Pressing `D` will toggle the value of Boolean `opt_hw04_tex`. When correctly solved textures will be shown on the dodecahedron faces only when `opt_hw04_tex` is `true`. Pressing `c` will toggle the value of Boolean variable `opt_hw04_stel`. When correctly solved the dodecahedra faces will be stellated when `opt_hw04_stel` is `true`. Also available for stellation is `float` variable `opt_hw04_height_stel`, which indicates the height of the stellation pyramid in dodecahedron local space units. A value of 0 indicates no stellation. The value can be changed using the UI for changing variables described above, it will be labeled using the variable name, `opt_hw04_height_stel`. The values of the Boolean variables are shown on the penultimate line of green text. See the screenshots on the first page of this assignment.

The balls can be rendered using three different sets of shaders, labeled `HW4-Tri`, `HW4-Clean`, and `Balls`, pressing `v` cycles between these shaders. Shader set `Balls` are the ones used in the links code, they show the links attached to balls. Shader sets `HW4-Tri` and `HW4-Clean` render dodecahedra instead of balls. Shaders for both of these are in file `hw04-shdrs.cc`. The `HW4-Tri` shaders, `vs_main`, `gs_main`, and `fs_main` are to be used for the solution to this assignment. The `HW4-Clean` shaders, `vs_main_clean`, `gs_main_clean`, and `fs_main_clean`, are there for reference or for light experimentation.

### Code Generation and Debug Support

The compiler generates an optimized version of the code, `hw04`, and a debug-able version of the code, `hw04-debug`. The `hw04-debug` version is compiled with optimization turned off, which makes it easier to debug. When needed, you are strongly encouraged to run `hw04-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 three Boolean tryout variables, `opt_tryout1` through `opt_tryout3`, 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 `x`, `y` and `Y` to toggle the values of the Boolean variables; their values are shown in the green text at the label `Tryout 1:`, `2:`, and `3:`. The user interface can also be used to modify host floating-point variable `opt_tryoutf` using the `Tab`, `+`, and `-` keys, see an earlier section.

### Code Overview

For this assignment only shader code, in file `hw04-shader.cc`, is to be modified. That said, the description here starts in the host code, which may be the subject of the next homework assignment.

This assignment is based on the links demo code, in directory `proj-base/v-links`. It has been modified so that the balls, by default, will be rendered using newly inserted dodechedron code. That code is in routine `World::render_doda`. (Pressing `v` will switch between the ball shader and the dodecahedron shader(s).)

Routine `World::render_doda` is called when the frame buffer (actually surface) is to be written, normally 60 times a second. The routine does any required pipeline setup, populates a buffer set to show each ball as a dodecahedron, and then *records* a *draw* with the buffer set providing vertex shader inputs to the pipeline. Be warned: the use of the pipeline is inefficient, something that will be fixed in a follow-up assignment.

Pipeline `pipe_doda` is set up by:

```
if ( !pipe_doda )
  pipe_doda
    .init( vh.qs )
    .ds_follow( puni_light_curr )
    .ds_follow( transform )
    .ds_uniform_use( "BIND_UNI_COMMON", buf_uni_common )
    .ds_use( sampler, texid_hw )
    .shader_inputs_info_set<pCoor, pColor, pMatrix>()
    .shader_code_set
    ("hw04-shdr.cc",
     "vs_main();",
     "gs_main();",
     "fs_main();", "#define HW4_TRI\n" )
    .topology_set( vk::PrimitiveTopology::eTriangleList )
    .create();
```

This sets up a Vulkan rasterization pipeline as follows. Call `shader_code_set` specifies that the shader code is to be read from from file `hw04-shdr.cc` and that the names of the vertex, geometry, and fragment shaders are executed using `vs_main();`, `gs_main();`, and `fs_main();`. The `.topology_set` call specifies that the inputs to the geometry shader are formed using a triangle list topology. The `shader_inputs_info_set` call specifies that the buffer set will include coordinates, colors, and $4 \times 4$ matrices.

The calls starting `ds_` specify other data available to shaders. (The "ds" is for *descriptor set*.) The `puni_light_curr` call provides lighting information. The solution to this assignment (Fall 2025 Homework 4) does not involve lighting, but those who are curious might look in `shader-include/light.h` and `shader-common-generic-lighting.h`. Call `ds_follow(transform)` specifies that `transform` is to be bound to a uniform. Look for `BIND_UNIFORM` in `hw04-shdr.cc`. Call `ds_uniform_use( "BIND_UNI_COMMON", buf_uni_common )` specifies another uniform. See file `links-shdr-common.h` for the members of that uniform. Finally, call `ds_use( sampler, texid_hw )` sets up a texture unit. Variable `sampler` indicates how the texture will be filtered, and `texid_hw` is the image object holding the texture. For this assignment the texture is constructed from a screenshot of the Wikipedia regular dodecahedron page. Remember that for this assignment the host code, including the code above, is not to be modified.

The code that follows writes, if necessary, dodecahedron coordinates to vector `dod_coors`. The dodecahedron coordinates are in a local space with the dodecahedron center at the origin and with an *inscribed radius* of 1. That is, a sphere of radius 1 would be inside the dodecahedron and touch the center of each dodecahedron face. This is the same code used in Homework 2 and 3.

Next, the code inserts vertices to render a dodecahedron for each ball. The code starts by computing a transformation matrix that will map a coordinate in the dodecahedron local space to

the global space used by ball (such as for `b->position`). The transformation translates the center of the dodecahedron to the ball center, scales the dodecahedron to the same radius as the ball, and rotates it to the same orientation:

```
  for ( Ball* b: balls )
    {
      pMatrix_Translate tra(b->position);
      pMatrix_Scale sca(b->radius);
      pMatrix rot(b->omatrix);
      pMatrix m = tra * sca * rot;
```

Next the code inserts dodecahedron vertex coordinates, colors, and the transformation matrix for this ball into the buffer set. It does so using the local-space coordinates in `dod_coors`, emitting three triangles for each face (just as in the solution to Homework 3 Problem 2):

```
      for ( size_t i=0; i<dod_coors.size(); i+=5 )
        for ( int j=1; j<4; j++ )
          bset_doda
            << dod_coors[ i ]
            << dod_coors[ i + j ]
            << dod_coors[ i + j + 1 ];
```

Note that in the code above each `i` loop iteration emits the three triangles for one face. All three of those triangles start with the same vertex, `dod_coors[i]`. This fact is important for solving this assignment.

Remember that the values in `dod_coors` are in the dodecahedron local space. We leave it to the shaders to transform them into global coordinates using `m`. The code for this assignment uses an inefficient method to send `m` to the shaders: as a vertex shader input. The next lines put `m` and also the color into the buffer set:

```
      for ( int i=0; i<3*3*12; i++ ) bset_doda << b->color << m;
}
```

The loop above iterates 108 times (three vertices per triangle, three triangles per face, twelve faces per dodecahedron). For each iteration it inserts the exact same data. That's wasteful!, why not send it just once? We will, but not in this assignment.

## Resources

A good reference for C++ is `https://en.cppreference.com/w/`. For an examples on how to insert coordinates into vertex lists (*e.g.*, `coors_os`, `colors`) see 2024 Midterm Exam Problem 1.

Information on OpenGL Shading Language can be found in the reference, OpenGL Shading Language 4.60, linked to the course references page, `https://www.ece.lsu.edu/koppel/gpup/ref.html`.

Like Problem 1 in this assignment, texture coordinates are the subject of 2023 Homework 4 Problem 3 and 2024 Homework 5 Problems 1 and 2. Unlike those past assignments, in this one the texture coordinates are to be computed in the shaders (rather than being provided as vertex shader inputs).

## 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.)

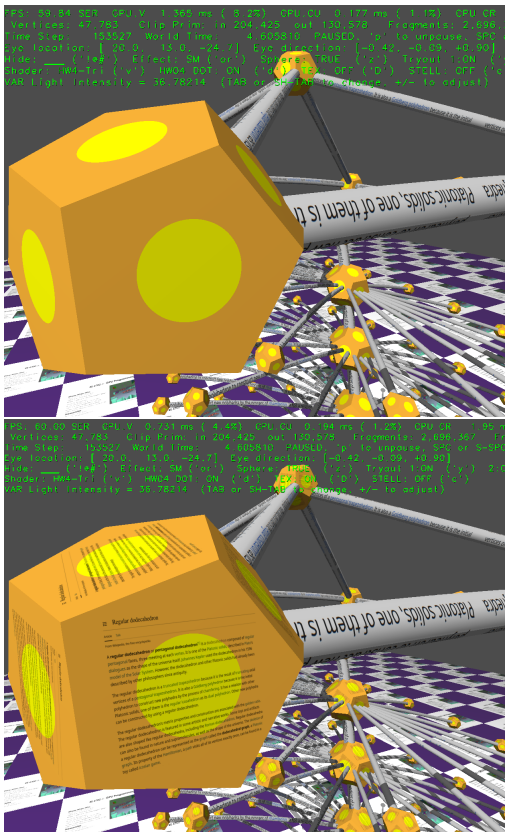
After availing oneself to these resources **each student is expected to be able to complete the assignment alone**. Test questions will be based on homework questions and **the assumed time needed to complete the question will be for a student who had solved the homework assignment on which it was based**.

### 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 process that takes effort, but one that also provides the satisfaction of progress and of developing skills and understanding.

**Problem 1:** As described below, modify the code to show the gold dot (top right screenshot) and the texture (lower right screenshot). These two tasks may seem unrelated, but they're not. In both cases one needs to provide the fragment shader with a face-local coordinate of the fragment. (Consider a fragment at the center of a face. Its face-local coordinate, which has two dimensions, might be $(0, 0)$. That face-local coordinate would be $(0, 0)$ for a fragment at the center of any dodecahedron face.) For the gold dot, the face-local coordinate could be used to determine the distance from the center (and so whether to use gold or the default color). For the texture, the face-local coordinate could be used to compute texture coordinates.



The code in the geometry shader computes quantities that might be useful for computing face-local coordinates. This includes `ctr_pentagon_e`, the center of the face. Also useful, is the fact that `In[0]` is the same vertex for all three triangles and so could be used, along with `tn`, the normal, for finding a face-local $a_x$ and $a_y$. Let $L_e$ be the origin of local space in eye-space coordinates (perhaps `ctr_pentagon_e`), and $a_x$ and $a_y$ be the local space axes (in eye space). Let $P_e$ be an eye-space coordinate. Then the $x$ component in local space would be $(P_e - L_e) \cdot a_x$, where $v \cdot u$ is the dot product of $v$ and $u$.

A face-local coordinate would need to be computed for each triangle vertex and written to a new geometry shader output, fragment shader input.

To use your own image(s) edit the lines:

```
texid_syl.init
  ( vh.qs, P_Image_Read( vh.gp_root_get() / "vulkan" / "gpup.png", 255 ) );
texid_emacs.init( vh.qs, P_Image_Read("mult.png",-1) );
texid_hw.init( vh.qs, P_Image_Read("wiki-dod.png",255) );
```

and change the file names, such as `wiki-dod.png`, to files containing your own images. The dodecahedron faces use texture id `texid_hw`, the balls (when not shown as dodecahedra) use `texid_emacs`, and the checkered platform uses `texid_syl`. The images are read from files by the Image Magick library. The library can read many formats, including JPEG, PNG, and PDF. Use the command `magick -list format` to get the list.

Those who are just not satisfied guessing their own RGB components for gold have multiple options to procrastinate. For more information on the old X11 colors used in the course `~/gp/include/colors.h` see `https://en.wikipedia.org/wiki/X11_color_names`. Those who want to strike just the right balance when choosing a color that will be adjacent to others might visit `https://www.sessions.edu/color-calculator/`. Those who want to reflect on our understanding of color might visit `https://xkcd.com/1882/`.

(*a*) Modify the code in `hw04-shdr.cc` so that when `opt_hw04_dot` is true a gold circle is shown centered on the face of each dodecahedron. Do this by changing the color in the fragment shader,

**not by emitting new primitives**. The radius, $r$, should be exactly $\frac{1}{2}$ the pentagon inscribed circle radius. That is, the smallest distance from the circle circumference to a pentagon edge should be $r$.

- ☐ Show the circle centered on each face.
- ☐ Do not emit new primitives, instead ☐ change the color in the fragment shader.
- ☐ Add new geometry shader output(s) and fragment shader inputs(s).
- ☐ The circle radius should be half the pentagon inscribed radius for each dodecahedron, regardless of size.
- ☐ For help in figuring out the radius look at the code computing `ctr_pentagon_e`.

(*b*) Modify the code in `hw04-shdr.cc` so that when `opt_hw04_tex` is true the texture bound to `tex_unit_0` is mapped to each dodecahedron face. The texture should be sized so that its corners touch the pentagon inscribed circle. Show exactly one copy on each face.

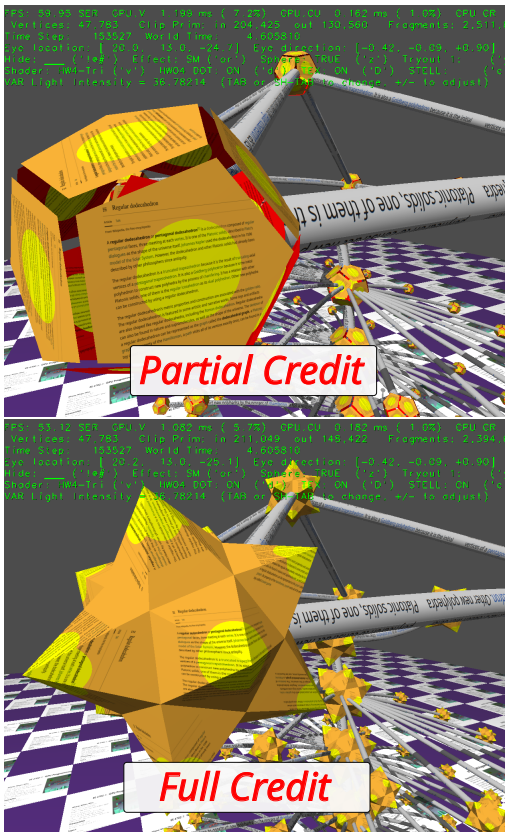
In the unmodified code a texel is retrieved using a constant coordinate and nothing is done with the texel. Use this as a starting point:

```
void fs_main()
{
  vec2 tc = vec2(0.5,0.5);
  vec4 texel = texture(tex_unit_0,tc);
```

- ☐ Show the texture on a dodecahedron face.
- ☐ Show one copy per face, the edges should touch the inscribed circle.
- ☐ The image should not be distorted (when stellation is off).

*There's another problem on the next page.*

**Problem 2:** Solve part (a) below for partial credit, part (b) for full credit. If part (b) is solved there is no need to solve part (a). If you solve both and just can't bring yourself to leave your work on part(a) dormant use `opt_tryout3` to switch between the two.




(*a*) *For partial credit solve this part instead of part b:* Modify the code in `gs_main` so that when `opt_hw04_stel` is true the pentagon is shrunk by about 10%. That is, each of the pentagon vertices moves 10% toward the pentagon center. Note that the coordinates of the pentagon center are already computed in the geometry shader. Also recall that the vertices provided to the geometry shader are pentagon vertices (none of them are inside the pentagon). See the screenshot to the upper right.

☐ Render a shrunken pentagon.

☐ This part can be skipped if the subpart below is solved.

(*b*) *Solve instead of (or in addition to part a), for Full Credit:* Modify the code in `gs_main` so that when `opt_hw04_stel` is `true` stellated dodecahedra with local-space height `opt_hw04_height_stel` are rendered. A stellated dodecahedron is obtained from a regular dodecahedron by putting a five-sided pyramid on each face. See the screenshot to the lower right.

Note that the coordinates of the pentagon center are already computed in the geometry shader. Another handy variable is `is_pentagon_edge`. It is `true` when coordinates `pt_i_e` and `pt_n_e` form a pentagon edge. (One or two edges of each triangle seen by the geometry shader are **not** pentagon edges.) To solve this the geometry shader may have to emit more than one triangle.

☐ Modify `gs_main` and surrounding code to emit a stellated dodecahedron.

☐ Update `max_vertices` if necessary.

☐ Make sure that `normal_e` written by the geometry shader is correct.

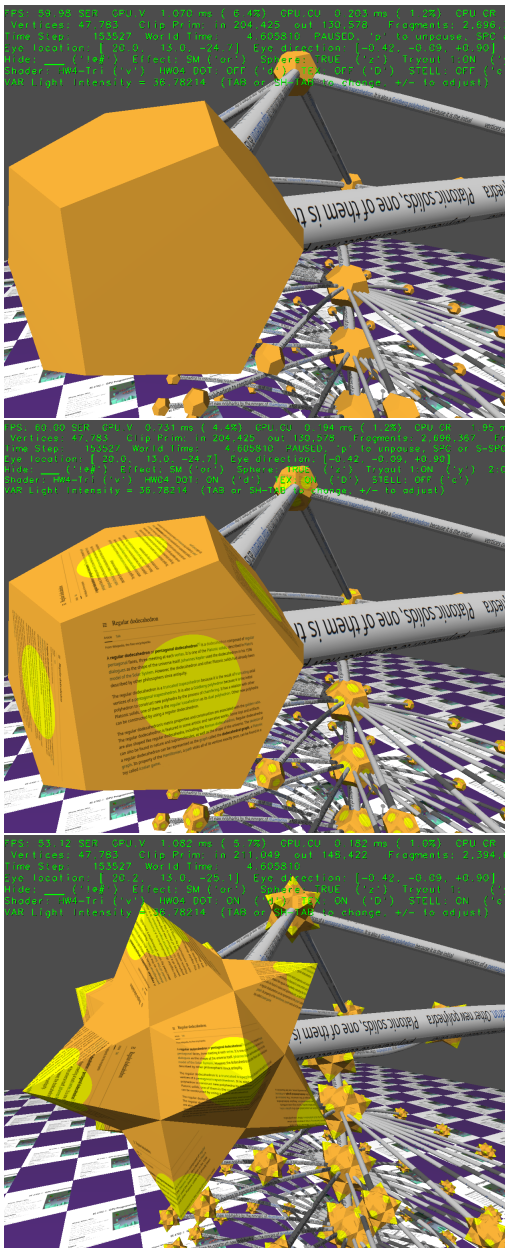
☐ The edges between the faces of the pyramid should be sharp (not blurry).

**LSU EE 4702-1** **Homework 5** **Due: 30 November 2025**

*Formatted 19:04, 22 November 2025*

*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/2025/hw05.cc.html` *and* `https://www.ece.lsu.edu/koppel/gpup/2025/hw05-shdr.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 canopy constructed from dodecahedra and links. In fact, it should look just like the scene from Homework 4. See the screenshot to the upper right. The code in this assignment is based on the solution to Homework 4, and the UI works the same way, except that there are more shaders. Solving this assignment correctly will not change what is show other than the performance data shown in the green text. The three screenshots to the right are being shown so avoid the need to find three different images.



User Interface

The code can display several scenes, numbered 0 through 7. The currently displayed scene is identified in the penultimate line of green text. For this assignment any scene can be used, Scene 1 is the default because the eye is positioned close to a large dodecahedron. 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. The code for scenes $0 \leq x \leq 7$ are in routine `World::ball_setup_`$x$.

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 `hw05.png` or `hw05-debug.png`. Press `F10` to start recording a video, and press `F10` to stop it. The video will be in file `hw05-1.webm` or `hw05-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. Normally, these variables specify things such as the gravitational acceleration, and 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

Variables and the UI for setting them have been provided for features to be coded in the problems. Pressing `d` will toggle the value of Boolean `opt_hw05_dot`. When correctly solved the gold dot will be shown only when `opt_hw05_dot` is `true`. Pressing `D` will toggle the value of Boolean `opt_hw05_tex`. When correctly solved textures will be shown on the dodecahedron faces only when `opt_hw05_tex` is `true`. Pressing `c` will toggle the value of Boolean variable `opt_hw05_stel`. When correctly solved the dodecahedra faces will be stellated when `opt_hw05_stel` is `true`. Also available for stellation is `float` variable `opt_hw05_height_stel`, which indicates the height of the stellation pyramid in dodecahedron local space units. A value of 0 indicates no stellation. The value can be changed using the UI for changing variables described above, it will be labeled using the variable name, `opt_hw05_height_stel`. The values of the Boolean variables are shown on the penultimate line of green text. See the screenshots on the first page of this assignment.

The balls can be rendered using five different sets of shaders, labeled `PLAIN`, `HW5-Tri`, `HW5-Pts-Cln`, `HW5-Pts-P1`, and `HW5-Inst-P2`. Pressing `v` cycles between these shaders. Shader set `PLAIN` are the ones used in the links code, they show the links attached to balls. The other shader sets render dodecahedra instead of balls. Shaders for all of these are in file `hw05-shdrs.cc`. Shader sets `HW5-Pts-P1` and `HW5-Inst-P2` are to be modified as part of the solution to this assignment. The shaders are named `vs_main_pts_prob_1`, `gs_main_pts_prob_1`, `vs_main_inst_prob_2`, and `gs_main_inst_prob_2`. The other shaders are for reference and for light experimentation.

## Code Generation and Debug Support

The compiler generates an optimized version of the code, `hw05`, and a debug-able version of the code, `hw05-debug`. The `hw05-debug` version is compiled with optimization turned off, which makes it easier to debug. When needed, you are strongly encouraged to run `hw05-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 three Boolean tryout variables, `opt_tryout1` through `opt_tryout3`, 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 `x`, `y` and `Y` to toggle the values of the Boolean variables; their values are shown in the green text at the label `Tryout 1:`, `2:`, and `3:`. The user interface can also be used to modify host floating-point variable `opt_tryoutf` using the `Tab`, `+`, and `-` keys, see an earlier section.

## Code Overview

For this assignment both shader code, in file `hw05-shader.cc`, and the host (CPU) code in file `hw05.cc` are to be modified.

This assignment is based on the links demo code, in directory `proj-base/v-links`. It has been modified so that the balls, by default, will be rendered using dodechedron code. That code is in routine `World::render_doda`. (Pressing `v` will switch between the ball shader and the dodecahedron shader(s).)

Routine `World::render_doda` is called when the frame buffer (actually surface) is to be written, normally 60 times a second. The routine prepares all of the dodecahedron shaders, so it's a bit busy. It starts by doing any required pipeline setup, populates buffer sets to show each ball as a dodecahedron, and then *records* a *draw* with the buffer set providing vertex shader inputs to the pipeline.

The routine starts by initializing Vulkan storage buffers. These only need to be initialized once. You will only need to modify this code if you want to add your own storage buffers (though you should not need to):

```
if ( !buf_hw05_color )
  {
    buf_dod_coors_l.init( vh.qs, vk::BufferUsageFlagBits::eStorageBuffer );
    buf_hw05_color.init( vh.qs, vk::BufferUsageFlagBits::eStorageBuffer );
    buf_hw05_matrix.init( vh.qs, vk::BufferUsageFlagBits::eStorageBuffer );
    buf_hw05_vector.init( vh.qs, vk::BufferUsageFlagBits::eStorageBuffer );
    buf_hw05_coor.init( vh.qs, vk::BufferUsageFlagBits::eStorageBuffer );
  }
```

Each storage buffer holds an array of items. Some or all of the storage buffers above will be needed in this assignment. If you'd like to add your own storage buffers look for occurrences of one of the buffers, say `buf_hw05_coor`, to see what needs to be done.

Next the pipelines are initialized. As with storage buffers, this only happens once for run. Consider the initialization of the pipeline for shader set `HW5-Pts-Cln`:

```
if ( !pipe_doda_pts_clean )
  {
    pipe_doda_pts_clean
      .init( vh.qs )
      .ds_follow( puni_light_curr )
      .ds_follow( transform )
      .ds_use( sampler, texid_hw )
      .ds_uniform_use( "BIND_UNI_COMMON", buf_uni_common );

    pipe_doda_pts_clean
      .ds_storage_follow
      ( "BIND_HW5_COLOR", buf_hw05_color, shader_stages_all );

    pipe_doda_pts_clean
      .shader_inputs_info_set<pCoor, pVect, pMatrix, int>()
      .shader_code_set
      ("hw05-shdr.cc",
       "vs_main_pts_clean();",
       "gs_main_pts_clean();",
       "fs_main();", "#define HW5_PTS_CLEAN\n" )
      .topology_set( vk::PrimitiveTopology::ePointList );

    pipe_doda_pts_clean.create();
  }
```

The initialization starts with the call `pipe_doda_pts_clean.init( vh.qs )` and then continues with member functions (operating on the returned reference to `pipe_doda_pts_clean`) specifying resources that will be accessed by the shaders in the pipeline. Many of these member function names start with `ds_`, the "ds" is for *descriptor set*. The first group above specifies lighting information, transformations, a texture/sampler pair, and a uniform. All of the pipelines used for this assignment access these same resources. The `puni_light_curr` call provides lighting information. The solution to this assignment (Fall 2025 Homework 5) does not involve lighting, but those who are curious might look in `shader-include/light.h` and `shader-common-generic-lighting.h`. Call `ds_follow(transform)` specifies that `transform` is to be bound to a uniform. Look for `BIND_UNIFORM` in `hw05-shdr.cc`. Call `ds_uniform_use( "BIND_UNI_COMMON", buf_uni_common )` specifies another uniform. See file `links-shdr-common.h` for the members of that uniform. Call `ds_use( sampler, texid_hw )` sets up a texture unit. Variable `sampler` indicates how the texture will be filtered, and `texid_hw` is the image object holding the texture. For this assignment the texture is constructed from a screenshot of the Wikipedia regular dodecahedron page.

Next the pipeline is set to use storage buffer `buf_hw05_color`. The string `BIND_HW5_COLOR` is defined as a macro in the shader code and its value is the buffer binding point. The third argument, `shader_stages_all`, specifies which shader stages can access the buffer.

The next group specifies the shader code. It consists of three calls, the first, `shader_inputs_info_set`, specifies the data types that will be used as inputs to the vertex shader, in this case `pCoor`, `pVect`, `pMatrix` (oh my!), and `int`. The buffer set used with the draw command, in this case `pipe_doda_pts_clean.record_draw( cb, bset_doda_pts_clean )`, must have buffers with those types populated. More on that later.

The call `shader_code_set` specifies: the name of the file holding the shader code, commands used to call the vertex shader, geometry shader, and fragment shader. The last argument specifies code to put at the top of the file when running this particular shader. The last call in the group specifies the topology used to group vertices at the input to the geometry shader. The geometry shader code must declare a consistent input topology. In this particular case the geometry shader must declare points as its inputs, which it does.

The last call is `pipe_doda_pts_clean.create()` which creates the pipeline. In the excerpt above the calls were broken into four groups to make it easier to describe them. In other cases there's just one group.

To solve both Problems 1 and 2 their respective pipelines will need to be modified. The modifications include adding or removing storage buffers, changing the inputs to the vertex shaders (by modifying `shader_inputs_info_set`), and perhaps changing the geometry shader input topology.

The code that follows sets up other dodecahedron pipelines. After that is the code that, if necessary, computes dodecahedron coordinates in local space. Those coordinates will be placed in Vulkan storage buffer `buf_dod_coors_l`. If the buffer is empty the coordinates are computed and sent to the GPU:

```
if ( buf_dod_coors_l.size() == 0 ) {
    pCoor center(0,0,0);
    pVect ax(1,0,0), ay(0,1,0), az(0,0,1);
    float ri = 1;
    const float len_edge = ri / sqrt( sqrt(5) * pow(numbers::phi,5) / 20 );
    const float rc_pentagon = len_edge / ( 2 * sinf(numbers::pi/5) );
    pCoor center_pentagon_0 = center + ri * az;
    float theta_dihedral = 2 * atanf( numbers::phi );
    const float d_theta = 2 * numbers::pi / 5;
    for ( int i=0; i<5; i++ ) {
        const float theta = d_theta * i;
```

```
        pVect dir = cosf(theta) * ax + sinf(theta) * ay;
        pCoor pos = center_pentagon_0 + rc_pentagon * dir;
        buf_dod_coors_l << pos;  }
    for ( int i=0; i<5; i++ ) {
        pCoor p0 = buf_dod_coors_l[i], p1 = buf_dod_coors_l[(i+1)%5];
        pMatrix_Translate to_p0( p0 );
        pMatrix_Rotation rot2( pNorm(p0,p1), -theta_dihedral );
        pMatrix_Translate to_Origin( pVect( p0, pCoor(0,0,0) ) );
        pMatrix m = to_p0 * rot2 * to_Origin;
        for ( int j=4; j>=0; j-- ) buf_dod_coors_l << m * buf_dod_coors_l[j]; }

    pMatrix_Rotation rot3(ay,numbers::pi);
    const int n_vtx_demi = buf_dod_coors_l.size();
    for ( int i=0; i<n_vtx_demi; i++ ) buf_dod_coors_l << rot3 * buf_dod_coors_l[i];
    buf_dod_coors_l.to_dev();
  }
```

After preparing the local coordinates the code sets up buffer sets and then does a draw using the chosen shader set (and pipeline) in the `switch( opt_shader )` construct. The switch uses enumeration constants to identify them, such as `SO_HW5_Triangles` for `HW5-Tri`.

Each shader set operates differently, though they will all result in the same images being shown. The shaders differ in how they get their data and in how much computation they do.

Shader set `HW5-Tri` (`SO_HW5_Triangles`) is similar to the Homework 4 shader. The input to a vertex shader invocation consists of a dodecahedron vertex local-space coordinate, a local-to-object transformation matrix (oh my!), and a color. Each face is rendered using three triangles, and so for each dodecahedron $3 \times 3 \times 12 = 108$ vertices are entered. Each vertex of a particular dodecahedron is given the same color and transformation matrix. How wasteful!

```
  case SO_HW5_Triangles:
    bset_doda.reset( pipe_doda );
    for ( Ball* b: balls ) {
        pMatrix rot(b->omatrix);
        pMatrix_Translate tra(b->position);
        pMatrix_Scale sca(b->radius);
        pMatrix m = tra * sca * rot;
        for ( size_t i=0; i<buf_dod_coors_l.size(); i+=5 )
          // Render the pentagon by emitting three triangles, all
          // share pentagon vertex 0 (index i).
          //
          for ( int j=1; j<4; j++ )
            bset_doda
             << buf_dod_coors_l[ i ] << buf_dod_coors_l[ i + j ] << buf_dod_coors_l[ i + j +
1 ];

        for ( int i=0; i<3*3*12; i++ ) bset_doda << b->color << m;
      }
    bset_doda.to_dev();
    pipe_doda.record_draw( cb, bset_doda );
```

The code above populates the buffer set, sends it to the device, and records a draw.

The second shader, `HW5-Pts-Cln` (`SO_HW5_Points_Clean`) uses a points topology instead of a triangle list. Each invocation of the geometry shader will render one pentagon (using the data from

the geometry one element shader input). Here the vertex shader input consists of a transformation matrix, the local-space coordinate of one vertex on a dodecahedron face, the face normal, and a *color index*. This pipeline, `pipe_doda_pts_clean`, binds the `buf_hw05_color` storage buffer. The code that sets up the buffer set also writes colors to the storage buffer. Let $n$ denote the number of dodecahedra to be rendered. The number of elements expected in the buffer set is $12n$ (one for each face). But since one color is used for an entire dodecahedron, the number of elements in `buf_hw05_color` is just $n$. The color index written to the buffer set is used by the shader code to access the color from `buf_hw05_color`. (The color is accessed in the geometry shader using `vec4 color = buf_hw05_color[In[0].color_idx];`.) Here is the code preparing the buffer set and storage buffer and the code recording the draw:

```
case SO_HW5_Points_Clean:
  bset_doda_pts_clean.reset( pipe_doda_pts_clean );
  buf_hw05_color.clear();
  for ( Ball* b: balls ) {
      pMatrix rot(b->omatrix);
      pMatrix_Translate tra(b->position);
      pMatrix_Scale sca(b->radius);
      pMatrix m = tra * sca * rot;
      const int color_idx = buf_hw05_color.size();
      buf_hw05_color << b->color;

      for ( size_t i=0; i<buf_dod_coors_l.size(); i+=5 ) {
          pNorm n( cross( buf_dod_coors_l[i+2],
                          buf_dod_coors_l[i+1],
                          buf_dod_coors_l[i  ] ) );
          bset_doda_pts_clean << m << buf_dod_coors_l[i] << n << color_idx;
        }}

  bset_doda_pts_clean.to_dev();
  buf_hw05_color.to_dev();
  pipe_doda_pts_clean.record_draw( cb, bset_doda_pts_clean );
```

Note that the geometry shader must render the pentagon using just one pentagon vertex, a normal, and a transformation matrix. That's possible because the coordinate is in local space, in which the dodecahedron center is at the origin. See the geometry shader `gs_main_pts_clean()` to see how that's done.

Alert students at this point may have realized at this point that `HW5-Pts-Cln` sends much less data from the CPU to the GPU, but that even less can be sent. One improvement is that the number of Vulkan vertices per dodecahedron has dropped from $3 \times 3 \times 12 = 108$ to just 12. And instead of sending a color, a color index is sent, which is one quarter the size. But, **but**, an entire transformation matrix is sent for each face even though it will be the same for each of the 12 faces. If a color index can be sent in place of a color, why not do the same for the transformation matrix? Yes, we could use the color index (which is really just a dodecahedron index) to retrieve the transformation matrix from a transformation matrix storage buffer. But Problem 1 goes further than that: not only will the transformation matrix be removed from the `HW5-Pts-P1` vertex shader input, but the vertex shader will have zero inputs. Nothing. Except for vertex shader built-in input `gl_VertexIndex` and geometry shader built-in input `gl_PrimitiveIDIn`. These provide a consecutive numbering of the shader inputs, starting at zero.

The third shader, `HW5-Pts-P1`, (`SO_HW5_Points_Prob_1`, `pipe_doda_pts_prob_1`) is for Problem 1. As described above, this is like `HW5-Pts-Cln`, except that the vertex shader has zero inputs

(and that should not be changed). Because it has zero inputs no buffer set is needed. The vertex shaders use `gl_VertexIndex` and geometry shaders use `gl_PrimitiveIDIn` to determine which item they are working on and retrieve any data they need from storage buffers. Because there is no buffer set, the draw command must be told how many times to invoke the vertex shader. In the unmodified assignment that's hardcoded to 1. See the code below, that sets up the color storage buffer and for reference has defined-out code for sending coordinates.

```
  case SO_HW5_Points_Prob_1:
    buf_hw05_color.clear();

    for ( Ball* b: balls )
      {
        pMatrix rot(b->omatrix);
        pMatrix_Translate tra(b->position);
        pMatrix_Scale sca(b->radius);
        pMatrix m = tra * sca * rot;

        int dod_idx = buf_hw05_color.size();
        buf_hw05_color << b->color;

        // Zero inputs, so defined-out code below not needed.
#if 0
        for ( size_t i=0; i<buf_dod_coors_l.size(); i+=5 ) {
            pNorm n( cross( buf_dod_coors_l[i+2], buf_dod_coors_l[i+1],
                            buf_dod_coors_l[i  ] ) );
            bset_doda_pts_prob_1 << m << dod_idx << buf_dod_coors_l[i] << n;
          }
#endif
      }
     const int n_vertices = 1; // This number is way too low.
     buf_hw05_color.to_dev();
     pipe_doda_pts_prob_1.record_draw( cb, n_vertices );
```

The last shader set `HW5-Inst-P2` (`SO_HW5_Inst_Prob_2`, `pipe_doda_inst_prob_2`) takes a different approach to reducing the amount of data sent from the CPU to the GPU. This pipeline is used for an *instanced draw*. Let $m$ denote the number of vertices in a buffer set and let $n$ denote the number of instances (the number of dodecahedra here). In an instanced draw the vertex shader is invoked $mn$ times. Suppose $n = 2$. The $m$ vertices will be streamed into the vertex shader twice (and the geometry shader will see primitives constructed from those vertices twice). If those shaders were not written for instanced draws, they would render the exact same objects twice, which would be a waste. But, if they were written for instanced draws, then something about the second time would be different, such as drawing the same object in a different location. When correctly solved the `HW5-Inst-P2` shaders will draw one dodahedron for each instance. The vertices in the buffer set will describe just one dodecahedron. In an instanced draw the vertex shader uses `gl_InstanceIndex` to determine the instance number. An example of an instanced draw is the code for rendering spheres in `links-shdr.cc`. Look for `vs_main_instances_sphere`. The input vertices are sphere coordinates in local space.

An advantage of an instanced draw, at least here, is that the buffer set does not need to be re-written each frame. That's because it is to describe a dodecahedron location in local space and of course that doesn't change. Instead, any per-instance data is placed in storage buffers. The

existing code just uses storage buffers for colors. The draw command for an instanced draw takes three arguments, the *command buffer* (used by most Vulkan calls recording command), the buffer set, and an integer, the number of instances. The existing code renders just one instance and that one instance renders all of the dodecahedra. That will need to be changed.

### Resources

A good reference for C++ is `https://en.cppreference.com/w/`. For an examples on how to insert coordinates into vertex lists (*e.g.*, `coors_os`, `colors`) see 2024 Midterm Exam Problem 1.

Information on OpenGL Shading Language can be found in the reference, OpenGL Shading Language 4.60, linked to the course references page, `https://www.ece.lsu.edu/koppel/gpup/ref.html`.

### 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 process that takes effort, but one that also provides the satisfaction of progress and of developing skills and understanding.

**Problem 1:** Modify the code in `hw05.cc` and `hw05-shdr.cc` so that when shader set `HW5-Pts-P1` is used the dodecahedra are rendered using a point list input topology with zero vertex shader inputs. The pipeline is already set for zero vertex shader inputs and a point topology. But other changes need to be made. To help you get started the code in `gs\_main\_pts\_prob\_1` has the "clean" shader code, but that code won't work because it lacks the vertex shader inputs.

- ☐ Modify the code setting up `pipe_dota_pts_prob_1` so that any additional storage buffers are bound to the pipeline. ☐ Don't overlook `buf_dod_coors_1`, which the already code prepares and sends to the GPU.
- ☐ Modify the code in the `SO_HW5_Points_Prob_1` switch case to ☐ populate and send any additional buffer objects and to ☐ call the `draw` with `n_vertices` set to the appropriate value.
- ☐ Modify the `gs_main_pts_prob_1` shader (and surrounding code) in `hw05-shdr.cc` to render the dodecahedron faces, using `gl_PrimitiveIDIn` to get the vertex number. (Because the input topology is points and there are no vertex shader inputs there's no need to modify the vertex shader.)
- ☐ Unlike the "clean" points shader, avoid computing the pentagon vertices, instead read them.

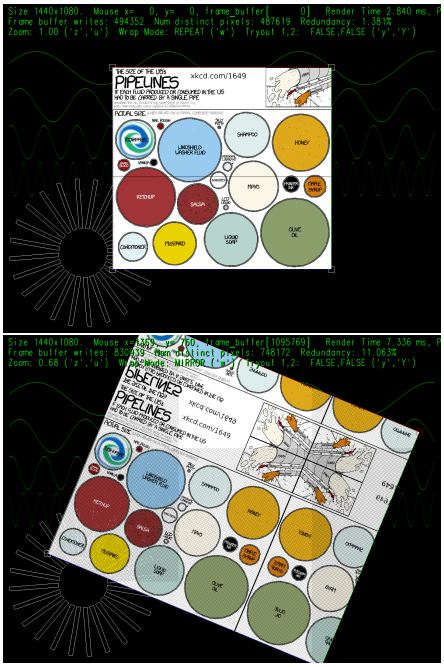
**Problem 2:** Modify the code in `hw05.cc` and `hw05-shdr.cc` so that when shader set `HW5-Inst-P2` is used the dodecahedra are rendered using an instanced draw. One big difference in an instanced draw is that the buffer set only need be sent once. So there is less of a need to keep it small. In this problem use the buffer set for triangle coordinates and texture coordinates, and anything else that's needed. Also, to get the stellation effect, insert coordinates for five triangles per face (one vertex is at the pentagon center, the other two form a pentagon edge). Have the geometry shader change the location of the center coordinate to get the stellation effect.

- ☐ Bind any additional storage buffers that are needed to `pipe_doda_inst_prob_2`. Don't bind more than are needed.
- ☐ Use the buffer set, `bset_doda_inst_prob_2`, for dodecahedron local coordinates and ☐ texture coordinates, and anything else that's needed.
- ☐ Write the buffer set once, not every frame.
- ☐ Write five triangles per face, with one vertex of each triangle in the pentagon center.
- ☐ Modify the shaders to use the instance number to get the appropriate data.
- ☐ The shaders should implement the stellation effect.

# 3 Fall 2024

## LSU EE 4702-1 Homework 1 Due: 14 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/hw01.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 some sine waves, radially arranged white rectangles, and an image of xkcd cartoon number 1649 (cropped). See the screenshot to the upper right. When the assignment is complete the image can be rotated and resized by dragging its corners, the image can be zoomed by pressing `z` and `u`, and the *wrap mode* can be changed by pressing `w` so that when more than one image fits in the image box it is either duplicated, mirrored, or just one copy is shown. The screenshot to the lower right shows a rotated, unzoomed, mirror-wrapped version.

### General 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 `hw01.png` or `hw01-debug.png`.

The value of two Boolean debug-support variables, `tryout1` and `tryout2`, are shown in the green text, pressing `y` toggles `tryout1` (between `true` and `false`) and pressing `Y` toggles `tryout2`. The variables are available in routine `render_hw01`.

### Problem-Specific User Interface

Pressing `z` will increase the zoom level for the image, pressing `u` will decrease the zoom level. The zooming won't be seen until Problem 2 is solved correctly.

At each corner of the image box is a small square. To change the size of the image box drag the upper-right or lower-left corner. Both the height and width can be freely changed, down to a minimum size. In the unmodified assignment the image will be scaled to fit in the image box, as long as the image box isn't rotated.

To rotate and scale the image box drag the upper-left or lower-right corner. This will rotate the image and change its size, but its aspect ratio won't change. That is, if it were square before the drag is will be a square after the drag, probably rotated and a different size. In the unmodified code the image will not be rotated.

When the zoom level is less than 1 it should be possible to fit multiple images in the image box. The *wrap mode* indicates what should be displayed (of course, when the assignment is correctly solved). The wrap mode is shown in the green text. Pressing `w` will cycle through wrap modes *repeat*, *mirror*, and *transparent*. See Problem X for a description of what these should do.

### Display of Performance-Related Data

The top green text line shows performance-related and other information. `Size` refers to the size of the window. `Mouse` refers to the coordinates of the mouse pointer. Coordinate $(0,0)$ is at the **lower** left of the window. Text `frame_buffer[N]` shows the index of the frame buffer corresponding to

the point under the mouse pointer. (In the assignment file `frame_buffer` is abbreviated to `fb`, for convenience.)

`Render Time` and `Potential Frame Rate` show the CPU time needed to write the frame buffer. They can be ignored for this assignment.

## Integer Coordinate and Vector Types

This assignment uses both integer and floating point types for coordinates and vectors. (Previous frame buffer assignments just used integer types.) The integer types are `iCoor` for 2D coordinates and `iVect` for 2D vectors. The assignment also uses type `pCoor` for floating-point 3D coordinates (actually 4D, we'll learn in a few weeks) and `pVect` for floating-point 3D vectors.

The `iCoor` and `iVect` classes each have two members, `x` and `y`:

```
class iCoor {
public:
  iCoor( int x, int y ):x(x),y(y){}
  int x, y;
};
class iVect {
public:
  iVect( int x, int y ):x(x),y(y){}
  int x, y;
};
```

The addition, subtraction, and division operators are overloaded with these types. Here are some examples of how to use them:

```
void demo() {
  iCoor c0(3,4);  // Set to x=3, y=4.
  iCoor c1;
  c1.x = 5; c1.y = 7;  // Set to x=5, y=7;

  iCoor c3 = c1;

  // Component-wise subtraction.
  // Equivalent to v1.x=c0.x-c1.x;  v1.y=c0.y-c1.y;
  iVect v1 = c0 - c1;

  // Component-wise addition.
  // Equivalent to c4.x=c1.x+v1.x;  c4.y=c1.y+v1.y;
  iCoor c4 = c1 + v1;

  // Not allowed: Can't add two coordinates.
  iCoor c5 = c4 + c1;
}
```

## Code Generation and Debug Support

The compiler generates an optimized version of the code, `hw01`, and a debug-able version of the code, `hw01-debug`. The `hw01-debug` version is compiled with optimization turned off, which makes it easier to debug. When needed, you are strongly encouraged to run `hw01-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. These are `tryout1` and `tryout2`. You can use these variables in your code (for

example, `if ( tryout1 ) { x += 5; }`) to help debug, to help familiarize yourself with how the code works, or to experiment with new ideas. Keys `y` and`Y` to toggle their values, which are shown in the green text at the label `Tryout 1,2:`.

## Homework Code Overview

For this assignment the code in routine `hw01_render` will be modified. That code writes the frame buffer with sine waves, a circle of rectangles radiating from a point, and fills the image box (described earlier) with an image read from a file. The code for the sine waves and circle of rectangles is there for references, but it does not need to be modified for this assignment.

The coordinates of the image box are placed in variables `p0`, `p1`, `p2`, and `p3`. The width and height of the image box (assuming it is not rotated) is computed by looking at the `x` and `y` members of the `pCoor` class:

```
// Image Box Vertices (Corners)
//
//   p3 --- p2
//   |       |
//   p0 --- p1
//
const pCoor p0 [[maybe_unused]] = img_box[0];
const pCoor p1 [[maybe_unused]] = img_box[1];
const pCoor p2 [[maybe_unused]] = img_box[2];
const pCoor p3 [[maybe_unused]] = img_box[3];

float box_wd = p1.x - p0.x;
float box_ht = p3.y - p0.y;
```

The units of `box_wd` and `box_ht` are pixels. The width and height of the image are placed in variables `img_wd` and `img_ht` then a scale factor is comptued:

```
const int img_wd = hw01_data.img_px_width;
const int img_ht = hw01_data.img_px_height;

float img_scale_x = img_wd / box_wd;
float img_scale_y = img_ht / box_ht;
```

Variable `img_scale_x` indicates how many image pixels are to fit into one frame buffer pixel, horizontally, `img_scale_y` is set similarly. These variables are set so that the entire image fits in the image box.

A loop nest iterates over the height and width of the image box, and at each iteration is writes one frame buffer pixel (at coordinates `fb_x` and `fb_y` with one image pixel (at coordinates `img_x` and `img_y`. Pay attention to how these sets of coordinates are computed in the code below (in which some parts are omitted):

```
for ( int y=0; y<box_ht; y++ )
  for ( int x=0; x<box_wd; x++ )
    {
      int fb_x = p0.x + x;
      int fb_y = p0.y + y;

      int img_x = x * img_scale_x;
      int img_y = y * img_scale_y;
```

```
    uint32_t img_pixel = img_pixels[ img_idx ];

    fb[ fb_y * win_width + fb_x ] = img_pixel;
  }
```

The code above correctly displays the image so long as the image box is not changed and zoom is left at 1. In the problems that follow the code is to be modified so that the image is correctly displayed when the image box is rotated, scaled, and zoom and wrapped options are changed.

### Resources

A good reference for C++ is `https://en.cppreference.com/w/`. See Homework 1 assigned in the past few semesters for similar problems.

### 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**.

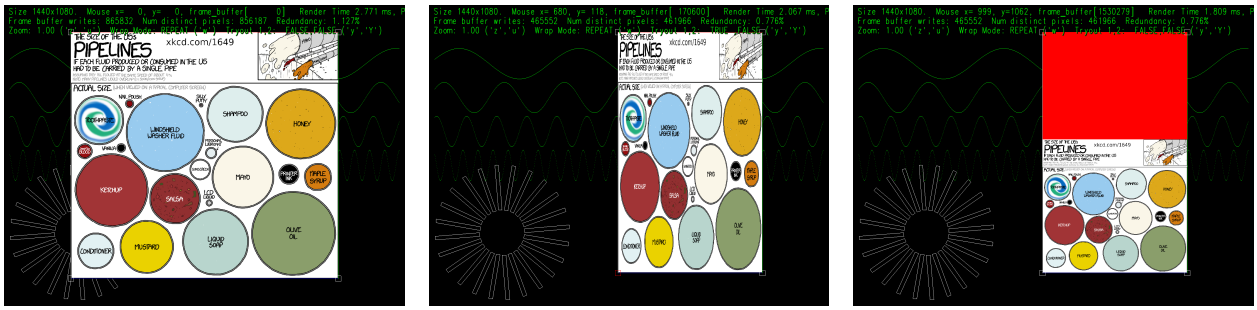
### Student Expectations

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:** The image box can be resized by dragging the lower-left or upper-right corners. (Dragging the other corners will rotate the image, which we don't want to do in this problem.) When the image box is resized this way the width and height can be changed independently, meaning if the aspect ratio were $w/h = 0.5$, where $w$ denotes the width and $h$ denotes the height) before the drag it, might be $w/h = 0.9$ after the drag. Since we are displaying images that's not a good thing, because circles should not look like ovals. The screenshot to the lower left shows the image before resizing, the one in the middle shows the image after resizing with the aspect

ratio wrong (circles are now ovals), the one on the right shows the image displayed with the correct aspect ratio. The red area will be fixed in Problem 3.

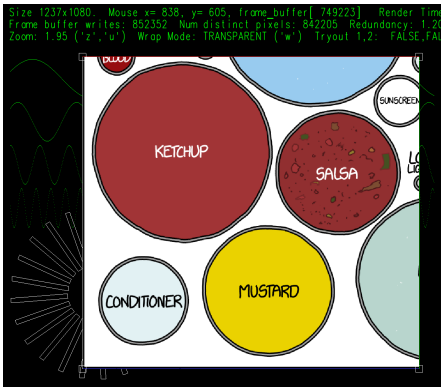



Modify the code in `render_hw01` so that the aspect ratio is preserved. When zoom is 1 (the default) one image should be across with width of the box, regardless of how the image box is resized. For this problem don't worry about unfilled parts of the image box (shown in red in the screenshot).

*Hint: This is easy, just one line needs to be changed.*

**Problem 2:** Let $z$ denote the value of variable `zoom`. Modify the code in `render_hw01` so that $\frac{1}{z}$ images are shown horizontally inside the image box. (The size of the image box itself should not be changed, and the image should not be written outside of the image box.) For example, when $z = 2$ the image should appear twice as large. The value of variable `zoom` can be changed by pressing `z`, `Z`, `u`, and `U` (for zoom and unzoom). The upper-case variants change the zoom level by larger amounts. The screenshot to the right shows the image at $z = 1.95$.

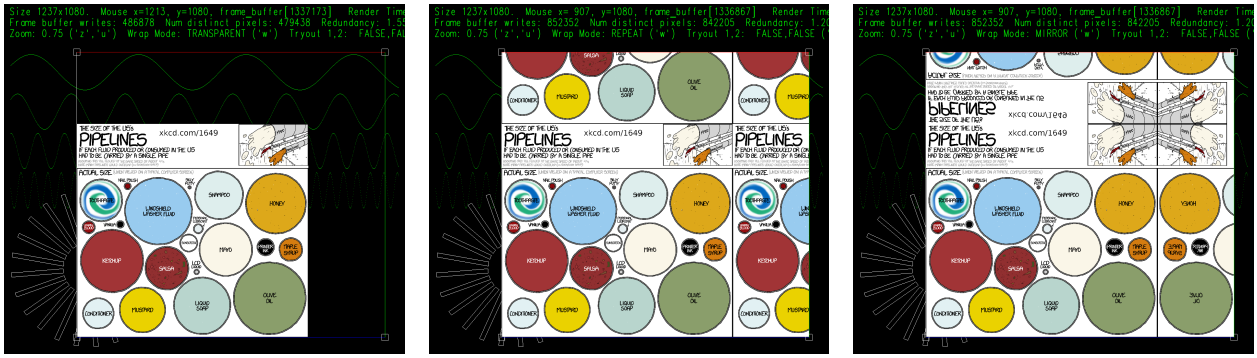
The solution to this problem is also easy. If you're writing a lot of code you're on the wrong track.




*There are more problems on the next page.*

**Problem 3:** When the zoom level is less than one, more than one image will fit in the image box. (That can happen too if the zoom level is one or larger but the image box is narrow.) In all cases the lower-left of the image box should hold a copy of the image (or as much as will fit). The value of variable `wrap_mode` indicates what should be displayed in the remaining space.

When `wrap_mode = WM_Transparent` nothing should be written in the remaining parts of the image box. One should be able to see whatever was there before, such as the sine waves. See the screenshot to the lower left. When `wrap_mode = WM_Repeat` the image should be repeated, like a pattern on wallpaper. See the screenshot in the middle. When `wrap_mode = WM_Mirror` the image should be repeated, but reflected as though the copy to the right of the original is its mirror image. The copy above should also be a mirror image. In fact, the mirroring should be done across each line separating copies of the image. See the screenshot to the lower right. (Such mirroring is often done with images intended to give an object a realistic texture, such as a wood grain or of concrete. Mirroring makes the boundaries less visible.)



Modify `render_hw01` so that the image box contents is set for the wrap modes. For your solving convenience in the loop writing the image to the frame buffer there is a `switch` statement with a case for each wrap mode.

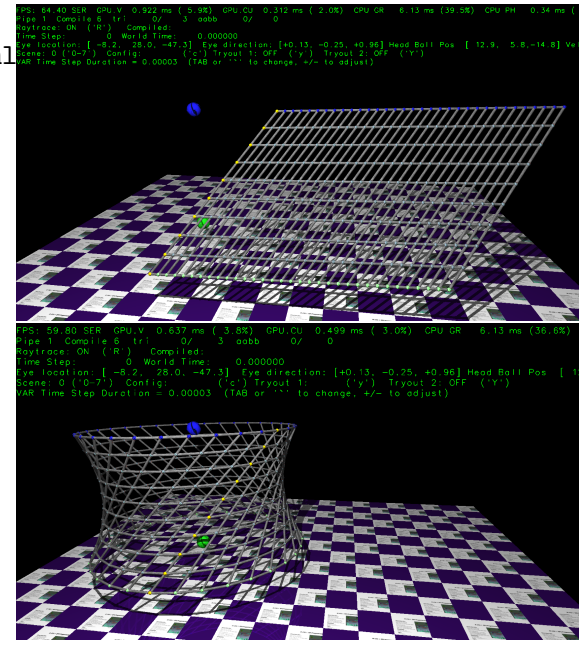
The transparent mode is easy, just don't write anything if the image pixel coordinates are out of range. The modulus operator, `%`, should come in handy for the other two wrap modes.

**Problem 4:** The image box can be rotated and resized by dragging its upper-left or lower-right corner. (Unlike when dragging the other two corners, the aspect ratio will be preserved.) Modify the code in `render_hw01` so that the image is placed in the image box, even after rotation, and is zoomed and wrapped based on `zoom` and `wrap_mode`. See the second screenshot on the first page of this assignment.

In a simple solution, and the one that is sufficient for full credit, the `x` and `y` loop limits and increment are left unchanged. (That is, the `for` statements aren't changed.) As before, set `box_wd` to the distance between points `p0` and `p1` and `box_ht` to the distance between points `p0` and `p3`, but note that when the image box is rotated `box_wd=p1.x-p0.x` does not compute the correct distance between `p0` and `p1`. The only other thing that needs to be done is to change how `fb_x` and `fb_y` are computed. This simple solution will skip over some pixels, the pattern of skipped pixels varies by rotation angle. That can be seen on the screenshot on the first page. Such a solution will get full credit. *The skipped pixels can be avoided by replacing the* `x,y` *loop nest with a loop nests that iterate over frame buffer pixels of the rotated image box. That would be too tedious for a problem 4 on a first assignment.*

# 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 );

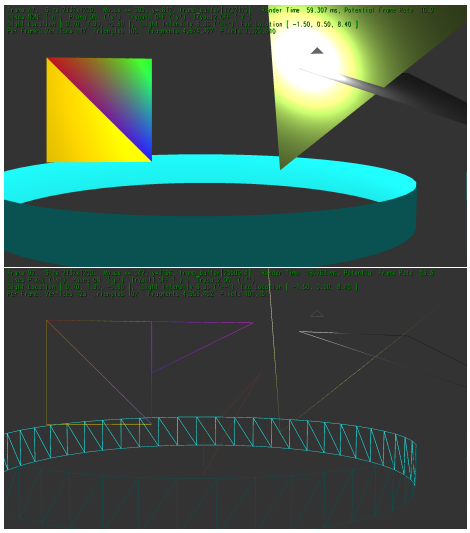
// Translate so that coordinate P is moved to P + grid_pitch.
pMatrix_Translate transl( grid_pitch );
pMatrix transform = transl;
```

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.

# LSU EE 4702-1 Homework 3 Due: 11 October 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/hw03.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 CPU-only rendering pipeline demo in directory `cpu-only/demo-06-rend-pipe.cc` and on the solution to 2023 Homework 3. The homework code should initially show a scene with a ring (or squat cylinder) and some triangles, one of which looks like it is dangerously close to a bright light, see the screenshot to the upper right. The screenshot on the lower right shows a correct solution to this assignment, in particular, Problem 3 in which only triangle edges are rendered, shown as pixel-width lines.



## Homework Overview

The code in the this assignment takes the frame buffer demo code, such as `demo-04-z-light`, and re-organizes it into a form closer to that of 3D acceleration libraries by separating the code that computes the location of triangles and the location of the eye, from the code rendering triangles. The homework code contains two major classes, `World` and `Our_3D`. Code in `Our_3D` is part of a fictional 3D rendering library, the kind of code that an OpenGL or Vulkan implementation would provide plus *shaders* provided by application programs. The code in `World` is application code that makes use of the 3D rendering library.

It does so by passing arrays of coordinates and other data to `Our_3D` and then by calling routine `Our_3D::draw_rasterization` to write the frame buffer based on those coordinates. In this assignment `World` draws a simple scene, but it is supposed to represent applications such as flight simulators, video games, or visualization. The code in `Our_3D` represents code provided as part of a device driver that came with your video card or operating system plus any application-programmer written shaders. The device-driver code would be written to run fast on a particular piece of hardware, such as a video card using an RTX 4090 GPU. In this assignment though the code is not even close to tuned to any hardware device.

Routine `World::render_scene` prepares a scene to be rendered. In the unmodified assignment it consists of several triangles and a ring (a short cylinder), which is itself made from triangles. The triangle vertex locations are specified by inserting coordinates into container `coors_os`, vertex normals into `norms_os`, and vertex colors (material properties) into `colors`. The suffix `_os` is for object space.

For example, the code below inserts coordinates and colors for two triangles (note that `<<` inserts an item, it is an overload of `vector::push_back`):

```
vector<pCoor> coors_os;
vector<pVect4> norms_os;
vector<pColor> colors;
```

```
coors_os << pCoor( 0, 0, 0 ) << pCoor( 9, 6, -5 ) << pCoor( 0, 7, -3 );
colors << color_white << color_white << color_white;

coors_os << pCoor(-4,2,-3) << pCoor(-2,2,-3) << pCoor(-2,0,-3);
colors << color_blue << color_green << color_red;
```

(The code samples here are condensed versions of the code in `hw03.cc`.) The vertex normals for the triangles are computed by a loop further below. Note that `coors_os` is declared in `World::render_scene` and so `Our_3D` can't automatically see these coordinates. `World::render_scene` also provides transformation matrices to `Our_3D`:

```
// Specify Transformation from Object to Eye Space: eye_from_object.
pMatrix_Translate center_eye(-eye_location);
pMatrix_Rotation rotate_eye(eye_direction,pVect(0,0,-1));
pMatrix eye_from_object = rotate_eye * center_eye;
gc.transform_eye_from_object_set(eye_from_object);

// Transform from Eye to Clip
pMatrix_Frustum clip_from_eye
  ( -width_m/2, width_m/2,  -height_m/2, height_m/2,  1, 5000 );
gc.transform_clip_from_eye_set(clip_from_eye);
```

After populating the `coors_os` container our user code computes the triangle normals:

```
for ( size_t i = 0;  i < coors_os.size();  i += 3 ) {
    pCoor p1 = coors_os[i], p2 = coors_os[i+1], p3 = coors_os[i+2];
    pVect4 n = cross( p1, p2, p3 ).normal();
    norms_os << n << n << n;  }
```

Notice that each group of three vertices describes a triangle. That's the case for the triangles specified by the code so far, but not for the ring. To actually render the scene the coordinate and color container information needs to be passed to `Our_3D` and then a *draw* command needs to be issued. That is done by the code below:

```
gc.topology_strip_set(false);  // Triangles *do not* form a strip.
gc.vtx_coors_set(coors_os);    // Copy address of coors_os.
gc.vtx_normals_set(norms_os);  // Copy address of norms_os.
gc.vtx_colors_set(colors);     // Copy address of colors.
gc.draw_rasterization();       // Render the scene.
```

`Our_3D` does not copy the containers, it only keeps an address, so additional items can be added up until `gc.draw_rasterization()` is called. Routine `gc.draw_rasterization()` paints the frame buffer (actually a pixel array which will eventually be composited into the frame buffer).

Routine `gc.draw_rasterization()` is called twice per frame, the first time to render the individual triangles, and the second time for the ring. It is possible to call `gc.draw_rasterization()` once per triangle, but that would be foolish because there is a substantial per-call overhead that would have to be borne for each triangle. So the code loads all the triangles into `coors_os` and calls `draw_rasterization` once. Calling draw routines fewer times with more triangles per call is especially important for "real" 3D acceleration libraries like OpenGL and Vulkan.

After preparing the individual triangles, the code prepares the ring (cylinder). The ring is described by variables declared near the top of the routine:

```
pCoor cyl_center(-1,-1,-3);
int n_segs = 50;
pVect cyl_axis( 0, 0.75, 0.05 );
float cyl_radius = 4;
```

Further ahead in the routine code inserts triangles for the ring. It starts by clearing (emptying) the containers and then computing axes, `ax` and `ay`, of the ring circles.

```
// Empty the containers so they can be used for the next draw.
coors_os.clear(); colors.clear(); norms_os.clear();

pNorm ax(0,-cyl_axis.z,cyl_axis.y);  // Find an axis orthogonal to cyl_axis.
pNorm ay = cross( cyl_axis, ax );
float delta_theta = 2 * M_PI / n_segs;
```

For the individual triangles above (before the ring) three vertices were inserted into `coors_os` for each triangle, this is referred to as an *individual triangle grouping*. For the ring we will tell `Our3D` to expect the vertices to be in a *triangle strip grouping*. In a triangle strip grouping the first triangle is constructed from `coors_os[0]`, `coors_os[1]`, and `coors_os[2]`; the second triangle from `coors_os[3]`, `coors_os[2]`, and `coors_os[1]`, the third from `coors_os[2]`, `coors_os[3]`, and `coors_os[4]`, etc. To render $T$ triangles requires $T + 2$ vertices rather than $3T$. (Sharp-eyed readers may have noticed that the order of vertices of the first and second triangle are different. That's intentional since vertex order determines which side is the front. The default is a counter-clockwise [right-handed] winding order.)

A triangle strip grouping works well for the ring. A loop iterates around the circle, with the first and last point overlapping. At each iteration a point on the lower circle, `c`, and upper circle, `c+cyl_axis` are computed and added to the `coors_os` container. Both of these points have the same surface normal, `n` in the code. That is added to `norms_os`. Note that `n` is the normal of the cylinder surface, not of the triangles used to approximate the cylinder. Code in the loop body also adds colors to the list.

```
for ( int i=0; i<=n_segs; i++ ) {
    float theta = i * delta_theta;
    pVect4 n = ax * cosf(theta) + ay * sinf(theta);
    pCoor c = cyl_center + cyl_radius * n;

    coors_os << c + cyl_axis << c;
    norms_os << n << n;
    colors << color_cyan << color_cyan;
  }
```

Finally, `Our3D` is told to expect a triangle-strip grouping, pointers to the containers are provided, and the draw routine is called:

```
gc.topology_strip_set(true);
gc.vtx_normals_set(norms_os);
gc.vtx_coors_set(coors_os).vtx_colors_set(colors);
gc.draw_rasterization();
```

In this assignment code in `Our_3D::draw_rasterization` will also have to be modified. This routine uses the vertex containers and transformations provided by `Our_3D::draw_rasterization`

to write the frame buffer. The code itself is based on code from `cpu-only` demos such as `demo-04-z-light`, but is re-organized to look a tiny bit more like acceleration code in a Vulkan implementation. Some of the code in `Our_3D::draw_rasterization` would be driver code in a Vulkan implementation (not normally seen by an application program) and some would be *shader code*, something application programs do write.

The code in `Our_3D::draw_rasterization` starts by preparing the window-from-clip coordinate space transformation:

```
Our_3D& Our_3D::draw_rasterization() {
  const uint win_width = frame_buffer.width_get();
  const uint win_height = frame_buffer.height_get();
  // Transform from Clip to Pixel
  //
  pMatrix_Translate recenter(pVect(1,1,0));
  pMatrix_Scale scale( win_width/2, win_height/2, 1 );
  pMatrix window_from_clip = scale * recenter;
  pMatrix ws_from_os = window_from_clip * clip_from_eye * eye_from_object;
  pCoor eye_location = -eye_from_object.col_get(3);
```

Notice that `clip_from_eye` and `eye_from_object` were prepared by application code, but that `ws_from_os` (window-space from object space) is prepared by `Our3D`. That's because `clip_from_eye` and `eye_from_object` depend on things that are of a concern to the application programmer, such as where the eye is and how big the monitor is.

After preparing convenience pointers like `zbuffer` and `colors` and updating statistics, like `frame_buffer.n_vtx_frame` (not shown here), the code starts the main loop which iterates over triangles:

```
  const size_t inc = topology_strip ? 1 : 3;
  for ( size_t i=0; i+2 < coors_os.size(); i += inc ) {
      frame_buffer.n_tri_frame++; // Count of number of triangles.
      // Get next triangle's object space coordinates ..
      pCoor o0 = coors_os[i+0],  o1 = coors_os[i+1],  o2 = coors_os[i+2];
      // .. convert them into unhomogenized clip-space coordinates ..
      pCoor u0( ws_from_os * o0 ), u1( ws_from_os * o1 ), u2( ws_from_os * o2 );
      // .. and homogenize them.
      pCoor_Homogenized w0( u0 ), w1( u1 ), w2( u2 );

      //  Extract colors and normals too.
      pColor c0 = colors[i], c1 = colors[i+1], c2 = colors[i+2];
      pVect4 n0 = vtx_normals[i], n1 = vtx_normals[i+1], n2 = vtx_normals[i+2];
```

The loop body starts by retrieving information about a triangle, which is described by three consecutive elements of `coors_os`, etc. Note that this loop works both with individual triangle and triangle strip groupings, the only difference is the value of `inc`. The retrieved object-space coordinates, `o0`,`o1`,`o2`, are converted to window-space coordinates in two steps. Coordinates in `w0`, `w1`, and `w2` are in window space, meaning that, for example, `w0.x` and `w0.y` could be used to compute a frame-buffer coordinate.

A deeper loop nest (`b0`,`b1`) iterates over barycentric coordinates. Recall that given barycentric coordinates $b_0$, $b_1$, and $b_2$, and points $P_0$, $P_1$, and $P_2$, point $S = b_0P_0 + b_1P_1 + b_2P_2$ is inside triangle $P_0P_1P_2$ if $b_0 \geq 0$, $b_1 \geq 0$, $b_2 \geq 0$, and $b_0 + b_1 + b_2 = 1$. Here $P_0$ is the coordinate of a point, but mathematically is treated like a vector from the origin to the point. (In class barycentric

coordinates were described in a different but equivalent way: $S = b_1\overrightarrow{P_0P_1} + b_2\overrightarrow{P_0P_2}$.) Some special cases: If $b_i = 1$ then $S = P_i$ for $i \in \{0, 1, 2\}$. If $b_0 = 0$ then $S$ is on the edge between points $P_1$ and $P_2$. *That may be useful for 2024 Problem 3.*

Since we aren't interested in points outside the triangle the inner loop limits $b_1 \in [0, 1 - b_0]$ and $b_2 = 1 - b_0 - b_1$. See the code below.

```
for ( float b0=0; b0<=1; b0 += db0 )
  for ( float b1=0; b1<=1-b0; b1 += db1 ) {
      frame_buffer.n_frag_frame++; // Count of number of fragments.
      const float b2 = 1 - b0 - b1;
      pCoor wf = b0*w0 + b1*w1 + b2*w2;
      if ( uint(wf.x) >= win_width || uint(wf.y) >= win_height ) continue;
      const size_t idx = wf.x + int(wf.y) * win_width;
```

Since the goal is to find a frame-buffer coordinate the barycentric coordinates are used in the code above to compute `wf` (window-space fragment) using window-space coordinates of the triangle vertices. After computing `wf` the code makes sure it is within the frame buffer.

To compute the lighted color of the fragment the code needs to find the object-space coordinates of the fragment. That **can not** be computed by the expression `b0*o0 + b1*o1 + b2*o2` because `b0`, `b1`, and `b2` are window-space barycentric coordinates. To find the fragment object-space coordinate and to interpolate any other vertex properties we need object-space (perspective-correct) barycentric coordinates. These are computed and used to interpolate the object-space coordinates, vertex normal, and color:

```
// Compute perspective-correct barycentric coordinates.
float bc0 = b0 / ( b2 * u0ou2 + b1 * u0ou1 + b0 );
float bc1 = b1 / ( b2 * u1ou2 + b0 * u1ou0 + b1 );
float bc2 = 1 - bc0 - bc1;

pCoor of =     bc0*o0 + bc1*o1 + bc2*o2;
pNorm n =      bc0*n0 + bc1*n1 + bc2*n2;
pColor color = bc0*c0 + bc1*c1 + bc2*c2;
```

For this assignment it is not important to understand the derivation of the object-space barycentric coordinates, however it is important to understand the difference between object-space and window-space coordinates.

With these, the lighted color can be computed and written to the frame buffer:

```
// Compute Lighted Color
pNorm frag_to_light(of,light_location);
pVect frag_to_eye(of,eye_location);
float cos_angle_eye = dot( n, frag_to_eye );
float cos_angle_light = dot( n, frag_to_light );
bool illumination_visible =
   ( cos_angle_light > 0 )  ==  ( cos_angle_eye > 0 );
float clamped_phase = illumination_visible ? fabs(cos_angle_light) : 0;
pColor lighted_color =
  ( light_ambient
    + clamped_phase / frag_to_light.magnitude * light_color ) * color;
// Write the frame (color) buffer with the lighted color
frame_buffer[ idx ] = lighted_color.int_rgb();
```

### 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 `hw03-`$x$`.png` or `hw03-debug-`$x$`.png`, where $x$ is the number of previous screenshots generated. The first screenshot may be garbled.

Initially the arrow keys, `PageUp`, and `PageDown`, can be used to move around the scene. The eye direction cannot be changed, but feel free to add this functionality if you want it. After pressing `l` the motion keys will move the light instead of the eye, and after pressing `e` the motion keys operate on the eye.

The light intensity can be adjusted by pressing `+` and `-`. (This only works for this assignment.)

### Assignment-Specific User Interface

Pressing `p` toggles between computing the lighted color per fragment or per vertex. The setting is shown in the second line of green text to the right of label Lighting per. The value is shown in variable `opt_light_frag`, set to `true` for per-fragment lighting.

Pressing `n` cycles through three ways of display triangle edges NONE, meaning the triangles are shown in the usual way, PIXEL, meaning only triangle edges are to be shown and that the edges are a few pixels wide, and WORLD, meaning that triangle edges are to be 0.1 object-space units wide. Variable `opt_lines` is cycled by `n`, see Problem 3 for more information. Until Problem 3 is solved correctly only NONE will work. (Of course, it's possible even that won't work. If that happens to you don't feel shy about asking for help.)

### Display of Performance-Related Data

The top green text line shows performance-related and other information. `Frame` shows the number of times `render_scene` has been called. It should only increment on key presses and certain window manager events (such as minimizing and unminimizing). `Size` refers to the size of the window. `Mouse` refers to the coordinates of the mouse pointer. Coordinate $(0, 0)$ is at the **lower** left of the window. Text `frame_buffer[N]` shows the index of the frame buffer corresponding to the point under the mouse pointer. (In the assignment file `frame_buffer` is abbreviated to `fb`, for convenience.)

`Render Time` and `Potential Frame Rate` show the CPU time needed to write the frame buffer. They can be ignored for this assignment.

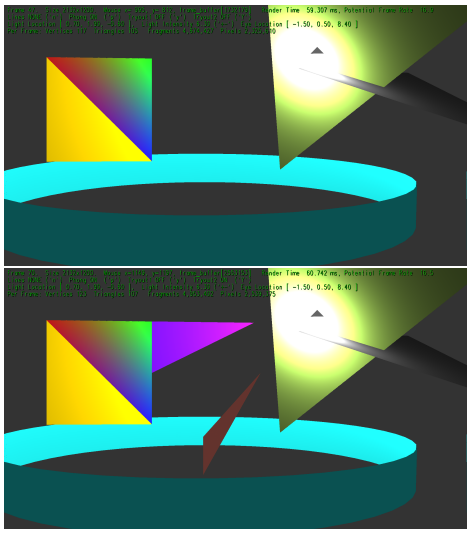
The last line of green text shows the number of items processed by the rendering pipeline per frame. One should be able to estimate these numbers. For example, if one is rendering using individual triangles one should be able to compute the number of triangles from the number of vertices. `Fragments` shows the number of fragments processed by what we will call the rasterizer. Recall that a fragment is a part of a triangle covering a pixel. In `draw_rasterization` look for the place where `n_frag_frame` is incremented. `Pixels` is the number of times that the frame buffer is written with a fragment. (Not every fragment is written.)

### Code Generation and Debug Support

The compiler generates an optimized version of the code, `hw03`, and a debug-able version of the code, `hw03-debug`. The `hw03-debug` version is compiled with optimization turned off, which makes it easier to debug. When needed, you are strongly encouraged to run `hw03-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` and `opt_tryout2`. 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` and `Y` toggle the value of host Boolean variables `opt_tryout1` and `opt_tryout2`.

**Problem 1:** For this problem modify the code in routine `World::render_scene()` so that it emits the two triangles described below. The screenshot to the upper-right shows the unmodified assignment, the screen shot to the lower right shows the scene with the two new triangles: a purple triangle in the the center and a salmon triangle (it looks brown) in the lower center.

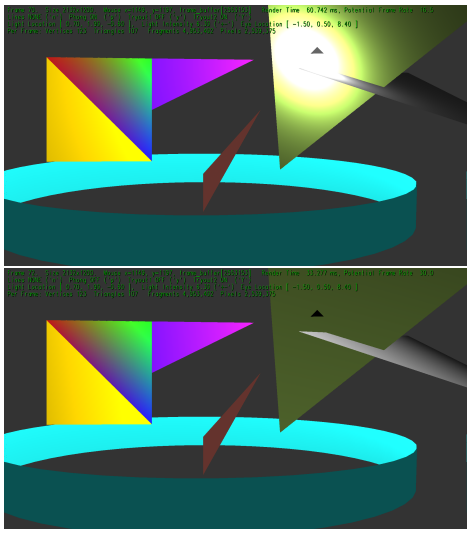


(*a*) Modify the code so it emits a purple triangle adjacent to the multicolored (red/green/blue) triangle. The top-left vertex of the purple triangle should be at the same coordinate as the green vertex of the multi-colored triangle. The lower vertex of the purple triangle should be halfway between the green and blue vertex. The upper edge of the purple triangle should be on the same line as the red/green edge and should be the same length as the red/green edge. *Note: This is an easy problem once you find the place in the code that emits the multi-colored triangle.*

(*b*) A common problem encountered by some working with graphics demos is figuring out where the light is. Here we will fix that problem by adding a triangle that points at the light: the brown (the color is actually salmon) triangle in the cyan ring.

Modify the code so that it renders the salmon triangle as follows. Two vertices are on the cylinder axis (one at the bottom and one at the top). The third vertex points toward the light. Place it on the line from the top vertex to the light. The length of the edge from the top vertex to the light should be two. Use variable `color_salmon` for the color, or choose your own color. The light location is in variable `World::light_location`.

**Problem 2:** When the code starts it shows a scene with the light close to a green (actually dark olive green) and a gray triangle. As with all demos covered in class so far, lighting is computed at each fragment. Back in the twentieth century the amount of computation needed to compute lighting was considered substantial. So rather than compute the lighted color for every fragment in a rasterized triangle, the lighted color would only be computed at each vertex. The lighted color at a fragment would be a blend of the lighted colors of the three vertices, using the barycentric coordinates as the blend weights. Note that the code already does such blending for color (material property) which can be seen in the multi-colored triangle.



Computing the lighted color only for vertices can save a substantial amount of computation. In our default scene (in a draft of this homework assignment) there are 57 vertices and 2141934 fragments, so our code does $\frac{2141934}{57} = 37577.8$ times more lighted color computations

than it would do if a lighted color were only computed for each vertex.

In the screenshot to the upper right a lighted color is computed for each fragment, in the screenshot to the lower right a lighted color is computed for each vertex. In some places the difference is obvious, such as the disappearance of the bright spot on the green triangle. There's no spot because lighted colors are a blend of the three vertices, and each vertex is further from the light than the center of the bright spot. The difference in lighting is much more subtle for primitives further from the light and for smaller primitives, such as the cyan ring.

(*a*) ☐ Modify the code in `Our_3D::draw_rasterization` so that when `opt_light_frag` is `false` lighting is only computed for each vertex. When `opt_light_frag` is `true` lighting should be computed the way it is now. ☐ Make sure that lighting for each vertex is computed just once, even when primitives are offered as a triangle strip.

To get a better understanding of how many times lighting is computed put the line `if ( opt_tryout1 ) frame_buffer.n_vtx_frame--;` in the place where the lighted color is being computed in your code. In the unmodified assignment that would be in this code:

```
float clamped_phase =
  illumination_visible ? fabs(cos_angle_light) : 0;
pColor lighted_color =
  ( light_ambient
    + clamped_phase / frag_to_light.magnitude * light_color )
  * color;

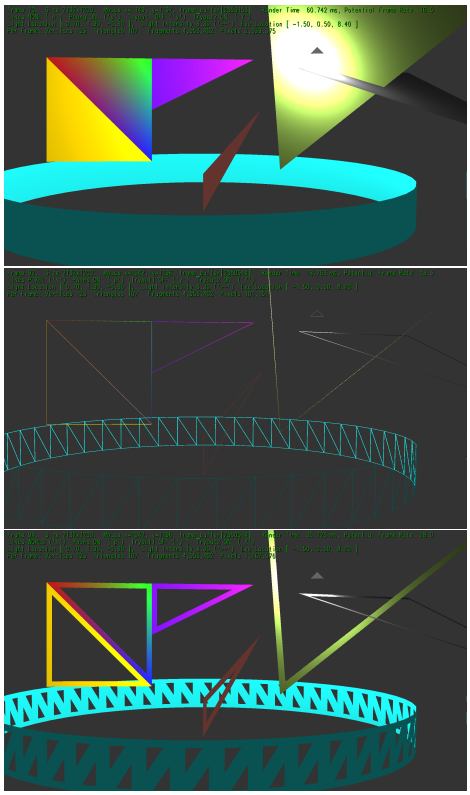
if ( opt_tryout1 ) frame_buffer.n_vtx_frame--;  // Add this for familiarization.

//
// Write the frame (color) buffer with the lighted color
//
frame_buffer[ idx ] = lighted_color.int_rgb();
```

When running the code with this modification look at the bottom line of green text near Vertices and press `y` (lower-case wye). If you've solved the assignment correctly the text should show Vertices 0 when Tryout1 is on (look at the middle of the second line of green text). If the number is instead a large negative value, say -1573915, then the code is computing lighted colors way too many times. It is also possible that it is a much smaller magnitude negative number, say $117 - 105 \times 3 = -198$. That's better but not a full credit answer.

*There's another problem on the next page.*

**Problem 3:** For many 3D graphics systems surfaces are approximated by triangles. To help with debugging, tuning, and familiarization it can be helpful to make the triangles' locations obvious by showing only the edges. See the two lower screenshots.



Variable `opt_lines` specifies how triangles should be rendered. When `opt_lines= Lines_None` the triangles should be filled, which is how they are rendered normally, shown in the upper screenshot. When `opt_lines= Lines_Pixel` rendered triangle edges should be only about two pixels thick, see the middle screenshot. When `opt_lines= Lines_World` triangle edges should be 0.1 object-space units thick, see the lower screenshot. Note that in all cases the lighted color of the edges is computed in the same way as the full triangle. The second line of green text shows the value of `opt_lines`, and its value can by cycled by pressing `n`. (The letter el was already used for light.)

There is an important difference in appearance between pixel mode and world mode. In pixel mode the rendered thickness is the same for every triangle, regardless of its size, distance from the eye, and orientation. In contrast in world mode the closer an edge is to the eye the larger its thickness in its projection on the frame buffer. Look closely at the thickness of the edges of the triangles making up the near and far portions of the cyan ring. The closer edges are thicker (even if your perceptual system is trying to tell you otherwise). Also look at the gray triangle in the upper left. Its edges appear thinner than nearby triangles because the gray triangle is facing up and so we see the triangle (and its edges) edge-on.

Modify the code in `Our_3D::draw_rasterization` so that it renders triangle edges as described above. One might be tempted to solve the problem by using a line-drawing routine (such as one from the early frame buffer demos or past homework assignments). Try to avoid that since it can not easily be made to work for world mode once edge thickness and lighting are taken into account. Instead, use the barycentric coordinates to determine how close a fragment is to an edge.

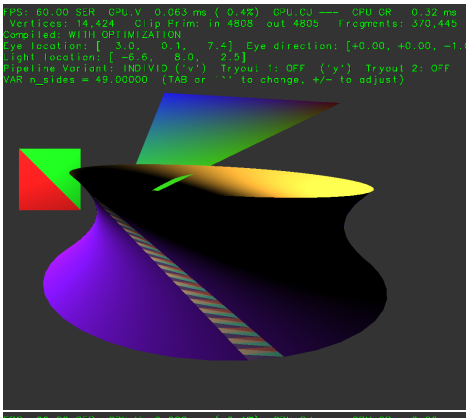
To help you get started there is a `switch ( opt_lines )` statement with the three cases. Code in a case should test the barycentric coordinates and if some condition is true, continue. For example,

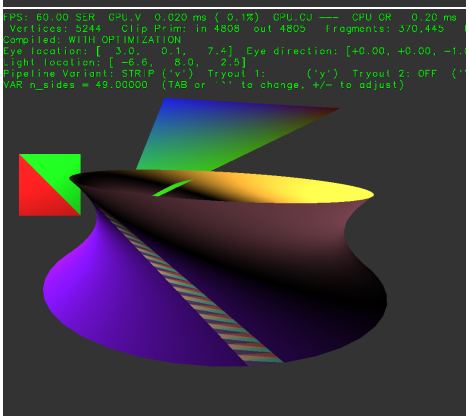
```
switch ( opt_lines ){
case Lines_None: break;
case Lines_Pixel:
  if ( bx0 ... ) continue;
  break;
```

where `bx0` is either `b0` or `bc0`. A correct solution depends on using the right barycentric coordinates in each case either the window-space coordinates, `b0,b1,b2`, or the perspective-correct coordinates, `bc0,bc1,bc2`. If necessary, re-read the material on barycentric coordinates in the homework overview above.

# LSU EE 4702-1 Homework 4 Due: 14 November 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/hw04.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 homework code should initially show a scene with a rotating hyperboloid of one sheet, see the screenshot on the upper right. The screenshot on the lower right shows a correct solution to this assignment, in particular, the hyperboloid is rendered using a triangle strip (notice the vertex count) and it can be illuminated from behind.

## Homework Overview

The code in this assignment explores methods of drawing a hyperboloid of one sheet in Vulkan. The baseline code in the unmodified assignment draws the hyperboloid using individual triangles.

## 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 `hw04.png` or `hw04-debug.png`. Press `F10` to start recording a video, and press `F10` to stop it. The video will be in file `hw04-1.ogg` or `hw04-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` 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

Pressing `1` generates a new hyperboloid. The first two are hard-coded, and after that they are randomly generated. Pressing `2` shows the first hyperboloid again.

Pressing `v` switches between using a pipeline using a triangle list (individual triangles) and a triangle strip. The setting is shown in the second line of green text to the right of label Pipeline Variant.

### 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 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, `hw04`, and a debug-able version of the code, `hw04-debug`. The `hw04-debug` version is compiled with optimization turned off, which makes it easier to debug. When needed, you are strongly encouraged to run `hw04-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`, available in CPU and shader code. 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.

### Resources

A good reference for C++ is `https://en.cppreference.com/w/`. Solutions to the shader programming problems may (will) require the use of library functions. See Chapter 8 of The OpenGL Shading Language Version 4.6 specification. Also see 2014 Homework 4-6 (especially 6), the spiral, to see examples of how the pipeline input topology can be chosen to provide data needed by the geometry shader, even though the topology class (point, line, triangle) does not geometrically correspond to what is being drawn.

**Problem 1:** The code in `World::render` prepares Vulkan for rendering a scene including some triangles, and a hyperboloid of one sheet. It also writes the green text. Two pipelines are used to render the hyperboloid, `pipe_hyperb` and `pipe_hyperb_strip`. Other than the primitive topology, both are identical.

The code computes hyperboloid vertex coordinates and normals and writes them to host containers `coords` and `norms`:

```
for ( int side = 0; side <= n_sides; side++ ) {
    float theta = side * delta_theta;
    pMatrix_Rotation rot(hyperb_az,theta);
    pCoor line_bot_pos = bot_ctr + rot * bot_vx;
    pVect l_dv = rot * line_dv;

    for ( int lev = 0; lev <= n_levels; lev++ ) {
        pCoor p_line = line_bot_pos + lev * l_dv;
        coords[lev][side] = p_line;
        pCoor ctr = bot_ctr + lev * delta_z;
        pVect tc = cross(hyperb_az,pVect(ctr,p_line));
        pNorm n = cross(tc,l_dv);
        norms[lev][side] = n;
      }
  }
```

Variable `lev` indicates the level (position along the hyperboloid axis) and `side` indicates a position around the circle at a level. Both are integers and are used to index the container. (See the complete code in the assignment file, and if necessary review Homework 2 which also worked with hyperboloids.) Notice that the `lev` loop moves along a straight line and that each iteration of the `side` loop rotates that line around the hyperboloid axis.

*Continued on next page.*

Variable `pipeline_variant` indicates which pipeline and vertex grouping to use. Pressing `v` toggles it between `PV_Individ` (triangle list [individual triangles]) and `PV_Strip` (triangle strip). Next, the code prepares a buffer set for the appropriate pipeline. In the unsolved assignment only individual triangles will work correctly.

```
switch ( pipeline_variant ) {
case PV_Individ:

  bset_hyperb.reset(pipe_hyperb);
  for ( int side = 0; side < n_sides; side++ )
    for ( int lev = 0; lev < n_levels; lev++ )
      {
        pCoor p00 = coords[lev][side];
        pCoor p01 = coords[lev][side+1];
        pCoor p10 = coords[lev+1][side];
        pCoor p11 = coords[lev+1][side+1];

        bset_hyperb << p00 << p01 << p10;
        bset_hyperb << p01 << p11 << p10;

        pNorm n00 = norms[lev][side];
        pNorm n01 = norms[lev][side+1];
        pNorm n10 = norms[lev+1][side];
        pNorm n11 = norms[lev+1][side+1];

        bset_hyperb << n00 << n01 << n10;
        bset_hyperb << n01 << n11 << n10;

        int c1 = side == 0 ? ( lev%2 ? 5 : 2 ) : 1;
        int c2 = side == 0 ? ( lev%2 ? 3 : 4 ) : 1;

        bset_hyperb << c1 << c1 << c1 << c2 << c2 << c2;
      }
break;
```

The code above inserts coordinates, normals, and color indices into buffer set `bset_hyperb` in the correct order for individual triangles. The color indices, `c1` and `c2`, are integers. They refer to colors stored in array `uni_hw03_colors`, which is prepared in `World::render`. A single index, such as 1, specifies both a front color and a back color. Though a different color index can be assigned to each vertex of a triangle, the geometry shader only uses the color index of the third vertex for coloring.

In the unsolved assignment the `PV_Strip` is identical to the code under `PV_Individ`. As a result the triangle strip pipeline will be fed with vertices in individual triangle order. As a result coloring will not be correct and the number of vertices will be 3× what they should be.

Modify the `PV_Strip` case in the `switch` (not shown above) so that vertices are inserted in the correct order for a triangle strip. In a correct solution the version using triangle strips will be identical in appearance to the one using a triangle list, but using one third the number of vertices.

To do this correctly one must use multiple triangle strips, either one per value of `side` or one per value of `lev`. When starting a new triangle strip (*restarting* a strip) use 0 for the color index of

the first two vertices. If triangle strips are not restarted properly extra triangles will be rendered, affecting at least coloring.

**Problem 2:** In the unmodified assignment the fragment shader, `fs_main` in file `hw04-shdr.cc`, applies a simple lighting algorithm that uses a diffuse coloring model to compute the lighted color of the fragment. First, the shader computes the value of `attenuation` which indicates how much light is reaching the fragment based on the distance from the fragment to the light and the angle, and on whether the light is on the side of the triangle that we can see (variable `lit_side` is `true`). If the light is on the opposite side of the triangle ( (variable `lit_side` is `false`) then attenuation is set to zero. The shader then combines the visible color material property, light color, and attenuation to compute the lighted color:

```
void fs_main() {
  // Vector from fragment to light.
  vec3 vec_vl = uni_light.position.xyz - vertex_e.xyz;

  // Distance squared.
  float dist_sq = dot( vec_vl, vec_vl );

  // False if the light illuminates the side we can't see.
  bool lit_side = dot( normal_e, vec_vl )>0 == dot( normal_e, -vertex_e.xyz )>0;

  // Amount of light reaching the fragment.
  float attenuation = lit_side ? abs( dot( normal_e, vec_vl ) / dist_sq ) : 0;

  // Material color.
  vec4 mat_color = gl_FrontFacing ? uc.front[color_idx] : uc.back[color_idx];

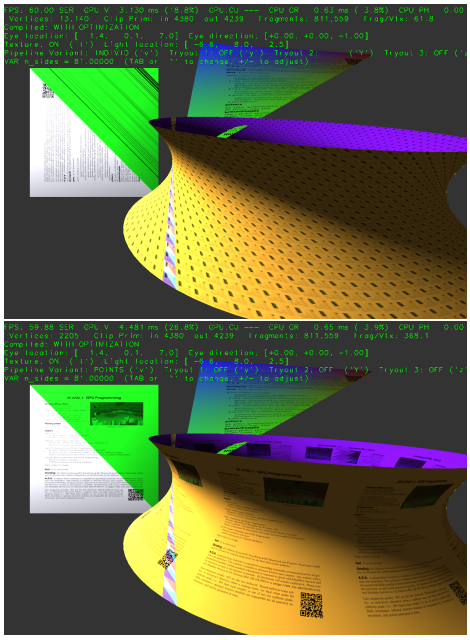
  // Compute lighted color.
  frag_color = mat_color * uni_light.color * attenuation;
}
```

The lighting code above models an opaque material. Modify the code above to modify a translucent material as follows: If the lit side of the fragment is visible then compute the lighted color as shown above (and in the unmodified assignment). If the lit side is not visible then combine the front and back colors in some way so that the visible color is a combination of the two, and is darker than if the light were on the visible side. See the lower screenshot on the first page of this assignment.

# LSU EE 4702-1 Homework 5 Due: 1 December 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/hw05.cc.html` *and* `https://www.ece.lsu.edu/koppel/gpup/2024/hw05-shdr.cc.html`.

**Problem 0:** In the unlikely event it was 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 homework code should initially show a scene with a rotating hyperboloid of one sheet, see the screenshot on the upper right. In the shot on the lower right, taken from a correctly solved assignment, the square formed by the light gray (it looks white but the name of the color is `color_light_gray`) and green triangles shows the course syllabus right-side-up, and the course syllabus is much larger and better positioned on the hyperboloid. Also notice that the hyperboloid is drawn using a points grouping which uses only half the number of vertices required by the triangle strip grouping.



## Homework Overview

The code in this assignment explores methods of applying textures and of using a rendering pipeline in which there are no inputs other than indices. It is up to the shaders to fetch the information they need from storage buffers.

## 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 `hw05.png` or `hw05-debug.png`. Press `F10` to start recording a video, and press `F10` to stop it. The video will be in file `hw05-1.ogg` or `hw05-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` 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

Pressing `1` generates a new hyperboloid. The first two are hard-coded, and after that they are randomly generated. Pressing `2` shows the first hyperboloid again.

Pressing `v` switches between using a pipeline using a triangle list (individual triangles), a triangle strip, and points. The setting is shown in the second line of green text to the right of label Pipeline Variant.

Pressing `t` toggles textures on and off.

### 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 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, `hw05`, and a debug-able version of the code, `hw05-debug`. The `hw05-debug` version is compiled with optimization turned off, which makes it easier to debug. When needed, you are strongly encouraged to run `hw05-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`, `opt_tryoutf`, and `opt_tryouti` available in CPU and shader code. 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 Boolean variables `opt_tryout1`, `opt_tryout2`, and `opt_tryout3`, available in host and shader code. The user interface can also be used to modify floating-point variables `opt_tryoutf` and `opt_tryouti` using the `Tab`, `+`, and `-` keys, see the previous section. In shader code use variables `opt_tryouti` and `opt_tryoutf`. In host code (2024 Homework 5) use variables `uni_misc->opt_tryoutf` and for for `opt_tryouti` use `uni_misc->opt_tryout.w`.

## Resources

A good reference for C++ is `https://en.cppreference.com/w/`. Solutions to the shader programming problems may (will) require the use of library functions. The names of many OGSL functions match corresponding functions in the C++ standard library, but not all of them including one needed for a complete solution to 2024 Homework 5. The OGSL library functions are described in Chapter 8 of The OpenGL Shading Language Version 4.6 specification. Also see 2014 Homework 4-6 (especially 6), the spiral, to see examples of how the pipeline input topology can be chosen to provide data needed by the geometry shader, even though the topology class (point, line, triangle) does not geometrically correspond to what is being drawn.

## 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++ and OGSL 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 each student's duty to himself or herself 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 process that takes effort, but one that also provides the satisfaction of progress and of developing skills and understanding.

*Problems start on the next page!*

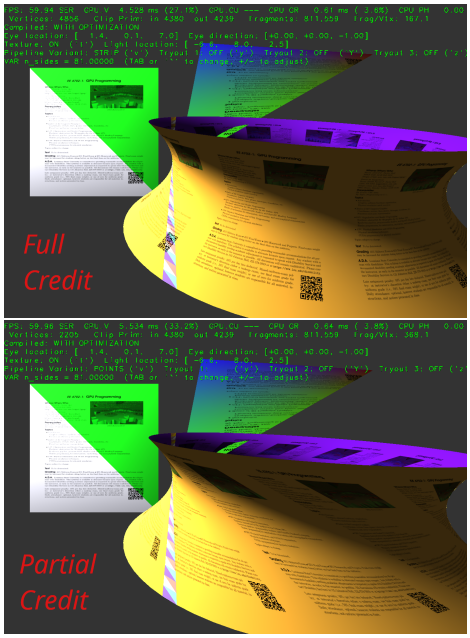
**Problem 1:** Modify the code in `World::render` so that the square formed by the green and light gray triangle shows the course syllabus right-side-up, as shown in the lower screenshot on the first page of the assignment. Do so by changing the texture coordinates inserted into the buffer set below the "Problem 1" comment:

```
/// Problem 1: Modify texture coordinates below.

// Add a square consisting of a light gray and green triangle.
bset_plain << pCoor(-1,-1,-2) << pCoor(-5,3,-2) << pCoor(-5,-1,-2);
bset_plain << color_light_gray << color_light_gray << color_light_gray;
bset_plain << pTCoor(0,0) << pTCoor(1,1) << pTCoor(0,1);  // Modify this line.

bset_plain << pCoor(-1,-1,-2) << pCoor(-1,3,-2) << pCoor(-5,3,-2);
bset_plain << color_green << color_green << color_green;
bset_plain << pTCoor(0,0) << pTCoor(1,1) << pTCoor(0,1);  // Modify this line.
```

**Problem 2:** The code in the unmodified assignment assigns texture coordinates to hyperboloid vertices in such a way that there are tiny syllabi repeated many times over the hyperboloid. See the top screenshot on the first page of this assignment. In the unmodified assignment and in the code fragment below texture coordinates are assigned for the triangle-strip hyperboloid so that the height of the syllabus matches one edge of a triangle and the width matches another edge, which is why each syllabus is so small.




```
case PV_Strip:
  bset_hyperb.reset(pipe_hyperb_strip);
  for ( int side = 0; side < n_sides; side++ )
    for ( int lev = 0; lev <= n_levels; lev++ ) {
        pCoor p00 = coords[ idxf( lev, side ) ];
        pCoor p01 = coords[ idxf( lev, side + 1 ) ];
        pNorm n00 = norms[ idxf( lev, side ) ];
        pNorm n01 = norms[ idxf( lev, side + 1 ) ];

        bset_hyperb << p00 << p01;
        bset_hyperb << n00 << n01;
        bset_hyperb << pTCoor(0,lev%2) << pTCoor(1,lev%2);
```

Modify this code so that there is one copy of the syllabus from top to bottom. The number of copies around the circumference can be chosen for readbility. For the solution shown in the upper screenshot there are $3\pi$ copies around the circumference. *The factor $\pi$ is a hint.*

The vertical axis of the texture (syllabus) must be aligned with the axis of the hyperboloid, as shown in the upper screenshot. The lower screenshot shows a **partial-credit** solution in which the vertical axis of the texture is slanted (following the line used to construct the hyperboloid).

**Problem 3:** Modify the code in `World::render` and the code in `hw05-shdr.cc` so that the hyperboloid is correctly drawn for a points topology (grouping). When the grouping is set to `PV_Points`

(shown as POINTS in the green text) the code uses pipeline pipe_hyperb_points which is set to a vk::PrimitiveTopology::ePointList topology. This pipeline is set up so that the vertex shader is not provided any attributes from the CPU. No color, no normal, not even a vertex coordinate. Since there are no vertex shader inputs there is no buffer set.

Though the vertex shader does not have inputs from the CPU it does have one input that may be useful: gl_VertexID. Suppose the vertex shader is invoked $n$ times. Then for the first invocation gl_VertexID=0, for the second invocation gl_VertexID=1, and so on. Ordinarily the number of invocations is the number of elements in the buffer set. But pipe_hyperb_points does not use a buffer set. Instead, the number of "vertices" is specified in the draw command:

```
case PV_Points: {
    const int n_vtx = 100; // Needs to be changed!
    pipe_hyperb_points.ds_set( transform * trot );
    pipe_hyperb_points.record_draw(cb, n_vtx );
  }
```

In the code above the number of vertices is hardcoded to 100, and so the vertex shader will be invoked 100 times. In a correct solution to this problem the value of n_vtx needs to be changed to the number of invocations needed to draw the hyperboloid.

The geometry shader has a similar input, gl_PrimitiveIDIn. Like the vertex shader, it is set to the number of invocations of the geometry shader. In a point grouping its value will range from 0 to n_vtx-1, where n_vtx is the value used in the record_draw above.

The shaders for pipeline pipe_hyperb_points have access to two arrays, hyperb_coords and hyperb_norms. These arrays have the same contents as coords and norms used by code in World::render to populate the buffer sets for individual triangle and triangle strip groupings. Containers coords and norms are used in the excerpt shown in Problem 2 and elsewhere in World::render(). Here is an excerpt from the solution to this problem showing how a shader can retrieve a coordinate and normal from the arrays:

```
int idx = level + side * uc.s_levels;
vec4 p = hyperb_coords[ idx ];
vec4 n = hyperb_norms[ idx ];
gl_Position = ut.clip_from_object * p;
```

The code above is retrieving a vertex coordinate, p, transforming it to clip space and writing it to a shader output variable. To retrieve p it computed idx using level and side. The value of level and side were themselves computed using one of the gl_..._ID variables. (That part of the solution won't be shown until after the ultimate due date.)

In the unsolved assignment hyperb_coords and hyperb_norms are available, as are the transformation matrices in ut. To solve the problem additional information from the host is needed. That should be placed in uniform uc, which in the CPU code is in variable uni_hw05 of type HW05_Uniform. Add needed members to HW05_Uniform and assign values to those members in uni_hw05. The code above uses member s_levels which does not exist in the unsolved assignment.

The input to the pipeline is points, but we need triangles to render the hyperboloid. The input to the geometry shader is set to points (that's correct, don't change it), and the output is set to triangle_strip, which is also correct. However, the value of max_vertices might need to be changed.

Here is an outline of what needs to be done to solve this problem.

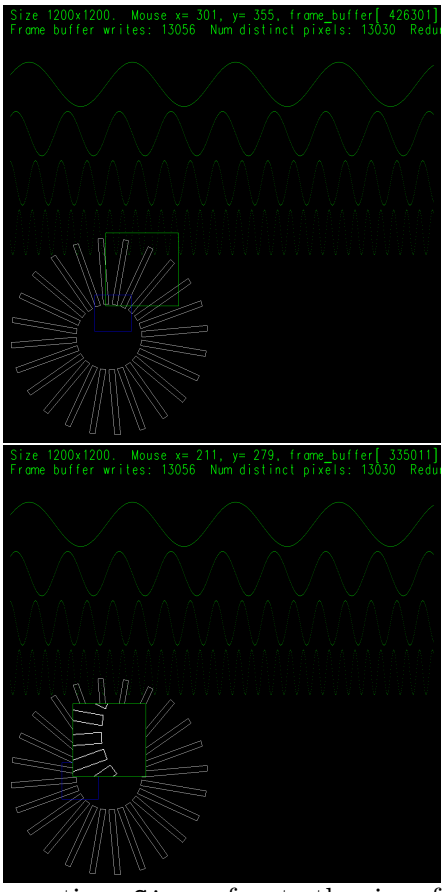
- [ ] Decide how many triangles a geometry shader invocation should emit. Call that $g$. Don't make $g$ too large.

- ☐ Let $T$ denote the total number of triangles in the hyperboloid. Set `n_vtx` (in the CPU code) to $T/g$.
- ☐ In the appropriate shader compute `level` and `side` from `gl_..ID`.
- ☐ Use `level` and `side` to retrieve coordinates and normals.
- ☐ Use the retrieved coordinates and normals to compute shader outputs. (See the shaders in `hw04-shdr.cc` to see what needs to be computed for the fragment shader and how to compute it.)
- ☐ Use `level` and `side` to compute the color index.
- ☐ Use `level`, `side`, and maybe coordinates to compute texture coordinates. ☐ For the full-credit texture appearance additional data will be needed from the host. The slanted textures are much easier.
- ☐ If needed, declare vertex-shader outputs and geometry shader inputs.

# 4 Fall 2023

# LSU EE 4702-1 Homework 1 Due: 1 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/hw01.cc.html`*. The solution code is at* `https://www.ece.lsu.edu/koppel/gpup/2023/hw01-sol.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 some sine waves and radially arranged white rectangles. See the screenshot to the upper right. Two boxes should follow the mouse pointer around the window. The mouse pointer should be in the center of the blue box. A green box is nearby. The relative position and size of the green box can be changed using the keyboard, as described further below. After this assignment is correctly solved the green box should show a zoomed version of what is in the blue box. That appears in the screenshot to the lower right.

## 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 `hw01.png` or `hw01-debug.png`.

To change the size of the green box use the arrow keys. The left and right keys change the width, the up and down keys change the height. Pressing shift and an arrow key will move the green box relative to the blue box. It is possible for the green box to completely cover the blue box.

## Display of Performance-Related Data

The top green text line shows performance-related and other information. `Size` refers to the size of the window. `Mouse` refers to the coordinates of the mouse pointer. Coordinate $(0, 0)$ is at the **lower** left of the window. Text `frame_buffer[N]` shows the index of the frame buffer corresponding to the point under the mouse pointer. (In the assignment file `frame_buffer` is abbreviated to `fb`, for convenience.)

`Render Time` and `Potential Frame Rate` show the CPU time needed to write the frame buffer. They can be ignored for this assignment. See Problem 2 for the second line, the one that starts `Frame buffer writes:`.

## Integer Coordinate and Vector Types

This assignment uses integer 2D coordinate type `iCoor` and integer 2D vector type `iVect`. These types will just be used for this assignment, but in other code we will be using similar types, `pCoor` and `pVect` which are for 3D and in which the components are floating-point types.

The `iCoor` and `iVect` classes each have two members, `x` and `y`:

```
class iCoor {
public:
  iCoor( int x, int y ):x(x),y(y){}
```

```
  int x, y;
};
class iVect {
public:
  iVect( int x, int y ):x(x),y(y){}
  int x, y;
};
```

The addition, subtraction, and division operators are overloaded with these types. Here are some examples of how to use them:

```
void demo() {
  iCoor c0(3,4);  // Set to x=3, y=4.
  iCoor c1;
  c1.x = 5; c1.y = 7;  // Set to x=5, y=7;

  iCoor c3 = c1;

  // Component-wise subtraction.
  // Equivalent to v1.x=c0.x-c1.x;  v1.y=c0.y-c1.y;
  iVect v1 = c0 - c1;

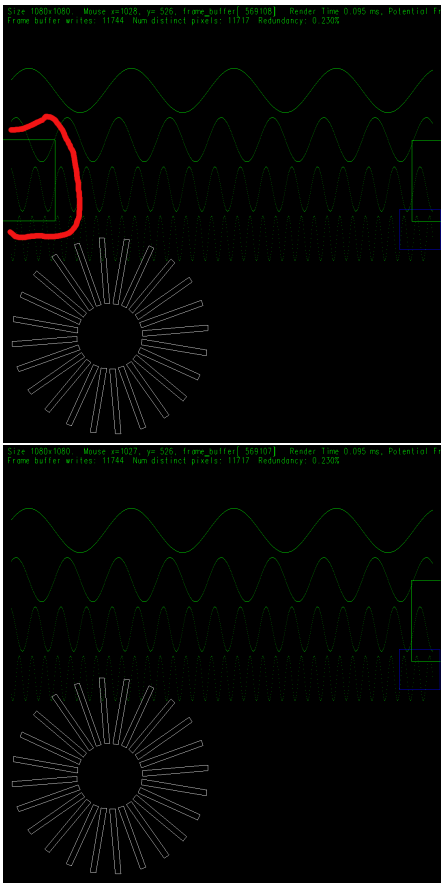
  // Component-wise addition.
  // Equivalent to c4.x=c1.x+v1.x;  c4.y=c1.y+v1.y;
  iCoor c4 = c1 + v1;

  // Not allowed: Can't add two coordinates.
  iCoor c5 = c4 + c1;
}
```

## Code Generation and Debug Support

The compiler generates an optimized version of the code, `hw01`, and a debug-able version of the code, `hw01-debug`. The `hw01-debug` version is compiled with optimization turned off, which makes it easier to debug. When needed, you are strongly encouraged to run `hw01-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.

**Problem 1:** In the unmodified assignment, when the mouse is near the right window edge the green box will wrap around to the left side of the window. See the upper screen-shot to the right, where the wrapped part is circled (almost) in red. The lower screen shot is from code in which this problem is correctly solved but Problem 2 is not started. The green box does not wrap to the left side.



The green box is drawn by calling `aa_rectangle_draw` (`aa` is for axis-aligned). The rectangle is specified by a lower-left coordinate, `p00`, a width, `wd`, and a height, `ht`. The routine draws the rectangle by calling a line drawing routine four times. Modify `aa_rectangle_draw` so that the rectangle it draws does not go beyond a window edge (and so the green box won't wrap).

The routine `aa_rectangle_draw` calls `line_draw` four times. What makes this problem easy is that each line is parallel to an axis. The $x$-axis values range from `p00.x` to `p00.x+wd`. These need to be checked and possibly adjusted to stay within the range 0 to `win_width-1`. Also note that if `p00.x` is less than zero, the left vertical line should not be drawn. Similar reasoning is used to skip drawing the other three lines of the rectangle.

*There is another problem on the next page.*

**Problem 2:** Code near the end of `render_hw01` draws the blue and green rectangles described in the homework introduction. Add code to the routine so that the contents of the frame buffer inside the blue rectangle is copied to the part of the frame buffer inside the green rectangle. A similar problem was asked in 2022 Homework 1, feel free to look at its solution to help get started.

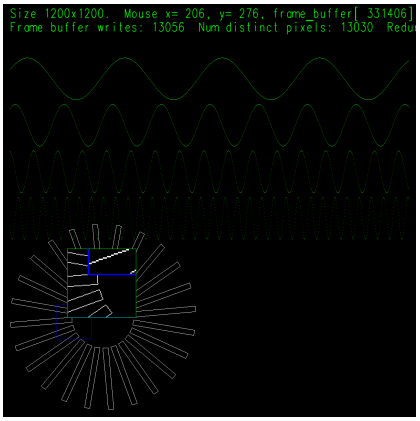
The coordinates of the lower-left of the blue rectangle, called the `mag_from` area in the code, is in variable `mag_from_ll`, a variable of type `iCoor`. (See description in the problem introduction.) The width and height are in `mag_from_diag`.

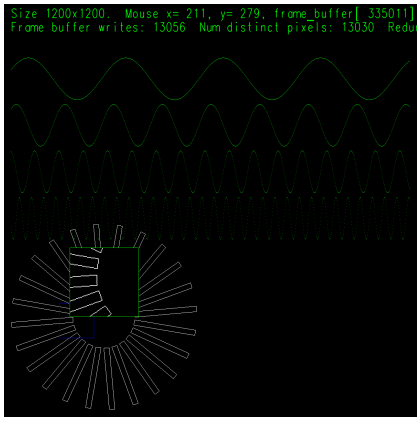
The coordinates of the lower-left of the green rectangle, called the `mag_to` area in the code, is in variable `mag_from_ll`, and the width and height are in `mag_to_diag`.

In order to solve this problem correctly when the blue and green box overlap, the contents of the blue box will need to be copied to intermediate storage before it is copied to the green box. Use array (actually a vector) `hw01_data.fb_area_dup` for this purpose. The size is `mag_from_diag.x * mag_from_diag.y` elements.

To help with your understanding of how to read and write the frame buffer an the intermediate storage there is code that draws diagonal lines into the green and blue rectangles, and into the intermediate storage. That code is guarded by `show_starter_code`. To actually toggle the visibility of those diagonal lines press `y`.

The screenshot to the lower right was taken from code with the correct solution. Notice that the green box only shows zoomed material and that one cannot see the blue box inside the green box. The shot to the left shows an incorrect solution, one in which intermediate storage was not used or not used properly.

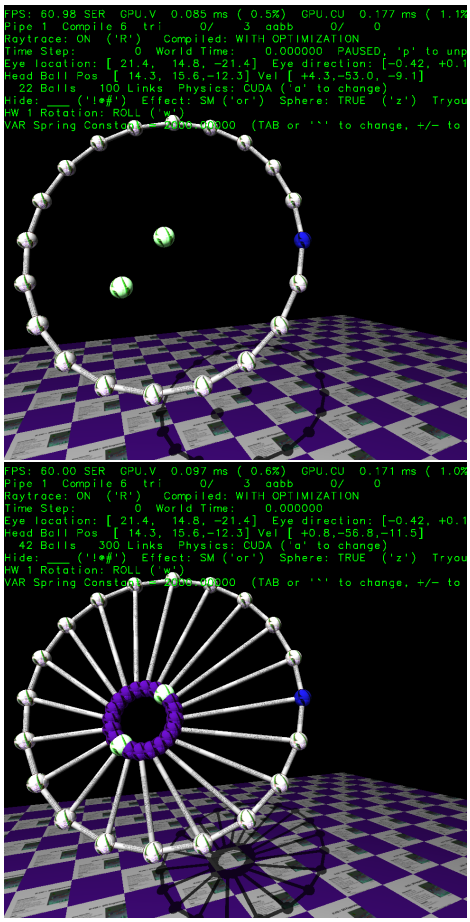



Make sure that your solution works correctly when the green box is moved and re-sized. Use the arrow keys to re-size and shift-arrow to move the green box.

## 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`.

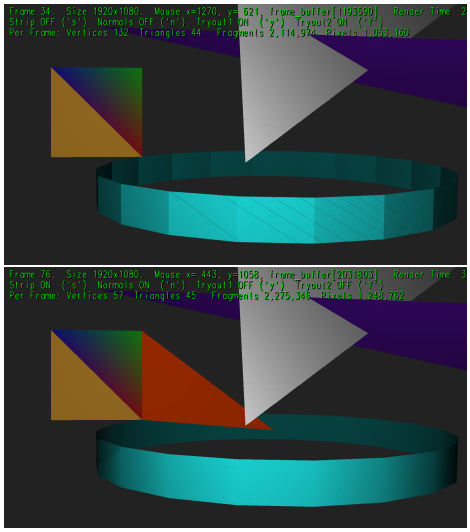
To determine the velocity of a ball one must find its minimum distance from the axis. (The `Rot_Roll` case is easier because it is only necessary to find the distance between the ball and the center of gravity, which is a point.)

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.

# LSU EE 4702-1 Homework 3 Due: 11 October 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/hw03.cc.html`*. The solution code is at* `https://www.ece.lsu.edu/koppel/gpup/2023/hw03-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 CPU-only rendering pipeline demo in directory `cpu-only/demo-06-rend-pipe.cc`. The homework code should initially show a scene with a ring and some triangles, see the screenshot to the upper right. The screenshot on the lower right is from a correct solution. There is a new red triangle (Problem 1), the scene is rendered using fewer vertices (Problem 2), and the ring looks rounder (Problem 3).



## Homework Overview

To solve this assignment one will have to consider the same issues considered in the design of 3D acceleration libraries like OpenGL and Vulkan. The homework code contains to major classes, `World` and `Our_3D`. Code in `Our_3D` is part of a fictional 3D rendering library, the kind of code that an OpenGL or Vulkan implementation would provide. The code in `World` is application code that makes use of the 3D acceleration library. It does so by passing arrays of coordinates and other data to `Our_3D` and then calling `Our_3D::draw_rasterization` to write the frame buffer based on those coordinates. In this assignment `World` draws a simple scene, but it is supposed to represent applications such as flight simulators, video games, or visualization. The code in `Our_3D` represents code provided as part of a device driver that came with your video card or operating system. It would be written to run fast on a particular piece of hardware, such as a video card using an RTX 4090 GPU. In this assignment though the code is not even close to tuned to any hardware device.

Routine `World::render_scene` prepares a scene to be rendered. In the unmodified assignment it consists of a ring plus four additional triangles. The routine fills `coors_os` (coordinates in object space) with the coordinates of each vertex of each triangle. A second container, `colors`, holds the color of each vertex. For example, the code below inserts coordinates and colors for two triangles (note that `<<` inserts an item, it is an overload of `push_back`):

```
vector<pCoor> coors_os;
vector<pColor> colors;

coors_os << pCoor( 0, 0, 0 ) << pCoor( 9, 6, -5 ) << pCoor( 0, 7, -3 );
colors << color_white << color_white << color_white;

coors_os << pCoor(-4,2,-3) << pCoor(-2,2,-3) << pCoor(-2,0,-3);
colors << color_blue << color_green << color_red;
```

(The code samples here are condensed versions of the code in `hw03.cc`.) Note that `coors_os` is declared in `World::render_scene` and so `Our_3D` can't automatically see these coordinates. `World::render_scene` also provides transformation matrices to `Our_3D`:

```
// Specify Transformation from Object to Eye Space: eye_from_object.
pMatrix_Translate center_eye(-eye_location);
pMatrix_Rotation rotate_eye(eye_direction,pVect(0,0,-1));
pMatrix eye_from_object = rotate_eye * center_eye;
gc.transform_eye_from_object_set(eye_from_object);

// Transform from Eye to Clip
pMatrix_Frustum clip_from_eye
  ( -width_m/2, width_m/2,  -height_m/2, height_m/2,  1, 5000 );
gc.transform_clip_from_eye_set(clip_from_eye);
```

To actually render the scene the coordinate and color container information needs to be passed to `Our_3D` and then a *draw* command needs to be issued. That is done by the code below:

```
gc.vtx_coors_set(coors_os);  // Copy address of coors_os
gc.vtx_colors_set(colors);   // Copy address of colors.
gc.draw_rasterization();     // Render the scene.
```

Note that `Our_3D` does not copy the containers, it only keeps an address, so additional items can be added before `gc.draw_rasterization()` is called.

In the unmodified code `gc.draw_rasterization()` is called once per frame. But it could be called multiple times, each time with some subset of triangles. For example:

```
coors_os << pCoor( 0, 0, 0 ) << pCoor( 9, 6, -5 ) << pCoor( 0, 7, -3 );
colors << color_white << color_white << color_white;

// WARNING: Example of bad code.  Better to call draw_rasterization fewer times.
// Draw the one triangle above. Just one.
gc.vtx_coors_set(coors_os).vtx_colors_set(colors).draw_rasterization();
coors_os.clear(); colors.clear();

coors_os << pCoor(7,2,1) << pCoor(-3,3,-3.5) << pCoor(9,0,0);
colors << color_lsu_spirit_purple << color_lsu_spirit_purple << color_lsu_spirit_purple;

// WARNING: Example of bad code.  Better to call draw_rasterization fewer times.
// Draw the one triangle above. Just one.
gc.vtx_coors_set(coors_os).vtx_colors_set(colors).draw_rasterization();
coors_os.clear(); colors.clear();

coors_os << pCoor(-4,0,-3) << pCoor(-4,2,-3) << pCoor(-2,0,-3);
colors << color_lsu_spirit_gold << color_lsu_spirit_gold
       << color_lsu_spirit_gold;

coors_os << pCoor(-4,2,-3) << pCoor(-2,2,-3) << pCoor(-2,0,-3);
colors << color_blue << color_green << color_red;

// WARNING: Example of bad code.  Better to call draw_rasterization fewer times.
// Draw the two triangles above. Just two. Still wasteful.
```

```
gc.vtx_coors_set(coors_os).vtx_colors_set(colors).draw_rasterization();
coors_os.clear(); colors.clear();
```

The code above is bad because it calls `draw_rasterization` multiple times for no reason. A good reason for calling `draw_rasterization` multiple times is that something needed to be changed, such as vertex grouping (Problem 2) or provision of normals (Problem 3).

Problem 1 can be solved by using `Our_3D` as described above. But for Problems 2 and 3 new functionality will be added to `Our_3D`. In Problem 2 a second way of grouping vertices to triangles, *triangle strips*, will be added to `Our_3D`. In Problem 3 the option of providing per-vertex normals will be added.

### 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 `hw03.png` or `hw03-debug.png`.

Initially the arrow keys, `PageUp`, and `PageDown`, can be used to move around the scene. The eye direction cannot be changed, but feel free to add this functionality if you want it. After pressing `l` the motion keys will move the light instead of the eye, and after pressing `e` the motion keys operate on the eye.

### Assignment-Specific User Interface

Pressing `s` toggles between rendering using triangle strips (`Strip ON` in the green text) and individual triangles (`Strip OFF` in the green text). Before Problem 2 is solved switching to triangle-strip mode will show only 1/3 of the ring's triangles. When Problem 2 is solved correctly switching between triangle strip and individual triangle rendering will have no visual impact (though fewer vertices will be sent down the pipeline).

After Problem 3 is solved correctly pressing `n` toggles between rendering using triangle normals (`Normals OFF` in the green text) versus application-provided normals (`Normals ON` in the green text).

### Display of Performance-Related Data

The top green text line shows performance-related and other information. `Size` refers to the size of the window. `Mouse` refers to the coordinates of the mouse pointer. Coordinate $(0, 0)$ is at the **lower** left of the window. Text `frame_buffer[N]` shows the index of the frame buffer corresponding to the point under the mouse pointer. (In the assignment file `frame_buffer` is abbreviated to `fb`, for convenience.)

`Render Time` and `Potential Frame Rate` show the CPU time needed to write the frame buffer. They can be ignored for this assignment. See Problem 2 for the second line, the one that starts `Frame buffer writes.`.

**Don't overlook** the last line of green text, which shows the number of items processed by the rendering pipeline per frame. One should be able to estimate these numbers. For example, if one is rendering using individual triangles one should be able to compute the number of triangles from the number of vertices. `Fragments` shows the number of fragments processed by what we will call the rasterizer. Recall that a fragment is a part of a triangle covering a pixel. In `draw_rasterization` look for the place where `n_frag_frame` is incremented. `Pixels` is the number of times that the frame buffer is written with a fragment. (Not every fragment is written.)

### Code Generation and Debug Support

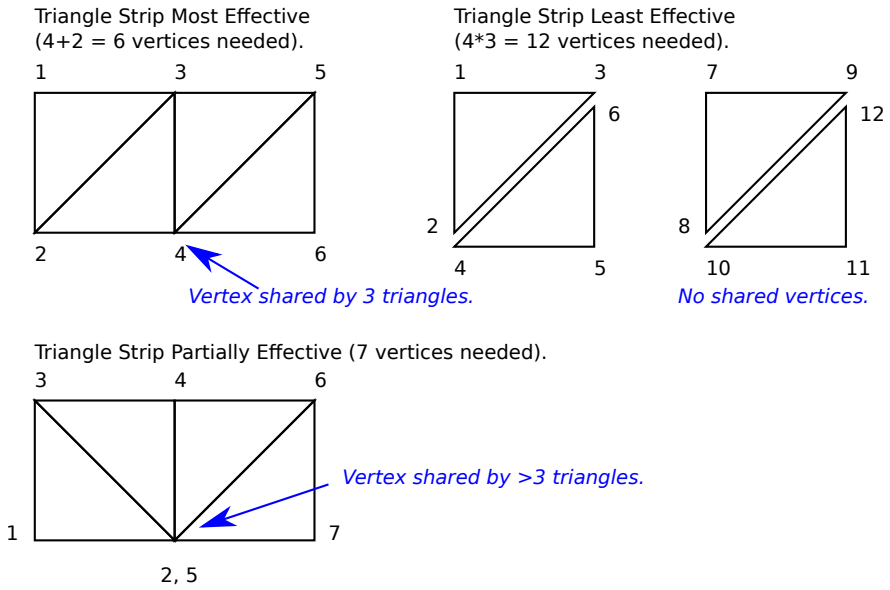
The compiler generates an optimized version of the code, `hw03`, and a debug-able version of the code, `hw03-debug`. The `hw03-debug` version is compiled with optimization turned off, which makes it easier to debug. When needed, you are strongly encouraged to run `hw03-debug` under the GNU

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` and `opt_tryout2`. 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` and `Y` toggle the value of host Boolean variables `opt_tryout1` and `opt_tryout2`.

**Problem 1:** *Warning: This is an easy problem, and will be awarded at most 2 points.* For this problem modify `World::render_scene()` so that a red triangle appears in scene as follows: two vertices should connect to the green and red vertices of the multicolored triangle. The third vertex should be on the ring's axis at the height of the top of the ring. See the screenshot at the beginning of the assignment.

**Problem 2:** In OpenGL and Vulkan a *triangle strip* is an arrangement of $n$ vertices numbered $v_1, v_2, \ldots, v_n$, describing $n-2$ triangles. Triangle $0 < i \leq n-2$ is constructed using vertices $v_i, v_{i+1}, v_{i+2}$. (This is not important for this problem, but when $i$ is even the order is $v_{i+2}, v_{i+1}, v_i$.) Note that a triangle strip consisting of three vertices is just one triangle. The illustration below shows three ways triangle strips can be used to render four triangles. The one on the upper left is best.




.

Modify `Our3D::draw_rasterization` so that optionally the coordinates passed by `Our_3D::vtx_coors_set(COORDS)` are interpreted as triangle strips rather than individual triangles. To solve this problem one must devise a way of telling `Our_3D` whether to expect a triangle strip or individual triangles. That could be another call such as `Our_3D::triangle_arrangement_set`. Also, modify `Our_3D::draw_raseterization` to honor the setting.

When `World::opt_strip` is true the code constructing the ring inserts coordinates in triangle-strip order. (That's pre-solved, already done.) Use your new API to render the ring using triangle

strips. The other triangles can still be rendered as individual triangles, perhaps by using a separate `draw_rasterization` call for the ring.

Pay attention to the number of vertices and triangles displaced in the green text when switching between the two modes. The number of triangles should not change, but the number of vertices should drop from 132 (or 135) to 54 (or 57) when triangle strips is turned on.

Do not expect a performance improvement. Though the size of the coordinate and color arrays will be much smaller, the number of fragments should not change, and that is where most of the work is done.

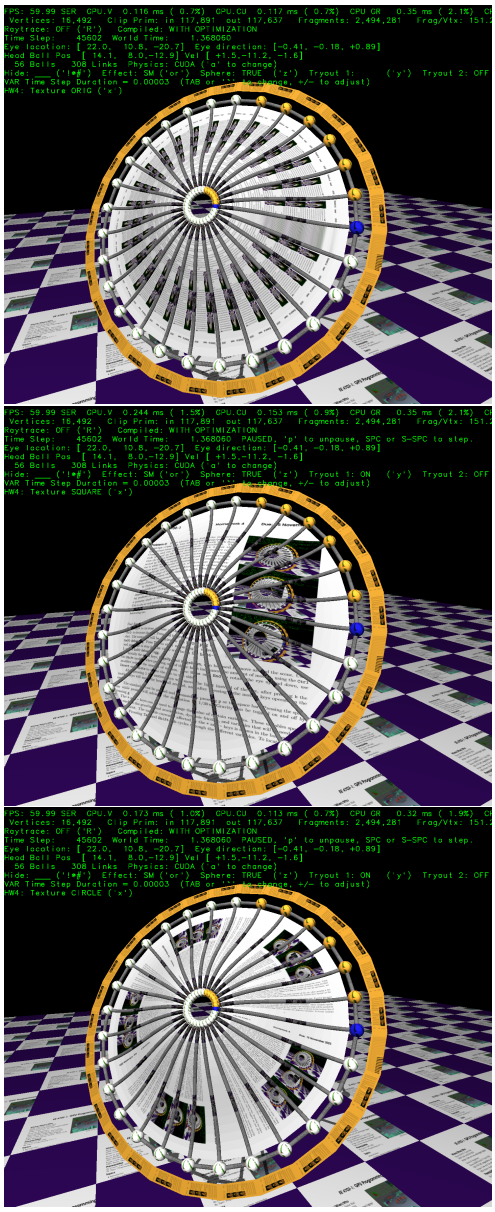
**Problem 3:** `Our_3D` is a bit unusual in that it computes triangle normals. In classic OpenGL the user code provides a normal for each vertex. (In modern OpenGL and Vulkan it is up to user-written shaders to compute lighting. Whether they do so by computing triangle normal themselves or reading a normal streamed in by the user is up to the writer of the shader, not OpenGL or Vulkan.)

Modify `World` and `Our_3D` so that when `opt_normal` is true `render_scene` passes normals to `Our_3D`, and `Our_3D` uses these normals. The normals for the ring should be computed based on the shape of the ring. (Those vectors are already computed.) (Just do this for the ring.) The rasterization loop in `Our_3D` should use these normals (when provided) for lighting. As it already does with colors and object-space coordinates, the rasterization loop should compute a normal for a fragment by blending the normals from the three vertices. When solved correctly, the ring will have a smoother appearance, as in the lower screenshot at the beginning of the assignment.

As in the previous problem, one needs to devise a mechanism to pass normals to `Our_3D`, and `Our_3D` must be able to work without normals as it does in the unmodified assignment.

## LSU EE 4702-1 Homework 4 Due: 15 November 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/hw04.cc.html` *(CPU code) and* `https://www.ece.lsu.edu/koppel/gpup/2023/hw04-shdr.cc.html` *(shader code).*



**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 wheel from Homework 2, but this time the spokes between the inner and outer ring are connected by flexible material. In the screenshots the material is white with an image of the first page of this homework assignment handout projected in various ways. The material is simulated as triangles interacting with air, and the material is rendered as a surface following the shape of the spokes. Also there is a flat ring surrounding the wheel, that is shown in gold, also showing this homework in the screenshots.

In Problem 1 the gold ring is to be drawn. This is a straightforward triangle strip drawing problem. It's supposed to be the confidence builder so if you're having problems on it **please ask for help!** In Problem 2 a geometry shader is used to compute surface normals. This is also an easy problem. Problem 3a, finding texture coordinates, is of medium difficulty. Problem 3b is the interesting one.

### 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 `hw04.png` or `hw04-debug.png`. Press `F10` to start recording a video, and press `F10` to stop it. The video will be in file `hw04-1.ogg` or `hw04-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 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 1 through 5. For this assignment use scene 4 (which is how the code starts). 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 wheel), 5 for a top (a spinning child's toy). Scene 1 is initialized by the code in World::ball_setup_1, etc. Parts of the solution to this assignment should be put in World::ball_setup_4.

The scene for this assignment, 4, initializes randomly each time it is reset. (That is to avoid solutions that accidentally only work for a special case of positioning, etc.)

In this assignment the projection of textures on to the spoke material can be changed by pressing x. There are four possible projections NONE, meaning no textures, ORIG, meaning the one shown on the top screenshot, SQUARE, which, when solved, shows the middle screenshot, and CIRCLE, which, when solved, shows the bottom screenshot. Pressing x cycles between these projections. Variable opt_texture is set to the current projection, which can be TX_None, TX_Orig, TX_Square, or TX_Circle. (In hw04.cc those are enumeration constants in hw04-shdr.cc those are integer constants.)

### 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, hw04, and a debug-able version of the code, hw04-debug. The hw04-debug version is compiled with optimization turned off, which makes it easier to debug. When needed, you are strongly encouraged to run hw04-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`, available in CPU and shader code. 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.

Resources

A good reference for C++ is `https://en.cppreference.com/w/`. Solutions to the shader programming problems may (will) require the use of library functions. See Chapter 8 of The OpenGL Shading Language Version 4.6 specification.

**Problem 1:** Modify the code in `render_hw04`, excerpted below, so that it renders a gold ring around the wheel (see the screenshots).

```
/// Put Homework 4, Problem 1 solution below.
//

// Compute center of outer ring.
pCoor ctr(0,0,0);
const auto& balls_outer = hw04_stuff.balls_outer;
for ( Ball* b: balls_outer ) ctr += b->position;
const size_t n_balls = balls_outer.size();
ctr.w = n_balls;
ctr.homogenize();
pNorm wz [[maybe_unused]] =    // Wheel z
  cross(ctr,balls_outer[0]->position,balls_outer[n_balls/4]->position);
for ( size_t i = 0; i < n_balls; i++ )
  {
    Ball *b = balls_outer[i];
    pCoor pos = b->position;
    pNorm n(ctr,pos);
    pCoor p = pos + 2 * b->radius * n;
    bset_hw04_p1 << p << color_cyan << pNorm(0,1,0) << pTCoor(0,0);
  }
```

To help with this problem the current center of the wheel is in variable `ctr` and an approximate normal is in `wz`. Also, there is a loop that iterates over each outer-ring ball. In that loop body a vector, `n`, and coordinate, `p`, are computed that can be used to construct the ring. These are used to compute a point, but not necessarily the point(s) needed to solve the problem. Also, a placeholder color, normal and texture coordinate are inserted.

Pipeline `pipe_hw04_p1` is used to render the ring. It expects a vertex coordinate, color, normal, and a texture coordinate. Placeholder values are inserted for each of these attributes, which must be modified for the correct solution. Textures are optional for this problem. Texture coordinates might be chosen to help understand how texture coordinates work, to assist in solving Problem 3.

**Problem 2:** Pipeline `pipe_hw04_p2` has only two inputs, a vertex coordinate and a texture coordinate. A normal is needed to compute lighting. Modify the shaders in `hw04-shdr.cc` so that they compute a normal based on triangle coordinates. *Hint: This is easy too.*

**Problem 3:** Texture coordinates for the spoke material is assigned in routine `World::ball_setup_4()`, search for Homework 4 in that routine. Coordinates are assigned to `Ball` members `Ball::tc.orig`, `Ball::tc.square`, and `Ball::tc.circle`.

```
for ( int i=0; i<n_balls; i++ ) {
    const float theta = d_theta * i;
    pVect pos_n = cosf(theta) * ax + sinf(theta) * ay;
    pCoor pos_o = center_pos + ring_outer_radius * pos_n;
    pCoor pos_i = center_pos + ring_inner_radius * pos_n;

    // Construct new ball and add it to the ball list.
    Ball* ball_o = balls_outer.emplace_back( new Ball(pos_o,ballo_default) );
    Ball* ball_i = balls_inner.emplace_back( new Ball(pos_i,balli_default) );
    Link *link = new Link(ball_o,ball_i, {.spring_constant=opt_spring_constant/200});
    links.push_back(link);
    hw04_stuff.ribs.push_back(link);

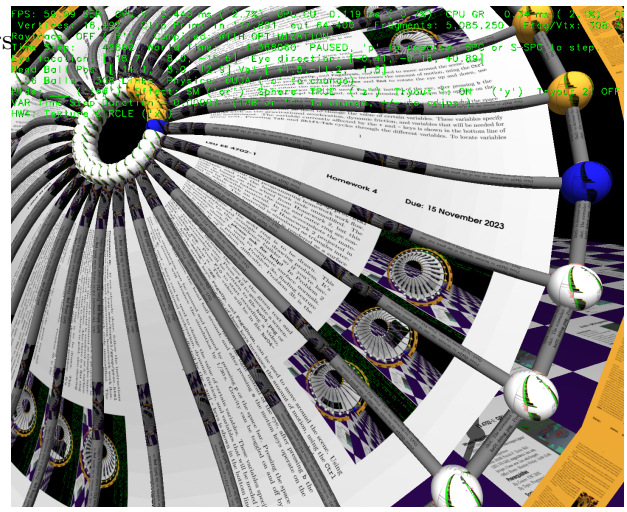
    // Homework 4 -- Change these values for experimentation, if you like.
    ball_o->tc.orig = pTCoor(i,0);     ball_i->tc.orig = pTCoor(i,1);

    //  HW 4: Modify the assignments to the tc.square and tc.circle members.
    ball_o->tc.square = pTCoor(i,0);   ball_i->tc.square = pTCoor(i,1);
    ball_o->tc.circle = pTCoor(i,0);   ball_i->tc.circle = pTCoor(i,1);
```

In the unmodified assignment all three are assigned the same texture coordinate. Note that the code assigns texture coordinates for an outer-ring ball, `ball_o`, and an inner-ring ball, `ball_i`.

(*a*) Assign `Ball:tc.square` so that the texture is mapped as a square. The blue ball is on the middle right-hand side and the yellow balls are on the upper-left quadrant. See the middle screenshot on *the first page, not the one below*.

(*b*) Assign `Ball:tc.circle` and modify the shaders so that the texture is mapped wrapping around the ring as shown in the screenshot to the right and the bottom screenshot on the first page. In those screenshots about four copies of the image are shown.




The exact number of copies is not important, however the text away from the center must be readable (not stretched or compressed). Notice that the lines of text are parallel to the spokes (actually parallel to their centers, not the surfaces).

This is more difficult than part a because it requires more than just assigning the right texture coordinates. To solve this correctly one might need to provide additional uniform variables. Here is how one might add a new variable `factor_x`.

*Step 1 – Declare in CPU Code*
Add the variable to `struct Uni_HW04`:

```
struct Uni_HW04
{
  pColor color;
  int opt_texture;

  float factor_x; // Added a new variable
};
```

*Step 2 – Assign in CPU Code*
Assign the value, a convenient place is in `hw04_render()`.

```
  uni_hw04_needs_update = true; // Set this if x changes every frame.

  if ( uni_hw04_needs_update )
    {
      uni_hw04_needs_update = false;
      uni_hw04->color = 1.0 * color_white;
      uni_hw04->opt_texture = opt_texture;
      uni_hw04->factor_x = x; // Assign new member.
      uni_hw04.to_dev();
    }
```

*Step 3 – Declare in Shader Code*
Declare in the uniform shown below. The type and order must match the CPU code. Putting things in the wrong order or using the wrong type will result in incorrect execution but there won't be an error message.

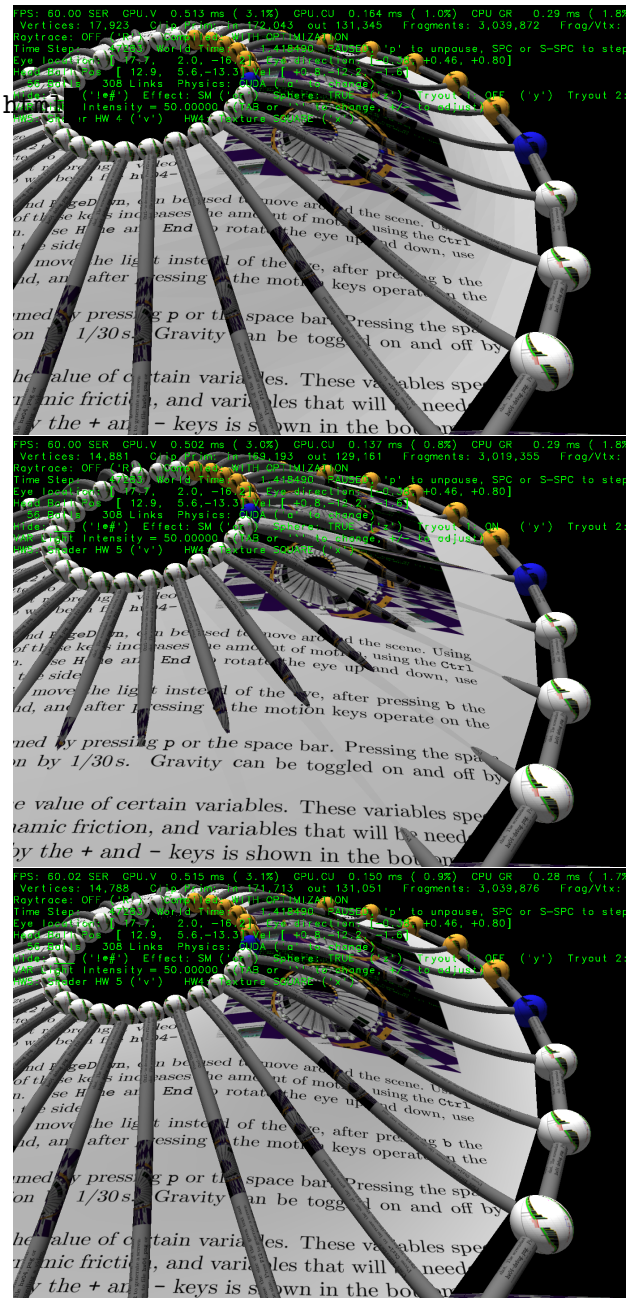
```
layout ( binding = BIND_HW04 ) uniform HW_04
{
  vec4 color;
  int opt_texture;
  float factor_x;  // New member.
};
```

*Step 4 – Use The Variable in Shader Code*
Here the value is assigned to the red component of the lit color.

```
void fs_main()
{
  vec4 lit_color = generic_lighting(In.vertex_e, color, In.normal_e);
  lit_color.r = factor_x;
```

# LSU EE 4702-1 Homework 5 Due: 22 November 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/hw05.cc.html` *(CPU code) and* `https://www.ece.lsu.edu/koppel/gpup/2023/hw05-shdr.cc.html` *(shader code).*



**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 solution to Homework 4.

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 `hw05.png` or `hw05-debug.png`. Press `F10` to start recording a video, and press `F10` to stop it. The video will be in file `hw05-1.ogg` or `hw05-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` 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 1 through 5. For this assignment use scene 4 (which is how the code starts). 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 wheel), `5` for a top (a spinning child's toy). Scene 1 is initialized by the code in `World::ball_setup_1`, etc. Parts of the solution to this assignment should be put in `World::ball_setup_4`.

The scene for this assignment, 4, initializes randomly each time it is reset. (That is to avoid solutions that accidentally only work for a special case of positioning, etc.)

The code for Scene 4 can be run with two shaders, called HW 4 and HW 5. The shader in use is shown on the bottom line of green text and can be switched by pressing `v`. The HW 4 shader is based on the solution to Homework 4. The HW 5 shader is to be modified in this assignment. Initially it shows a cruder version of the wheel, in which the surface does closely follow the shape of spokes when the spokes are bent. The code setting up the HW 4 shader is in `hw05.cc:World::render_hw04` and the shaders themselves are in `hw04-shdr.cc`. The code setting up the HW 5 shader is in `hw05.cc:World::render_hw05` and the shaders themselves are in `hw05-shdr.cc`.

The projection of textures on to the surface can be changed by pressing `x`. There are three possible projections NONE, meaning no textures, SQUARE, which shows one copy of the texture mapped over the entire wheel, and CIRCLE, which shows the texture wrapped around the wheel, perhaps requiring several copies of the texture. Pressing `x` cycles between these projections. Variable `opt_texture` is set to the current projection, which can be `TX_None`, `TX_Orig`, `TX_Square`, or `TX_Circle`. (In `hw05.cc` those are enumeration constants in `hw05-shdr.cc` those are integer constants.)

### 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, `hw05`, and a debug-able version of the code, `hw05-debug`. The `hw05-debug` version is compiled with optimization turned off, which makes it easier to debug. When needed, you are strongly encouraged to run `hw05-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`, available in CPU and shader code. 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.

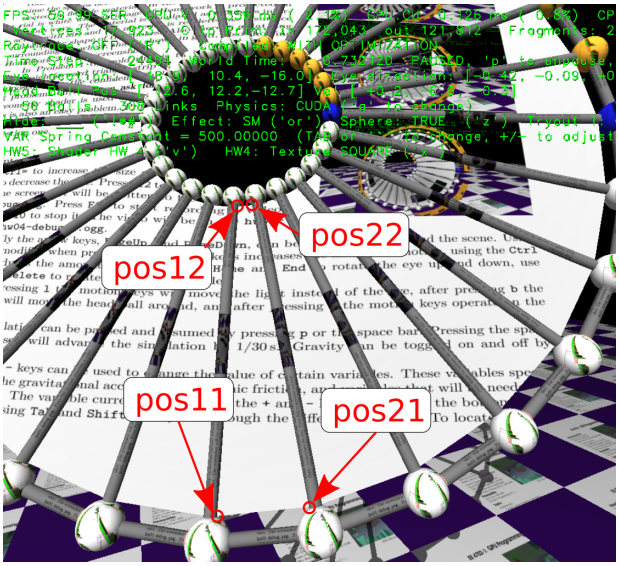
## Resources

A good reference for C++ is `https://en.cppreference.com/w/`. Solutions to the shader programming problems may (will) require the use of library functions. See Chapter 8 of The OpenGL Shading Language Version 4.6 specification. Also see 2014 Homework 4-6 (especially 6), the spiral, to see examples of how the pipeline input topology can be chosen to provide data needed by the geometry shader, even though the topology class (point, line, triangle) does not geometrically correspond to what is being drawn.

*The bottom half of this page intentionally left blank (other than this message).*

## Code Overview

The `render_hw04` routine draws the surface of the wheel using a pipeline set up for a triangle strip. The outer loop in that routine, extracted below, iterates over spokes. In each iteration it uses information from two adjacent spokes, `spoke1` and `spoke2`, to construct the strip. Each spoke has an index, such as `li1`, and that index can be used to get information about the spoke (or link). Array `lis_pos1` holds the coordinate of one spoke endpoint and `lis_pos2` holds the coordinate of the other endpoint. In the code below the two coordinates from each of the two spokes are extracted and placed in variables `pos11`, `pos12`, `pos21`, and `pos22`. The diagram to the right shows the relationship of the coordinates to the spokes and wheel.




```
const size_t n_spokes = hw05_stuff.spokes.size();
for ( size_t i = 0; i < n_spokes; i++ )
  {
    size_t i2 = (i+1) % n_spokes;
    Link *spoke1 = hw05_stuff.spokes[i], *spoke2 = hw05_stuff.spokes[i2];
    const int link_idx1 = spoke1->idx;
    const int link_idx2 = spoke2->idx;
    pCoor pos11 = lis_pos1[link_idx1];
    pCoor pos12 = lis_pos2[link_idx1];
    pCoor pos21 = lis_pos1[link_idx2];
    pCoor pos22 = lis_pos2[link_idx2]; }
```

It would be possible to use these four coordinates to draw two triangles, but those triangles would not match the shape of the spokes, which are cubic curves. An inner loop, shown below, finds points between the endpoints using routine `bez`. Point `pos1t` is between `pos11` and `pos12`, following the curved path defined by the endpoints and vectors `v11` and `v12`. It is points `pos1t` and `pos2t` that are inserted into the buffer set for use in drawing the surface. In addition to the coordinates, texture coordinates are also inserted. The `mix` routine blends the endpoint texture coordinates (not shown in the exceprt) using parameter `t`.

```
    pVect v11 = lis_v1[link_idx1], v12 = lis_v2[link_idx1];
    pVect v21 = lis_v1[link_idx2], v22 = lis_v2[link_idx2];
    const float delta_t = 1.0/opt_segments;
    for ( int s=0; s<=opt_segments; s++ ) {
        const float t = s * delta_t;
        pCoor pos1t = bez(pos11,pos12,v11,v12,t);
        pCoor pos2t = bez(pos21,pos22,v21,v22,t);
        bset_hw04 << pos1t << mix(tex11,tex12,t)
                  << pos2t << mix(tex21,tex22,t);
   }
```

The code in `render_hw05` prepares coordinates for pipeline `pipe_hw05`. The pipe has two inputs, an `int` and a texture coordinate. The pipeline also binds four arrays, `lis_pos1`, `lis_pos2`,

`lis_v1`, and `lis_v2`. Theses are the arrays holding link data used above, and they can be accessed in shader code using C array syntax.

The main loop in `render_hw05` inserts just two vertices (there is no `s` loop) in the pipeline for each spoke (and does the first spoke again at the end):

```
  const size_t n_spokes = hw05_stuff.spokes.size();
  for ( size_t i = 0; i <= n_spokes; i++ ) {
      Link* const spoke1 = hw05_stuff.spokes[i%n_spokes];
      const int link_idx = spoke1->idx;
      pTCoor tex11 = spoke1->ball1->tca[opt_texture];
      pTCoor tex12 = spoke1->ball2->tca[opt_texture];

      bset_hw05 << link_idx << tex11;
      bset_hw05 << link_idx << tex12;
    }
```

Each vertex consists of an integer, `link_idx`, and a texture coordinate. Note that the same index is used twice. The integer is used to index the `lis` arrays. A placeholder vertex shader uses the index, named `in_link_idx` in the shader code, to extract a position:

```
void vs_hw05() {  // Vertex Shader in hw05-shdr.cc
  vec4 p1 = pos1[in_link_idx];  // Outer ring.
  vec4 p2 = pos2[in_link_idx];  // Inner ring.
  vec4 p = ( gl_VertexIndex & 1 ) == 0 ? p1 : p2;
  Out.vertex_o = p;
  Out.vertex_e = ut.eye_from_object * p;
  Out.tex_coor = in_tex_coor;
}
```

The code above actually finds two coordinates. It uses `p1` for even-numbered vertices, and `p2` for odd-numbered vertices. Built-in variable `gl_VertexIndex` provides the vertex number. (Zero for the first vertex in a draw.)

The pipeline is set to group vertices for a triangle strip topology, so the input to the geometry shader is a triangle. The placeholder geometry shader computes the triangle normal and a clip-space coordinate, and emits the triangle:

```
void gs_hw05() {
  vec3 normal_o =
    cross( In[1].vertex_o.xyz - In[0].vertex_o.xyz,
           In[2].vertex_o.xyz - In[0].vertex_o.xyz );
  vec3 normal_e = mat3( ut.eye_from_object ) * normal_o;

  for ( int i=0; i<3; i++ ) {
      gl_Position = ut.clip_from_object * In[i].vertex_o;
      Out.normal_e = normal_e;
      Out.vertex_e = In[i].vertex_e;
      Out.tex_coor = In[i].tex_coor;
      EmitVertex();
    }
  EndPrimitive();
}
```

To help in solving the assignment the shader file has a `bez` routine that can be used for finding points along the spoke. It also has a routine, `bez_dt`, that computes the derivative of the curve

between the two points (with respect to the parametric parameter, `t`). Routine `bez_dt` can be used for finding surface normals.

**Problem 1:** Modify code in `hw05.cc:World::render_05` and the shaders in `hw05-shdr.cc` so that the geometry shader computes the triangles between two spokes. Each invocation of a geometry shader should do the same work as the `s` loop from `render_hw04` described above.
Here is what needs to be done:

*Modify Pipeline Input Topology*
The geometry shader needs information on two adjacent spokes (links) so that it can find the four points. That's not easily possible using a triangle strip topology. Modify the topology used by `pipe_hw05` to better provide this information. This can be done by modifying the argument to `.topology_set` called where the pipeline is set up in `render_hw05`. Consider the approach used in 2014 Homework 6, in which a spiral was constructed using a line list.

*Modify Pipeline Inputs*
In the unmodified assignment `pipe_hw05` is set for two inputs, an int and a texture coordinate. Those are set by the `.shader_inputs_info_set` call by the types between the angle brackets. A limited number of types are accepted and they can't be repeated. The valid types are `int`, `ivec2`, `ivec4`, `pCoor`, `pColor`, and `pVect`. (This is a limitation of the course library, not Vulkan.) An input type does not have to be used for the purpose implied by its name. So a `pColor`, for example, can be used to hold four floats used for any purpose. Note that `ivec2` is a 2-element integer vector.

*Modify Shader Inputs*
For any change made to pipeline inputs (by modifying `.shader_inputs_info_set`) a corresponding change must be made in the shader code. For your convenience some inputs are pre-defined, guarded by ifdefs, such as `in_int4` for an `ivec4` type.

*Modify the Shader Interface Blocks*
A correct solution to this assignment will require changing the information sent from the vertex to the geometry shader. Modify the interface blocks for that. (These are the structure-like constructs with tags like `Data_to_GS`.) Remember that the block describing the output of the vertex shader must match the block at the input to the geometry shader.

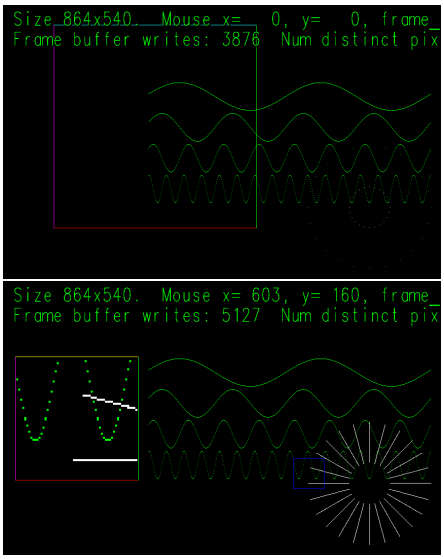
*Modify the Geometry Shader Input and Output Layouts*
The geometry shader output topology should remain a triangle strip, but the `max_vertices` setting should change. Compute `max_vertices` from `opt_segments`, which is available as a constant in the shader file (trust me on that). The input layout to the geometry shader must match what was set on `pipe_hw05`.

*Modify the Geometry Shader to Emit the Triangles*
Modify the geometry shader so that it computes and emits the triangles between the spokes. There should be something like the `s` loop, but adapted for a geometry shader.

*Compute the Correct Surface Normal Based on the Shape of the Curve*
Use `bez_dt` to help get the normal. Note that `bez_dt` does not provide the surface normal, instead it is a vector in a direction of the curve from `posx1` to `posx2` at point `t`.

# 5 Fall 2022

## LSU EE 4702-1 Homework 1 Due: X 2 September 2022




**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 the square from the `cpu-only/demo-01-frame-buffer.cc` code used in class, along with some sine waves and faint dotted concentric circles. See the screenshot to the upper-right. The lower screen shot is from a correctly solved assignment. The square is positioned on the upper-left, and it now shows a magnified version of what is in a blue square (which is where the mouse pointer is). Lines now connect the dots that formed the two concentric circles.

### 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 `hw01.png` or `hw01-debug.png`.

### Display of Performance-Related Data

The top green text line shows performance-related and other information. `Size` refers to the size of the window. `Mouse` refers to the coordinates of the mouse pointer. Coordinate $(0, 0)$ is at the **lower** left of the window. Text `frame_buffer[N]` shows the index of the frame buffer corresponding to the point under the mouse pointer. (In the assignment file `frame_buffer` is abbreviated to `fb`, for convenience.)

`Render Time` and `Potential Frame Rate` show the CPU time needed to write the frame buffer. They can be ignored for this assignment. See Problem 2 for the second line, the one that starts `Frame buffer writes.`.

### Code Generation and Debug Support

The compiler generates an optimized version of the code, `hw01`, and a debug-able version of the code, `hw01-debug`. The `hw01-debug` version is compiled with optimization turned off, which makes it easier to debug. When needed, you are strongly encouraged to run `hw01-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.

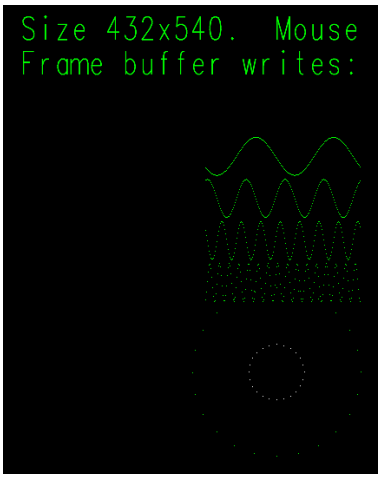
**Problem 1:** The code in routine `render_hw01` includes a fragment that draws a square (by writing the frame buffer), which is based on what was done in class on Wednesday, 24 August 2022:

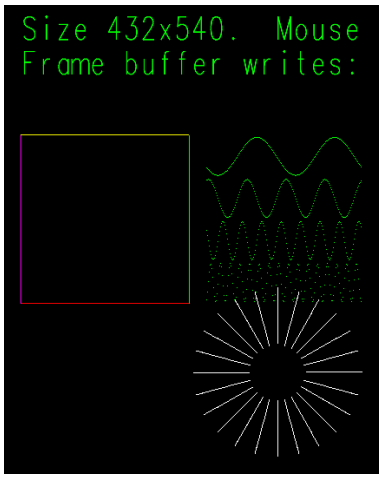
```
for ( int x=100; x<500; x++ )
  {
    fb[ 100 * win_width + x ] = color_red;
    fb[ 500 * win_width + x ] = 0xffff;
    fb[ x * win_width + 100 ] = 0xff00ff;
    fb[ x * win_width + 500 ] = 0xff00;
  }
```

The position of this square is hard-coded to coordinates $(100, 100)$ (meaning $x = 100$, $y = 100$) lower-left and $(500, 500)$ upper-right. That will place the square in the lower-left portion of the window.

Modify the routine so that the square is drawn at (`sq_x0,sq_y0`) lower-left and (`sq_x1,sq_y1`) upper-right, where `sq_x0`, `sq_y0`, `sq_x1`, and `sq_y1`, are variables in the code. Do this by using these variables in the routine that draws the square. If it helps, variable `sq_slen` can also be used. If done correctly the square will be at the upper-left of the window vertically aligned with the sine waves, and the size of the square will be determined by the minimum of the window width and height. The square will adjust whenever the window is resized. See the lower screenshot at the beginning of this assignment.

**Problem 2:** The homework file has a placeholder routine `line_draw(fb,x0,y0,x1,y1,th,co)` which is to draw a line of color `co` from coordinate `x0,y0` to `x1,y1`. Add code to `line_draw` so that it draws such a line. The placeholder code writes a pixel at either end of the line. Code in `render_hw01` uses `line_draw` to draw lines radiating from a point. With the placeholder code, that results in two concentric circle of dots, see the screenshot to the lower left. If the `line_draw` routine is written correctly the lines should be drawn as shown on the screenshot to the right.




The line should appear solid, not dotted. Also, try not to write a pixel more than once when drawing the line. The green text to the right of `Frame buffer writes:` shows how many times the frame buffer was accessed. The text to the write of `Num distinct pixels:` shows how many different pixels were accessed. If the two numbers are the same then each pixel that was accessed was accessed just once. If `Frame buffer writes:` is higher than `Num distinct pixels:` then some pixels were written multiple times. This might happen, for example, when two lines cross. The pixel at their intersection is written twice. That's not a problem. But, it can also happen if code is carelessly written, for example, drawing the same line twice (in exactly the same place).

The text to the right of `Redundancy:` shows the percentage of pixels that were written more than once. When solving this problem make sure that the redundancy stays below 50%.

**Problem 3:** Modify the code near the end of `render_hw01` so that a blue square is drawn around the mouse pointer, and the contents of the frame buffer covered by the blue square is copied to the square in the upper left. (See the lower screenshot at the beginning of this assignment. The mouse pointer is not visible in the screenshots.) Set the length of an edge of the square to `(sq_slen-2)/zoom`, where `sq_slen` and `zoom` are variables in the code. Variable `zoom` is hardcoded to 4, so the square should show a 4× magnification of what is near the mouse pointer. (The length is `(sq_slen-2)/zoom` rather than `sq_slen/zoom` so that the copied pixels do not overwrite the sides of the square.)

To get you started, there is code to copy one pixel from the mouse pointer to the center of the square:

```
const int mouse_x = fb.mouse_x;
const int mouse_y = fb.mouse_y;
const int zoom = 4;

fb[ ( sq_y1 + sq_slen/2 ) * win_width + sq_x0 + sq_slen/2 ]
  = fb[ mouse_y * win_width + mouse_x ];
```

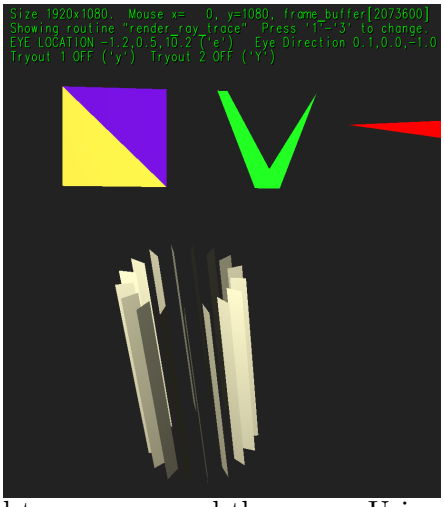
The code above copies just one pixel. To complete the assignment that needs to be replaced by a loop nest that will copy each pixel in the blue square to `zoom*zoom` locations in the big square.

The location of the mouse pointer is in variables `mouse_x` and `mouse_y`. You may use the `line_draw` routines to draw the blue square.

Pay attention to the following:

- All code for this problem must be placed after the `fb.fb_stats_off();` statement.
- The code to draw the blue box must be placed after the code that copies the frame buffer.
- Do not overwrite the square edges.

# LSU EE 4702-1 Homework 2 Due: 4 October 2022




**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 the square and triangles from the `cpu-only/demo-05-ray-tracing.cc` code used in class. See the screenshot to the right, which shows the correctly solved assignment. Solving this adds the green V and the paddle wheel (the object below the square and green V).

### 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.png`.

Initially the arrow keys, `PageUp`, and `PageDown` can be used to move around the scene. Using the `Shift` modifier 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. Press `l` to move the light around and `e` to move the eye (which is what the arrow keys do when the program starts).

### Assignment-Specific User Interface

The scene is rendered by using a cpu-only ray tracing routine. (So don't complain if it's slow.) There are two ray-tracing routines, the one in use is shown in the green text to the right of `Showing routine`. Pressing `1` switches to `render_ray_trace`, this should be used for Problems 1 and 2. Pressing `2` switches to `render_ray_trace_os` (the `os` is for object space). Routine `render_ray_trace_os` won't work until Problem 3 is solved correctly.

### Code Generation and Debug Support

The compiler generates two versions of the code, `hw02` and `hw02-debug`. Use `hw02` to measure performance, but use `hw02-debug` for debugging. The `hw02-debug` version is compiled with optimization turned off. 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.

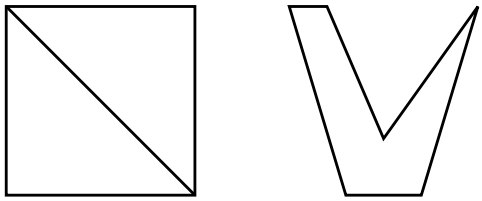
To help you debug your code and experiment in one way or another, the user interface lets you change variables. In this assignment these include the variables `hw02_info.opt_tryout1` and `hw02_info.opt_tryout2`. You can use these variables in your code (for example, `if ( hw02_info.opt_tryout1 ) { x += 5; }`) to help debug, to help familiarize yourself with how the code works, or to experiment with new ideas. Keys `y` and `Y` toggle the value of these variables. Their values are shown in the green text.

### Display of Performance-Related Data

The top green text line shows performance-related and other information. `Size` refers to the size of the window. `Mouse` refers to the coordinates of the mouse pointer. Coordinate $(0, 0)$ is at the **lower** left of the window. Text `frame_buffer[N]` shows the index of the frame buffer corresponding to the point under the mouse pointer. (In the assignment file `frame_buffer` is abbreviated to `fb`, for convenience.)

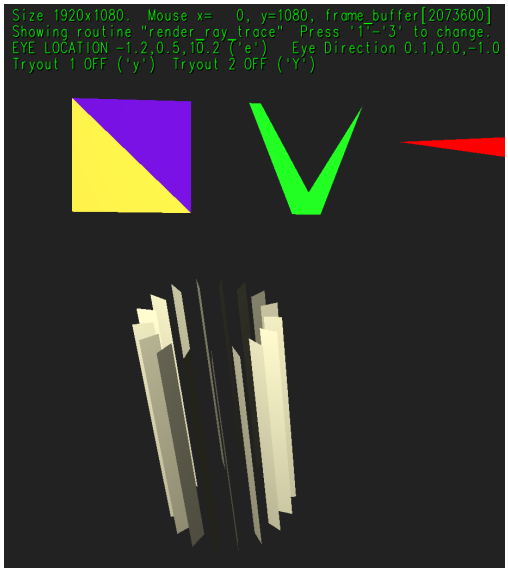
`Render Time` and `Potential Frame Rate` show the CPU time needed to write the frame buffer. They can be ignored for this assignment. See Problem 2 for the second line, the one that starts `Frame buffer writes.`.

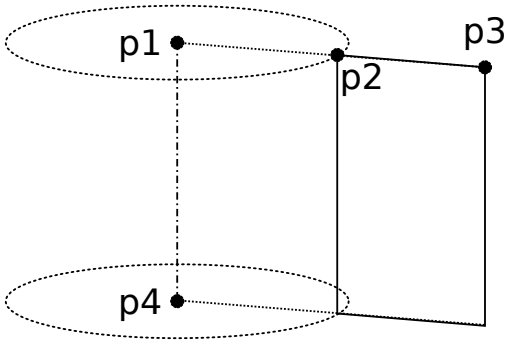
**Problem 1:** The code in routine `prep_secene` includes a fragment that draws a square (by writing the frame buffer) consisting of a purple and gold triangle. Add code beneath that which draws the V-like figure illustrated to the right. Draw the figure using only a few triangles, and do so without overlapping triangles. It might help to make each triangle a different color. The screenshot at the beginning of the assignment shows the V-like figure in green.



See 2016 Midterm Exam Problem 1 and 2018 Final Exam Problem 1 for a roughly similar problems.

**Problem 2:** Modify the code in `prep_scene` below the comment reading `Problem 2` so that it renders a paddle-wheel-like object, as shown in the lower part of the screenshot to the right. The position and shape of the paddle wheel is determined by variables `p1`, `p2`, `p3`, `p4`, and `n_pieces`, see the diagram below the screenshot. Points `p1` and `p4` define the axis of a cylinder (actually two cylinders sharing the same axis but of different radii). Point `p2` is on the surface of one cylinder and point `p3` is on the surface of the other, larger, cylinder. Points `p1`, `p2`, and `p3` will always form a line, and that line is orthogonal to the cylinder axis. The first paddle wheel blade is positioned as shown in the diagram (the rectangle with solid lines). A complete paddle wheel should have `n_pieces-1` additional blades equally spaced around the cylinder axis.




Each time `1` (or `2` when Problem 3 is solved) is pressed new values for the variables will be chosen. Pressing those keys might help in debugging.

*Hint: Review the code for drawing a circle from the Circles section of the math slides. For this problem there will be four circles, two at the top of the cylinder, and two at the bottom. At each iteration of an* `n_pieces` *iteration loop find a point on each circle and connect them to form a blade. The illustration shows just one top circle and one bottom circle. Point* `p2` *is on the illustrated top circle, and point* `p3` *is on a top circle that's not shown.*

**Problem 3:** Modify routine `render_ray_trace_os` so that checks for ray/triangle intercepts using object-space coordinates. This routine is used after `2` is pressed. Routine `render_ray_trace_os` starts out by computing the pixel coordinate (`px_e`) and `ray` in eye space:

```
for (uint yw=0;  yw < win_height; yw++ ) for (uint xw=0;  xw < win_width; xw++ )
    {
      // Eye-Space Coordinate of Pixel.
      pCoor px_e = window_ll_e + window_dx_e * xw + window_dy_e * yw;

      // Ray From Eye to Pixel in Eye Space.
      pVect ray(px_e);
```

But the routine iterates over the object space coordinates of triangles:

```
      for ( auto it = coors_os.begin(); it != coors_os.end(); ) {
          pCoor o0 = *it++,  o1 = *it++,  o2 = *it++;
          uint32_t color = *ic++;

          pVect tn(o0,o1,o2);  // Triangle normal.
          float t = dot( pVect(o0), tn ) / dot( ray, tn ); /// WRONG!
          pCoor s = t * ray; /// WRONG!
```

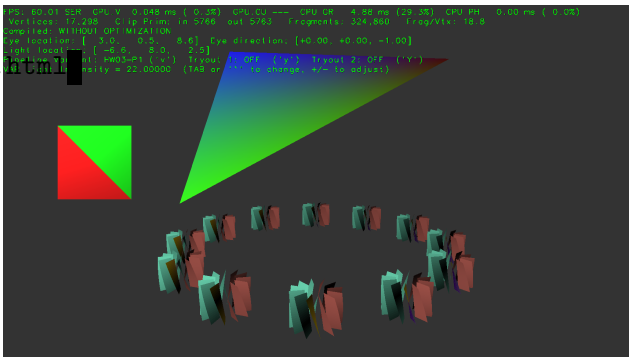
This code is incorrect because the ray is in eye space but the triangle normal and vertex `o0` are in object space. Also, `s` is not computed correctly.

Fix these problems by computing a `px_o` and `ray`, both in object space. **Do not** fix this by using a transformation matrix within the loop nest. Instead directly compute a `px_o` in object space, consider using variables `window_ll_o`, and others to compute `px_o`. Also, please remove the code for `px_e`.

When this problem is correctly solved there should be no difference between rendering with `1` and `2`.

Please review the material on line/plain intercepts to help with this problem.

# LSU EE 4702-1 Homework 3 Due: 17 October 2022



**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 the square and triangle from the `cpu-only/demo-03-vulkun-one.cc` code used in class, but with spinning turbines. See the screenshot to the right, which shows the correctly solved assignment. In the unsolved code all the turbines are gray.

## 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 `hw03.png` or `hw03-debug.png`.

Initially the arrow keys, `PageUp`, and `PageDown` can be used to move around the scene. Using the `Shift` modifier 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. Press `l` to move the light around and `e` to move the eye (which is what the arrow keys do when the program starts).

## Assignment-Specific User Interface

The scene can be reset by pressing `1` and `2`. When `1` is pressed the number of turbines and their positions is randomly chosen. When `2` is pressed the number and positions are the same each time. Pressing `p` will stop and start the turbines from spinning.

The scene can be rendered using three different pipelines, *Plain*, *HW03-P1*, and *HW03-P2*. The particular set of code in use is shown by the green text to the right of `Pipeline Variant`. Pressing `v` switches between the variants.

## Code Generation and Debug Support

The compiler generates two versions of the code, `hw03` and `hw03-debug`. Use `hw03` to measure performance, but use `hw03-debug` for debugging. The `hw03-debug` version is compiled with optimization turned off. You are strongly encouraged to run `hw03-debug` under the GNU debugger, `gdb`. See the material under "Running and Debugging the Assignment" on the course procedures page.

To help you debug your code and experiment in one way or another, the user interface lets you change variables. In this assignment these include the variables `opt_tryout1` and `opt_tryout2`, which are available both in CPU code and in shader code. 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` and `Y` toggle the value of these variables. Their values are shown in the green text.

## 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 rasterization. (Other assignments will use ray tracing.) 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.

The `+` and `-` keys can be used to change the value of certain variables. These variables specify things such as the light intensity, sphere radius, 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 Background

The key element of this assignment is the rendering of will be called turbines here. (They may not be shaped like real turbines.) Structure `Fan_Info` describes the shape of a turbine and how the blades should be colored. The description is similar though not identical to the way the paddle wheel from Homework 2 was described.

```
struct Fan_Info {
  pCoor pos_axis_top, pos_axis_bottom;
  pVect axis_to_cyl;
  float outer_radius;
  int n_blades;
  pColor blade_0_front_upper, blade_0_front_lower;
  pColor blade_0_back_upper, blade_0_back_lower;
  pColor blade_i_front_upper, blade_i_front_lower;
  pColor blade_i_back_upper, blade_i_back_lower;
  int serial;
};
```

The values of coordinates and vectors used above are in a local coordinate space. A transformation matrix is used to move the turbine to its intended position. In this assignment there is one `Fan_Info` object `fan_info`, and a array of transformation matrices, `fans_xforms`. (The code to generate triangles corresponding to the turbines has already been written. Look for `case PV_Plain` in the code.)

Routine `fan_setup` writes both `fan_info` and `fans_xforms`, it is called when `1` or `2` is pressed. Each time `fan_setup` executes it increments `fan_info.serial`. The purpose of `fan_info.serial` is to make it possible for other code to detect when `fan_info` has changed. (That is relevant to this assignment.)

In this assignment the turbines are to be rendered by the three pipeline variants, plain, hw03p1,

and hw03p2. (See the assignment-specific user interface section, above.) The code for the plain variant is complete (except maybe for the colors, see Problem 1).

The plain variant uses `VPipeline pipe_plain` and buffer set `bset_plain`. Here is an excerpt of the code, omitting the normals for brevity (the complete code is in `hw03.cc`):

```
case PV_Plain:
  for ( auto& m: fans_xforms )
    for ( int i=0; i<f.n_blades; i++ ) {
        float theta = theta_0 + i * delta_theta;
        pCoor p1_last, p2_last;
        for ( int j=0; j<=n_slices; j++ ) {
            float eta = theta + j * delta_eta;
            pVect v = ax * cosf(eta) + ay * sinf(eta);
            pCoor pa = f.pos_axis_top + delta_vz * j;
            pCoor p3 = m * ( pa + r1 * v );
            pCoor p4 = m * ( pa + f.outer_radius * v );
            if ( j ) {
                bset_plain << p1_last << p3 << p2_last;
                bset_plain << p2_last << p3 << p4;
                bset_plain << gray << gray << gray << gray << gray << gray;
              }
            p1_last = p3; p2_last = p4;
          }
      }
```

The `j` loop computes triangles for one blade, the `i` loop computes triangles for one turbine, and the `m` loop computes the set of turbines. Those concerned about computational waste will be appalled by the loop nest above. Why? First, the same color is used over and over, surely there must be a better way of specifying the same color a zillion times. That will be fixed in Problem 1, and in a way that allows both front and back colors to be used. Another wasteful issue is the fact that each iteration of the `m` loop does almost exactly the same thing, the only difference is the value of `m`. That will be fixed in Problem 2.

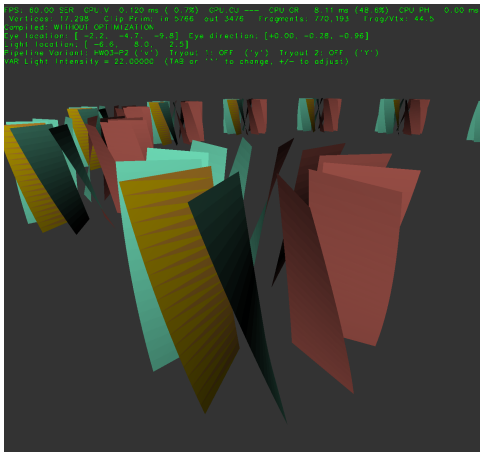
## Code Overview

Routine `World::render` is called each time the frame buffer is to be prepared. It is similar to other render routines used in class, the parts relevant to this assignment will be discussed here.

Objects `pipe_plain` and `pipe_hw03` are `VPipeline` instances. `World::render` initializes both of them, though in the solution to this assignment the initialization of `pipe_hw03` will need to be modified.

Some notes on how to work with the `VPipeline`, `VVertex_Buffer_Set`, and the buffer types can be found in this under-construction Vulkan course library note set.

**Problem 1:** Modify the code in `render` so that the turbine uses colors specified in `fan_info` and helpfully written to variables `b0_fl`, etc. The `blade_0` colors are to be used by the blade emitted when `i=0` in the `case PV_Plain` loop nest. The `blade_i` colors are to be used when `i>0`. The front color is to be used for the front of the blade, and back for the back. (See the fragment shader routine, `fs_main`, in `hw03-shdr.cc`.) Each iteration of the `j` loop emits two triangles. The first is the upper triangle and the second is the lower triangle. See the screenshot to the right. In the screenshot blade 0 has distinct upper- and lower-triangle colors, but in the other blades they are the same. Don't assume that this will always be the case.



There are three parts to this problem, only the last, part c, really needs to be solved. But solving them in order may be helpful.

(*a*) Modify the code for the `PV_Plain` case so that it uses the front colors from the `fan_info` structure, as described above. Just the front colors for this part because the `PV_Plain` pipeline variant can't take back colors. Note that this part is much easier than the others.

(*b*) For this part (and the next one) write a new set of shaders in `hw03_shdr.cc` and modify the CPU code (in many places in `hw03.cc`) so that instead of using a color as a vertex attribute, an integer is used instead. Call this integer a *color set index*. The idea is to replace the code that uses colors:

```
bset_hw03 << color_red << color_blue << color_red;
```

with code that uses indices

```
bset_hw03 << 1 << 2 << 1;
```

The indices refers to colors that will have been placed in a uniform object as part of the solution. The example above implies that 1 refers to position 1 somewhere in the uniform (maybe an array element), which presumable holds red. But for this problem the color index should be used to refer to a pair of colors, front and back. For this subproblem use an existing uniform object to store the colors, but in the next part declare your own. To make things easier, the shader code already expects that there might be an integer input and there might not be a color input:

```
// Vertex Shader Inputs
layout (location = LOC_IN_POS) in vec4 in_vertex_o;
layout (location = LOC_IN_NORMAL) in vec3 in_normal_o;
#ifdef LOC_IN_COLOR
layout (location = LOC_IN_COLOR) in vec4 in_color;
#endif
#ifdef LOC_IN_INT1
layout (location = LOC_IN_INT1) in int in_color_idx;
#endif
```

The following changes need to be made to the shader code:

- A color array(s) must be added to an existing uniform. Note that a color can be declared pColor in CPU code but must be declared `vec4` in shader code.

- The changes to the uniform in the shader code must be made to the corresponding structure in the CPU code.
- Modify the shaders so that they get the correct color using the color set index, named `in_color_idx` in the shader code.
- The shaders should no longer use the `in_color` input.
- Compute a front and back lighted color in the vertex shader.
- Modify the vertex shader `out` declaration and the fragment shader `in` declaration to accommodate the back color.
- Modify the fragment shader to use the back color when appropriate.

On the CPU side the following changes are needed:

- Add the color declarations to the struct used for colors (corresponding to the uniform used in the shader code).
- Populate the uniform with the `fan_info` colors.
- Modify the code in the `PV_HW03_P1` case so that it uses color set indices rather than colors.

When this problem is solved correctly the colors should match the screenshots.

(*c*) In the previous subpart the colors might have been placed in the `Uni_Light_Simple` or `Uni_Misc` structures because they were already there. In this subproblem define a new structure and use that for a new buffer. The new buffer will need to be declared in `World`, initialized and later destroyed in `World::setup_and_run`. In `World::render` the new uniform will need to be bound to the `pipe_hw03` descriptor set (that's what `ds` stands for) when `pipe_hw03` is initialized, and explicitly copied `to_dev`, preferably only when the colors change (see the `serial` member of `fan_info`). Most of the steps needed for your new buffer will match those for `uni_light_simple` and `uni_misc`.

**Problem 2:** As mentioned in the introductory material, the code in `PV_Plain` and `PV_HW03_P1` is wasteful because it re-computes the triangles for all the turbines, even though each turbine is identical (except for position and orientation). In this problem add code to the `PV_HW03_P2` case that writes triangles to `bset_hw03_p2` for just one turbine. In fact, only write these when the serial in `fan_info` changes. Make sure not to reset the buffer set only when it needs to be re-populated. Next, record a draw for each transformation matrix. The code should look something like this:

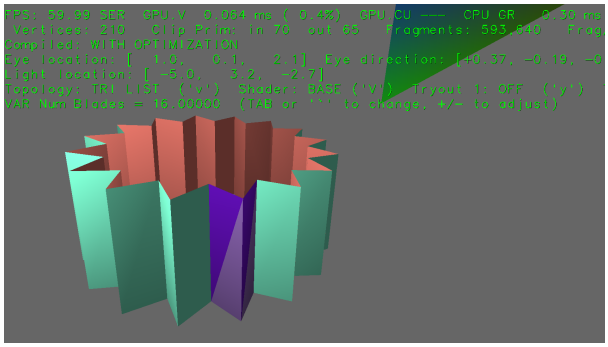
```
for ( auto& m: fans_xforms )
  {
    pMatrix global_from_local = ..;
    pipe_hw03.ds_set( transform * global_from_local );
    pipe_hw03.record_draw(cb, bset_hw03_p2);
  }
```

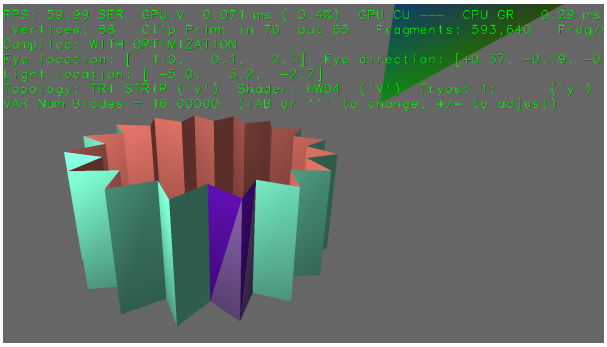
The `pipe_hw03.ds_set` line is setting the `eye_from_object` matrix used by the pipeline. A draw is then recorded using the one-turbine buffer set. The loop iterates for each pipeline. In contrast, the code for the plain and problem 1 variants record one draw for all the turbines. They must spend more time preparing the buffer set, and it takes longer to send the buffer set from the CPU to the GPU. When this problem is solved correctly the buffer set is much smaller. Though it is used many times, it is only sent from the CPU to the GPU once (or each time it changes). There is a downside to this method: there is the overhead of setting up the draw, including changing the transform. That will be fixed in later assignments (using instances and storage arrays)

If this is solved correctly then the appearance using the problem 1 and 2 variants should be the same.

## LSU EE 4702-1 Homework 4 Due: 6 November 2022, 23:59

**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 look like the screenshot below on left. This is the gear from the midterm exam, which itself is similar to the paddle wheel from Homework 2 (though done in Vulkan). The screenshot on the right is taken from a correctly assignment. They look the same, but there is an important difference: the right image was rendered using fewer vertices.





### 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 `hw04.png` or `hw04-debug.png`.

Initially the arrow keys, `PageUp`, and `PageDown` can be used to move around the scene. Using the `Shift` modifier 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. Press `l` to move the light around and `e` to move the eye (which is what the arrow keys do when the program starts).

### Assignment-Specific User Interface

The scene can be reset by pressing `1`, `2`, and `3`. When `1` is pressed the number of turbines and their positions is randomly chosen. When `2` is pressed the number and positions are the same each time. When `3` is pressed the view will switch to the view used in taking the screenshots in this assignment. Pressing `p` will stop and start the turbines from spinning.

The gear can be rendered using two different shaders, *Base*, and *HW04*. Pressing `V` (upper-case) toggles between the two. The shaders' pipelines can be set to group vertices based on two topologies, *triangle lists*, and *triangle strips*. The `v` (lower-case) toggles between the two. The currently selected shader and topology are shown in the green text on the line starting with `Topology`.

The number of blades in the gear (okay, I should have called them teeth, or maybe cogs) can be changed by changing `VAR Num Blades`.

### Code Generation and Debug Support

The compiler generates two versions of the code, `hw04` and `hw04-debug`. Use `hw04` to measure performance, but use `hw04-debug` for debugging. The `hw04-debug` version is compiled with optimization turned off. You are strongly encouraged to run `hw04-debug` under the GNU debugger, `gdb`. See the material under "Running and Debugging the Assignment" on the course procedures page.

To help you debug your code and experiment in one way or another, the user interface lets you change variables. In this assignment these include the variables `opt_tryout1` and `opt_tryout2`, which are available both in CPU code and in shader code. 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` and `Y` toggle the value of these variables. Their values are shown in the green text.

### 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 rasterization. (Other assignments will use ray tracing.) 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.

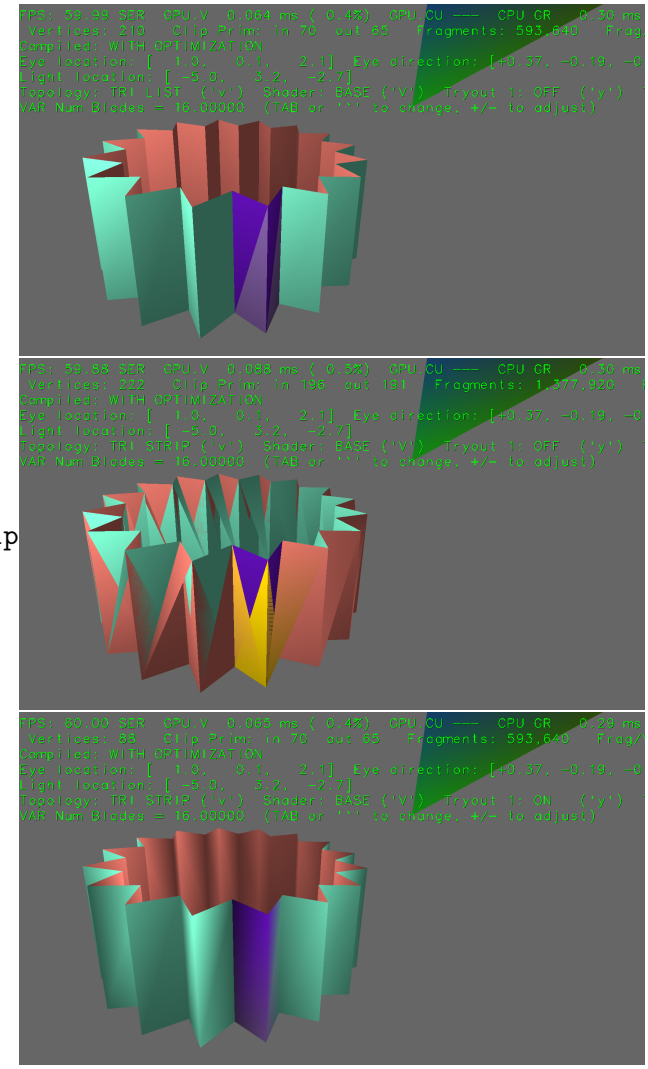
The `+` and `-` keys can be used to change the value of certain variables. These variables specify things such as the light intensity, sphere radius, and variables that will be needed for this assignment. The variable currently affected by the `+` and `-` keys is shown in the bottom line of green text. 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.

### Code Overview

Routine `World::render` is called each time the frame buffer is to be prepared. It is similar to other render routines used in class, the parts relevant to this assignment will be discussed here.

Some notes on how to work with the `VPipeline`, `VVertex_Buffer_Set`, and the buffer types can be found in this under-construction Vulkan course library note set.

**Problem 1:** The code in `render` prepares and uses two buffer sets, `bset_gear_list` and `bset_gear_strip`. In the unmodified code both of these contain vertices that correctly describe the gear for a triangle list topology. Buffer set `bset_gear_list` is used when the rendering is set to use the triangle list topology, which is correct. (The topologies are toggled by pressing `v` [lower-case].) Buffer set `bset_gear_strip` is used when the topology is set to a triangle strip. Since in the unmodified code the vertices in `bset_gear_strip` are arranged for a triangle list using it in this way is wrong, the middle screenshot shows the result.



Modify the code preparing `bset_gear_strip` so that the vertices are in triangle-strip order. Search for "Problem 1" in `hw04.cc` to find the code to modify. The bottom screenshot shows the gear with this problem correctly solved, but when using the base shader. Notice that the number of vertices in the bottom image is about one third the number in the other two. (In the next problem the blurriness will be fixed.)

**Problem 2:** When Problem 1 is correctly solved (and before this one is solved) the gear will appear blurry. This is because the base shader computes the lighted color at a vertex using the color and normal of the vertex. That does not work for a triangle strip because one vertex can be used for three triangles, not all of which have the same normal and color.

Modify the shaders in file `hw04-shdr.cc` to fix this problem plus the code preparing pipeline `pipe_gear_hw04` in `hw04.cc`. These are the shaders labeled `HW04` in the green text.

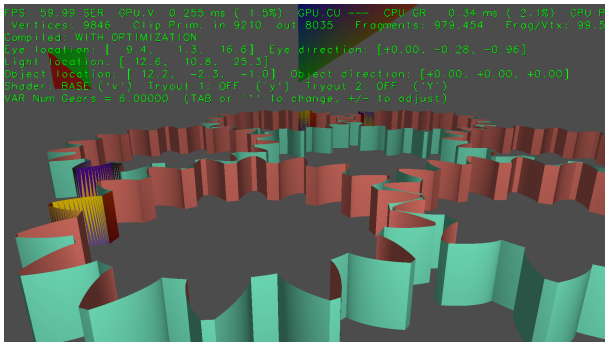
To fix the problem try the following: First, don't even bother sending normals into the rendering pipeline. Instead, have the rendering pipeline compute the normals. This should be done in the geometry shader. To fix the color problem, use a color associated with a single vertex when computing the lighted color. This too should be done in the geometry shader. And, if you've done those an still feeling like coding, try removing the integer pipeline input used for the color index, and instead pack the color index into the $w$ component of the vertex coordinate. That will reduce the amount of data per vertex to just four floats.

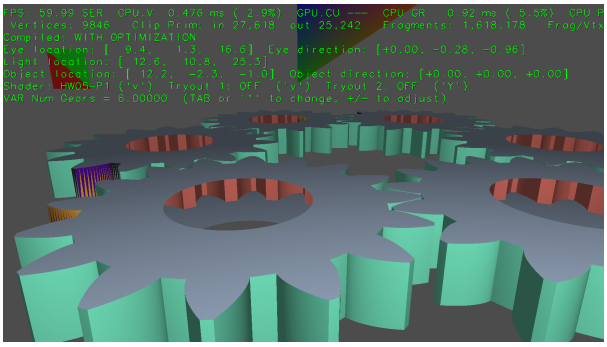
When this problem is correctly solved the result should look like the right-hand screenshot on the first page of this assignment. (The number of vertices should be only slightly larger than the number of primitives.)

See the Open GL Shading Language 4.60 documentation, linked to the course References page, for a list of available functions, such as for computing a cross product, and other details on the language.

## LSU EE 4702-1 Homework 5 Due: 28 Nov 2022, 23:59 CST

**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 look like the screenshot below on left. This shows involute gears, which are much better than the crude gears in the midterm exam. The screenshot on the right is taken from a correct solution to Problem 1.





### 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 `hw05.png` or `hw05-debug.png`.

Initially the arrow keys, `PageUp`, and `PageDown` can be used to move around the scene. Using the `Shift` modifier 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. Press `l` to move the light around, press `e` to move the eye (which is what the arrow keys do when the program starts), and press `b` to move the gears around.

### Assignment-Specific User Interface

The scene can be reset by pressing `1`. Pressing `2` and `3` set up scenes, but those two haven't been tested. Pressing `p` will stop and start the gears from spinning. The gears can be rendered using three different shaders (each using their own pipeline), *Base*, *HW05-P1*, and *HW05-P2*. Pressing `v` (upper or lower-case) cycles through the shaders.

The number of teeth in the gears can be changed by adjusting `VAR Num Teeth`, and the number of gears can be changed by adjusting `Num Gears`. When Problem 2 is correctly solved rendering a large number of gears will be more efficient.

### Code Generation and Debug Support

The compiler generates two versions of the code, `hw05` and `hw05-debug`. Use `hw05` to measure performance, but use `hw05-debug` for debugging. The `hw05-debug` version is compiled with optimization turned off. You are strongly encouraged to run `hw05-debug` under the GNU debugger, `gdb`. See the material under "Running and Debugging the Assignment" on the course procedures page.

To help you debug your code and experiment in one way or another, the user interface lets you change variables. In this assignment these include the Boolean variables `opt_tryout1` and `opt_tryout2`, and floating-point variable `opt_tryoutf`, which are available both in CPU code and in shader code. 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` and `Y` toggle the value of the Boolean variables. Their values are shown in the green text. The `VAR` mechanism can be used to change `opt_tryoutf`.

## 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 rasterization. (Other assignments will use ray tracing.) 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.

The `+` and `-` keys can be used to change the value of certain variables. These variables specify things such as the light intensity, sphere radius, 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.

## Code Overview

Routine `World::render` is called each time the frame buffer is to be prepared. It is similar to other render routines used in class, the parts relevant to this assignment will be discussed here.

All shaders use buffer set `bset_gear_strip`, in which vertices are arranged for a triangle strip. The gear shape consists of cylindrical portions called *lands*, and also includes *faces*, the part of a tooth that makes contact with a tooth on another gear. The shape of a face is an *involute* of a circle. With this face shape the point of contact between two gears follows a straight line.

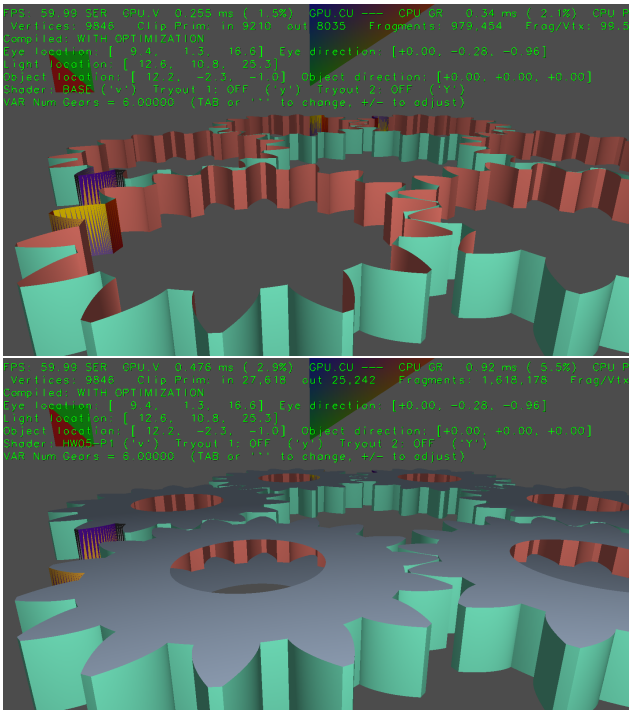
Recall that in Homework 4 the surface normal was eliminated from the buffer set so as to reduce the amount of data sent. Here the surface normal is back and for a reason. The normal is for the approximated surface (cylinder or involute) not of a triangle.

The buffer set holds the coordinates of a gear in a local coordinate space. In that space the gear axis is aligned with the $z$, with the center of the gear at $x = 0, y = 0$. Array `buf_gear_xforms` holds a transformation matrix for each gear that moves it to the desired position. The gears are arranged in a circle and are positioned so that the teeth mesh. Gear rotation is done using matrix `rot`, computed in the render routine. For the Base and Problem 1 shaders, `rot` is and should be used to update the set of transformation matrices sent to the shaders. (That's done in the `ds_set` member function, see Problem 2.) For the Problem 2 shader it is used to update the transformation matrices, but shouldn't be.

In this assignment there is no need to change the code that prepares the buffer set.

Some notes on how to work with the `VPipeline`, `VVertex_Buffer_Set`, and the buffer types can be found in this under-construction Vulkan course library note set.

**Problem 1:** The gears in the upper screenshot consists only of *faces* and *lands*, but not the sides of the gears. The sides are visible in the lower screenshot.



Modify the shaders in the Problem 1 area of `hw05-shdr.cc` and code in `hw05.cc` so that the pipeline `pipe_gear_prob1` renders the sides of the gears, with the radius of the hole set to `gear_info.radius_inner * 0.5`. The color should be set to color set 6 (light steel blue).

It should come as no surprise that the sides will be emitted in the geometry shader. Most of the solution should be in the shader code. Do not change the vertex shader inputs. It will be necessary send over some data as uniforms, `Uni_Misc` can be used.

It might be helpful to look at built-in variable `gl_VertexIndex`, available in the vertex shader. For the first vertex its value is zero, for the second its value is one, and so on. Because of the way the gear is constructed, odd values of `gl_VertexIndex` will be on one side of the gear (perhaps facing up), and even values will be on the other.

- Do not change the vertex shader inputs.
- Set the radius of the hole to `gear_info.radius_inner * 0.5`.
- Set the color to index 6.
- Be sure to correctly compute the normal.

**Problem 2:** As pointed out in an earlier assignment, performing multiple draws can be inefficient, this is the case for the Problem 1 pipeline because a draw is performed for each element in `buf_gears_xforms`:

```
case HW_Shader_Prob1:
  for ( auto m: buf_gears_xforms )
    {
      pipe_gear_prob1.ds_set( transform * m * rot );
      pipe_gear_prob1.record_draw(cb, bset_gear_strip );
    }
  break;

case HW_Shader_Prob2:
  pipe_gear_prob2.ds_set( transform * buf_gears_xforms[0] * rot ); // This is wrong.
  pipe_gear_prob2.record_draw_instanced( cb, bset_gear_strip, opt_n_gears );
  break;
```

The Problem 2 pipeline is rendered using an instanced draw. In an instanced draw the vertices (the ones in `bset_gear_strip` in the example above) are sent to the pipeline multiple times. In the example above they are sent `opt_n_gears` times (once for each gear). In the vertex shader the value of built-in variable `gl_InstanceIndex` is set to 0 for the first gear (or instance), 1 for the second gear, and so on.

```
void vs_main_prob2() {
  int inst = gl_InstanceIndex;
  vertex_c = ut.clip_from_object * in_vertex_o;
```

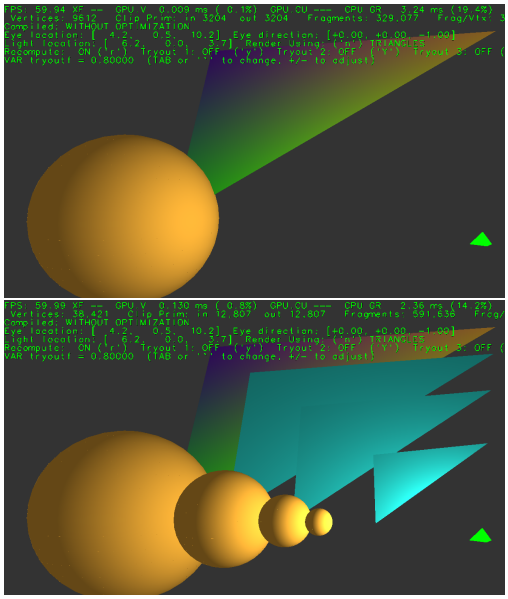
For your solving convenience, the contents of `buf_gears_xforms` has been sent to the GPU and made available as a storage buffer named `gears_xform`. That storage buffer is only sent over when the position of the gears changes. The storage buffer **is not sent over** for gear rotation and it should not be sent over for rotation, so don't change that.

The unmodified code will seem to show just one gear when using the Problem 2 shader. It is actually showing `opt_n_gears` gears, but they are all in the same position, which is not what we want. Modify the code in the Problem 2 section of `hw05-shdr.cc` and in `hw05.cc` so that the instanced draw works correctly (showing the gears in the same place as the other shaders) and is reasonable efficient. There is no need to include the sides that were completed for Problem 1.

- Modify the code so that the instanced draw works.
- The gears should turn.
- Do not send over `buf_gears_xforms` just to update rotation.
- If solved correctly CPU time should go down, especially with a large number of gears.

# 6 Fall 2021

# LSU EE 4702-1 Homework 1 Due: 8 October 2021



**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 the sphere from the `demo-05-more-vulkan.cc` code, see the upper screenshot on the right.

## User Interface

Initially the arrow keys, PageUp, and PageDown can be used to move around the scene, but they can be set to move the sphere or the light. Press `l` to move the light around, `b` to move the sphere (ball), and `e` to move the eye (which is what the arrow keys do when the program starts). Pressing `Shift` and an arrow key will move by a larger distance (than if `Shift` were not pressed) and pressing `Ctrl` and an arrow key will move by a smaller distance. The eye can be aimed up and down by pressing `Home` and `End` and the eye can be rotated by pressing `Insert` and `Delete`.

Press `Ctrl=` to increase the size of the green text and `Ctrl-` to decrease the size.

## 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.

GPU.V 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 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 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`.

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.

The `+` and `-` keys can be used to change the value of certain variables. These variables specify things such as the light intensity, sphere radius, 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.

## Code Generation and Debug Support

The compiler generates two versions of the code, `hw01` and `hw01-debug`. Use `hw01` to measure performance, but use `hw01-debug` for debugging. The `hw01-debug` version is compiled with optimization turned off and with Vulkan validation turned on. You are strongly encouraged to run `hw01-debug` under the GNU debugger, `gdb`. See the material under "Running and Debugging the Assignment" on the course procedures page.

When Vulkan validation is on (which is currently in both debug and normal versions) helpful error and warning messages will be printed about misuse or abuse of the Vulkan API. These will appear on the terminal window (which might be a gdb session) from which `hw01-debug` was started. Currently, the code is set so that all warnings are fatal (except for a select few, which won't generate messages).

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.

## Overview of the Vulkan Helper Classes

The Vulkan Helper Classes are a set of classes and other C++ code intended to make Vulkan easier to learn and use for students in LSU EE 4702 (GPU Programming). Its development started in the summer of 2021 and it continues to evolve rapidly, so this description may well be obsolete after the 2021 Homework 1 assignment deadline.

## Using the Vulkan Helper Code in this Assignment

For this assignment it is important to understand how to use the `VVertex_Buffer_Set` objects to specify vertices for a draw, and how to specify transformation matrices using `VTransform`. These classes are described below with more than enough detail to solve this assignment. In addition to the descriptions in this handout, look for the comments in the `render` routine.

For this discussion one needs some understanding of what a rendering pipeline is and of transformations and coordinate spaces used in graphics.

The code in `hw01.cc` has a routine named `render` in which most of the solution is to be placed. The `render` routine is called each time the window needs to be updated, which on many computers will be 60 times a second.

The render routine sets up two rendering pipelines, one for the triangle, `pipe_lonely`, and one for the sphere, `pipe_sphere`, and prepares a set of vertices for each, `bset_lonely` and `bset_sphere`. It also prepares transformation matrices so that the triangles formed by the vertices in the two vertex sets are placed in the desired place in the window. Most importantly it records draws using each pipeline with its vertex set. See the following sections to understand how that is done.

In this assignment it will be necessary to add new vertices forming new triangles (Problem 1) and to use transformation matrices so that additional spheres can be rendered by re-using the existing sphere coordinates.

## Using Vertex Buffer Sets

The vertex buffer set class, `VVertex_Buffer_Set`, manages the vertices that will be streamed into a rendering pipeline. Objects of this class by convention have names that start `bset`. In the Homework 1 assignment there are two such objects `bset_lonely` and `bset_sphere`. Here, *vertex* refers to a collection of attributes associated with a part of a geometric object. Typical attributes are coordinate, normal, and color.

The first step in using a vertex buffer set object is to reset it. A vertex buffer set object is reset by calling the `reset` member function with a pipeline as an argument, for example `bset_sphere.reset(pipe_sphere);`. This will clear any vertex attributes that the object is carrying and will also set the object to expect only those attributes needed by the pipeline given in the argument. For example, if `pipe_sphere` in the example above does not use texture coordinates (and it doesn't in Homework 1) then `bset_sphere` will not expect texture coordinates and will exit with an error if an attempt is made to insert texture coordinates.

After being reset a vertex buffer set is ready for new vertex attributes to be inserted. This is done using the insertion operator, `<<`. Each attribute has a different type and so there is no need to specify which attribute is being inserted. For example, in the code below ...

```
bset_lonely.reset(pipe_lonely);
pCoor p0 = { 0, 0, 0 };
pNorm tri_norm = cross( p0, p1, p2 );
pColor color_tri( .470, .553, .965 ); // Red, Green, Blue
bset_lonely << color_tri << tri_norm << p0;
```

... three different attributes are being appended to `bset_lonely`, a vertex coordinate (`p0`), a vertex normal (`tri_norm`), and a color (`color_tri`). After execution of the code above `bset_lonely` holds one vertex and the vertex has three attributes. The three attributes could have been inserted in any order (relative to each other) and using one to three statements. For example, the following are valid ways of inserting two more vertices:

```
pCoor p1 = { 9, 6, -4 };
pCoor p2 = { 0, 5, -3 };
bset_lonely << p1 << color_lsu_spirit_gold << tri_norm;
bset_lonely << color_lsu_spirit_purple;
bset_lonely << tri_norm;
bset_lonely << p2;
```

After the execution of the second code fragment above `bset_lonely` holds three vertices, each with three attributes.

Currently a vertex buffer set object can manage six attributes: vertex coordinates (type `pCoor` in member `pos`), colors (type `pColor` in member `color`), normals (type `pVect4` in member `normal`), texture coordinates (type `pTCoor` in member `tcoord`), scalar integers (type `int` in member `int1`), and 2-element integer vectors (type `ivec2` in member `int2`). The first four are used by the default Vulkan Helper shaders and correspond to the vertex attributes defined by OpenGL. The `int1` and `int2` attributes can only be used with custom shaders, and those shaders will know how to interpret those integers. (The member names above may change by Homework 2, for example, `pos` may be changed to `coor` and `tcoord` may change to `tcoor`.)

The insertion operation appends the attribute to an array on the host (a C++ std vector). To copy the attributes to the device (GPU) use the member function `to_dev()`, for example, `bset_lonely.to_dev();`. The vertex buffer set object does not keep track of changes to the attribute arrays, and will copy the arrays to the device even if no changes have been made since the last copy. That is, `bset_lonely.to_dev(); bset_lonely.to_dev();` will copy the attributes twice. This behavior may change by Homework 2.

The `to_dev()` member function copies data immediately. That is in contrast to other code in the `render` routine that records commands to a command buffer for later execution.

The purpose of a vertex buffer object is to provide inputs to a pipeline. A `VPipeline` object's `record_draw` member function takes a vertex buffer set object as a parameter and records a draw command in which the buffer set is streamed into the pipeline. For example, `pipe_lonely.record_draw(cb, bset_lonely);` records a draw command (actually several setup commands followed by a draw command) in which pipeline `pipe_lonely` is sent vertex attributes from `bset_lonely`. The commands

recorded into `cb` will be executed after `render` returns, so `bset_lonely` should not be changed until the next call to `render`.

The sequence of commands `bset_lonely.reset(pipe_lonely);`, `bset_lonely << color_green << tri_norm << p0;`, `bset_lonely.to_dev();` only need to be executed when the vertices change. There is no need to re-execute them on later calls to `render` if the exact same vertex attributes would be inserted. In the homework assignment package they *are* re-executed every time `render` is called, which is wasteful, but something that is to be fixed (for the sphere) in the assignment.

The assignment code also calls member function `destroy()`. This frees resources consumed by the vertex buffer set object. This should be called when the object is no longer needed, especially in long-running programs in which the GPU can become stuffed with no-longer-needed resources.

## Using Transformation Matrices

Transformation matrices are used by the rendering pipeline's shaders to transform vertex coordinates from one space to another. The shaders in this assignment are provided as part of the Vulkan Helper code, and so the transformation matrices must be in the form that they expect. The `VTransform` class holds a set of transformation matrices and allows them to be conveniently changed.

In a correct solution to Problem 2 the transformation matrix used in draws of the sphere will have to be changed. That is done using the `VTransform::global_from_local_set_for` member function, so keep reading.

The `VTransform` class recognizes four coordinate spaces, *local*, *global*, *eye*, and *clip*. The *Local* space coordinate space is one that is convenient for a particular object. For example, the sphere is constructed in a local coordinate space in which the center is at the origin and the radius is one. The *Global* space is one that is shared by all parts of a scene. In this assignment the triangle coordinates and eye coordinates are specified in global space. In the *eye* coordinate space the eye is at the origin and is looking in the $-z$ direction. Eye space historically was used for lighting calculations, and is the space in which perspective projection transforms are defined. In *clip* space the coordinates of all visible items are in a cube with vertices at $\pm 1$.

*Object space* is another term used when describing rendering pipelines. Object space is defined as the coordinate space of the vertices used as input to a rendering pipeline. That can be either local or global space using the definition above.

The `VTransform` keeps track of three transformations, *eye from global*, *clip from eye*, and *global from local*. The eye from global transform is chosen based on the location of the eye in global space and the direction it is looking. It is set using the `eye_from_global_set` member function. The clip from eye transform is chosen based on the position and size of the user's monitor in eye space (to achieve a perspective projection), it is set using the `clip_from_eye_set` member function. These should be set once during a render pass. The code below sets these two transformations using the UI-modifiable `eye_location` and `eye_direction` variables and the aspect ratio of the window Vulkan is rendering into (obtained ultimately from the window manager):

```
transform.eye_from_global_set
  ( pMatrix_Rotation(eye_direction,pVect(0,0,-1))
    * pMatrix_Translate(-eye_location) );

const int win_width = vh.get_width(), win_height = vh.get_height();
const float aspect = float(win_width) / win_height;

transform.clip_from_eye_set
  ( pMatrix_Frustum(-.8,  .8,    -.8/aspect,  .8/aspect, 1,    5000) );
       // Frustum: left, right, bottom,      top,       near, far
```

(In the actual assignment code the clip from eye transform also flips the $y$ axis direction. In later assignments the $y$ axis will point up without such a trick.)

Before recording a draw command, a pipeline needs to be given a transform. (That means, it needs to be given a `vk::Buffer` handle of a uniform object that will contain the transform matrices.) For pipelines using a global transform that is done using the `VTransform::use_for` member function called with a pipeline as an argument, for example, `transform.use_for(pipe_lonely);`. This only needs to be done once per rendering pass, since the global transform does not change. (It could even be done once period, if `VTransform` preserves the buffer, which it might one day do.)

The local space can be changed multiple times in a rendering pass. In the unmodified assignment it is changed once, for the sphere. The solution to problem 2 would have the local space modified multiple times.

To specify that a pipeline is to use a local transformation matrix call the `VTransform::global_from_local_set_for(XFORM,PIPE)` member function. The first argument is the transformation matrix, the second is a `VPipeline` object. For example,

```
pMatrix glo_from_loc =
  pMatrix_Translate( sphere_location ) * pMatrix_Scale(sphere_radius);
transform.global_from_local_set_for( glo_from_loc, pipe_sphere );
```

scales the sphere so that it is radius `sphere_radius` and moves it to location `sphere_location`. Each time `global_from_local_set_for` is called (in the first render) it will create a new Vulkan `vk::Buffer` to hold the transformation. So if it is called 10 times in a render pass it will create 10 buffers, but those buffers will be re-used in the next pass. Calling it thousands of times will likely exhaust storage, meaning that other methods of managing local space are needed.

## Using Pipelines

The pipeline class, `VPipeline`, manages Vulkan rendering pipeline objects and information needed to use them to draw. In this assignment there will be no need to modify the pipelines themselves, but that will change in future assignments. This brief description is provided for background and to provide a better overall understanding of what the code is doing.

A Vulkan pipeline object consists of a sequence of fixed-function and programmable stages along with information on the resources to be accessed including vertex shader input types, uniform types, and the type of frame buffer it expects to access. To use a pipeline object the pipeline and the data it access must all be *bound* (identified, sort of like specifying call arguments in advance) and then a Vulkan draw command is recorded. The argument to the draw itself is little more than the number of vertices. The vertices come from the bound vertex buffers. A `VPipeline` object records a draw using the `record_draw` member function. This function automatically binds the vertex buffers, pipeline, and other resources. In this assignment see, for example, `pipe_sphere.record_draw(cb, bset_sphere);`.

A `VPipeline` object can be set to use user-provided shaders or it can use its own shaders, the latter of which is done by the Homework 1 code. A `VPipeline` object must be created before use. Pipeline creation starts with a call to the `init` member function, followed by calls to functions setting options, specifying shaders, specifying data to use, and so on, creation is completed with a call to the `create` member function. Most of these member functions return a reference to the object, so they can be conveniently stringed together, for example:

```
pipe_sphere
  .init( vh.qs )
  .use_uni_light( uni_light )
  .lighting_on()
  .topology_set( prim_want )
  .create( vh.qs.render_pass );
```

A pipeline object when cast to a Boolean (as in `if ( !pipe_sphere )`) will return false until it is created. Options such as the topology (how to group vertices into primitives) can only be set before the pipeline is created. The only way to change them is to call `destroy` and then re-create the pipeline. This is time consuming and so should not be done frequently.

The code for this assignment is based on the `demo-05-more-vulkan` code used in class which shows how to render a sphere and a triangle. Routine `render()` renders the sphere and triangle—the lonely triangle.

**Problem 1:** Modify the code in `render` so that the lonely triangle has `opt_n_objects` friends, where `opt_n_objects` is the name of a user-interface modifiable variable. Those new triangles should be placed between the lonely triangle and the light so that each vertex of a new triangle is on a line between an original vertex and the light. See the second screenshot on the first page of the assignment. The color of the new triangles should be different than the lonely triangle (or any of the lonely triangle's vertices).

The coordinates of the lonely triangle vertices are in objects `p0`, `p1`, and `p2` and for your convenience have been placed in vector `tri_vtx`. The coordinates of the light are in object `light_location`.

Use `bset_lonely` for the shader inputs; it should hold shader inputs for the lonely triangle (as it does before any changes are made) along with the shader inputs for the new triangles. For this problem there is no need to change `pipe_lonely` or how it is used.

- Compute the coordinates of the new triangles and insert them into `bset_lonely`.
- Compute the normal of the new triangle and insert that into `bset_lonely`.
- Also insert the new colors.
- The number of coordinates inserted into `bset_lonely` must match the number of colors and normals.

To help debug your code it might be helpful (or at least mildly interesting) to move the light (using the UI) and to change the value of `opt_n_objects`.

**Problem 2:** Modify the code so that `opt_n_objects` new spheres are rendered. The centers of the new spheres should be on a line between the center of the original sphere and the light. The size of the new spheres should be smaller than the original (though the exact size is not important) and the spheres should be positioned so that they exactly touch but do not intersect. See the second screenshot on the first page of this assignment.

Unlike the triangles, each new sphere should be rendered by reusing the shader inputs used to render the original, `bset_sphere`, and by reusing the rendering pipeline `pipe_sphere`. To place the a new sphere specify a new local-to-global transformation matrix and record a new pipeline draw. See the use of `transform.global_from_local`.

**Problem 3:** The code in `demo-05-more-vulkan` and this assignment is actually quite wasteful. In fact, more experienced programmers might cringe just looking at it. I am referring in particular to the code computing the sphere coordinates. The problem is that those coordinates are computed each frame even when the sphere does not change. (The sphere can change if the value of `slices` is changed using the user interface.)

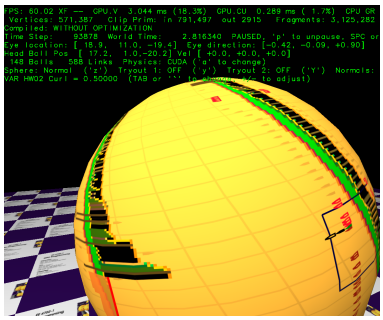
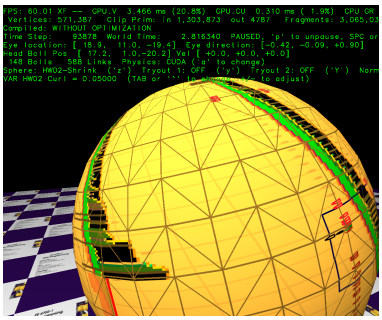
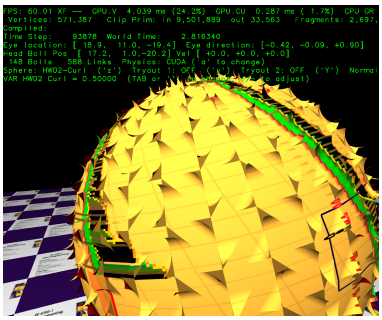
(*a*) Modify the code in `render` and elsewhere so that when `opt_recompute` is false the sphere is only recomputed when necessary. Keep in mind that `bset_sphere` will hold its contents until reset, and the data sent to the device using `bset_sphere.to_dev()` will stay there until removed, so the data sent for rendering one frame can be used to render the next frame.

(*b*) Investigate the impact on performance by switching `opt_recompute` on and off and noting its effect on performance. Note that the time needed to execute `render` is shown to the right of `GPU GR` in the green text.

- Indicate that GPU being used. (That can be found in the program output to the terminal near the top. Look for a line starting `Props driver`.
- Does optimized code make a big difference?
- In optimized code, who many slices does it take to degrade frame rate with and without re-computation?

**LSU EE 4702-1** **Homework 2** **Due: 9 November 2021**

**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 scene from the Vulkan links code (in `projbase/v-links`), the one showing a vaguely tree-like form, the *silly tree*, constructed from flexible links and balls. In the original links code the balls are ordinary spheres, which look brand-new no matter how long you run the program. See the screenshot below on the left. In this assignment the spheres are to show effects of aging. Imagine that the texture were applied to the sphere by applying triangular stickers. As time passes two kinds of degradation are possible. For Problem 1 that degradation is shrinkage. The triangular stickers should shrink, reveling the untextured and slightly darker sphere underneath. See the screenshot below in the center. For Problem 2 the triangles should start pealing off at the corners. See the screenshot below on the right.

Non-Assignment-Specific User Interface

Press digits `1` through `4` to initialize different scenes, the program starts with scene 1. Scene 1 starts with the balls arranged in the tree-like form.

Press `p` to pause the simulation.

Press `Ctrl=` to increase the size of the green text and `Ctrl-` to decrease the size. Initially the arrow keys, `PageUp`, and `PageDown` can be used to move around the scene. Using the `Shift` modifier 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. Press `l` to move the light around and `e` to move the eye (which is what the arrow keys do when the program starts).

The `+` and `-` keys can be used to change the value of certain variables. These variables control things like light intensity and options needed for this assignment. The variable currently affected by the `+` and `-` keys is shown in the bottom line of green text next to VAR. Pressing `Tab` cycles forward through the different variables.

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

Assignment-Specific User Interface

The sphere can be rendered by four different shaders, `Normal`, `HW02-Shrink`, `HW02-Curl`, and `True`. The shader being used is shown to the right of `Sphere` in the green text. To cycle through the shaders (except for `True`) press `z`. To toggle between the `True` shader and some other press `Z`.

The amount of curling or shrinkage is specified by variable `opt_hw02_curl`. (This variable should affect curling and shrinkage.) This can be modified using the UI, look for that variable name to the right of VAR.

Pressing `n` toggles between computing sphere lighting based on triangle normals, `TRI`, and sphere normals, `SPHERE`. When solving the problems it might help to rendering using triangle

normals so that you can see the triangle boundaries. It might also be helpful to adjust light intensity to make effects more visible.

### Graphics and Performance Investigation Options

The user interface can be used to toggle various rendering options and for generating a screenshot.

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.

Pressing `F12` will write a screenshot to file with a name like `hw02-00.png`. The number at the end will start at `00` and will be incremented for each screenshot *within a run*. When the code is run again the count starts at zero, so be sure to rename files that you want to keep.

The rendering of shadows is toggled by `o` and the rendering of reflections it toggled by `r`. Their state is shown in the green text next to Effect:. Pressing `n` will toggle how surface normals are computed for tessellated spheres, the possibilities are to use the triangle normal or the sphere normal. The use of the triangle normals makes it easier to see the triangles from which the sphere was tessellated.

### 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.

GPU.V 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 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 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`.

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.

The `+` and `-` keys can be used to change the value of certain variables. These variables specify things such as the light intensity, sphere radius, 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.

### Code Generation and Debug Support

The compiler generates two versions of the code, `hw02` and `hw02-debug`. Use `hw02` to measure performance, but use `hw02-debug` for debugging. The `hw02-debug` version is compiled with optimization turned off and with Vulkan validation turned on. 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.

When Vulkan validation is on (which is currently in both debug and normal versions) helpful error and warning messages will be printed about misuse or abuse of the Vulkan API. These will

appear on the terminal window (which might be a gdb session) from which `hw02-debug` was started. Currently, the code is set so that all warnings are fatal (except for a select few, which won't generate messages).

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.

### Instanced Draws, and the Assignment Sphere Instance Shaders

The code in the assignment package can render the sphere two ways, using the true sphere shader, which is not a part of this assignment (but still can be run), and the instanced shaders which are.

The sphere rendering that was described in `demo-09-shader.cc` rendered a single sphere. Recall that an input buffer set was prepared holding coordinates of the surface of the sphere in an order appropriate for triangle strips. The coordinates are for a sphere with its center at the origin and of radius one. A transformation matrix was used to scale and move the sphere to wherever we like. In homework 1 multiple spheres are rendered by recording multiple draws each with the same sphere buffer set but with different transformation matrices.

An instanced draw does something like that using a single draw command. The Vulkan draw command, `vkCmdDraw` and the equivalent Vulkan hpp draw member function `CommandBuffer::draw` (See Vulkan Chapter 21.3) take an `instanceCount` argument that specifies the number of instances to draw. So far in this class `instanceCount` has been set to 1, but when using our Sphere Instanced shaders the `instanceCount` will be set to the number of spheres we want to render. The code preparing buffer sets and recording the instanced draw is not part of this assignment. But for those who are curious, the Vulkan record draw calls are made in `VPipeline::record_draw_general` in file `include/vutil-pipeline.h`. Code using the course helpers might call `pipe.record_draw_instanced(cb, bufset, n_instances)`. For the homework assignment spheres `record_draw_instanced` is called in not-well-organized `Sphere::render_bunch_render` in file `shapes.h`. Remember, the host-side code discussed above is not part of the assignment. (Shader code is, which is discussed below.)

Suppose `instanceCount` is set to 2. Then the buffer set will be streamed into the pipeline inputs twice. The first time OpenGL Shading Language variable `gl_InstanceIndex` will be set to 0, the second time it will be set to 1. It is up to the shader code to use the value of `gl_InstanceIndex` to retrieve, in the case of our spheres, the location, radius, and orientation of the sphere. That is done by the vertex shader `vs_main_instances_sphere` in file `hw02-shdr.cc` (and other files). It reads information from three storage buffers, `sphere_pos_rad`, `sphere_rot`, and `sphere_color`, and uses these to compute clip- and eye-space coordinates, as well as texture coordinates. (Note that it would be possible to replace `sphere_pos_rad` and `sphere_rot` by a single transformation matrix.) The subsequent shader stages are no different than in an un-instanced draw.

In this assignment a geometry shader is also used to apply the special affects described in the problems. In the unmodified assignment the geometry shaders, `gs_main_shrink` and `gs_main_curl`, just omit an ordinary triangle.

**Problem 1:** Modify `gs_main_shrink` (in file `hw02-shdr.cc`) and the fragment shader `fs_main_common` so that they show a shrunken triangle as described below. The amount of shrink should be based on `opt_hw02_curl`. A value of 0 indicates no shrink, a value of 1 indicates that the triangle should shrink to a point. This shader is used when `HW02-Shrink` is shown in the user interface.

The shrunken triangle should carry the entire texture. (That's an easy thing to do.) Underneath the shrunken triangle there should be another triangle at the original size, but with a darker color and without the texture applied.

Here are some things that need to be done:

- Find the coordinates of the shrunken triangle. Do so so that the center of both triangles is the same, except that the shrunken triangle is slightly further from the sphere center.
- Find a way of rendering the original triangle so that it does not show texture and has a darker color. Changing the color is easy. Suppress the color by sending something down the rendering pipeline as a signal to the fragment shader. Don't just set the texture coordinates to zero.
- Don't forget to increase the number of output vertices in the geometry shader declaration.
- Note that the fragment shader uses red for the back face of primitives.

**Problem 2:** Modify `gs_main_curl` (in file `hw02-shdr.cc`) and the fragment shader `fs_main_common` so that they show some triangles peeling off at one vertex. See the screenshot. Base the fraction of the triangle peeled off on the value of `opt_hw02_curl`.

- The peeling should start at a vertex of your choosing, and the peeled part should be in the shape of a cylinder.
- As with the shrunken triangles, show an untextured, darker-colored triangle underneath.
- The normals on the unpeeled part should be sphere normals (as they are in the unmodified code). The normals on the peeled part should be correctly chosen for the cylinder.
- The cylinder should be placed so the peeling starts smoothly, as in the screenshot. The radius of the cylinder can be chosen to taste.

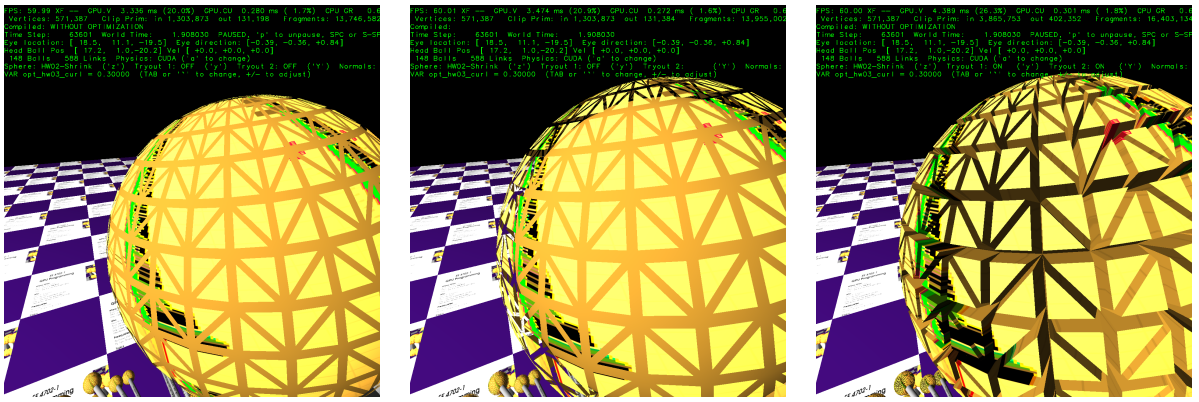
As a resource for solving this problem look at the shader rendering the bump in demo-10. (Those files are in the `gp/vulkan/` subdirectory.) Also check out past homework OpenGL shading language assignments.

To help you get started `gs_main_curl` has code to compute the coordinates of vertices of the peeled triangles. Sorry for spoiling the fun.

- Do not expect shadows to work correctly. In particular, light won't go through the holes interrupting the shadow cast by the part of the sphere that is present.
- Don't forget to increase the number of output vertices in the geometry shader declaration. Note that a single value is used for both. Use the value of `slices` for that. Set the value to the smallest value that would work for both problems.
- Note that the fragment shader uses red for the back face of primitives.

# LSU EE 4702-1 Homework 3 Due: 17 November 2021

**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 scene from the solution to Homework 2, the one showing the silly tree with shrunken triangles. The code includes the Homework 2 solution. In this assignment the shrunken (but not the curled triangles) are to be given depth, as shown in the screenshots below.



## Non-Assignment-Specific User Interface

Press digits `1` through `4` to initialize different scenes, the program starts with scene 1. Scene 1 starts with the balls arranged in the tree-like form.

Press `p` to pause the simulation.

Press `Ctrl=` to increase the size of the green text and `Ctrl-` to decrease the size. Initially the arrow keys, `PageUp`, and `PageDown` can be used to move around the scene. Using the `Shift` modifier 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. Press `l` to move the light around and `e` to move the eye (which is what the arrow keys do when the program starts).

The `+` and `-` keys can be used to change the value of certain variables. These variables control things like light intensity and options needed for this assignment. The variable currently affected by the `+` and `-` keys is shown in the bottom line of green text next to VAR. Pressing `Tab` cycles forward through the different variables.

Look at the comments in the file `hw03.cc` for documentation on other keys.

## Assignment-Specific User Interface

The sphere can be rendered by four different shaders, `Normal`, `HW03-Shrink`, `HW03-Curl`, and `True`. The shader being used is shown to the right of `Sphere` in the green text. To cycle through the shaders (except for `True`) press `z`. To toggle between the `True` shader and some other press `Z`.

The amount of curling or shrinkage is specified by variable `opt_hw03_curl`. (This variable should affect curling and shrinkage.) This can be modified using the UI, look for that variable name to the right of VAR.

Pressing `n` toggles between computing sphere lighting based on triangle normals, `TRI`, and sphere normals, `SPHERE`. When solving the problems it might help to rendering using triangle normals so that you can see the triangle boundaries. It might also be helpful to adjust light intensity to make effects more visible.

## Graphics and Performance Investigation Options

The user interface can be used to toggle various rendering options and for generating a screenshot.

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.

Pressing `F12` will write a screenshot to file with a name like `hw03-00.png`. The number at the end will start at `00` and will be incremented for each screenshot *within a run*. When the code is run again the count starts at zero, so be sure to rename files that you want to keep.

The rendering of shadows is toggled by `o` and the rendering of reflections it toggled by `r`. Their state is shown in the green text next to Effect:. Pressing `n` will toggle how surface normals are computed for tessellated spheres, the possibilities are to use the triangle normal or the sphere normal. The use of the triangle normals makes it easier to see the triangles from which the sphere was tessellated.

## 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.

GPU.V 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 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 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`.

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.

The `+` and `-` keys can be used to change the value of certain variables. These variables specify things such as the light intensity, sphere radius, 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.

## Code Generation and Debug Support

The compiler generates two versions of the code, `hw03` and `hw03-debug`. Use `hw03` to measure performance, but use `hw03-debug` for debugging. The `hw03-debug` version is compiled with optimization turned off and with Vulkan validation turned on. You are strongly encouraged to run `hw03-debug` under the GNU debugger, `gdb`. See the material under "Running and Debugging the Assignment" on the course procedures page.

When Vulkan validation is on (which is currently in both debug and normal versions) helpful error and warning messages will be printed about misuse or abuse of the Vulkan API. These will appear on the terminal window (which might be a gdb session) from which `hw03-debug` was started. Currently, the code is set so that all warnings are fatal (except for a select few, which won't generate messages).

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.

### Instanced Draws, and the Assignment Sphere Instance Shaders

The code in the assignment package can render the sphere two ways, using the true sphere shader, which is not a part of this assignment (but still can be run), and the instanced shaders which are.

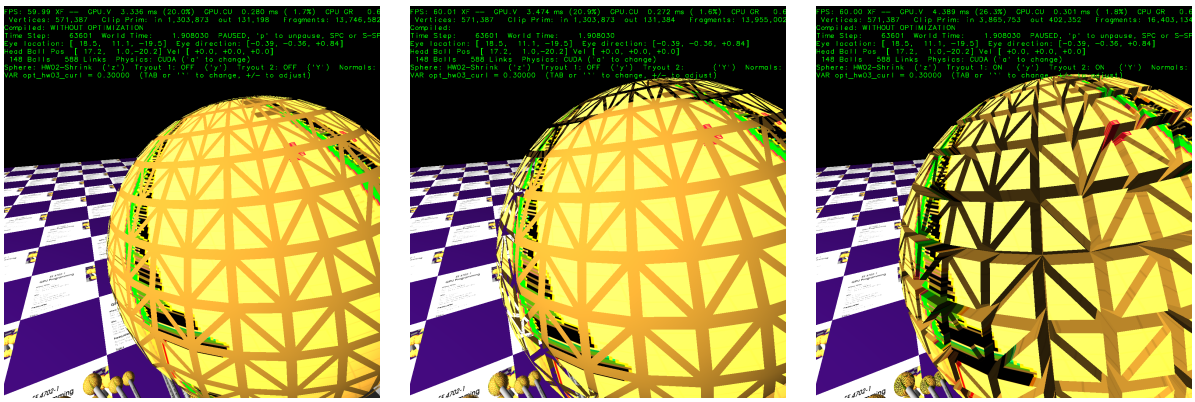
The sphere rendering that was described in `demo-09-shader.cc` rendered a single sphere. Recall that an input buffer set was prepared holding coordinates of the surface of the sphere in an order appropriate for triangle strips. The coordinates are for a sphere with its center at the origin and of radius one. A transformation matrix was used to scale and move the sphere to wherever we like. In homework 1 multiple spheres are rendered by recording multiple draws each with the same sphere buffer set but with different transformation matrices.

An instanced draw does something like that using a single draw command. The Vulkan draw command, `vkCmdDraw` and the equivalent Vulkan hpp draw member function `CommandBuffer::draw` (See Vulkan Chapter 21.3) take an `instanceCount` argument that specifies the number of instances to draw. So far in this class `instanceCount` has been set to 1, but when using our Sphere Instanced shaders the `instanceCount` will be set to the number of spheres we want to render. The code preparing buffer sets and recording the instanced draw is not part of this assignment. But for those who are curious, the Vulkan record draw calls are made in `VPipeline::record_draw_general` in file `include/vutil-pipeline.h`. Code using the course helpers might call `pipe.record_draw_instanced(cb, bufset, n_instances)`. For the homework assignment spheres `record_draw_instanced` is called in not-well-organized `Sphere::render_bunch_render` in file `shapes.h`. Remember, the host-side code discussed above is not part of the assignment. (Shader code is, which is discussed below.)

Suppose `instanceCount` is set to 2. Then the buffer set will be streamed into the pipeline inputs twice. The first time OpenGL Shading Language variable `gl_InstanceIndex` will be set to 0, the second time it will be set to 1. It is up to the shader code to use the value of `gl_InstanceIndex` to retrieve, in the case of our spheres, the location, radius, and orientation of the sphere. That is done by the vertex shader `vs_main_instances_sphere` in file `hw03-shdr.cc` (and other files). It reads information from three storage buffers, `sphere_pos_rad`, `sphere_rot`, and `sphere_color`, and uses these to compute clip- and eye-space coordinates, as well as texture coordinates. (Note that it would be possible to replace `sphere_pos_rad` and `sphere_rot` by a single transformation matrix.) The subsequent shader stages are no different than in an un-instanced draw.

In this assignment a geometry shader is also used to apply the special affects described in the problems. In the unmodified assignment the geometry shaders, `gs_main_shrink` and `gs_main_curl`, just omit an ordinary triangle.

**Problem 1:** In the Homework 2 solution the shrunken triangle was very close to the sphere surface, see the screenshot below on the left. In the unmodified shader code for this assignment the triangle is further away from the sphere, see the center screenshot. Modify `gs_main_shrink` so that the shrunken triangle is connected to the sphere by trapezoidal surfaces colored as described below. The shape made by the shrunken triangle, the original triangle, and the new faces should be a pyramid with the top cut off (a frustum). See the screenshot below on the right.



The new faces are formed using vertices of the original triangle and the shrunken one. (There is no need to render the original triangle.) The new faces should use a darker color and texture coordinates should be applied so that it appears that whatever texture color appears at a point on the edge of the shrunken triangle is used on the path down toward the sphere surface. See the screenshot. (This is an easy thing to do.)

Be sure to supply the correct normals for the faces. The faces are flat, not curved.

Let $c$ denote the value of `opt_hw03_curl`. Remember that $c \in [0, 1]$. As with Homework 2, the size of the shrunken triangle should be proportional to $(1 - c)$, so that for $c = 0$ the shrunken triangle is the same size as the original. For this assignment $c$ should also change the distance of the shrunken triangle from the original sphere surface. Let $d(c)$ denote the distance of the shrunken triangle from the sphere computed in terms of $c$. In your code choose $d(c)$ such that the volume bounded by the shrunken triangle, faces, and original triangle does not change. That is, when $c$ gets larger $d(c)$ should get larger such that the volume does not change.

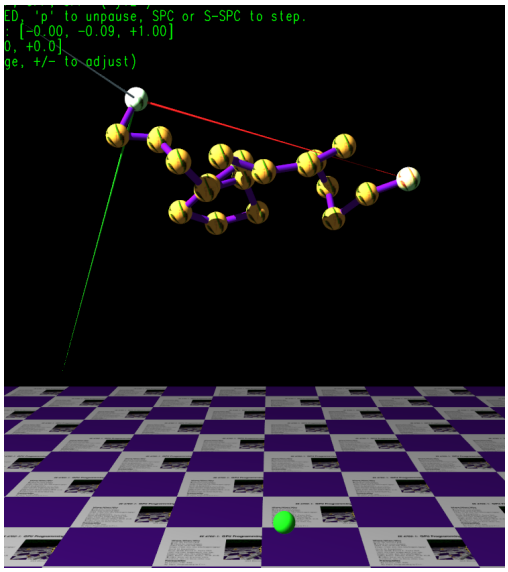
Note that unlike Homework 2, the fragment shader uses the same color for the front and back of primitives.

**Problem 2:** Sorry, no Problem 2. Trying to keep this short. Would have asked that the curl be given thickness too. A potential problem is exceeding the limit on the number of output floats of the geometry shader (1024 on my system). So as part of this unasked question, the instanced draw would have to be re-done such that each instance (sphere) was rendered multiple times. The first time the upper part of the curl would be rendered, the second time the edges. Some other time.

# 7 Fall 2020

# LSU EE 4702-1 Homework 1 Due: 23 September 2020

**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 string of balls compressed between two points, the first and last balls are fixed in place. The balls quickly expand and eventually will form an arc-like shape. The screenshot to the right, taken when the simulation was paused, shows the balls uncompressing, with pointy markers showing local axes, and a green ball showing the approximate place where the lowermost part of the string will be once the balls come to rest.



## User Interface

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 tail 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 `Shift` and an arrow key will move by a larger distance (than if `Shift` were not pressed) and pressing `Ctrl` and an arrow key will move by a smaller distance. The eye can be aimed up and down by pressing `Home` and `End`, and the eye can be rotated by pressing `Insert` and `Delete`.

Pressing `p` will pause and un-pause the simulation. While the simulation is paused time can be advanced by one frame by pressing the space bar, and by one time step by pressing `Shift` and the space bar.

Press digits `1` through `4` to initialize different scenes, the program starts with scene 1. In the unmodified code scenes 1, 2, and 3 are identical: they start with the balls arranged in a line, and separated by springs at less than the relaxed distance. Once this assignment is solved scenes 1, 2, and 3 will behave differently.

Pressing `h` (head) will grab or release one end (to be precise, the ball at one end) and pressing `t` (tail) will grab or release the other end. (Actually, those keys toggle between the `OC_Locked` and `OC_Free` constraint of their respective balls.)

Press `Ctrl=` to increase the size of the green text and `Ctrl-` to decrease the size.

The `+` and `-` keys can be used to change the value of certain variables. These variables specify things such as the light intensity, spring constant, 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` cycles through the different variables.

## Code Generation and Debug Support

The compiler generates two versions of the code, `hw01` and `hw01-debug`. Use `hw01` to measure performance, but use `hw01-debug` for debugging. The `hw01-debug` version is compiled with optimization turned off and with OpenGL error checking turned on. You are strongly encouraged to

run `hw01-debug` under the GNU debugger, `gdb`. See the material under "Running and Debugging the Assignment" on the course procedures page.

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. These variables are intended for debugging and trying things out.

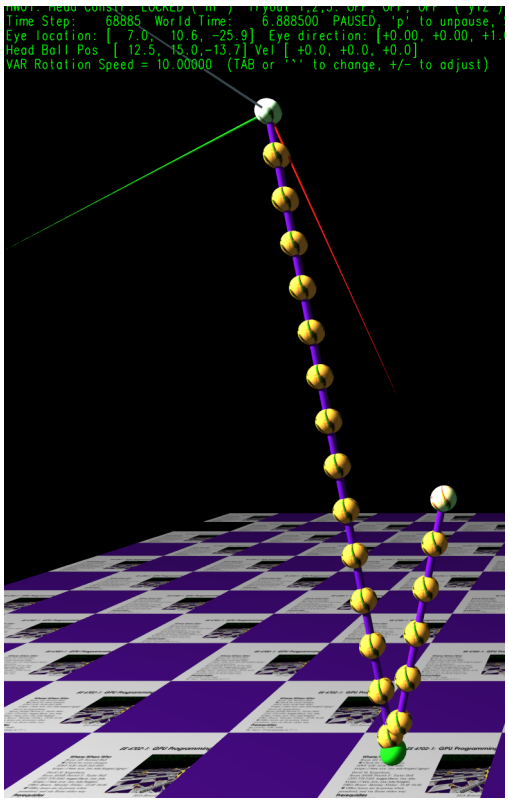
**Problem 1:** Scene 1 starts with the balls arranged in a line and the first and last balls locked in place. The distance between adjacent balls is set to less than the spring's relaxed distance and so the balls are quickly pushed away from each other. They will eventually come to rest in an arc. The goal of this problem is to modify the setup routine so that the initial position of the balls are chosen such that the balls are closer to equilibrium when the scene starts.

In the ideal solution to this problem the balls are arranged to form a *catenary*, the name given to the curve made by an ideal cable with two ends fixed in space. A catenary is close in shape to a parabola.

This problem will be solved with a much cruder approximation: two straight lines. The code in `ball_setup_hw01` chooses a first and last position for the balls, `first_pos` and `last_pos`, denote these $P_f$ and $P_l$. Let $n$ denote the value of `chain_length`, the number of balls, and let $l_r$ denote the value of `distance_relaxed`. The setup routine computes `nadir_pos`, denote that $P_n$.

Call $L = (n-1)l_r$ the (relaxed) length of the chain. Assume for a moment that the chain were an un-stretchable string, with the ends still fixed at $P_f$ and $P_l$. Take a pencil and pull the string in some direction until it is taut. (That is, until the string forms a straight line from $P_f$ to the pencil tip and $P_l$ to the pencil tip.) Draw a dot. Pull in some other direction and draw a dot. Repeat $\infty$ times. Perhaps you've guessed that shape drawn this way is an ellipse. Points $P_f$ and $P_l$ are the foci. Since an ellipse is defined on a plane, we need a normal. The normal used is parallel to the $xz$ plain. The point $P_n$ is on this ellipse, and it is chosen so that it is the point on the ellipse with the minimum $y$ value.

To help understanding these concepts the location of $P_f$, $P_l$, $P_n$, and the axes in the ellipse's local space can be visualized. To do so pause the simulation (using `p`) in scene 1, 2, or 3. (Switch scenes using `1`, `2`, etc.) The ellipse local $x$, $y$, and $z$ axes are shown by red, green, and blue needles. Positions $P_f$, $P_l$, and $P_f$ are shown by white, red, and green featureless spheres (which may be hidden by balls). See the screenshot to the right and the code below `hw01.markers_show` in `ball_setup_hw01` to see which colors are assigned.




$P_n$ is chosen such that $\|\overrightarrow{P_f P_n}\| + \|\overrightarrow{P_n P_l}\| = L$. This means that the first several balls can be arranged on the line from $P_f$ to $P_n$ and the remaining balls can be arranged on the line from $P_n$ to $P_l$, while keeping the distance between adjacent balls to $l_r$ with possibly one exception. The exception occurs when no ball is exactly at $P_n$. In the exception case consider the two balls closest to $P_n$. The distance from the first of these balls, to $P_n$ to the second of these balls will be $l_r$, but the distance between these two balls will be less than $l_r$.

(*a*) Modify the code in `ball_setup_hw01` so that the balls are arranged in a straight line starting from `first_pos` reaching exactly or close to `nadir_pos`, and ending at `last_pos`. The distance between adjacent balls must be `distance_relaxed` except for the exception condition described above.

No ball should be placed below `nadir_pos`. The first ball must be at `first_pos` and the last ball must be at `last_pos`. See the screenshot above.

If this is solved correctly the two balls near `last_pos` should twitch (because they are closer than their relaxed distance). Initially the two lowest balls should be near the green marker (the location of `nadir_pos`) and the white and red spherical markers should not be visible.

To help debug your solution pause the simulation using `p`, then start the scene, `1`. Compare your placement of the balls to the marker balls.

(*b*) In this problem that twitching when a scene is started will be fixed. The solution is not ideal from a physical model point-of-view, but it makes a decent question. Modify the code so that each spring can have its own relaxed distance. Do so by adding a relaxed distance member to `Ball` and using it appropriately. The code in `ball_setup_hw01` must set this new member to `distance_relaxed` for all balls except for one. That one ball is the one that's closer to its neighbor. Set it to the appropriate distance if `opt_special_dist_relaxed` is true, otherwise set it to `distance_relaxed`. Modify the code in `time_step_cpu` so that the per-ball `distance_relaxed` values will be used. Part of this problem is deciding whether the value of `distance_relaxed` set for a ball applies to its connection to its predecessor in the `balls` array or to its connection with its successor.

Use scene 2 for this problem. In scene 2 (and 3) `opt_special_dist_relaxed` is true.

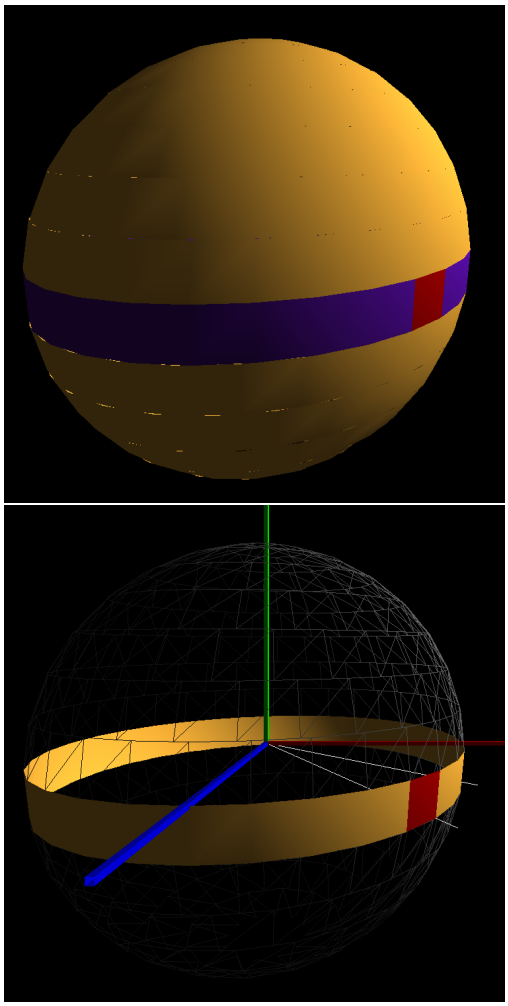
If this is solved directly scene 2 should start without a sudden twitch. Instead the balls will gently come to rest in the correct place.

(*c*) Modify the code in `ball_setup_hw01` so that when `opt_spin` is true (which is the case in scene 3) the balls are given an initial velocity so that they rotate rigidly around the line defined by `first_pos` and `last_pos`. To do this set the ball velocity. Note that balls closer to the line will have a lower velocity.

If this is solved correctly the balls will swing around the axis while generally keeping their shape.

# LSU EE 4702-1 Homework 2 Due: 8 October 2020

**Problem 0:** If necessary, 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 the sphere from the `demo-5-more-ogl.cc` code, but with a *slice* and pair of triangles on the slice, a *segment*, deferentially colored. The segment is red, the rest of the slice is purple. See the upper screenshot to the right. The slice and segment can be moved using the user interface. The goal in this assignment is to provide visual aids to help understand how the sphere is constructed. The sphere's local coordinate axes are to be shown as blocks, and lines to *vertex 2* and to the $\theta = 0$ position on the slice are to be shown. See the lower screenshot to the right.



## User Interface

Initially the arrow keys, PageUp, and PageDown can be used to move around the scene. To change $\eta$ and $\theta$ press (lower-case) `s` (if necessary) and then use the arrow keys. The current value is shown in one of the last lines of green text. Pressing `c` toggles between showing the sphere in show-construction mode (the default) and in normal mode.

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 `Shift` and an arrow key will move by a larger distance (than if `Shift` were not pressed) and pressing `Ctrl` and an arrow key will move by a smaller distance. The eye can be aimed up and down by pressing `Home` and `End`.

Press `Ctrl=` to increase the size of the green text and `Ctrl-` to decrease the size.

The `+` and `-` keys can be used to change the value of certain variables. These variables specify things such as the light intensity, spring constant, 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` cycles through the different variables.

## Code Generation and Debug Support

The compiler generates two versions of the code, `hw02` and `hw02-debug`. Use `hw02` to measure performance, but use `hw02-debug` for debugging. The `hw02-debug` version is compiled with optimization turned off and with OpenGL error checking turned on. 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.

When OpenGL error checking is on (as it is in the debug version) helpful error and warning messages will be printed about misuse or abuse of the OpenGL API. These will appear on the terminal window (which might be a gdb session) from which `hw02-debug` was started.

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. These variables are intended for debugging and trying things out.

**Problem 1:** The routine `render()` renders the sphere, and is based on the `demo-5-more-ogl` code. New variables to pay attention to are `opt_show`, `opt_show_eta`, and `opt_show_theta`. When `opt_show` is true the sphere should be shown in construction mode, otherwise it's a plain gold sphere. Also, when `opt_show` is true the lonely triangle should be hidden. Variable `opt_show_eta` identifies the slice which is to be highlighted and `opt_show_theta` emphasizes the segment. Look how these are used to compute `show_eta` and `show_theta`, and how `show_eta` and `show_theta` are used to specify colors. Did you look? This assignment cannot be completed without understanding how these variables are used, and without understanding how the sphere is rendered of the code in render, especially the loop nest.

(*a*) Modify the code so that when in construction mode (`opt_show` is true) the slice is shown in gold (not purple), the segment is shown in red (as it is), and only the edges are shown for triangles which are not part of the highlighted slice; the edges should be gray.

There are several ways to solve this. The least disruptive (in terms of the original code) is to end and start rendering passes within the loop nest, switching between triangles and lines. Do not do more rendering passes than are necessary.

(*b*) Draw two lines that start at the center of the sphere and that pass through points labeled `Vertex 2` in the code. (These are shown as white in the second screenshot on the first page.) The length of these lines should be 1.1 in the sphere's local coordinate space. Both lines should be drawn for points in which `eta0 == opt_show_eta` (for which `show_eta` is true). Draw one line for `theta0 = 0` and draw the other for `theta0 = opt_show_theta`. The second line should pass through a corner of the highlighted segment. Draw these using the `GL_LINES` primitive, and make them white.

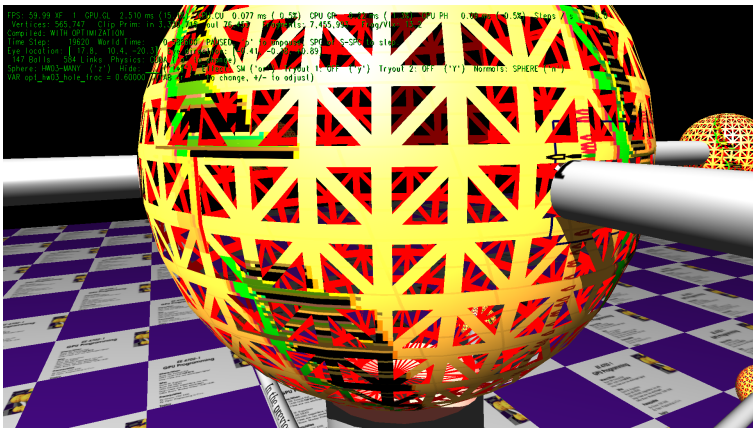
Try putting this code after the `slice` / `seg` loop nest.

(*c*) Show the coordinate axes of the sphere local space using red, green, and blue blocks for the $x$, $y$, and $z$ axes. The width of the block should be .02 in the sphere local coordinate space.

Set the normals correctly for the blocks. That is, the surface facing up should have a normal of $\begin{bmatrix} 0 \\ 1 \\ 0 \end{bmatrix}$, etc. If the normals are set correctly then the edges of the blocks will be sharp and the more a block face faces the light the more brightly it will be lit.

## LSU EE 4702-1 Homework 3 Due: 6 November 2020

**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 scene from the links code (in `projbase/links`), the one showing a vaguely tree-like form, the *silly tree*, constructed from flexible links and balls. In the original links code the balls are ordinary spheres. In this assignment the spheres are to have triangular holes, as shown in screenshot below, which shows a closeup after Problem 1 or 2 is solved correctly. (Problems 1 and 2 do the same thing in different ways.)



### Non-Assignment-Specific User Interface

Press digits `1` through `4` to initialize different scenes, the program starts with scene 1. Scene 1 starts with the balls arranged in the tree-like form.

Press `p` to pause the simulation.

Press `Ctrl=` to increase the size of the green text and `Ctrl-` to decrease the size. Initially the arrow keys, `PageUp`, and `PageDown` can be used to move around the scene. Using the `Shift` modifier 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. Press `l` to move the light around and `e` to move the eye (which is what the arrow keys do when the program starts).

The `+` and `-` keys can be used to change the value of certain variables. These variables control things like light intensity and options needed for this assignment. The variable currently affected by the `+` and `-` keys is shown in the bottom line of green text next to VAR. Pressing `Tab` cycles forward through the different variables.

Look at the comments in the file `hw03.cc` for documentation on other keys.

### Assignment-Specific User Interface

The sphere can be rendered by three different shaders, `HW03-ONE`, `HW03-MANY`, and `HW03-TRI`. The shader being used is shown to the right of `Sphere` in the green text. To cycle through the shaders press `z`.

The size of the hole is specified by variable `opt_hw03_hole_frac`. This can be modified using the UI, look for that variable name to the right of VAR.

Pressing `n` toggles between computing sphere lighting based on triangle normals, `TRI`, and sphere normals, `SPHERE`. When solving the problems it might help to rendering using triangle normals so that you can see where your hole is positioned within the triangle. But then switch to sphere normals to make sure the lighting is done correctly.

### Graphics and Performance Investigation Options

The user interface can be used to toggle various rendering options and for generating a screenshot.

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.

Pressing `F12` will write a screenshot to file `hw03.png`. Any existing screenshot will be silently overwritten so be sure to rename files that you want to keep.

The rendering of shadows is toggled by `o` and the rendering of reflections it toggled by `r`. Their state is shown in the green text next to Effect:. Pressing `n` will toggle how surface normals are computed for tessellated spheres, the possibilities are to use the triangle normal or the sphere normal. The use of the triangle normals makes it easier to see the triangles from which the sphere was tessellated.

### Display of Performance-Related Data

The top green text line shows performance in various ways. XF shows the number of display frames per frame buffer update. An ideal number is 1. A 2 means that two display frame update were done in the time needed to update the frame buffer once, 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).

### Code Generation and Debug Support

The compiler generates two versions of the code, `hw03` and `hw03-debug`. Use `hw03` to measure performance, but use `hw03-debug` for debugging. The `hw03-debug` version is compiled with optimization turned off and with OpenGL error checking turned on. You are strongly encouraged to run `hw03-debug` under the GNU debugger, `gdb`. See the material under "Running and Debugging the Assignment" on the course procedures page.

When OpenGL error checking is on (as it is in the debug version) helpful error and warning messages will be printed about misuse or abuse of the OpenGL API. These will appear on the terminal window (which might be a gdb session) from which `hw03-debug` was started.

Keys `y`, `Y`, and `Z` toggle the value of host Boolean variables `opt_tryout1`, `opt_tryout2`, and `opt_tryout3`. and corresponding shader variables `tryout.x`, `tryout.y`, and `tryout.z`. The user interface can also be used to modify host floating-point variable `opt_tryoutf` and corresponding shader variable `tryoutf` using the `Tab`, `+`, and `-` keys, see the previous section. These variables are intended for debugging and trying things out.

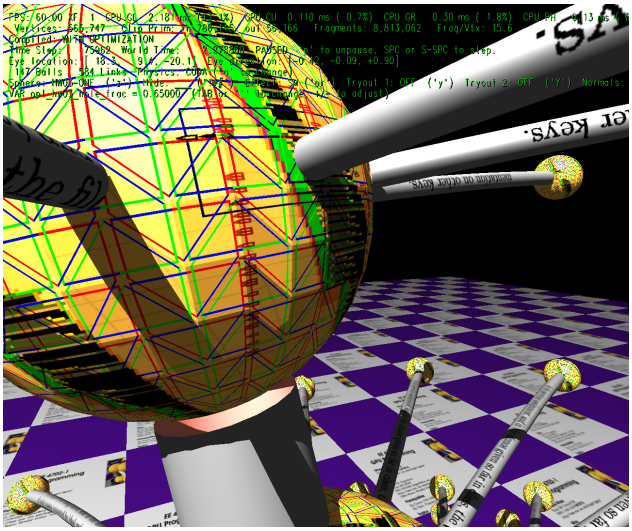
**Problem 1:** Modify the code in `gs_main_many_triangles` and perhaps `fs_main_many_triangles` in file `hw03-shdr.cc` so that each triangle used to tessellate the sphere has a triangular hole in it. (See the screenshot on the first page.) The relative size of the triangle should be based on the value of `opt_hw03_hole_frac`. Variable `opt_hw03_hole_frac` ranges from 0 to 1 (and can be controlled using the UI). When `opt_hw03_hole_frac` is 0.05 there should be a tiny hole, when `opt_hw03_hole_frac` is 0.95 there should be a large hole. At 0, there is no hole, at 1 the sphere itself should be invisible.

In this problem the hole effect should be achieved by having the geometry shader emit primitives for the area surrounding the hole.

- It should be possible to see the inside sphere surface and beyond through the hole.
- The texture should be applied in the same position as it would be if there was no hole. That is, when switching between the `HW03` and `TRI` shaders the position of the texture should not move.
- Don't forget to set the number of output vertices in the geometry shader. Don't set them to more than you need.
- Be sure to set normals correctly. To check that they are correct cycle through the different shaders using the `z` key. The coloring of the sphere should be the same for the `HW03` and `TRI` shaders.
- Do not expect shadows to work correctly. In particular, light won't go through the holes interrupting the shadow cast by the part of the sphere that is present.

**Problem 2:** Modify the code in shader routines `fs_main_one_triangle` and perhaps routine `gs_main_one_triangle` so that each triangle used to tessellate the sphere has a triangular hole in it, the same kind of hole as in the previous problem. In this problem do so by discarding a fragment if it would be within the hole.

To assist with this solution the geometry and fragment shaders have been modified to draw lines, see the screen shot to the right. Figure out how those lines are drawn, and modify the shaders so that there is a hole. Note that calling built-in function `discard` in the fragment shader discards the fragment. For your convenience there is a variable `hole_here` that can be set for those fragment at a hole.



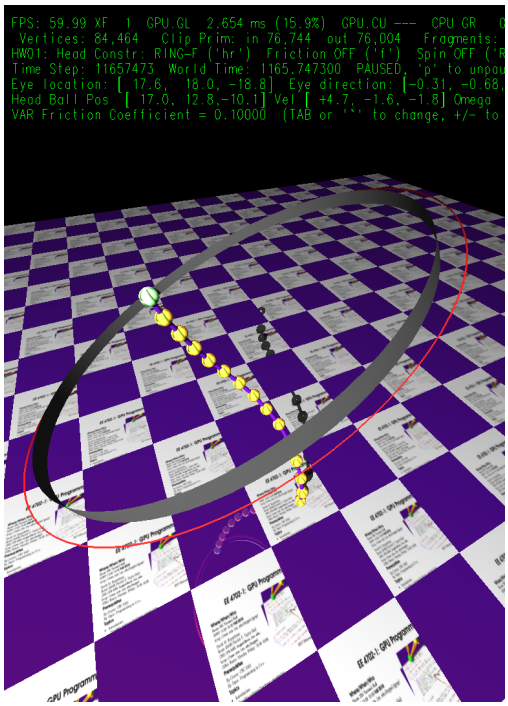
- The size of the holes in the one-triangle shader must match the size in the many-triangle shader.

**Problem 3:** Which appears to be the better method? That is, which uses more computational resources? See if you can determine any difference in performance between the two.

# 8 Fall 2019

# LSU EE 4702-1 Homework 1 Due: 12 September 2019

**Problem 0:** Follow the instruction 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 string of beads hanging from a ring. The ring is shown as a cylinder surrounded by a red circle. The red circle is rendered using code written in class (with a few changes), see the code in file `hw01-graphics.cc` near the comment `Live Classroom Demo`.




## Assignment-Specific Model and User Interface

simulation starts the head (first, zero) ball is attached to the ring and can slide freely along the ring. Pressing `r` (lower-case) will toggle between the head ball sliding freely and the head ball rotating along the ring at a fixed rate. The green text line starting with HW01: shows the state of the ball's attachment to the ring and of the ring itself. On this line the text to the right of Head Contr (head ball constraint) is RING-F if the ball is sliding freely, RING-A (animated) if the ball is rotating with the ring, FREE if the ball is not attached to anything, and LOCKED if the ball is fixed in space (does not change its position). The last two states can be obtained by pressing `h` (head). (Press `h` and `r` to familiarize yourself with the behavior, and be sure to read the next Common User Interface section which describes other available UI features, such as moving the viewer and pausing simulation.)

When in mode RING-A the rotation rate is determined by variable VAR Rotation rate (variable `hw01.omega` in the code). See the Common User Interface section below to see how to change that and other variables.

For Problem 1 ring friction will be added to the simulation model. Pressing `f` toggles friction on and off (variable `opt_hw01_do_friction`, the state is shown to the right of Friction and by the color of the ring. When friction is turned on the ring color is yellowish, like sandpaper. This will have no effect until Problem 1 is properly solved. VAR Friction Coefficient is the coefficient of friction, variable `opt_hw01_fric_coefficient` in the code.

For Problem 2 the ring will be made to spin. When properly solved pressing `R` should turn this feature on and off, (variable `opt_hw01_spin`).

## Common User Interface

release one end (to be precise, the ball at one end) and pressing `t` (tail) will grab or release the other end. (Actually, those keys toggle between the `OC_Locked` and `OC_Free` constraint of their respective balls.)

Press digits `1` through `4` to initialize different scenes, the program starts with scene 1. Scene 1 starts with the balls arranged almost vertically. Scene 2 starts with the balls arranged horizontally,

and they will start swinging. Each time scene 1 and 2 are initialized the ring is positioned to a new position. In scenes 3 and 4 the ring is always positioned the same way.

Press `Ctrl=` to increase the size of the green text and `Ctrl-` to decrease the size. 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 tail 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).

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.

Look at the comments in the file `hw01.cc` for documentation on other keys.

## Code Describing the Ring

The code in this assignment is based on `demo-2-springs.cc`, which shows balls connected by links. In the underlying physical model balls are modeled as point masses and links are modeled as ideal springs. The same is true for this assignment. In `demo-2-springs.cc` the head and tail balls could either be locked or free. When a ball is locked its position doesn't change. That's simulated by simply setting a locked ball's velocity to zero. Free balls are simulated based on the ideal-mass-and-springs model.

In this assignment a ball can be under one of four possible constraints, Free, Locked, Ring-Animated, and Ring-Free. The constraint imposed is in member `ball->constraint`. The Free and Locked constraints are carried over from `demo-2-springs.cc`. The position and velocity of a ball under the Ring-Animated constraint is determined solely by the rotating ring and so the forces on such a ball have no effect. A ball under the Ring-Free constraint can move freely along the circumference of a ring, its motion is determined by forces tangent to the ring at its location.

Variables describing the ring and some other features in this assignment are in `struct HW01_Stuff` which is declared in the `World` class as `HW01_Stuff hw01`. In this discussion they will be referred to using the member name, `hw01`. The coordinate of the center of the ring is in `hw01.center`, the ring axis (normal) is in `hw01.axis`, and its radius is in `hw01.radius`. These three variables define the ring. Additional variables are provided for convenience and to describe the ball's position on the ring. The local $x$- and $y$-axis of the ring are in `hw01.x` and `hw01.y`.

The location of the ball on the ring (in radians) is specified by `hw01.theta`. The rotation rate (in radians per second) around the ring is specified by `hw01.omega`. The following code computes the position of the ball described by these variables:

```
pCoor pos =
  hw01.center +
  hw01.radius * ( cosf(hw01.theta) * hw01.x + sinf(hw01.theta) * hw01.y );
```

In `time_step_cpu` lambda function `pos_compute` computes this quantity.

## Initialization of Ring Position

The variables describing the ring are initialized by code in `time_step_cpu` below the comment `Initialize Ring`. The initialization is performed whenever variable `hw01.rail_inited` is false. It is set to `false` when a scene is initialized, and when the head is released and then re-ringed. (Press `h` then `r`.)

The ring position is initialized in three possible ways. In all cases the ring is positioned so that the ring passes through the head ball. If the ball is not moving then the ring axis is set to either a random direction (scenes 1 and 2) or a predetermined direction (scenes 3 and 4). If the ball is moving, the local $y$ axis, `hw01.y`, of the ring is set to the ball's direction of motion and the local $x$ axis is chosen to be orthogonal to the global $y$ axis. A passenger on the ball might feel that the

ball was smoothly captured by the ring. The rotation rate, `hw01.omega` is set to 1 if the ball is under constraint Ring Animated, otherwise it is set to zero.

Here is a condensed version of the code:

```
pNorm vdir(hball.velocity);
hw01.theta = 0;

if ( vdir.magnitude < 0.00001 ) {
    switch ( curr_setup ) {
    case 1: case 2:  // Random angle.
      hw01.axis = pNorm(.9*drand48(),1,.9*drand48()); break;
    case 3: // Almost straight up.
      hw01.axis = pNorm(0.2,1,-0.2); break;
    case 4: // Orthogonal to direction of chain.
      hw01.axis =
        cross( balls[chain_length-1].position - balls[0].position, pVect(0,1,0) );
      break;}

    hw01.x = hw01.axis.z != 0 && hw01.x != 0
      ? pNorm(hw01.axis.z,0,-hw01.axis.x) : pNorm(1,0,0);
    hw01.center = hball.position - hw01.x * hw01.radius;
    hw01.y = cross(hw01.axis,hw01.x);
    hw01.omega = hball.constraint == OC_Ring_Animated ? 1 : 0;
  } else {
    hw01.y = vdir;
    hw01.x = pNorm( cross( hw01.y, pVect(0,-1,0) ) );
    hw01.axis = cross(hw01.x,hw01.y);
    hw01.center = hball.position - hw01.radius * hw01.x;
    hw01.omega = vdir.magnitude / hw01.radius; // Match ball speed.
  }
```

## The Time Step Routine and Ring Simulation

The time step routine has three main parts. In the first part the force on each ball is determined and written to `ball->force`. The second part, described above, initializes the ring if that needs to be done. The third part updates the velocity and position of each ball. The method used to update these is determined by `ball->constraint`. The code for the Free and Locked cases is similar to that in `demo-2-springs`, though the placement of the code is different.

The code for Ring-Animated computes the position and velocity based on the fixed motion. Here is a condensed version:

```
case OC_Ring_Animated:
    // Ball is attached to ring and rotates with it at fixed rate omega.

    // Compute amount by which ring will rotate during this time step.
    const double delta_theta = delta_t * hw01.omega;

    // Update velocity vector.
    ball->velocity = hw01.omega * hw01.radius *
      ( -sinf(hw01.theta) * hw01.x + cosf(hw01.theta) * hw01.y );
```

```
    // Update the position angle and the position Cartesian coordinate.
    hw01.theta += delta_theta;
    ball->position = pos_compute(hw01.theta);
  break;
```

The code for the Ring-Free constraint uses `ball->force`, but not all of it. Because the ball is constrained to move along the ring the only force that matters is force in the direction along the ring (the tangent). The code starts by computing the tangent, `dtan`, and using that to find the amount of force in that direction, `force_tan`:

```
pVect center_to_ball = pVect(hw01.center,ball->position) / hw01.radius;
pVect dtan = cross(hw01.axis,center_to_ball);
const float force_tan = dot( ball->force, dtan );
```

Force `force_tan` is then used to compute the change in velocity, `delta_v` and from that a change in rotation rate, `delta_omega`. Then `hw01.omega` is updated and used to update the velocity, and `hw01.theta` is updated and used to update the position:

```
const float delta_v = force_tan / ball->mass * delta_t;
const float delta_omega = delta_v / hw01.radius;
hw01.omega += delta_omega;
hw01.theta += hw01.omega * delta_t;
ball->velocity = hw01.omega * hw01.radius * dtan;
ball->position = pos_compute(hw01.theta);
```

The alert student might suspect that `ball->velocity` isn't needed for the head ball attached to the ring. True, it's not needed to determine the position, but it is used when determining whether the spring is gaining or loosing energy, and it is needed when the ball's constraint is changed of Free.

**Problem 1:** In the unmodified assignment code the model does not account for friction on the ring. In this problem account for such friction when the ball is sliding along the ring and variable `opt_hw01_do_friction` is `true`. The code to be modified is in routine `time_step_cpu` near the comment `Put Problem 1`.

As you may remember from physics, the amount of dynamic frictional force on an object sliding over a surface is determined by the force of the object on the surface and a coefficient of friction. Speed of motion is irrelevant.

Recall that the forces on the ball are accumulated in `ball->force`. As described in Problem 0, the force along the tangent accelerates the ball, and so this component of force does not contribute to friction.

Let $f_t$ denote the magnitude of the ball's force along the tangent (`force_tan` in the code) and let $f_r$ denote the magnitude of the ball's force on the ring (something that must be solved for this problem). Let $k$ denote the friction coefficient, (`opt_hw01_fric_coefficient`). Then the magnitude of frictional force will be $kf_r$ and the direction is in the opposite direction of the ball's motion.

As with ordinary force, the frictional force will result in a change in rotation rate, $\Delta\omega = \frac{kf_r\Delta t}{mr}$, where $m$ is the ball mass and $r$ is the ring radius, with an important caveat: the interval $\Delta t$ can not be so long as to result in a change in direction. Instead, $\omega$ should be just set to zero.

So, to solve the problem first compute $f_r$ and with that $\Delta\omega = \frac{kf_r\Delta t}{mr}$. Computing $f_r$ is straightforward. Of course, I don't need to remind anyone that $f_r \neq f - f_t$, where $f$ is the total force on the ball.

Next, figure out whether to add or subtract $\Delta\omega$ from `hw01.omega`. If `hw01.omega` changes sign, set it to zero.

If this is solved correctly the chain should come to a stop before reaching the lowest point on the ring, unless the ring is steeply titled or the chain is bouncing around.

**Problem 2:** When variable `opt_hw01_spin` is true and the ring is visible, spin the ring around an axis parallel to the $xz$ plane. Press `R` to toggle the state of `opt_hw01_spin`. The ring should spin at rate `hw01.opt_spin_omega`.

Put most of the solution for this at the end of `time_step_cpu`, search for `Put Problem 2` in the code. An `if` statement with an appropriate condition has been prepared.

To solve this problem, first compute a rotation axis and matrix. This only needs to be done once each time a new ring is chosen. Put the rotation matrix and any other variables that are initialized once used each time step in `HW01_Stuff`. The code below shows how to construct a transformation matrix that rotates around axis `spin_axis` by `theta` radians, and to use it to rotate a vector. The rotated vector is re-normalized to correct small errors that change vector length.

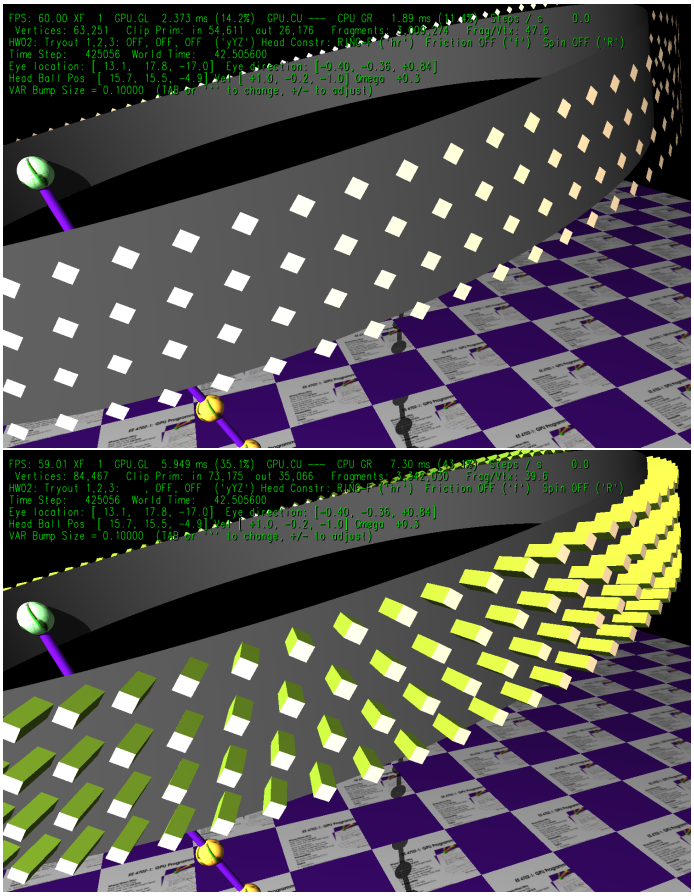
```
pMatrix xform_spin; // Put this in HW01_Stuff.
hw01.xform_spin.set_rotation( spin_axis, theta );
pNorm before(2,1,2);  // Some unit vector.
pVect after = pNorm( hw01.xform_spin * before );
```

Then use the rotation matrix each time step to rotate the appropriate variables describing the ring. The default spin rate, in `hw01.opt_spin_omega`, is 1 radian per second, and so the ring should make a complete rotation in $2\pi \approx 6.283$ seconds.

Take care not to re-compute the rotation matrix unnecessarily.

**LSU EE 4702-1** **Homework 2** **Due: 2 October 2019**

**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 string of beads hanging from a ring. The ring is shown as a cylinder surrounded by tan-colored squares (or diamonds). See the screenshot on the upper right. The cylinder and the squares are rendered by code in routine `World::render_cylinder`, near the end of `hw02.cc`.




As you may have guessed, this code is based on the solution to Homework 1, and the user interface is similar. The goal in this assignment is to add sides to the squares so that they appear as blocks protruding from the cylinder surface, as in the screenshot on the lower right. The goal in the next assignment, Homework 3, will be to render these efficiently. (There will be a separate handout for Homework 3.)

## Assignment-Specific Model and User Interface

When the simulation starts the head (first, zero) ball is attached to the ring and can slide freely along the ring. Pressing `r` (lower-case) will toggle between the head ball sliding freely and the head ball rotating along the ring at a fixed rate. The green text line starting with HW02: shows the state of the ball's attachment to the ring and of the ring itself. On this line the text to the right of Head Contr (head ball constraint) is RING-F if the ball is sliding freely, RING-A (animated) if the ball is rotating with the ring, FREE if the ball is not attached to anything, and LOCKED if the ball is fixed in space (does not change its position). The last two states can be obtained by pressing `h` (head). (Press `h` and `r` to familiarize yourself with the behavior, and be sure to read the next Common User Interface section which describes other available UI features, such as moving the viewer and pausing simulation.)

When in mode RING-A the rotation rate is determined by variable VAR Rotation rate (variable `hw01.omega` in the code). See the Common User Interface section below to see how to change that and other variables.

Pressing `f` toggles friction on and off, the state is shown to the right of Friction and by the color of the ring. When friction is turned on the ring color is yellowish, like sandpaper. Pressing `R` spins the ring.

## Common User Interface

Press `h` (head) will grab or release one end (to be precise, the ball at one end) and pressing `t` (tail) will grab or release the other end. (Actually, those keys toggle between the `OC_Locked` and `OC_Free` constraint of their respective balls.)

Press digits `1` through `4` to initialize different scenes, the program starts with scene 1. Scene 1 starts with the balls arranged almost vertically. Scene 2 starts with the balls arranged horizontally, and they will start swinging. Each time scene 1 and 2 are initialized the ring is positioned to a new position. In scenes 3 and 4 the ring is always positioned the same way.

Press `Ctrl=` to increase the size of the green text and `Ctrl-` to decrease the size. 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 tail 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).

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.

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

## Code Generation and Debug Support

The compiler generates two versions of the code, `hw02` and `hw02-debug`. Use `hw02` to measure performance, but use `hw02-debug` for debugging. The `hw02-debug` version is compiled with optimization turned off and with OpenGL error checking turned on. 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.

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` and corresponding shader variable `tryoutf` using the `Tab`, `+`, and `-` keys, see the previous section. These variables are intended for debugging and trying things out.

*There are problems on the next page.*

**Problem 1:** Please submit the solution to this problem in writing or in the comments of your assignment. (Problem 2 is a computer assignment.) A question similar to this may be asked on the midterm exam. Each iteration of the loop below, taken from `World::render_cylinder`, correctly renders one diamond. Notice that there is one rendering pass per iteration.

```
for ( auto& e: info )
  {
    glBegin(GL_TRIANGLE_STRIP);
    glNormal3fv(e.nc); glVertex3fv(e.ctop);
    glNormal3fv(e.nl); glVertex3fv(e.cl);
    glNormal3fv(e.nr); glVertex3fv(e.cr);
    glNormal3fv(e.nc); glVertex3fv(e.cbot);
    glEnd();
  }
```

In contrast, the code below performs just one rendering pass. Assume that the contents of the `info` container is the same in the code above and below.

```
glBegin(GL_TRIANGLE_STRIP);
for ( auto& e: info )
  {
    glNormal3fv(e.nc); glVertex3fv(e.ctop);
    glNormal3fv(e.nl); glVertex3fv(e.cl);
    glNormal3fv(e.nr); glVertex3fv(e.cr);
    glNormal3fv(e.nc); glVertex3fv(e.cbot);
  }
glEnd();
```

(*a*) Explain why the second code fragment will not render the diamonds correctly. You should be able to sketch the result on paper.

(*b*) How could the code be modified to correctly render all the diamonds in one rendering pass? A correct solution requires changing one word and moving some code around. Feel free to use a deprecated primitive.

**Problem 2:** Modify the code in file `hw02.cc`, routine `World::render_cylinder` so that the tan-colored squares form the top of a block which touches the surface of the cylinder. To do so render the sides of these blocks and assign them color `color_olive_drab`, or some other color different than the squares. See the screenshots at the top of this assignment.

It will be helpful when solving this assignment to move the eye and light location around. The instructions above explain how to do so.

- Pay attention to efficiency. The code should not be less efficient than the code that's already there.
- Don't call `glColor` and `glNormal` more often than is necessary.
- Be sure to set the normal correctly. If the normal is set correctly adjacent faces will be shaded differently. If the same normal is used for adjacent faces then it will not be possible to tell where one stops and the other starts.

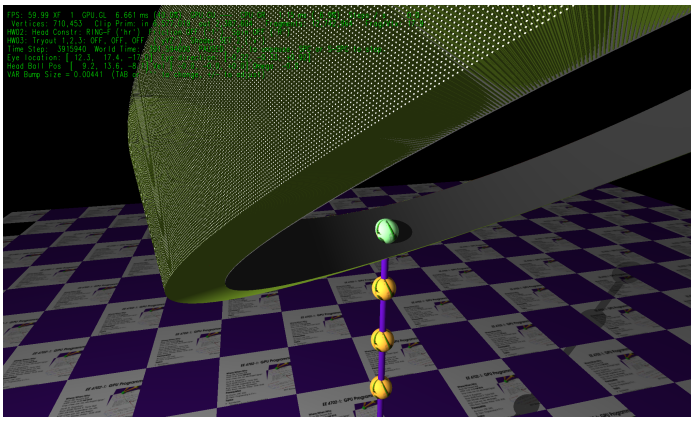
The easier part of this assignment is using `glVertex` calls in a triangles or triangle strip rendering pass to emit the coördinates of a block. See 2016 Homework 1 Problem 1a for an easy

example of how to use `glVertex` calls to draw shapes. (The code is in the repo, so you can look at the unsolved and solved versions.)

The more difficult parts are figuring out exactly what the code in `World::render_cylinder` is doing and then getting the geometry right. Try drawing a diagram of the cylinder and a diamond, and label it with the variables used in the code, such as `ctr`, `nc`, `ctop`, etc.

# LSU EE 4702-1 Homework 3 Due: 1 November 2019

**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 string of beads hanging from a ring. The ring is shown as a cylinder surrounded by green-colored protrusions with tan-colored tops, based on the solution to Homework 2. You may notice that in the screenshot to the right there are a large number of protrusions.



If the Homework 2 solution were used to render so many protrusions the frame rate would be very low, but as can be seen (perhaps after zooming) the code generating the screenshot above can keep up (the frame rate is about 60 FPS and there are no skipped frames [XF is 1]). That code is the solution to Problem 2 in this assignment, in which the protrusions are generated using shaders.

As you may remember one possible reason the code is slow is because protrusion geometry is re-computed each frame. That was true in Homework 2, but in the code for this assignment the geometry is only re-computed when something changes. Therefore performance problems are only due to the overhead of sending vertices to the rendering pipeline one-by-one (rather than using arrays).

## Assignment-Specific User Interface

The code in this assignment uses one of three shaders, shown to the right of Shader on the HW03 line of the green text. The shader in use can be changed by pressing `v`. Shader FIXED uses the OpenGL default shaders, as was used in previous assignments. Shader STRIP-PLUS is for use in Problem 1. When STRIP-PLUS is in use the protrusions are rendered using a single triangle strip. They won't look good until Problem 1 is solved. When shader POINTS is used the protrusions are rendered using primitive type `GL_POINTS` in a `glDrawArrays` rendering pass. Code for this will be completed as part of Problem 2. The shaders are in file `hw03-shdr.cc`. STRIP-PLUS uses vertex shader `vs_strip_plus` and geometry shader `gs_strip_plus`, while POINTS uses vertex shader `vs_points` and geometry shader `gs_points`. Both use the same fragment shader (which is not part of this assignment).

To change the protrusion size modify variable Bump Size as shown to the right of VAR.

## Previous Assignment User Interface

When the simulation starts the head (first, zero) ball is attached to the ring and can slide freely along the ring. Pressing `r` (lower-case) will toggle between the head ball sliding freely and the head ball rotating along the ring at a fixed rate. The green text line starting with HW02: shows the state of the ball's attachment to the ring and of the ring itself. On this line the text to the right of Head Contr (head ball constraint) is RING-F if the ball is sliding freely, RING-A (animated) if the ball is rotating with the ring, FREE if the ball is not attached to anything, and LOCKED if the ball is fixed in space (does not change its position). The last two states can be obtained by pressing `h` (head). (Press `h` and `r` to familiarize yourself with the behavior, and be sure to read the

next Common User Interface section which describes other available UI features, such as moving the viewer and pausing simulation.)

When in mode RING-A the rotation rate is determined by variable VAR Rotation rate (variable `hw01.omega` in the code). See the Common User Interface section below to see how to change that and other variables.

Pressing `f` toggles friction on and off, the state is shown to the right of Friction and by the color of the ring. When friction is turned on the ring color is yellowish, like sandpaper. Pressing `R` spins the ring.

## Common User Interface

Press `h` (head) will grab or release one end (to be precise, the ball at one end) and pressing `t` (tail) will grab or release the other end. (Actually, those keys toggle between the `OC_Locked` and `OC_Free` constraint of their respective balls.)

Press digits `1` through `4` to initialize different scenes, the program starts with scene 1. Scene 1 starts with the balls arranged almost vertically. Scene 2 starts with the balls arranged horizontally, and they will start swinging. Each time scene 1 and 2 are initialized the ring is positioned to a new position. In scenes 3 and 4 the ring is always positioned the same way.

Press `Ctrl=` to increase the size of the green text and `Ctrl-` to decrease the size. 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 tail 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).

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.

## Code Generation and Debug Support

The compiler generates two versions of the code, `hw03` and `hw03-debug`. Use `hw03` to measure performance, but use `hw03-debug` for debugging. The `hw03-debug` version is compiled with optimization turned off and with OpenGL error checking turned on. You are strongly encouraged to run `hw03-debug` under the GNU debugger, `gdb`. See the material under "Running and Debugging the Assignment" on the course procedures page.

Keys `y`, `Y`, and `Z` toggle the value of host Boolean variables `opt_tryout1`, `opt_tryout2`, and `opt_tryout3`. and corresponding shader variables `tryout.x`, `tryout.y`, and `tryout.z`. The user interface can also be used to modify host floating-point variable `opt_tryoutf` and corresponding shader variable `tryoutf` using the `Tab`, `+`, and `-` keys, see the previous section. These variables are intended for debugging and trying things out.

*There are problems on the next page.*

**Problem 1:** One frustration with triangle strips is that in order to start a new strip you either need to start a new rendering pass (which slows things down) or else you need to duplicate both the vertex that ends the old strip and starts the new strip (which is inelegant). Fortunately this frustration will be in the past once this problem is solved!

When shader STRIP-PLUS is used the protrusions will all be rendered using a single triangle strip. The shader is already written so that only the normal of the third (provoking) vertex is used for lighting.

Modify the `_strip_plus` shaders in file `hw03-shdr.cc` and the code in `World::render_cylinder` in file `hw03.cc` so that a new strip will be started when needed without having to start a new rendering pass. Signal the start of a new strip by sending something down the rendering pipeline. For example, you might set the normal of the first two vertices of a strip to the zero vector.

**Problem 2:** The `_points` shaders are to be used in a `glDrawArrays` rendering pass with input primitive point and with no inputs to the vertex shaders other than `gl_VertexID`.

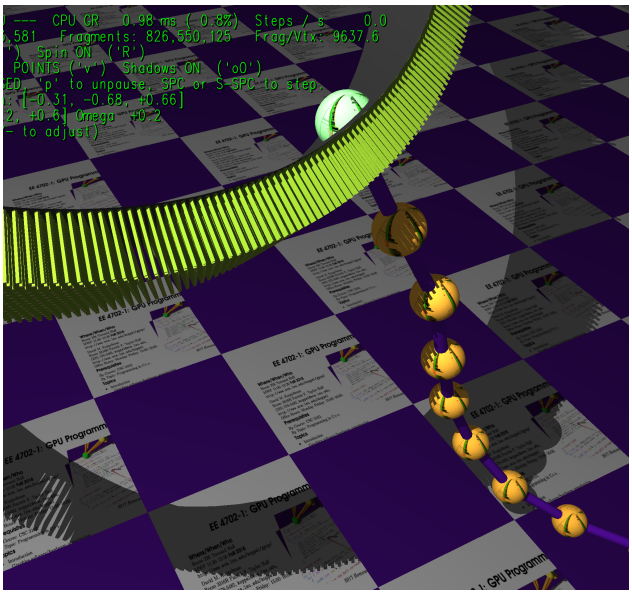
Modify the `_points` shaders so that they generate the protrusions themselves using only `gl_VertexID` and uniform variables. As a convenience some useful uniform variables are set, but you'll need to set others to complete the problem. See the code in `render_cylinder`.

When this problem is successfully solved Bump Size (bump was the old name of protrusion) can be set small without impacting performance).

- Do not source any arrays into the rendering pipeline.
- Delete existing code in the `points` shaders that's not needed.
- Don't forget to modify the interface blocks (both vertex shader output and geometry shader input).
- Make sure that normals are correctly computed. Check normals by switching between shaders and comparing the shading of the edges.

# LSU EE 4702-1 Homework 4 Due: 15 November 2019

**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 string of beads hanging from a ring. The ring is shown as a cylinder surrounded by green-colored protrusions with tan-colored tops, based on the solution to Homework 3. You may notice that in the screenshot to the right that the protrusions are casting shadows. Efficiently casting those shadows is the goal of Problem 2 in this assignment.



### Assignment-Specific User Interface

The setting for the rendering of shadows is shown to the right of Shadows on the HW04 line. There are three possible states, OFF, in which no shadows are rendered, ON, in which shadows are to be rendered normally and VOL, in which the protrusions' shadow volumes are visible. The rendering of shadows is toggled using `o` (lower-case oh) and the rendering of shadow volumes is toggled using `O` (upper-case oh). When shadow volume rendering is turned on the ring is not shown (because it gets in the way). Also, the outside (light-facing) face of shadow volumes is colored blue, the inside (shadow-facing) face is colored red. The colors are based on frequency shift of moving objects. Imagine that the front side is coming towards you and so it is blue shifted, and the back face is moving away, so it is red shifted. If the eye (actually the near plane of the view volume) is not itself in a shadow, all shadow volumes should appear blue.

In all of the shaders the balls, links, and ring should cast shadows. In the unmodified code the protrusions cast shadows only with the STRIP-PLUS shader, when Problem 2 is correctly solved protrusions will cast shadows with the POINTS shader, and will do so with much fewer fragments than with STRIP-PLUS.

The code in this assignment uses one of three shaders, shown to the right of Shader on the HW04 line of the green text. The shader in use can be changed by pressing `v`. Shader FIXED uses the OpenGL default shaders, as was used in previous assignments. Shader STRIP-PLUS is for use in Problem 1. When STRIP-PLUS is in use the protrusions are rendered using a single triangle strip but with the strip-restart feature developed in Homework 3.

When shader POINTS is used the protrusions are rendered using primitive type `GL_POINTS` in a `glDrawArrays` rendering pass. The shaders are in file `hw03-shdr.cc`. STRIP-PLUS uses vertex shader `vs_strip_plus` and geometry shader `gs_strip_plus`. When POINTS uses vertex shader `vs_points` and geometry shader `gs_points_common` with argument `shadow_volumes` set to `false`.

When the STRIP-PLUS shader option is selected shadow volumes are rendered by code in `vs_strip_plus_sv` and `gs_strip_plus_sv`. This code correctly renders the shadow volumes, the only changes to be made are for the improved triangle re-start to be implemented for the Strip Plus+ shader in Problem 1. When the POINTS shader option is selected shadow volumes are rendered by the code in `vs_points` and `gs_points_common` with argument `shadow_volumes` set to

`true`. In the unmodified `gs_points_common` there is an `if` statement guarding an area where the shadow volume code is to be placed. To find it search for "Put Problem 2 Solution".

In non-shadow rendering passes `fs_main` is used for both STRIP-PLUS and POINTS. In shadow rendering passes `fs_main_sv` is used for both of the shaders.

To change the protrusion size modify variable Bump Size as shown to the right of VAR.

### Previous Assignment User Interface

When the simulation starts the head (first, zero) ball is attached to the ring and can slide freely along the ring. Pressing `r` (lower-case) will toggle between the head ball sliding freely and the head ball rotating along the ring at a fixed rate. The green text line starting with HW02: shows the state of the ball's attachment to the ring and of the ring itself. On this line the text to the right of Head Contr (head ball constraint) is RING-F if the ball is sliding freely, RING-A (animated) if the ball is rotating with the ring, FREE if the ball is not attached to anything, and LOCKED if the ball is fixed in space (does not change its position). The last two states can be obtained by pressing `h` (head). (Press `h` and `r` to familiarize yourself with the behavior, and be sure to read the next Common User Interface section which describes other available UI features, such as moving the viewer and pausing simulation.)

When in mode RING-A the rotation rate is determined by variable VAR Rotation rate (variable `hw01.omega` in the code). See the Common User Interface section below to see how to change that and other variables.

Pressing `f` toggles friction on and off, the state is shown to the right of Friction and by the color of the ring. When friction is turned on the ring color is yellowish, like sandpaper. Pressing `R` spins the ring.

### Common User Interface

Press `h` (head) will grab or release one end (to be precise, the ball at one end) and pressing `t` (tail) will grab or release the other end. (Actually, those keys toggle between the `OC_Locked` and `OC_Free` constraint of their respective balls.)

Press digits `1` through `4` to initialize different scenes, the program starts with scene 1. Scene 1 starts with the balls arranged almost vertically. Scene 2 starts with the balls arranged horizontally, and they will start swinging. Each time scene 1 and 2 are initialized the ring is positioned to a new position. In scenes 3 and 4 the ring is always positioned the same way.

Press `Ctrl=` to increase the size of the green text and `Ctrl-` to decrease the size. 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 tail 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).

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.

### Code Generation and Debug Support

The compiler generates two versions of the code, `hw03` and `hw03-debug`. Use `hw03` to measure performance, but use `hw03-debug` for debugging. The `hw03-debug` version is compiled with optimization turned off and with OpenGL error checking turned on. You are strongly encouraged to run `hw03-debug` under the GNU debugger, `gdb`. See the material under "Running and Debugging the Assignment" on the course procedures page.

Keys `y`, `Y`, and `Z` toggle the value of host Boolean variables `opt_tryout1`, `opt_tryout2`, and `opt_tryout3`. and corresponding shader variables `tryout.x`, `tryout.y`, and `tryout.z`. The user interface can also be used to modify host floating-point variable `opt_tryoutf` and corresponding

shader variable `tryoutf` using the `Tab`, `+`, and `-` keys, see the previous section. These variables are intended for debugging and trying things out.

*There are problems on the next page.*

**Problem 1:** Recall that in Homework 3 Problem 1 a Strip Plus [tm] shader was developed in which a triangle strip could be restarted in the middle of a rendering pass by setting the normal in the first two vertices to zero. With Strip Plus it is possible to render several separated triangle strips in a single rendering pass, avoiding the overhead of multiple render passes that would have to be borne using ordinary triangle strips. Strip Plus can only be used, though, when the normal of a provoking vertex is to be applied to the entire triangle. That's not the case when triangles are approximating a curved surface, but lets just consider situations in which we are rendering flat surfaces and so using the provoking vertex normal is fine. Further, lets consider rendering situations in which we always intend to provide the true triangle normal. In that case we don't need to provide a normal at all, the geometry shader can easily provide one. With The Strip Plus + shader the beginning of a triangle strip is identified by setting the $w$ component of the first two vertices to zero. The geometry shader computes geometric normals for the triangles.

($a$) Modify the code in `hw04.cc` and the strip plus shaders in `hw04-shdr.cc` so that the $w$ component of the vertex coordinate is used to mark the first two vertices of a triangle strip and so that the geometry shader computes the normals.

($b$) Compare the amount of data sent by the Strip Plus (Homework 3) and the Strip Plus + shaders.

**Problem 2:** Routine `gs_points_common(bool shadow_volumes)` is similar to the shader in the solution to Homework 3 Problem 2, except that it has a Boolean argument. When that argument is true the geometry shader is supposed to emit shadow volumes rather than triangles forming the protrusions. In the unmodified routine it just returns when `shadow_volumes` is true.

Modify the routine so that it renders efficient shadow volumes. Efficient means that the shadow volume has six faces. That's in contrast to a shadow volume with 30 faces that would result in constructing a 3-face shadow volume for each of the 10 triangles making up the protrusion. For an example of how to construct a shadow volume for a triangle see routine `gs_strip_plus_sv`.

If the problem is solved correctly shadows will match those cast using STRIP-PLUS and the number of fragments will be about $\frac{1}{5}$ the number used by STRIP-PLUS.

# 9 Fall 2018

# LSU EE 4702-1 Homework 1 Due: 6 September 2018

**Problem 0:** Follow the instruction 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 string of beads stretched out vertically and falling to a platform. Pressing `h` (head) will grab or release one end (to be precise, the ball at one end) and pressing `t` (tail) will grab or release the other end. (Actually, those keys toggle the fixed-in-space status of their respective balls.)

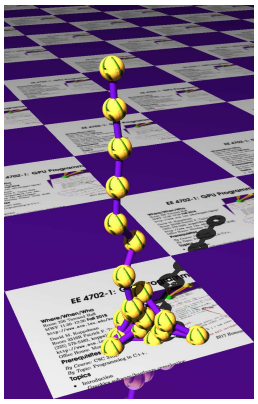
User Interface

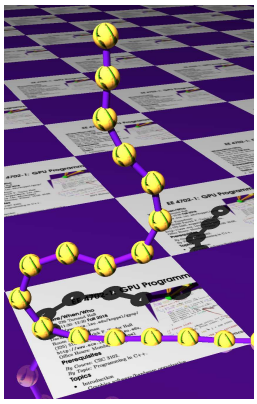
Press digits `1` through `4` to initialize different scenes, the program starts with scene 1. Scene 1 starts with the balls arranged almost vertically, and they fall into a jumbled mess. Scene 2 starts with the balls arranged horizontally, and they will start swinging. Scenes 3 and 4 are initially similar to scene 1, they are to be modified for this assignment.

Press `Ctrl=` to increase the size of the green text and `Ctrl-` to decrease the size. 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).

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 `hw01.cc` for documentation on other keys.

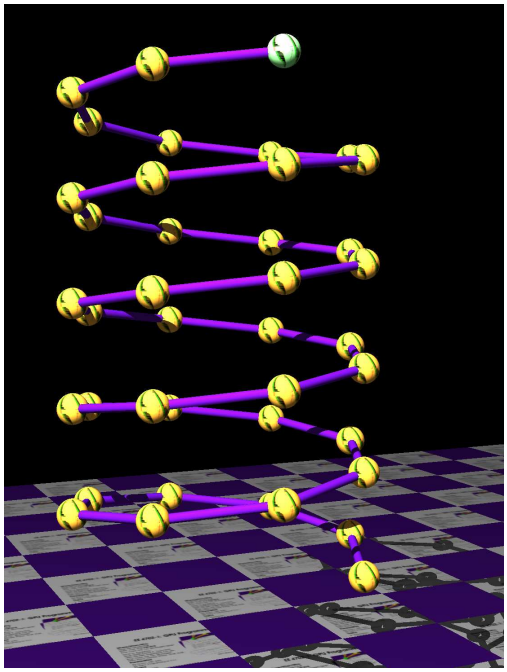




**Problem 1:** Scene 1 starts with the balls falling into a jumbled mess because the links connecting the balls can attach to the balls at any angle, see the left screen-shot to the right. (Press `1` to reset the scene and see the balls fall again.) In the right-hand screen shot the cord connecting the balls appears stiffer. The stiffer cord is modeled by adding springs between balls at a distance of 2, these are in addition to the springs at a distance of 1. (Balls `ball[0]` and `ball[1]` are at a distance of 1, balls `ball[0]` and `ball[2]` are at a distance of 2.

Modify routine `time_step_cpu_hw01` so that it models the stiffer cord as described above. A correct solution will require modifying only three lines of code.

The simulation has two time step routines, `time_step_cpu_hw01`, which needs to be modified for this problem, and `time_step_cpu_orig`, which should be left unchanged. Initially the two routines are identical (except for the comments). Press `v` to switch between the two time step routines.



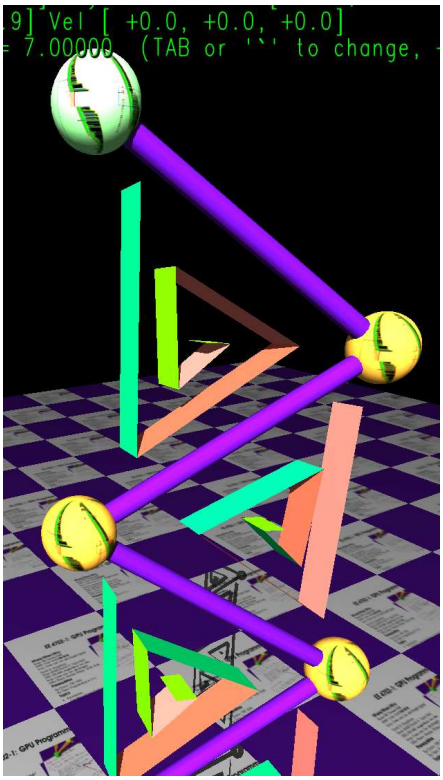
**Problem 2:** Routine `hw01` has three arguments, coordinate `center`, vector `dir`, and scalar `r`. It currently places the balls in a straight line starting at `center` in the direction `dir`. Modify the routine so that it arranges the centers of the balls in a spiral on the surface of a cylinder as described below. See the screenshot to the right. Coordinates `center` and `center + dir` are both on the cylinder axis. The cylinder has a radius of `r`. The first ball should be placed on the cylinder on a point closest to `center`, the last ball should be on the cylinder on a point closest to `center + dir`, and the balls should form a spiral. Set `distance_relaxed` to the distance between adjacent balls.

Routine `hw01` is used both in Scene 3 and Scene 4. In Scene 3 `dir` point straight down, in Scene 4 `dir`, `center`, and `r` are randomly chosen.

The cylinder mentioned above is only meant to describe where the balls should be. But to actually see the cylinder, press `c`.

**Problem 3:** The following is a preview problem, no credit will be given for Homework 1, but it may appear in a future assignment or as a test question. Modify the simulation so that the cylinder is a solid object and so the balls bounce off it, just as they do the platform.

# LSU EE 4702-1 Homework 2 Due: 21 September 2018




**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 string of beads connected in a zig-zag pattern. Pressing `h` (head) will grab or release one end (to be precise, the ball at one end) and pressing `t` (tail) will grab or release the other end. (Actually, those keys toggle the fixed-in-space status of their respective balls.)

## General User Interface

Press digits `1` through `4` to initialize different scenes, the program starts with scene 1. Scene 1 starts with the balls arranged almost vertically. The other scenes are not used in this assignment.

Press `Ctrl=` to increase the size of the green text and `Ctrl-` to decrease the size. 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).

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.

Keys `y` and `Y` toggle the value of Boolean variables `opt_tryout_1` and `opt_tryout_2`. They are intended for debugging and trying things out.

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

## Code Generation

The compiler generates two versions of the code, `hw02` and `hw02-debug`. Use `hw02` to measure performance, but use `hw02-debug` for debugging. The `hw02-debug` version was compiled with optimization turned off and with OpenGL error checking turned on.

## Physics Model, Graphics, and Problem-Specific UI

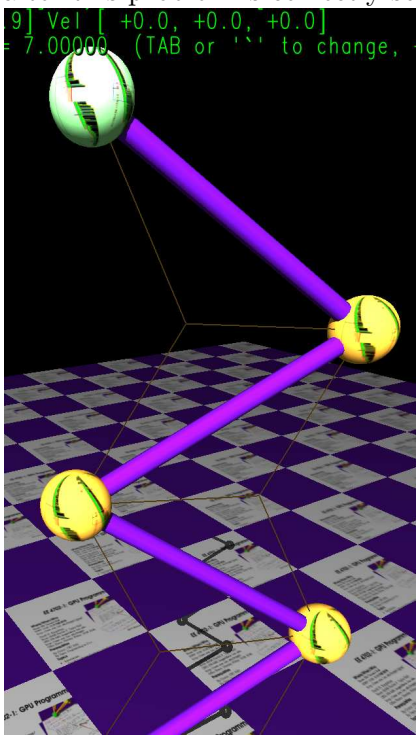
The physics model in the Homework 2 code is similar to the one implemented in Homework 1 Problem 1, in which the cord was given some stiffness by adding a spring between balls at distance 2, except that the relaxed length of all the springs are the same. This has the effect of pulling the cord into a zig-zag pattern.

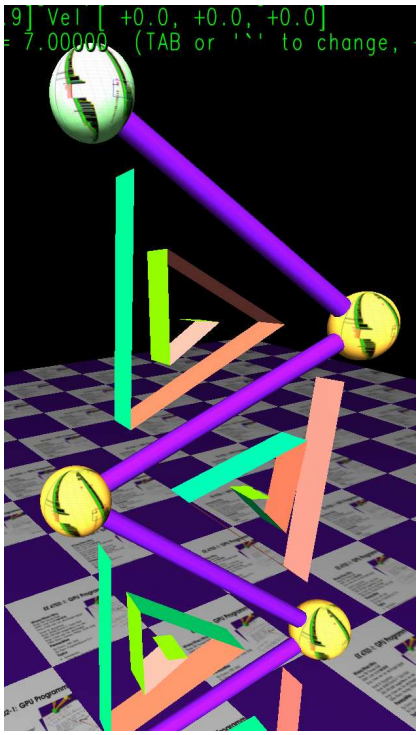
Pressing `f` will cycle through different methods of filling the triangles formed by the balls. The possibilities are none, wye, and louvers, the screen shots in Problem 1 show the intended appearances for wye and louvers. The code in routine `hw02_render` provides the needed fill. The

routine already correctly renders the wye option, in this assignment the louver option will be implemented.

Pressing `w` will toggle shadows on and off. Pressing `W` will toggle the visibility of shadow volumes. Shadow volumes are used to render shadows, see Problem 2.

**Problem 1:** The two screen shots below show Scene 1 rendered with fill option set to Wye and Louver. None and Wye should work with unmodified code, the screen shot for louver shows the appearance after this problem is correctly solved.




Consider a group of three balls forming a triangle. In the Wye image there are faint yellow lines from each ball to the center of the triangle. This is drawn by the code in routine `hw02_render` guarded by `opt_fill == Fill_Wye` .

The louver fill option renders what looks like a strip folded into a triangular spiral. Each fold on the strip occurs on a line between a ball and the triangle center (the yellow lines shown with the wye fill option). The strip starts in the triangle center.

Let $s$ denote the number of segments, which is the value of variable `opt_n_segs`. Let $l_0$, $l_1$, and $l_2$ be the distance from ball 0, 1, and 2, to the triangle center, where ball 0, 1, and 2 refer to *any* three adjacent balls forming a triangle. Let *fold 0* refer to the end of the strip in the center (so there's no real fold). The distance from fold $j$, $0 \geq j \geq s$ to the triangle center is $\frac{0.8j}{s} l_{(j \bmod 3)}$.

Here are some other details:

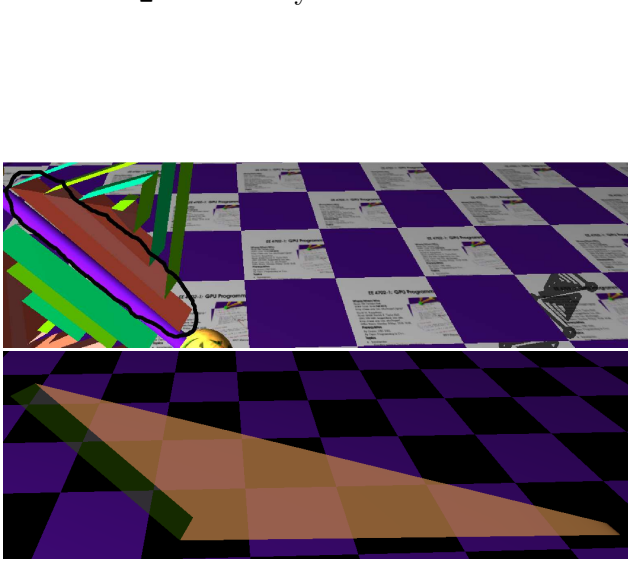
- The width of the louvers should be 0.1 units.
- The front of the louvers should be facing the center of the triangle.
- Use colors in the array `fcolors` for the front colors and `bcolors` for the back colors. Alternate the colors in these arrays to enhance visibility.
- Be sure to set the surface normals correctly.

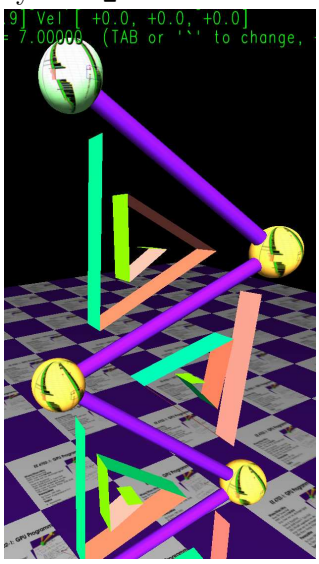
- Use `opt_n_segs` for the number of segments. Make sure that the code reacts when its value is changed with the UI.
- Before solving Problem 2, don't render anything when `shadows` true.

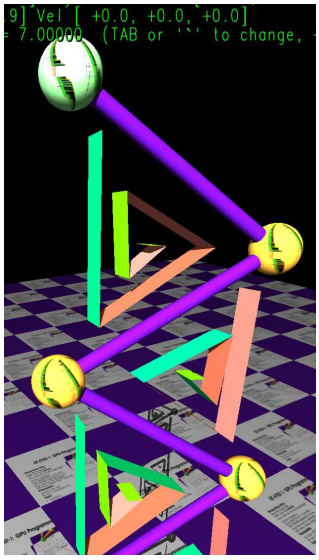
Modify `hw02_render` so that when `opt_fill == Fill_Louver` it renders the louvers as described above.

**Problem 2:** One way to render shadows is by using shadow volumes. A point is in the *shadow volume* of some light and some object if a line from the point to the light passes through the object. That is, everything in the shadow volume is shaded from the light by the object.

One way to create shadows in OpenGL uses triangles (or other primitives) placed on the surface of the shadow volume and facing outside of the volume. The surfaces start at the object being shaded and end some large distance away. Consider the triangle defined by points $P_1$, $P_2$, $P_3$, and a light at location $L$. One surface would be the quadrilateral defined by points $P_1$, $P_2$, $P_2 + d\overrightarrow{LP_2}$, and $P_1 + d\overrightarrow{LP_1}$, where $d$ is some large distance. Similar quadrilaterals can be defined for points $P_2$ and $P_3$ and for points $P_3$ and $P_1$. The pair of screenshots below on the left show a shadow volume for one segment of the louver. Pressing `W` (upper-case) toggles the visibility of shadow volumes. When `opt_fill != Fill_Louver` all shadow volumes are shown, when `opt_fill == Fill_Louver` only shadow volumes rendered by `hw02_render` are shown.



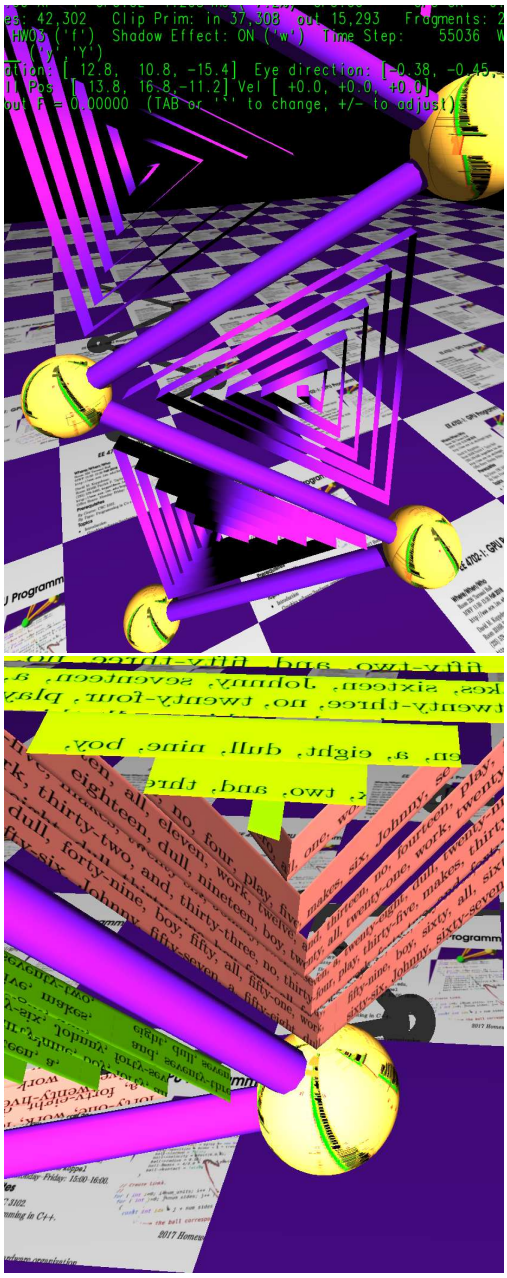




Modify the code in `hw02_render` so that when `shadows` is true and `opt_fill == Fill_Louver` it emits shadow volumes for the louvers. The two screenshots above right show the louvers without and with shadows.

It is important that the shadow volume triangles (or other primitives) face outside the volume.

**LSU EE 4702-1** **Homework 3** **Due: 11 October 2018**



**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 the scene from the solution to Homework 2, in which balls are connected by links in a zig-zag pattern, and the triangles formed by each adjacent group of three balls are filled with a strip folded into a triangular spiral (the louvers). See the screenshot to the upper right (press `f` twice to see that coloring). The screenshot to the lower right is from a completely solved assignment.

## Non-Assignment-Specific User Interface

Pressing `h` (head) will grab or release one end (to be precise, the ball at one end) and pressing `t` (tail) will grab or release the other end. (Actually, those keys toggle the fixed-in-space status of their respective balls.)

Press digits `1` through `4` to initialize different scenes, the program starts with scene 1. Scene 1 starts with the balls arranged almost vertically. The other scenes are not used in this assignment.

Press `Ctrl=` to increase the size of the green text and `Ctrl-` to decrease the size. 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).

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.

Look at the comments in the file `hw03.cc` for documentation on other keys.

## Code Generation and Debug Support

The compiler generates two versions of the code, `hw03` and `hw03-debug`. Use `hw03` to measure performance, but use `hw03-debug` for debugging. The `hw03-debug` version was compiled with optimization turned off and with OpenGL error checking turned on.

Keys `y` and `Y` toggle the value of host Boolean variables `opt_tryout_1` and `opt_tryout_2` and corresponding shader variables `tryout.x` and `tryout.y`. The user interface can also be used to modify host floating-point variable `opt_tryoutf` and corresponding shader variable `tryoutf` using

the `Tab`, `+`, and `-` keys, see the previous section. These variables are intended for debugging and trying things out.

### Shaders and Rendering Passes

The spiral triangle can be rendered using three different sets of shaders, Fixed, Plain, and HW03. The shader in use is shown in the green text near the upper left. Pressing `f` cycles through the shaders. The code in routine `hw03_render` sets up and uses these shaders. For the `Fixed` and `Plain` shaders it will emit each triangular spiral using many triangle strips, the method used in Homework 2; see the code near the comment "Small Strips Rendering." For the `HW03` shader each triangular spiral is rendered using one long triangle strip, see the code near the comment "Large Strip Rendering." By now everyone should see that the Large Strip Rendering code is more efficient since it uses fewer rendering passes (thus suffering less overhead setting up a rendering passes) and because it sends fewer vertices into the rendering pipeline.

### Physics Model and Graphics

The physics model in the Homework 3 code is the same as the one used in Homework 2, in which springs are arranged to maintain the balls in a zig-zag pattern.

Pressing `w` will toggle shadows on and off. The triangular spirals do not cast shadows, and shadows are not part of this assignment.

**Problem 1:** When run with the Fixed shader the triangular spiral is colored green (okay, chartreuse) on the outside and salmon on the inside. These colors are obtained by setting the material front and back properties. With the HW03 shaders the color is a purple gradient, see the screen shot at the beginning of the assignment. The gradient is due to the interpolation of the normals from one fold of the triangle strip to the next.

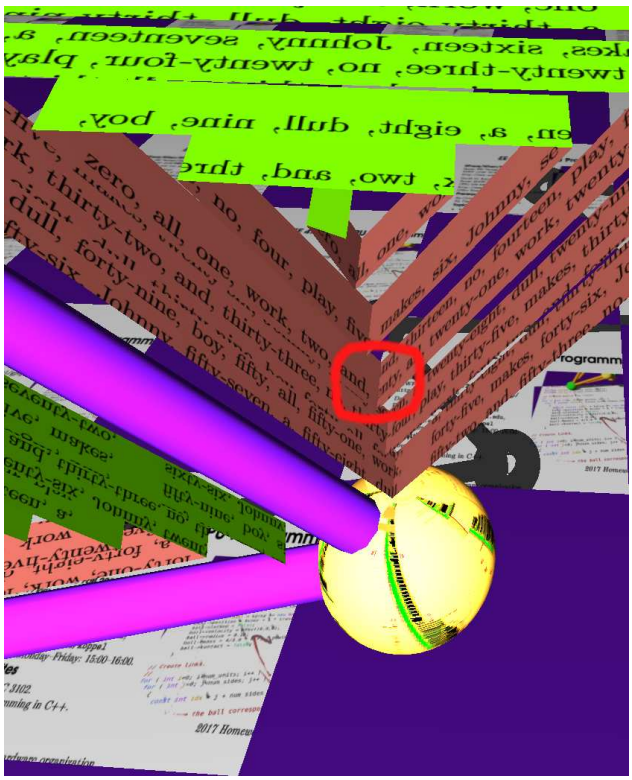
(*a*) Modify the `hw03` vertex and fragment shader related code in `hw03-shdr.cc` so that the correct normal is used to compute lighting. *Hint: A correct solution requires modifying a declaration in the shader code. That's one word to be inserted.*

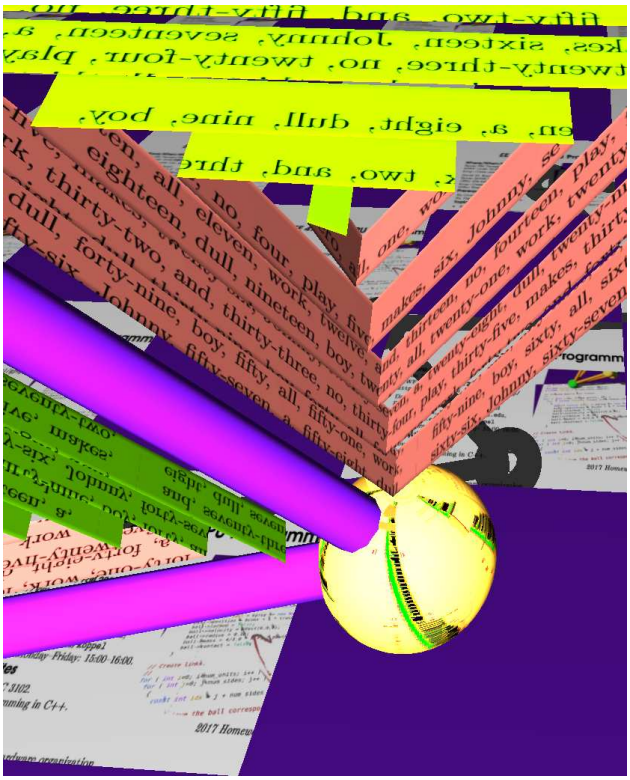
(*b*) Modify the `hw03` vertex and fragment shaders in `hw03-shdr.cc` so that the front and back colors are used when the `hw03` shaders are active. The front and back colors are set by `glMaterialfv` calls in `hw03_render`.

- Use the OpenGL Shading Language documentation to find the built-in names of the uniform variables holding these values. (Look for the Built-In Variables chapter.)
- **Do not** declare your own variables for the front and back colors. Ordinarily declaring your own variables for material properties would be a good thing in this situation because it avoids the use of deprecated functionality. However the point of this problem is familiarization with OGSL documentation.
- Don't needlessly increase the amount of data sent between shaders.

**Problem 2:** Notice that in routine `hw03_render` (in `hw03.cc`) a texture unit is set up with a texture object named `texid_spiral_image`. Modify code in `hw03_render` and the `hw03` shaders in `hw03-shdr.cc` so that the texture is applied to the spiral so that the start of the text on the texture image is at the center of the spiral and so that text can be continuously read following the spiral from the center on outward. The text includes consecutive integers starting at zero.

The screenshot to the lower right shows the texture correctly applied (and with other parts of this assignment solved). The screenshot to the lower left shows a flaw. This problem can be solved two ways, the first will receive only partial credit, a correct solution to the second will receive full credit.





For both solutions:

- The size of the text must be the same on all parts of the spiral. That is, the inner segments **should not** show squished text and the aspect ratio should be consistent.

- It should be possible to continuously read the text starting from the spiral center. That is, it should be as though the texture image was cut into horizontal strips and these strips were glued together to make one long strip, which was folded into the spiral. The text at the end of the spiral does not need to reach the text at the end of the texture image.

(*a*) **For Partial Credit:** Modify the code in `hw03_render` in and around the "Small Strips Rendering" block to emit texture coordinates that will result in the texture appearing approximately as described above. In limiting changes to `hw03_render` this way flaws such as the one circled in red above are unavoidable. **There is no need to solve this part if the subproblem below is solved.**

(*b*) Modify the code in `hw03_render` in and around the "Large Strip Rendering" block and in the `hw03` shaders in file `hw03-shdr.cc` so that the texture appears as described above. The texture coordinates or equivalent emitted for this problem can also be used to solve the next problem. For full credit there should be no repetition and no gaps (except for the end of the spiral not reaching the end of the texture image).

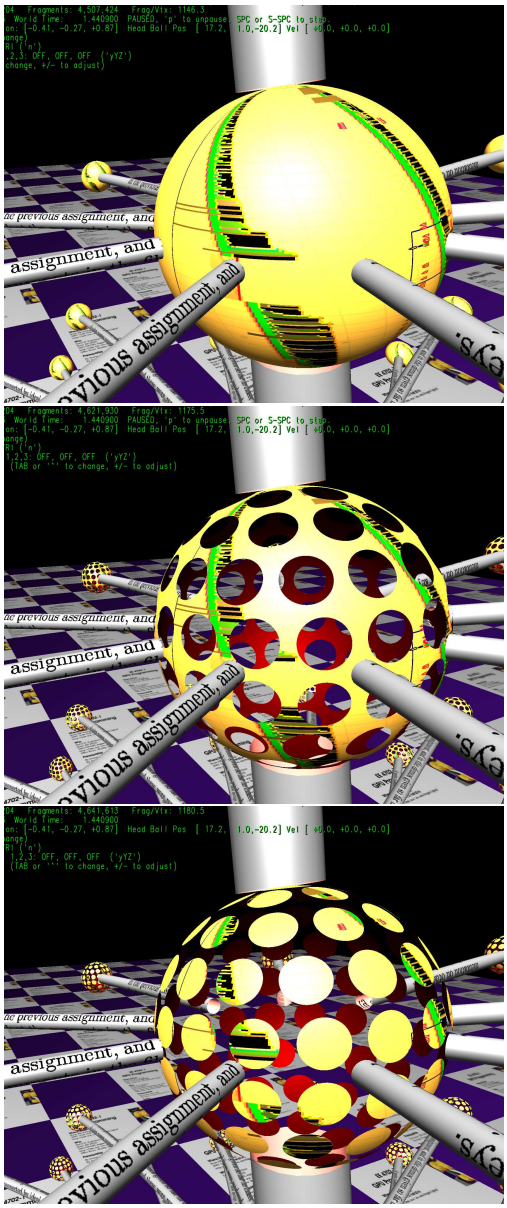
**Problem 3:** Modify the code in `hw03_render` in and around the "Large Strip Rendering" block and in the `hw03` shaders in file `hw03-shdr.cc` so that the edges of the strip appear rounded, like the edges of a metal tape measure. The screenshot on the upper right has the rounding effect, the one on the upper left does not have the rounding effect.

Do so by rotating the normal used for lighting away from the center of the spiral. Perform the rotation by combining the normal currently used for lighting with the triangle normal (`nz` in

`hw03_render`). The blending should be done near the edges of the strip, elsewhere just use the existing normal for lighting.

- There is starter code in the fragment shader to solve this problem. Finish it.
- Make sure that the normal is computed correctly. The more closely the normal points to the light, the more brightly lit the object. Pay attention to the shading in the screenshot for the correct solution.
- Do not waste data bandwidth delivering the triangle normal to the shaders.
- Consider using the texture coordinates from the previous problem to detect distance from the strip edge.

**LSU EE 4702-1** **Homework 4** **Due: 16 November 2018**



**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 scene from the links code, the one showing a vaguely tree-like form, the *silly tree*, constructed from flexible links and balls. See the screenshot to the upper right. The screenshots at the middle and lower right were taken from correctly solved code.

## Non-Assignment-Specific User Interface

Pressing h (head) will grab or release one end (to be precise, the ball at one end) and pressing t (tail) will grab or release the other end. (Actually, those keys toggle the fixed-in-space status of their respective balls.)

Press digits 1 through 4 to initialize different scenes, the program starts with scene 1. Scene 1 starts with the balls arranged in the tree-like form.

Press Ctrl = to increase the size of the green text and Ctrl - to decrease the size. 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).

The + and - keys can be used to change the value of certain variables. These variables control things like light intensity and options needed for this assignment. The variable currently affected by the + and - keys is shown in the bottom line of green text next to VAR. Pressing Tab cycles through the different variables.

Look at the comments in the file `hw04.cc` for documentation on other keys.

## Code Generation and Debug Support

The compiler generates two versions of the code, `hw04` and `hw04-debug`. Use `hw04` to measure performance, but use `hw04-debug` for debugging. The `hw04-debug` version was compiled with optimization turned off and with OpenGL error checking turned on.

Keys y, Y, and Z toggle the value of host Boolean variables `opt_tryout1`, `opt_tryout2`, and `opt_tryout3` and corresponding shader variables `tryout.x`, `tryout.y`, and `tryout.z`. The

user interface can also be used to modify host floating-point variable `opt_tryoutf` and corresponding shader variable `tryoutf` using the Tab, +, and - keys, see the previous section. These variables are intended for debugging and trying things out.

### Sphere Rendering

The sphere can be rendered using two methods, the *tessellated sphere*, and the *true sphere* methods. The option in use is toggled by pressing (lower case) z and the current setting is shown next to Sphere in the green text. Both TRUE and HW04 indicate the true sphere method, TESS indicates the tessellated sphere method.

The tessellated sphere method is the one initially covered in class in which the sphere is approximated by triangles. A buffer object holds the triangles' vertex coordinates arranged for a triangle strip rendering pass. Those coordinates are in a local coordinate space in which the sphere center is at the origin and the sphere has a radius of 1. The number of triangles is based on variable `opt_slices`, larger values result in a closer approximation to a sphere.

The rendering pass for both tessellated and true spheres is done in routine `render_bunch_render` in file `shapes.h`. For both cases three additional buffer objects are prepared, one holding the sphere orientation, `sphere_rot`, one holding the sphere coordinates and radius, `sphere_pos_rad`, and one holding the sphere color `sphere_color`. There is one element in each buffer object for each sphere. An element of `sphere_rot` is a $4 \times 4$ matrix, which rotates a coordinate from local to global space (but does not perform translation or scaling). Because of differences in matrix layout, the corresponding matrix in shader code rotates from global to local space. Each element of `sphere_pos_rad` is a 4-element vector. The first three components form the coordinate of the sphere center in global space. The 4th component is the sphere's radius. Each element of `sphere_color` is a 4-element vector, which holds the red, green, blue, and alpha color components.

The tessellated spheres are rendered in an instanced pass. The pipeline inputs are taken from the vertex buffer object, and the number of instances is set to the number of spheres. The shaders in `hw04-shdr-sphere.cc` render the sphere (and its shadow volumes). The input to the vertex shader, `vs_main_instances_sphere()`, is a triangle (sphere surface) coordinate in local space. It converts it into global (object) space using the instance ID to get the location, orientation, and radius from the buffer objects described above. The color is retrieved by the fragment shader.

The rendering pass for the true spheres is also started in `render_bunch_render`. In this rendering pass there is no vertex shader input other than the vertex ID. The input primitive is a point and the number of vertices is set equal to the number of spheres. This pass uses code in `hw04-shdr-hw04.cc`. The geometry shader, `gs_main`, computes the coordinates of a square that will frame the sphere as seen from the user, and emits two triangles that form this square. The input to the fragment shader, `fs_main`, is the coordinate on this square corresponding to the fragment location. It computes the point on the sphere surface intersected by a line from the eye to the fragment location. If there is no intersection the fragment is discarded, otherwise a normal and texture coordinates are computed. To compute the texture coordinates the sphere rotation must be taken into account.

The code in `fs_main` has some variables to be used in this assignment, they are described in the problems.

### Graphics and Performance Investigation Options

The user interface can be used to toggle various rendering options and for generating a screenshot.

Pressing F12 will write a screenshot to file `hw04.png` (`FOO.png`, where `FOO` is the name of the executable, such as `hw04-debug`). Any existing screenshot will be silently overwritten so be sure to rename files that you want to keep.

The rendering of shadows is toggled by o and the rendering of reflections it toggled by r. Their state is shown in the green text next to Effect:. Pressing n will toggle how surface normals are computed for tessellated spheres, the possibilities are to use the triangle normal or the sphere normal. The use of the triangle normals makes it easier to see the triangles from which the sphere was tessellated.

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.

### Display of Performance-Related Data

The top green text line shows performance in various ways. XF shows the number of display frames per frame buffer update. An ideal number is 1. A 2 means that two display frame update were done in the time needed to update the frame buffer once, 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).

**Problem 1:** The code in `fs_main` (in file `hw04-shdr-hw04.cc` so that when `holes` is true (see the code) the sphere is not visible in the locations described below and when `lenses` is true the sphere is only visible in those same locations, in both cases show the inside of the sphere when appropriate. See the screen shots at the top of the assignment.

Variable `opt_n_holes_eqt` specifies roughly how many holes there should be along a circumference (such as the equator on a globe). Distribute the holes roughly evenly and so that 80% of a line along the sphere connecting the centers of two adjacent holes would be over holes (40% over one hole, 20% on the rendered surface, 40% over the other hole).

The hole locations should rotate with the sphere, just as texture coordinates do. That is, the part of a texture that touches a hole should not change as the sphere rotates.

Note that the fragment shader computes the same kind of `eta` and `theta` angle values used to construct the sphere, and from those computes the texture coordinates. Those same `eta` and `theta` values can be used to find hole locations. For example, if $n$ is the number of holes along the equator, the $\eta_h$ value for a hole might be $\eta_h = \lfloor \eta \frac{n}{\pi} + \frac{1}{2} \rfloor \frac{\pi}{n}$, where $\eta$ is the `eta` value for the fragment. The $\eta_h$ and $\theta_h$ value can be used to find the surface coordinates of a hole, and from that one can find the distance from the hole center to the fragment. The fragment is in the hole if this distance is below some threshold.

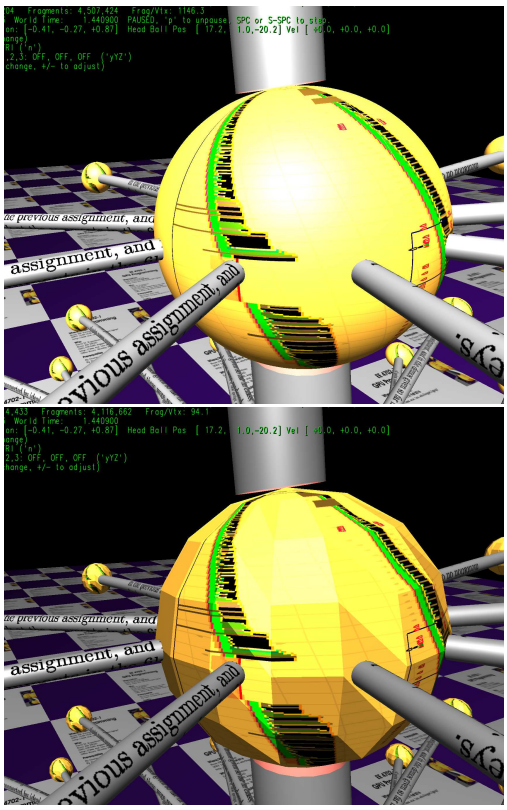
($a$) Modify `fs_main` so that the sphere surface is not rendered where there are holes when `holes` is true, or so that the sphere surface is rendered only where there are holes when `lenses` is true.

($b$) If there is a hole in the front surface of a sphere render the back surface of the sphere. When doing so compute the correct normal and check for the presence of a hole. If the back surface is visible set `front` to false.

Pay attention to:

- Shader compilation and link errors. These are sent to stdout when the program starts to run.
- Coordinate spaces. The code uses suffix `_l` for sphere local space, `_e` for eye space and `_o` for object (global) space.

**LSU EE 4702-1** **Homework 5** **Due: 14 November 2018**



**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 scene from the links code, the one showing a vaguely tree-like form, the *silly tree*, constructed from flexible links and balls. See the screenshot to the upper right. The screenshots to the right. This assignment uses the Homework 4 code. Homework 4 does not need to be solved correctly to run this assignment.

## Non-Assignment-Specific User Interface

Pressing h (head) will grab or release one end (to be precise, the ball at one end) and pressing t (tail) will grab or release the other end. (Actually, those keys toggle the fixed-in-space status of their respective balls.)

Press digits 1 through 4 to initialize different scenes, the program starts with scene 1. Scene 1 starts with the balls arranged in the tree-like form.

Press Ctrl = to increase the size of the green text and Ctrl - to decrease the size. 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).

The + and - keys can be used to change the value of certain variables. These variables control things like light intensity and options needed for this assignment. The variable currently affected by the + and - keys is shown in the bottom line of green text next to VAR. Pressing Tab cycles through the different variables.

Look at the comments in the file `hw04.cc` for documentation on other keys.

## Code Generation and Debug Support

The compiler generates two versions of the code, `hw04` and `hw04-debug`. Use `hw04` to measure performance, but use `hw04-debug` for debugging. The `hw04-debug` version was compiled with optimization turned off and with OpenGL error checking turned on.

Keys y, Y, and Z toggle the value of host Boolean variables `opt_tryout1`, `opt_tryout2`, and `opt_tryout3` and corresponding shader variables `tryout.x`, `tryout.y`, and `tryout.z`. The user interface can also be used to modify host floating-point variable `opt_tryoutf` and corresponding shader variable `tryoutf` using the Tab, +, and - keys, see the previous section. These variables are intended for debugging and trying things out.

## Sphere Rendering

The sphere can be rendered using two methods, the *tessellated sphere*, and the *true sphere* methods. The option in use is toggled by pressing (lower case) z and the current setting is shown next to Sphere in the green text. Both TRUE and HW04 indicate the true sphere method, TESS indicates the tessellated sphere method.

The tessellated sphere method is the one initially covered in class in which the sphere is approximated by triangles. A buffer object holds the triangles' vertex coordinates arranged for a triangle strip rendering pass. Those coordinates are in a local coordinate space in which the sphere center is at the origin and the sphere has a radius of 1. The number of triangles is based on variable `opt_slices`, larger values result in a closer approximation to a sphere.

The rendering pass for both tessellated and true spheres is done in routine `render_bunch_render` in file `shapes.h`. For both cases three additional buffer objects are prepared, one holding the sphere orientation, `sphere_rot`, one holding the sphere coordinates and radius, `sphere_pos_rad`, and one holding the sphere color `sphere_color`. There is one element in each buffer object for each sphere. An element of `sphere_rot` is a $4 \times 4$ matrix, which rotates a coordinate from local to global space (but does not perform translation or scaling). Because of differences in matrix layout, the corresponding matrix in shader code rotates from global to local space. Each element of `sphere_pos_rad` is a 4-element vector. The first three components form the coordinate of the sphere center in global space. The 4th component is the sphere's radius. Each element of `sphere_color` is a 4-element vector, which holds the red, green, blue, and alpha color components.

The tessellated spheres are rendered in an instanced pass. The pipeline inputs are taken from the vertex buffer object, and the number of instances is set to the number of spheres. The shaders in `hw04-shdr-sphere.cc` render the sphere (and its shadow volumes). The input to the vertex shader, `vs_main_instances_sphere()`, is a triangle (sphere surface) coordinate in local space. It converts it into global (object) space using the instance ID to get the location, orientation, and radius from the buffer objects described above. The color is retrieved by the fragment shader.

The rendering pass for the true spheres is also started in `render_bunch_render`. In this rendering pass there is no vertex shader input other than the vertex ID. The input primitive is a point and the number of vertices is set equal to the number of spheres. This pass uses code in `hw04-shdr-hw04.cc`. The geometry shader, `gs_main`, computes the coordinates of a square that will frame the sphere as seen from the user, and emits two triangles that form this square. The input to the fragment shader, `fs_main`, is the coordinate on this square corresponding to the fragment location. It computes the point on the sphere surface intersected by a line from the eye to the fragment location. If there is no intersection the fragment is discarded, otherwise a normal and texture coordinates are computed. To compute the texture coordinates the sphere rotation must be taken into account.

The code in `fs_main` has some variables to be used in this assignment, they are described in the problems.

## Graphics and Performance Investigation Options

The user interface can be used to toggle various rendering options and for generating a screenshot.

Pressing F12 will write a screenshot to file `hw04.png` (`FOO.png`, where `FOO` is the name of the executable, such as `hw04-debug`). Any existing screenshot will be silently overwritten so be sure to rename files that you want to keep.

The rendering of shadows is toggled by o and the rendering of reflections it toggled by r. Their state is shown in the green text next to Effect:. Pressing n will toggle how surface normals are computed for tessellated spheres, the possibilities are to use the triangle normal or the sphere

normal. The use of the triangle normals makes it easier to see the triangles from which the sphere was tessellated.

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.

### Display of Performance-Related Data

The top green text line shows performance in various ways. XF shows the number of display frames per frame buffer update. An ideal number is 1. A 2 means that two display frame update were done in the time needed to update the frame buffer once, 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).

**Problem 1:** Compare the tessellated and true sphere code as requested below. The true-sphere method requires more work in the fragment shader, while the tessellated-sphere method requires that more work be done in the vertex and fragment shaders. The impact on performance depends upon a number of factors, so that in some situations the tessellated-sphere code is faster and in others the true sphere code is faster. The true-sphere code will always have the best appearance but the tessellated-sphere code may be good enough or as good in some situations.

For the questions below set up the scene for best measuring performance of the sphere rendering code. This includes setting graphics options and choosing the eye location and scene. Review the description of the graphics and other options described in the first part of this assignment.

For this problem don't use the HW04 sphere rendering option, use TRUE instead, since we don't want to measure the impact of the holes (or have to wait for a correctly solved hole shader).

(*a*) Find a view in which the true sphere code takes more time than the tessellated sphere code. Use the default number of sphere slices. Include a screenshot and explain the following:

- Explain why the true sphere code takes longer in this view but not others.
- Explain what it is about the view that results in the true-sphere code taking longer.
- Indicate whether this is a fair comparison. The comparison would not be fair, for example, if a minor modification to the true-sphere code would make it faster in this case, or if the tessellated spheres' appearance was unusually bad.

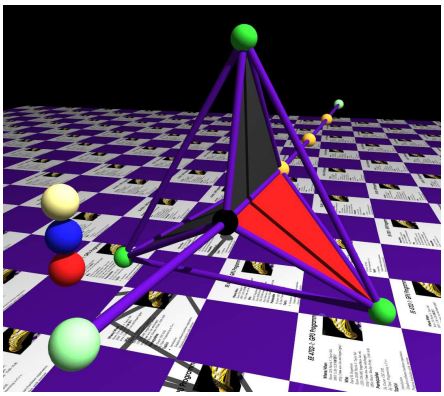
(*b*) In the initial view, the one showing the silly tree, the tessellated sphere code should take more time than the true-sphere code. A critic might argue that this is not a fair comparison because of the way `opt_sphere_slices` is set *for the various spheres. Hint: think about what "for the various spheres" is intended to convey.*

- Explain why this initial view might be unfair.

- Set up a new view which is fair, describe why you think it's fair, and indicate the difference in performance.
- Include a screenshot for each view.

# 10 Fall 2017

## LSU EE 4702-1 Homework 1 Due: 20 September 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 string of beads stretched out horizontally with either end fixed in space by some invisible force, see the illustration to the right. 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.)

The cable includes a vaguely paddle-wheel like thing that includes triangular sheets of material which will be called blades. The blades are red on one side and dark gray on the other. Each corner of the blade connects to a different colored ball. The corner connected to the black ball is called the *base vertex*, the corner connected to the green ball is called the *circ vertex*, and the corner connected to the yellow ball is called the *apex vertex*.

The scene also includes (initially) three stationary balls, colored khaki, blue, and red. These are called *markers* and are intended for debugging purposes. See routine `World::render_p1` for an example of how to use the marker balls. Another aid for debugging are Boolean variables `opt_tryout1` and `opt_tryout2`. Pressing `y` toggles `opt_tryout1` between `true` and `false` and `Y` toggles `opt_tryout2`. The state of these variables is shown in the penultimate line of green text. Feel free to use the marker balls and tryout variables to help write and debug your code.

There three versions of the scene, *Experiment*, *Problem 1*, and *Problem 2*. The version currently being displayed is shown in the penultimate line of green text (not shown in the illustration). Pressing `v` cycles between the different versions. When the Experiment version is active the code in `World::render_p0` is executed. Changes to this routine won't be graded, it's intended to try things out or at least to have something clean to look at after having extensively modified other routines. The code in `World::render_p1` and `World::render_p2` are active when scenes Problem 1 and Problem 2 are active. The changes to be made in these routines are described in the problems below.

Press digits `1` through `2` to initialize different scenes, the program starts with scene 2. Scene 1 was described above, scene 2 shows a . . . a . . . make up your own name for it. Scene 2 is not part of this assignment.

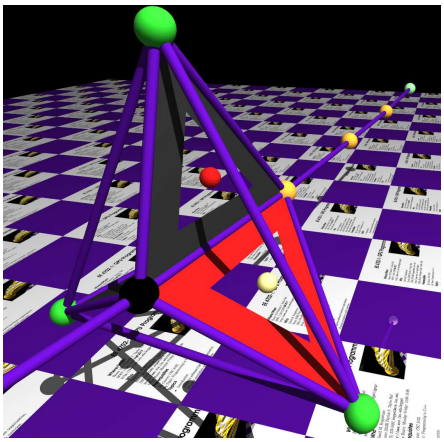
Initially the arrow keys, PageUp, and PageDown can be used to move around the scene. Press (lower-case) `b` and then use the arrow and page keys to move the first ball around. Press `l` to move the light around and `e` to move the eye (which is what the arrow keys do when the program starts).

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.

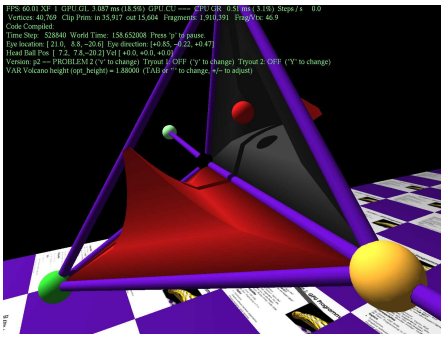
Look at the comments in the file `hw01.cc` for documentation on other keys.

*There is a problem on the next page.*

**Problem 1:** Version Problem 1 is initially the same as the Experiment version, except that a marker ball is placed at the center of each blade. Modify the code in `World::render_p1` so that the blade has a triangular hole in the center, see the illustration to the right. Do so by using a triangle strip for each blade.



**Problem 2:** Version Problem 2 is initially the same as the Experiment version, except that a marker ball is placed at the center of each blade. Modify the code in `World::render_p2` so that the blade is shaped sort of like a volcano, as in the illustration to the right and as described below.



Let $a$, $b$, and $c$ denote the coordinates of the apex, base, and circ vertices. Let $m$ denote the coordinate of the center of the triangle (shown as `mid` in the code). Let $\overrightarrow{n}$ denote the triangle normal and following our usual notation, $\overrightarrow{am}$ is a vector from $a$ to the triangle center.

The shape of the volcano will be defined by the parametric line $P_a(t) = a + t^e\overrightarrow{am} + td\overrightarrow{n}$, where $e$ and $d$ are constants. Note that $P_a(0)$ is point $a$ (the apex) if $e = 1$ and $d = 0$, point $P_a(1)$ is the center of the triangle. For other values of $e$ and $d$ the line ascends the volcano as $t$ increases. Define $P_b(t)$ and $P_c(t)$ similarly. For this assignment set $e$ to the value of `opt_e` (default, 0.3) and $d$ to the value of variable `opt_dist`. The volcano is rendered by varying $t$ from 0 to 0.9. (If $t$ varied from 0 to 1 it would be a mountain, not a volcano.) Use variable `opt_layers` for the number of values of $t$ (evenly spaced between 0 and 0.9).

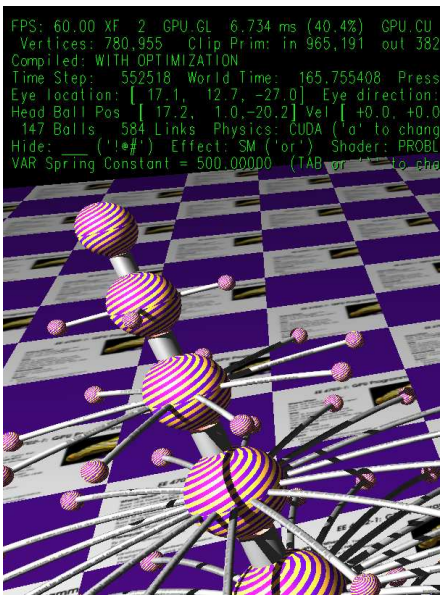
($a$) Modify the code in `World::render_p2` so that it renders each side of the volcano using a triangle strip. (One side is between points $a$ and $b$, one side is between $b$ and $c$, and one is between $c$ and $a$.)

($b$) Choose the normals based on the shape of the volcanoes defined by parametric equation. If this is done correctly the volcano will be smoothly colored consistent with the lighting, as illustrated.

Note that using the normal of one of the triangles adjacent to a vertex as the vertex's normal is an approximation. An exact surface normal can be found by taking the cross product of two vectors that lie on the surface. Taking the derivative of the parametric equation with respect to $t$ provides a vector along the surface. So $\frac{dP_a(t)}{dt}$ gives a vector on the $ca$ and $ab$ sides.

# 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.)




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 $\omega$ is the value of variable `opt_omega`. The line $P(\Delta_\theta, t)$ for $0 \le t \le 1$ 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 $0 \le t \le 1$, 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.

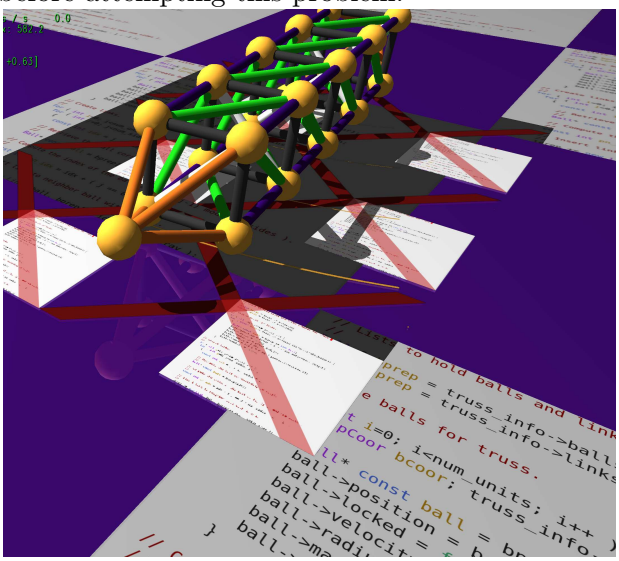
(*b*) Render a sphere by rotating and reflecting this spiral. A hemisphere can be rendered by rendering the spiral $p$ times and rotating the spiral by $\frac{2\pi}{p}i$ on the $i$'th rendering, where $p$ is the value of `opt_spirals`. When $i$ is odd use the ball's own color, when $i$ is even use purple 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.

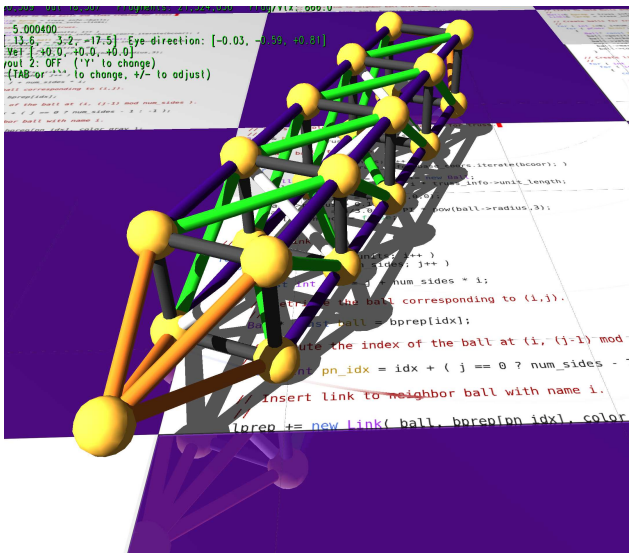
- As in Problem 1, render all of the spheres in the `balls` list.
- Make sure that the balls rotate appropriately. In scenes other than 1 it is easy to see whether ball rotation is correct.
- Don't forget to alternate color between purple and the ball's color.
- As with the first problem, avoid doing 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.

**LSU EE 4702-1 Homework 3 Due: XX 23 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. This assignment is in directory `hw/gpup/2017/hw03`. Compile and run the homework code unmodified. This assignment is based on 2015 Homework 3, and uses some of its code. Please see the 2015 assignment and its solution before attempting this problem.





The code should start in Scene 2, showing a truss rotating around the platform leaving colored scuff marks. The screenshot to the upper left shows Scene 2 before any changes are made and the screenshot to the right shows Scene 2 after the assignment has been solved (though not perfectly). Notice that the balls sliding along the platform appear to have smeared the text on the non-mirrored tiles. The image to the upper right has a minor imperfection: the faint black lines between overlays.

Pressing `w` will spin the object around some center, in Scene 2 that's the head ball. Pressing `w` several times will spin the Scene 2 truss fast enough to wobble and bounce hard against the platform, providing a means to test the smearing to be implemented in Problem 2.

Pressing `h` (head) will release or lock the head ball and pressing `t` (tail) will release or lock the tail ball. Scene 2 starts with the head ball locked. Pressing digits `1` through `4` will initialize different scenes, the program starts with Scene 2.

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).

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.

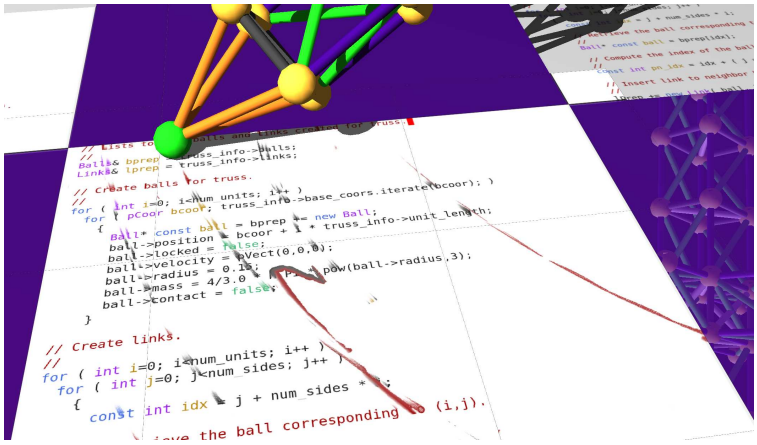
Look at the comments in the file `hw03.cc` for documentation on other keys.

**Problem 1:** The code in routine `My_Piece_Of_The_World::platform_obj_to_tile` converts object-space coordinates into *tile* coordinates, where tile refers to the tiles visible on the platform. (Which are not to be confused with the overlays.) A tile coordinate of $\begin{bmatrix} 0.1 \\ 0 \\ 3.5 \end{bmatrix}$ indicates the tile in the first column of tiles (in the $x$ direction) and the fourth in row of tiles (in the $z$ direction). The coordinate is near the left-hand side (along the $x$ axis) and in the middle along the $z$ axis. The routine also sets the $w$ component to true if the tile has a texture (showing some code from the assignment) and to false if the tile is mirrored. The routine checks itself for correctness, which should help in solving this problem.

Modify the routine so that it uses a transformation matrix to compute the tile coordinate. The transformation matrix should compute the $x$ and $z$ components, but the existing code should set the $w$ component. The $y$ component can be set to anything. A solution should contain a line like `rv = my_matrix * pos_obj;` , preceded by code setting `my_matrix`.

**Problem 2:** Modify routine `World::time_step_cpu` so that instead of writing scuff marks, it smears the underlying text when balls slide along the platform. Search for "Problem 2" to find the place to put the solution. The `Ball` class has a vector called `glop` that can be used to store color values that the ball has picked up. Other members can be added to `Ball`.

**Problem 3:** The code in `My_Piece_Of_The_World::po_get` initializes a new overlay. It is called when a ball touches a part of the platform which does not yet have an overlay. The screenshot to the right is from a run of the solution, in which the new overlay's texture is set up to match the texture that it covers. In the screenshot the texture created for the overlay is shown in a lighter shade of white than the texture it covers. (The difference in shade is automatic.) In the unmodified code the new overlay's texture shows a big red ex and includes the code image (see the screenshot on the first page), but the code image is not positioned so that it matches what is underneath.



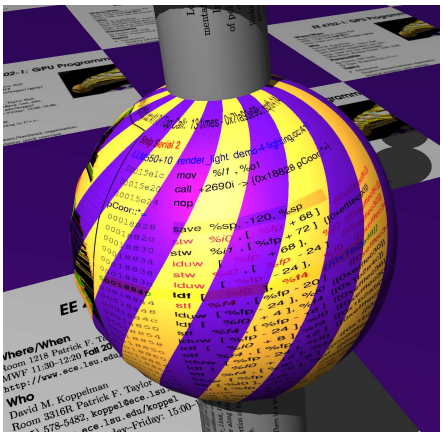
In this problem modify the texture initialization code in `po_get` so that the texture does not include the red ex and so that the code image is zoomed and positioned to match what it is covering. The texture for the code image is in array `syl_pixels`. Either use this array, or if you like use OpenGL calls to read back pixels from the texture object used for the textured tiles.

The parts of the overlay texture over mirrored tiles should have the alpha value set to zero (and the red, green, and blue components set to zero). The parts of the overlay texture over the image tiles should show the same image as the tiles. Use `platform_obj_to_tile` to determine what kind of tile (mirrored or image) a coordinate is over, and to determine where in the tile a coordinate is located. From that one can determine what parts of the `syl_pixels` image to load into the texture. See `sample_tex_make` for how to load the image into a texture.

In principle this is an easy problem, but it requires that one keep unflagging attention on the multiple coordinate spaces involved. Those spaces are: Object-space coordinates, for example, used for `ball->position`. Tile coordinates (see Problem 1), which can be used to determine which tile a point is over, and more importantly the relative location in a tile (such as upper-left, center, bottom-center, etc.). The relative location determines which part of the code image to use. There is the pixel coordinate space of the image (used for reading from `syl_pixel`) and the texel coordinate space of the overlay texture.

## LSU EE 4702-1 Homework 4 Due: 6 November 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 1 after this assignment is completed.



### Scene Interaction and User Interface

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.)

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).

### Display of Performance-Related Data

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).

### User Interface for Navigation and Interaction

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 hw04.cc and hw04-shdr.cc for documentation on other keys.

### Switching Between Shaders

The program can use three different sets of shaders for the spheres, SPHERE, PROBLEM 0 (render_p1), and PROBLEM 1 (render_p1). Pressing v switches between them.

### Description of Instanced Rendering in render_p1 and vs_main_p0

The code in render_p1 contains code adapted from the solution to Homework 2. The routine computes the coordinates of a spiral slice of a sphere and places it in a buffer object. There are two differences with Homework 2. First, the $w$ component of a spiral slice coordinate is set to the value of $\theta$ with which it was computed. The vertex shader uses this to compute texture coordinates. Second, unlike Homework 2, the slice goes from pole to pole (as opposed to pole to equator as in Homework 2).

The routine then puts information about the balls in three buffer objects, pos_rad, rotations, and colors. Each element of buffer object pos_rad is a pCoor (vec4 in shader code). The x, y, and z members hold the world-space coordinate of the ball center and the w member holds its radius. The rotations buffer object holds the ball orientation (as a $4 \times 4$ transformation matrix) and colors holds the ball color. The render_p1 routine binds these buffer objects to OpenGL array buffers, so that they can be accessed in shader code.

Next, the routine installs either the *Problem 0* shaders, vs_main_p0, gs_main_p0, and fs_main_p0 or the *Problem 1* shaders, vs_main_p1, gs_main_p1, and fs_main_p1, both sets of which are in file hw04-shdr.cc. It then starts an instanced rendering pass using slice coordinates as vertex shader inputs.

Your solutions should be placed in the Problem 1 shaders. Leave the Problem 0 shaders unchanged or use them for minor experiments.

### Instanced Rendering

In an instanced rendering pass a set of vertices, $\mathcal{V}$, is sent through the rendering pipeline $I$ times, where $I$ is an integer. In render_p1 the OpenGL command glDrawArraysInstanced starts the instanced rendering pass, the last argument to this command is the number of instances. The first time that $\mathcal{V}$ is sent through the rendering pipeline vertex shader built in variable gl_InstanceID will be set to 0, the second time gl_InstanceID will be set to 1, etc.

The code in vs_main_p0 uses gl_InstanceID to retrieve the ball's position and orientation and uses these to convert vertex shader input gl_Vertex, the sphere normal in the sphere local coordinate space, into an object-space coordinate, vertex_o. It then performs eye- and clip-space conversions on vertex_o.

The code in vs_main_p0 computes texture coordinates based upon the sphere's local $y$ coordinate and the $\theta$ value provided. Study vs_main_p0 to understand how that works.

### Debugging Help

Boolean variables opt_tryout1 and opt_tryout2 can be used for debugging on the CPU. Pressing y and Y toggles their state. In the shader code use tryout.x and tryout.y for these two variables.

To see the difference between your code and the original code issue the Emacs command Ctrl x v = or from the shell the command git diff.

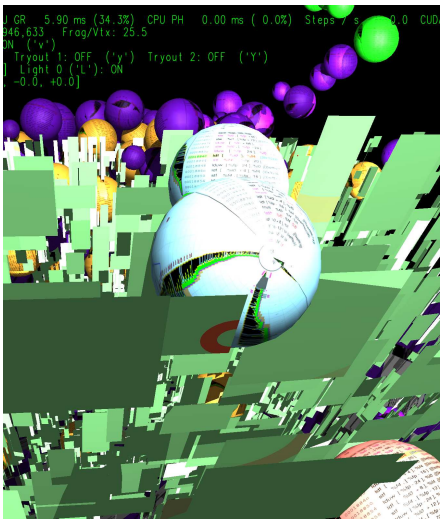
**Problem 1:** Modify the code in vs_main_p1, gs_main_p1, fs_main_p1, and possibly render_p1 so that it renders the entire sphere by using the geometry shader to emit opt_spirals tri-

angles, for values $\leq 20$. As in Homework 2, color odd spirals purple. For your convenience `color_lsu_spirit_purple` is defined in the shader file.

- Use one geometry shader invocation to generate triangles for all spirals.
- Be sure to generate approximately correct texture coordinates.
- The code must be reasonably efficient. Avoid trigonometric operations if these can be pre-computed and re-used.
- Color alternate spirals purple (with texture applied).
- **Don't forget to update the triangle layout.**

# LSU EE 4702-1 Homework 5 Due: 27 November 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 Scene 1, a cloud of green rectangles, *tiles*, with balls dropping on them. (The familiar water wheel is in scene 2.)



## Scene Interaction and User Interface

Pressing `d` (drip) will toggle ball dripping on and off. Pressing `x` will toggle a shower of balls. Pressing digits `1` through `2` will initialize different scenes, the program starts with scene 1. Pressing `t` starts the lots of balls scene (t is for test). Pressing `T` drops fewer balls.

The scenes differ in the number of objects, which include spheres, tiles, 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, tiles, 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.

## Display of Performance-Related Data

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).

## User Interface for Navigation and Interaction

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 `hw05.cc` and `hw05-shdr.cc` for documentation on other keys.

### Switching Between Shaders

The program can use three different sets of shaders for the spheres, PLAIN (tiles_0), PROBLEM 1 (tiles_1), and PROBLEM 2 (tiles_2). Pressing `v` switches between them.

### Description of Rendering

Scene 1 is filled with 8000 light green tiles arranged randomly in some volume, but all facing one point. The CPU code rendering tiles (all tiles, not just the light green ones described above) is in routine `World::render_tiles` in file `hw05.cc`. This routine launches one of the three shaders for rendering tiles. Shader `s_hw05_tiles_0`, which will be called `tiles_0` here, is used with a rendering pass in which the vertices are grouped as individual triangles. The shaders for this version are in file `hw05-shdr.cc` and have names ending in `tiles_0`. The `tiles_0` work correctly in the unmodified code, and they provide one special effect: a tile changes to red when touced by a ball and remains red for two seconds. The `tiles_0` shaders are for at most experimentation, they don't need to be modified for the homework. The `tiles_1` shaders are used in a rendering pass that also groups vertices as individual triangles, but the values used for `glVertex` and `glColor` are unconventional (see the code). The `tiles_2` shaders are used in a rendering pass in which vertices are not grouped (meaning points). For `tiles_2` the vertex data is placed in buffer objects, and the buffer objects are only updated when the underlying data changes. The only vertex shader input for the `tiles_2` is `gl_VertexID`. Initially the `tiles_1` and `tiles_2` shaders just call their `tiles_0` counterparts and so they won't render properly.

### Debugging Help

Boolean variables `opt_tryout1` and `opt_tryout2` can be used for debugging on the CPU. Pressing `y` and `Y` toggles their state. In the shader code use `tryout.x` and `tryout.y` for these two variables.

To see the difference between your code and the original code issue the Emacs command `Ctrl x v =` or from the shell the command `git diff`.

**Problem 1:** Modify the `tiles_1` shaders in `hw05-shdr.cc` so that they correctly render the tiles. They must operate with the data provided by `World::render_tiles` case 1, unconventional though it is. Do not change what data is provided by the CPU for this rendering pass.

- Work with the data provided from the CPU, don't send additional information.
- The code must be reasonably efficient.
- Pay attention to the geometry shader layout declarations **including** `max_vertices`.

**Problem 2:** Modify the code in `tiles_2` so that it renders the tiles correctly. They must operate with the data provided by `World::render_tiles` case 2, which is much more straightforward than case 1. Note that case 2 uses points as the primitive.

- Work with the data provided from the CPU, don't send additional information.
- The code must be reasonably efficient.
- Pay attention to the geometry shader layout declarations **including** `max_vertices`.

**Problem 3:** Modify the code in any of the shaders so that when a tile is struck by a ball an expanding red ring is drawn centered at the point of contact. The screenshot at the top of the assignment shows one ring in the center of the image, and some larger faded regions below.

Each tile has a `tact_pos` member that holds the coordinates of the point on the tile that the ball has struck, that in the `x`, `y`, and `z` components, and the time that the contact occurred, in the `w` component. All three rendering passes provide this data in one way or another, see `World::render_tiles`. Also, notice that `world_time` is provided to shaders.

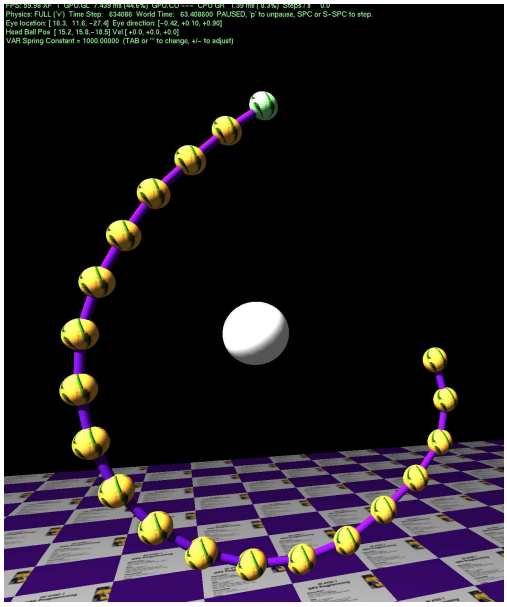
The ring should be easily visible and extend past the edges in a few seconds.

- The width of the ring should be some fixed object-space size, don't make it one pixel wide.
- Try to reduce the amount of work done by the fragment shader by doing extra work in the vertex or geometry shaders.
- Also try reducing the amount of work done by the rasterization stage.

# 11 Fall 2016

## LSU EE 4702-1 Homework 1 Due: 9 September 2016

*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, generated by code very similar to demo-2-springs.cc used in class. One difference with the classroom demo code is The Globe, a fixed white ball in the center of the scene. Pressing `o` toggles The Globe between white and yellow, but has no other effect—until Problem 3 is solved correctly. The illustration to the right is from the solution to Problem 3, in which The Globe repels other balls.

Press digits `1` through `4` to initialize different scenes, the program starts with scene 2. Initially Scenes 1 and 2 are identical, and Scene 3 shows the circle code done in class. *Promptly report any problems.*

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.

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).

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:** The code in `ball_setup_2` contains a coordinate `top_pos` and a normal vector `dir`. Modify `ball_setup_2` so that the balls are arranged in a straight line starting at `top_pos` and in the direction `dir` spaced `distance_relaxed` units apart. If the problem is solved correctly (and before Problem 2 is solved) the balls should smoothly swing and only gradually become disorganized.

Note: this is an easy problem.

**Problem 2:** In `ball_setup_2` (some as Problem 1) set the initial velocity so that the balls rotate around the axis passing through `top_pos`, going straight down. All balls should rotate around the axis at the same rate (*e.g.*, in revolutions per second, radians per second). If the problem is solved correctly the scene should start with the balls smoothly rotating around the axis.

This problem is similar to problems assigned in past semesters. Please make a strong effort to solve it yourself, rather than seeking a solution. If necessary ask the instructor for help.

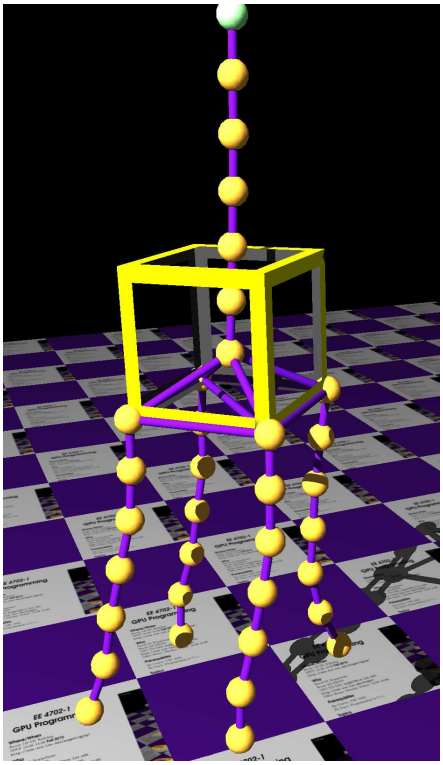
**Problem 3:** Now to The Globe. Modify the code in `time_step_cpu_full` so that when variable `opt_globe` is true the balls will be repelled by The Globe. The repulsion force on a ball should be

$\overrightarrow{f} = \frac{q_g m_b}{r^2}\widehat{GB}$, where $q_g$ is the value of variable `opt_globe_charge`, $m_b$ is the mass of the ball (in member `mass`), $r$ is the distance between the centers of The Globe and the ball, and $\widehat{GB}$ is a unit vector pointing from The Globe to the ball. The force affects the ball, but not The Globe, it is fixed in place.

The position of The Globe can be found in `globe.position`. Variable `opt_globe_charge` can be adjusted using the user interface with the `TAB`, `+`, and `-` keys.

If solved correctly the string of balls should avoid The Globe.

## LSU EE 4702-1 Homework 2 Due: 21 September 2016



**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 some objects on the left and a swinging thing of beads on the right. The screenshot to the right shows what the thing should look like after the assignment is solved correctly.

Press digits `1` through `2` to initialize different scenes, the program starts with scene 2. Scene 1 shows a plain string of beads. *Promptly report any problems.*

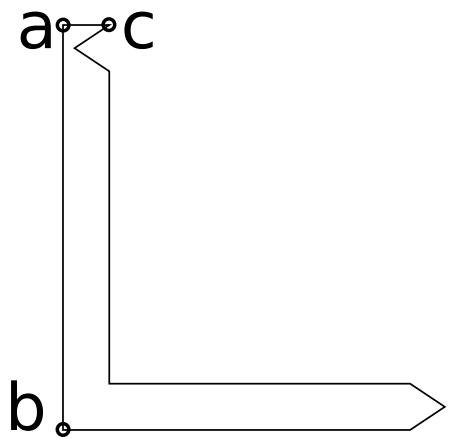
Use key `h` to toggle between the first (head) ball being locked in place and free. Use key `t` to do the same for the last (tail) ball.

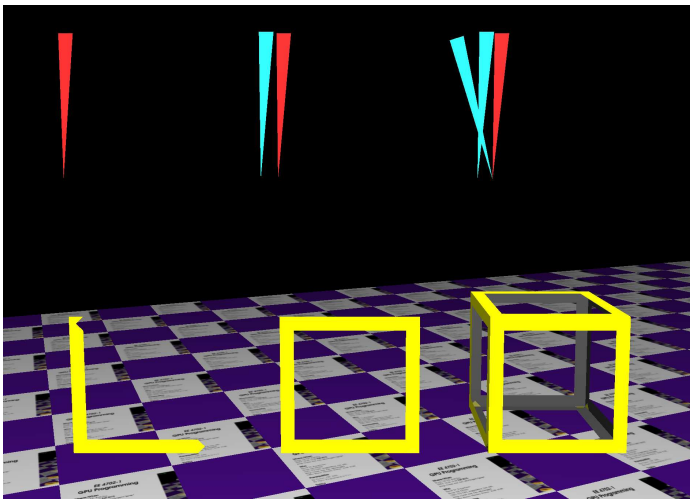
Initially the arrow keys, PageUp, and PageDown can be used to move around the scene. Press (lower-case) `b` and then use the arrow and page keys to move the first ball around. Press `l` to move the light around and `e` to move the eye (which is what the arrow keys do when the program starts).

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

*There is a problem on the next page.*

**Problem 1:** The scene contains a swinging thing of beads on the right-hand side (see the screenshot above) and six *samples* on the left-hand side (see the screenshot below). Both screenshots show the appearance after the problem is solved correctly. The six samples are arranged into two rows of three. The upper row is generated by code in `render_it_demo`, that code does not need to be changed for this assignment. The lower row is generated by code in `render_it`, that routine should be changed. The leftmost sample is generated when the `version` argument of `render_it` is 0, the middle sample is generated when `version` is 1, and the rightmost sample is generated when `version` is 2. After the assignment is solved correctly, the leftmost sample should look like the shape illustrated below, the middle sample should look like a square (or frame), and the rightmost sample should be a cube. Routine `render_it` with `version` set to 2 is also used to render the object about halfway down the thing of beads.



(*a*) The illustration on the upper left shows a shape with points labeled, a, b, and c. Routine `render_it` (in hw02.cc) has three coordinate arguments by the same name. Modify `render_it` so that when argument `version` is 0, it draws the illustrated shape. The routine currently draws a triangle with a, b, and c as vertices (shown in the upper row of the screenshot). The shape can be drawn using individual triangles or using a triangle strip.

The labeled vertices must correspond to the coordinates passed to `render_it`. The other vertices must be chosen so that when rotated it fits together to form a square (see the next subproblem). The front faces of the triangles must be facing the user, and should be yellow. The back faces should be gray.

Remove the original code in `render_it` (if it's not needed).

(*b*) Further modify `render_it` so that when `version` is 1 it draws the shape twice, once in its original orientation, and a second time rotated so that it and the original form a square. Perform the rotation by transforming the coordinates that you found for the previous subpart. Don't individually find coordinate locations for the rotated shape. The routine has sample code showing how to transform coordinates placed in a list.

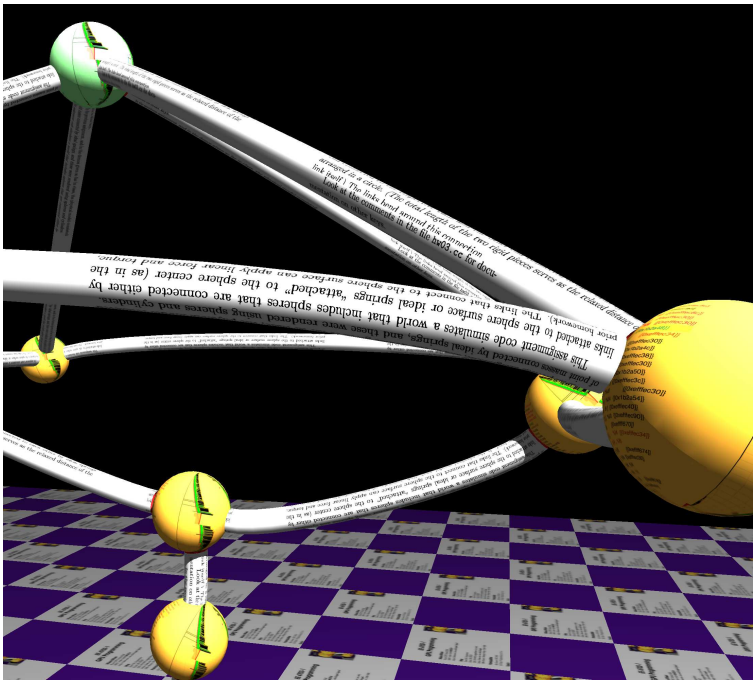
Try to render the entire square in a single rendering pass. (That is, one list should have the coordinates for the original and rotated shape.)

(*c*) Finally, modify the `render_it` so that when `version` is 2 the vertices from the previous subpart are transformed so that they form the faces of a cube. This can be achieved by rotating the vertices from the previous part. Be sure to appropriately rotate the normal too. It is okay to do a render pass for each face.

Rotate the faces so that the outsides are yellow and the inside is gray.

# LSU EE 4702-1 Homework 3 Due: 30 September 2016

**Problem 0:** If necessary, follow the instructions 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 vaguely umbrella-shaped object hanging in space.



The screenshot on the right shows what the object should look like after Problem 1 is solved correctly.

Press digits `1` through `2` to initialize different scenes, the program starts with scene 2. Scene 1 shows a balls connected in a rectangular spiral. *Promptly report any problems.*

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.

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).

Look at the comments in the file `hw03.cc` for documentation on other keys.

In the previous assignment, and in the demos given so far in class, the physical model consisted of point masses connected by ideal springs, and these were rendered using spheres and cylinders.

This assignment code simulates a world that includes spheres that are connected either by links attached to the sphere surface or ideal springs "attached" to the sphere center (as in the prior homework). The links that connect to the sphere surface can apply linear force and torque, and they can bend. In the physical model each link consists of two perfectly rigid pieces, one end connected to the respective balls and the other end connected to each other by ideal springs arranged in a circle. (The total length of the two rigid pieces serves as the relaxed distance of the link itself.) The links bend around this connection.

Since a link can bend it would not make sense to render it as a cylinder. Instead they are rendered as cubic Hermite curves. Consider a link that connects balls $B_1$ and $B_2$ with centers at $C_1$ and $C_2$. Let $P_1$ and $P_2$ denote the coordinate where the link attaches to $B_1$ and $B_2$. The link is rendered as a cubic Hermite curve parametrically described by $L(t) = (2t^3 - 3t^2 + 1)\overrightarrow{0P_1} + (-2t^3 + 3t^2)\overrightarrow{0P_2} + s(t^3 - 2t^2 + 1)\overrightarrow{C_1P_1} + s(t^3 - t^2)\overrightarrow{P_2C_2}$, where $s$ is the stiffness of the links and is set to $s = \|2.5\overrightarrow{P_1P_2}\|$ in the code.

In `hw03.cc` routine `render_link_1` and `render_link_2` render these new curved links as triangle strips. The `i` loop moves along the curve, the value of $L(t)$ is assigned to variable `ctr`.

The derivative $\frac{dL(t)}{dt}$ gives the direction of the curve. That is assigned to `tan`. The `j` loop draws a cylinder (sort of) around the curve.

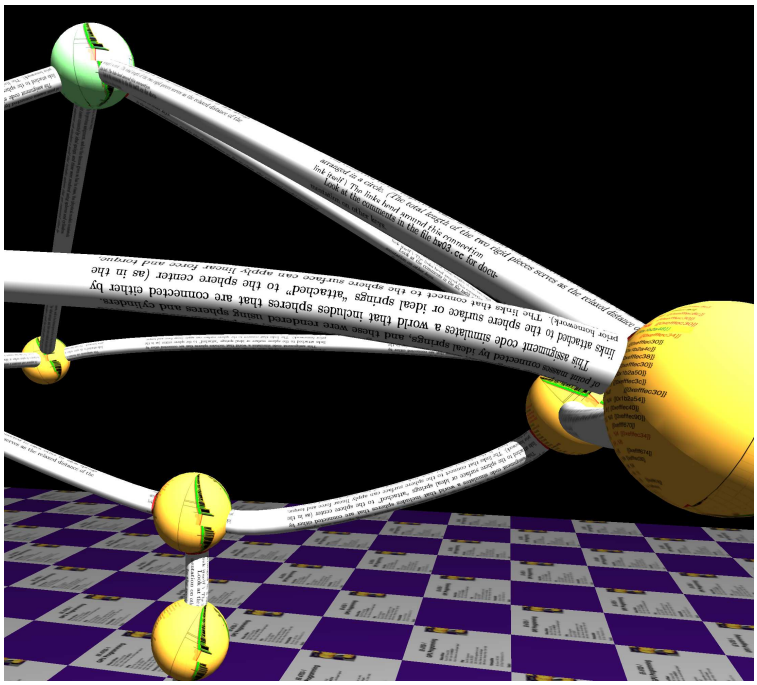
**Problem 1:** Modify `render_link_1` so that the link is textured. The texture unit and texture objects are already set up, all that needs to be done is to emit texture coordinates. The texture is an image of this assignment handout. Emit the texture coordinates so that text is along the links. **The size of the text should be constant in object space.** That is, long links (such as the ones that connect to the uppermost ball) should not have stretched-out text and short links (such as the vertical links, see the screenshot above) should not have compressed text. Pressing `v` will toggle between the ordinary view and one where triangles stand out. This may help in debugging.

*Note: As originally assigned, Homework 3 had an additional problem. That problem is now part of Homework 4.*

## LSU EE 4702-1 Homework 4 Due: 10 October 2016

*Note: These problems were originally part of Homework 3. Solve this assignment in file* `hw03.cc`*, the same one used for Homework 3.*

**Problem 0:** If necessary, follow the instructions 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 vaguely umbrella-shaped object hanging in space.



The screenshot on the right shows what the object should look like after Problem 1 of Homework 3 is solved correctly.

Press digits `1` through `2` to initialize different scenes, the program starts with scene 2. Scene 1 shows a balls connected in a rectangular spiral. *Promptly report any problems.*

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.

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).

Look at the comments in the file `hw03.cc` for documentation on other keys.

In Homework 2 and in the demos given so far in class, the physical model consisted of point masses connected by ideal springs, and these were rendered using spheres and cylinders.

This assignment code simulates a world that includes spheres that are connected either by links attached to the sphere surface or ideal springs "attached" to the sphere center (as in the prior homework). The links that connect to the sphere surface can apply linear force and torque, and they can bend. In the physical model each link consists of two perfectly rigid pieces, one end connected to the respective balls and the other end connected to each other by ideal springs arranged in a circle. (The total length of the two rigid pieces serves as the relaxed distance of the link itself.) The links bend around this connection.

Since a link can bend it would not make sense to render it as a cylinder. Instead they are rendered as cubic Hermite curves. Consider a link that connects balls $B_1$ and $B_2$ with centers at $C_1$ and $C_2$. Let $P_1$ and $P_2$ denote the coordinate where the link attaches to $B_1$ and $B_2$. The link is rendered as a cubic Hermite curve parametrically described by $L(t) = (2t^3 - 3t^2 + 1)0\overrightarrow{P_1} + (-2t^3 + 3t^2)0\overrightarrow{P_2} + s(t^3 - 2t^2 + 1)\overrightarrow{C_1P_1} + s(t^3 - t^2)\overrightarrow{P_2C_2}$, where $s$ is the stiffness of the links and is set to $s = \|2.5\overrightarrow{P_1P_2}\|$ in the code.

In `hw03.cc` routine `render_link_1` and `render_link_2` render these new curved links as triangle strips. The `i` loop moves along the curve, the value of $L(t)$ is assigned to variable `ctr`. The derivative $\frac{dL(t)}{dt}$ gives the direction of the curve. That is assigned to `tan`. The `j` loop draws a cylinder (sort of) around the curve.

**Problem 1:** Modify `render_link_2` so that it uses buffer objects for vertices, normals, and texture coordinates as described below. (To switch between `render_link_1` and `render_link_2` press `z`.)

(*a*) Modify the routine so that it uses a buffer object to hold texture coordinates. The buffer object should be written just once and used each time. (This will result in text appearing compressed on short links and stretched on long ones.)

(*b*) Modify the routine so that it uses a set of buffer objects to hold the vertices and normals of curved links. (See the next part.) These buffer objects will have to be updated each time `render_link_2` is called.

(*c*) Modify the routine so that it uses another set of buffer objects to hold the vertices and normals of straight links. These buffer objects should be written to once and used many times. Straight links, because they use re-cycled buffer objects, should be shown in pale green.

A link is straight if the link directions (`dir1` and `dir2` in the code) are the same direction as the line connecting the link endpoints (`pos1` and `pos2`) in the code. Consider a link straight if the link directions are close enough to the line connecting their endpoints. After the scene calms down, the vertical links in the umbrella should be considered straight, and perhaps the four links on top. If gravity is turned off (`g`) all links should be considered straight.

All straight links are in the shape of a cylinder. They differ however in their lengths and orientations and placement. For straight links you will need to use a transformation so that the straight link used to populate the buffer object is transformed for the link for which it will be used.

To solve this correctly storage will be needed for the buffer object names, and information about the link it describes. Declare members in World for this purpose. Do not declare variables in global scope.

For this problem, the best way to construct a rotation matrix is by combining unit vectors to form either the rows or columns of a matrix. These can be constructed using `pMatrix_Rows` and `pMatrix_Cols`. For example `pMatrix_Rows rot(v1,v2,v3);` constructs a matrix `rot` whose rows are `v1`, `v2`, and `v3`, respectively. Let `v1`, `v2`, and `v3` be some orthogonal unit vectors in some global space. Matrix `rot` can be used to convert a point to some local space, such that direction `v1` in object space is the $x$ axis in the local space, etc. With `pMatrix_Cols rot(v1,v2,v3);` a point can be transformed to a global space. For this homework problem, one might set `v3` to the link direction of the straight link (making the link direction the $z$ axis in a local space), see the code for suggestions for $x$ and $y$ directions.

To solve this problem two transformations are needed. One, to transform the straight link in the buffer object to some local space. A second one is needed to transform it for the link that's needed.

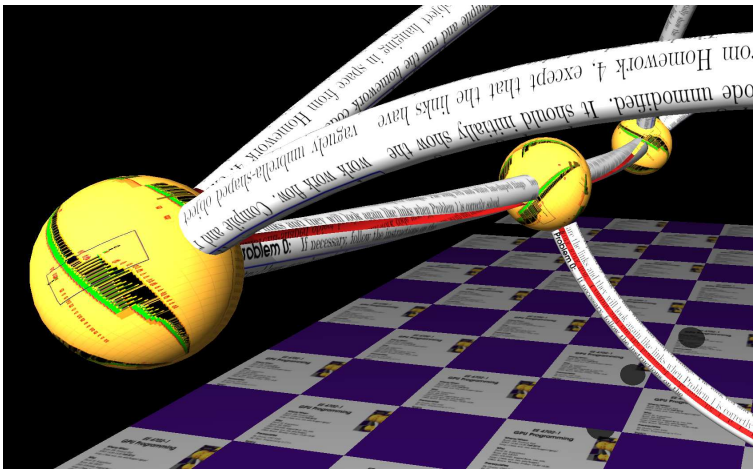
**Do not** construct a link in some local space and re-use that. It's not a bad thing to do in general, but don't do it here.

## LSU EE 4702-1 Homework 5 Due: 17 October 2016

**Problem 0:** If necessary, follow the instructions on the `http://www.ece.lsu.edu/koppel/gpup/proc.html` page for account setup and programming homework work flow. For this assignment only edit file `hw05-shdr-links.cc`. Compile and run the homework code unmodified. It should initially show the vaguely umbrella-shaped object hanging in space from Homework 4, except that the links have been replaced by long red and white fan-shaped things. The long red and white fan-shaped things are the links and they will look again like links when Problem 1 is correctly solved.

Pressing `v` will cycle through three different sets of shaders. The shader set that is being used is shown in the penultimate line of green text. Shader set PLAIN is a conventional set of shaders, and is there for comparison purposes. Shader set SET 1 is comprised of vertex shader `vs_main`, geometry shader `gs_main_1`, and fragment shader `fs_main_1`, all in file `hw05-shdr-links.cc`, and is used for Problem 1. Shader set SET 2 is comprised of vertex shader `vs_main`, geometry shader `gs_main_2`, and fragment shader `fs_main_2`, also in file `hw05-shdr-links.cc`, and is used for Problem 2.

The screenshot on the right shows what the object should look like after Problem 1 is solved correctly.



Press digits `1` through `2` to initialize different scenes, the program starts with scene 2. Scene 1 shows a balls connected in a rectangular spiral. *Promptly report any problems.*

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.

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).

When using the arrow and other keys to move the eye, light, or ball using `Shift` will move by a 5× greater amount and using `Ctrl` will move by one $\frac{1}{5}$ the amount than the motion without either modifier.

Look at the comments in the file `hw05.cc` for documentation on other keys.

**Problem 1:** One flaw in Homeworks 3 and 4 was in the use of triangle strips. Recall that a link was rendered in segments, and that each segment was roughly a cylinder. The link would have been rendered correctly if each individual segment were one triangle strip, but instead the code rendered the entire link as a triangle strip, resulting in two extra triangles at the beginning of each segment. The flaw persists in the Homework 5 code and is exasperated because the links are no longer supposed to be complete cylinders, instead there should be a gap revealing the red insides of the link. See the screenshot above.

An inefficient solution would be to perform a rendering pass for each segment. Or one might use an OpenGL draw command that can re-start strips periodically. But in this assignment we'll

fix the problem another way: by having the geometry shader recognize that a triangle should not be rendered and abandoning it.

On the CPU side in routine `render_link_1` the value of variable `theta` has been placed into the $w$ component of the vertex coordinate. (I'll pause for a moment while you think about this.) Hopefully some of you realized that it's the $w$ component that explains the long red and white fan-like things, since the $w$ acts like a scale factor.

Modify the Set 1 shaders so that the value of the $w$ component, `theta`, is used to determine whether a triangle should be rendered, and if not to discard the triangle. And make whatever other changes are needed so that the links look like links again.

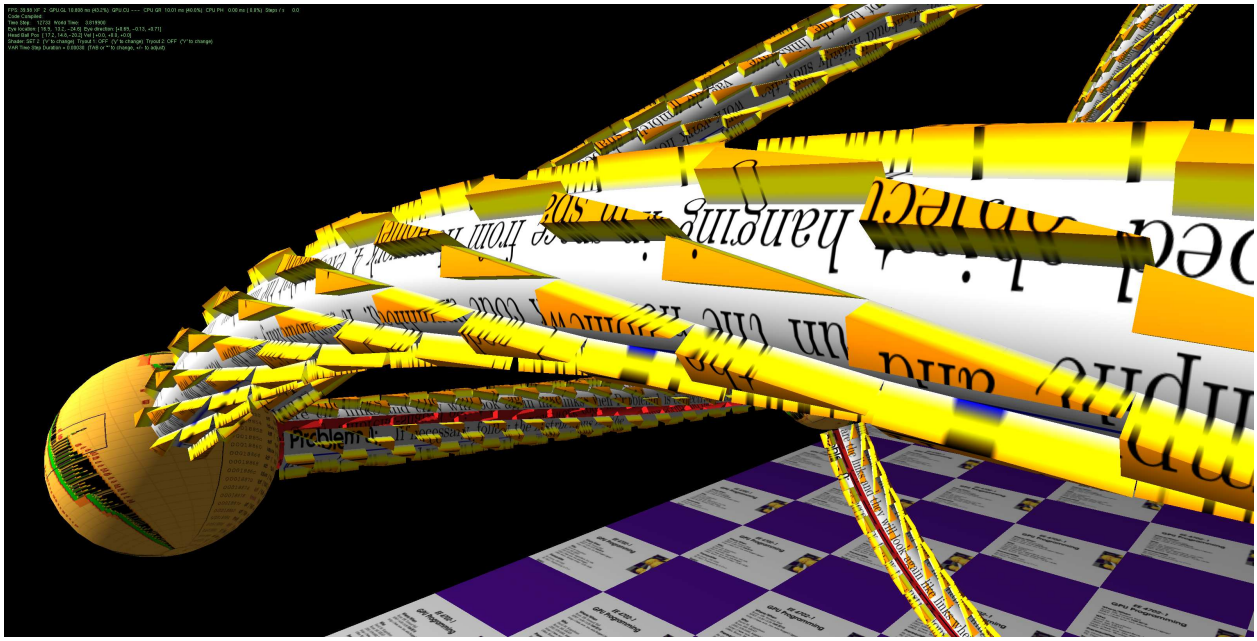
A solution to this problem will require the following: declaring a variable to communicate the value of theta between the vertex shader and geometry shader, modifying `vs_main_1` to extract theta from the vertex coordinate and then making sure that the correct vertex coordinate is used, and modifying `gs_main_1` so that it does not emit any triangles if theta is zero.

This triangle-strip issue is fixed using an alternative approach in the shaders for Problem 2. You might want to inspect that code for hints.

Note that packing unrelated information ($\theta$) into the $w$ component vertex coordinate is bad from the point-of-view of code readability however because GPUs handle four-element vectors very efficiently it might result in better performance than using a separate variable for $\theta$.

**There is another problem on the next page!**

**Problem 2:** Modify the Set 2 shaders so that some triangles making up the link appear raised and colored as in the screenshot below. Details are given in the subproblems.



(*a*) Modify the shaders so that some triangles that make up the surface of the link are raised above the surface `thickness` units (where `thickness` is a uniform variable) with additional *side* primitives added to connect the raised triangle to the link surface. The raised triangles are the solid orange triangles in the screenshot above and the side triangles are colored orange/yellow/white. It is not important which triangles are raised, so long as some raised triangles don't share edges with other raised triangles.

Try to render the side triangles using a triangle strip. If done properly, a triangle strip will make this part easier, but it will make specifying the proper normals more difficult.

(*b*) Use the value of `color_top` for the color of the raised triangle, `color_bottom` for the link surface, and a blend of `color_side` and the other two colors for the side primitives. The color of the side primitives starting at the raised triangle and going towards the link surface should shift gradually from `color_top` (orange in the screenshot) to `color_side` (yellow in the screenshot), and then from `color_side` to `color_bottom` (light gray in the screenshot). The color should be completely `color_side` from 33% to 66% of the way down. (The colors should still be lighted.)

(*c*) The normals for the raised triangle should be the same as the normals that would be used on the link surface. The normals for the side triangles must be their geometric normals, so that the side triangles appear flat and the corners are sharp, not rounded.

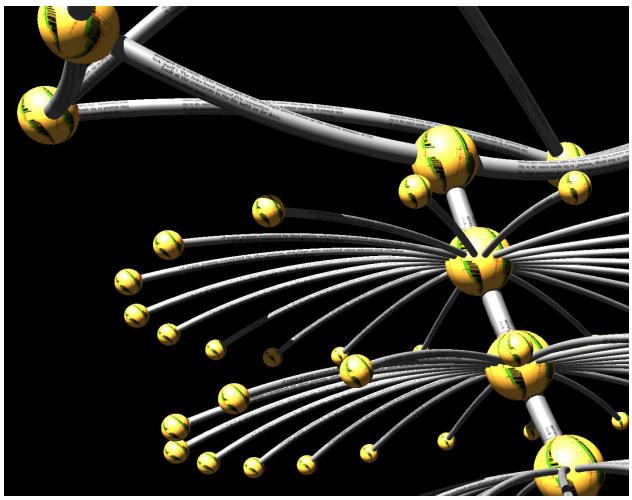
(*d*) Render the text (texture) on the side triangles so that as one goes from the raised triangle to the link surface the text fades, completely fading halfway down, then darkening again. (See the screenshot.)

# LSU EE 4702-1 Homework 6 Due: 16 November 2016

**Problem 0:** If necessary, follow the instructions on the `http://www.ece.lsu.edu/koppel/gpup/proc.html` page for account setup and programming homework work flow. For this assignment only edit files `hw06-shdr-links.cc` and `hw06.cc`. Compile and run the homework code unmodified. It should initially show Scene 3, a spikey ball. The vaguely umbrella-shaped object hanging in space from prior assignments but with bristles or something on the vertical tail of balls is in Scene 2 and Scene 1 shows a tree (the kind that grows outdoors).

Pressing `v` will cycle through three different sets of shaders. The shader set that is being used is shown in the penultimate line of green text. Shader set PLAIN is a conventional set of shaders, and is there for comparison purposes. Shader set SET 1 is comprised of vertex shader `vs_main_1`, geometry shader `gs_main_1`, and fragment shader `fs_main`, all in file `hw06-shdr-links.cc`, and is used for Problem 1. Shader set SET 2 is comprised of vertex shader `vs_main_2`, geometry shader `gs_main_2`, and fragment shader `fs_main`, also in file `hw06-shdr-links.cc`, and is used for Problem 2.

The plain set of shaders should be slow, while the other sets show the links in a crude manner. Options that affect performance are: turning shadows on and off `o` and turning mirroring on and off `r`. (Pressing `O` will show shadow volumes.) Note that the links still don't cast shadows.



Press digits `1` through `2` to initialize different scenes, the program starts with scene 2. Scene 1 shows a balls connected in a rectangular spiral. *Promptly report any problems.*

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.

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).

When using the arrow and other keys to move the eye, light, or ball using `Shift` will move by a 5× greater amount and using `Ctrl` will move by one $\frac{1}{5}$ the amount than the motion without either modifier.

Look at the comments in the file `hw06.cc` for documentation on other keys.

A goal of Homework 3 was to reduce the amount of communication between the CPU and GPU by using buffer objects, especially for almost straight links. In this assignment two methods will be used to improve performance: computing the surface of the curved link in the GPU, and using an instanced draw to reduce the number of rendering passes.

See the OpenGL Shading language documentation linked to the references page, `http://www.ece.lsu.edu/koppel/gpup/ref.html`, for syntax and library functions that can be used in shader code.

**Problem 1:** The code in `vs_main_1` and `gs_main_1` are used to crudely render a link with an instanced draw. Examine the code in `vs_main_1` and `render_link_2_render`. Built-in variable `gl_InstanceID` indicates which link is being rendered, its value can range from 0 to one minus the number of links (or instances). The number of instances is determined by the last argument to `glDrawArraysInstanced`. The built-in vertex shader `gl_VertexID` indicates the vertex number being rendered, its range of values is determined by the third argument to `glDrawArraysInstanced`.

Note that in the rendering pass set up in `render_link_2_render` *no vertex coordinates nor any other vertex shader inputs* are sent to the vertex shader. The shader must rely on `gl_InstanceID` and `gl_VertexID` to find its inputs.

Notice that `vs_main_1` reads link endpoint coordinates from buffer objects `pos1` and `pos2`, and using a scaled version of `gl_VertexID`, computes a point partway between the endpoints, `vertex_o`. That coordinate is transformed and sent to the geometry shader for constructing triangles.

Since it renders a straight link there is no reason to use more than two vertices per link. (The curved link will be rendered using `vs_main_2` an `gs_main_2`.)

(*a*) Modify `vs_main_1`, `gs_main_1`, and `render_link_2_render` so that the complete straight link is rendered using only two vertices per instance without changing the primitive at the input to the geometry shader (a line strip).

- Split work between the vertex shader and the geometry shader to void duplication of effort.
- Make sure that changes to `render_link_2_render` don't affect `vs_main_2`.

**Problem 2:** The code in `vs_main_1` and `gs_main_1` which also crudely render a link, are placeholders for code rendering the curved link.

(*a*) Modify `vs_main_2` and `gs_main_2` so that they render the curved link based on the data copied into buffer objects by routines `render_link_2_gather` and `render_link_2_render`. Base your code on the code in `render_link_1`. Be sure that any changes made for the prior problem don't interfere with this one.

- Split work between the vertex shader and the geometry shader to void duplication of effort.
- Write your routine for a link with a maximum of 20 sides.

**Problem 3:** Estimate the amount of data sent from CPU to GPU for each link for the plain and set 1 and set 2 shaders. (The set 1 and set 2 shaders should send the same amount of data.) Do this by examining the code in `render_link_2_gather` and `render_link_2_render`.

## LSU EE 4702-1 Homework 7 Due: 1 December 2016

**Problem 0:** If necessary, follow the instructions on the `http://www.ece.lsu.edu/koppel/gpup/proc.html` page for account setup and programming homework work flow. For this assignment only edit files `hw07-cuda.cu` and `hw07.cc`. **Find a machine with GPUs of CC (compute capability) 3.0 or higher,** see the table on `http://www.ece.lsu.edu/koppel/gpup/sys-status.html`. Compile and run the homework code unmodified. It should initially show the spring from classroom demo `demo-cuda-04-acc-pat.cc`.

The physics for the spring can be performed using three different sets of code, the one in use is shown at the beginning of the last line of green text next to Physics. The physics code can be changed by pressing `a`. Physics code *CPU* uses the CPU and will be very slow. Physics code *CUDA-M1* and *CUDA-M2* use the GPU running code in file `hw07-cuda.cu`. Code *CUDA-M1* uses routine `time_step_intersect_1` to detect and resolve intersection of helix segments, and *CUDA-M2* uses routine `time_step_intersect_2`.

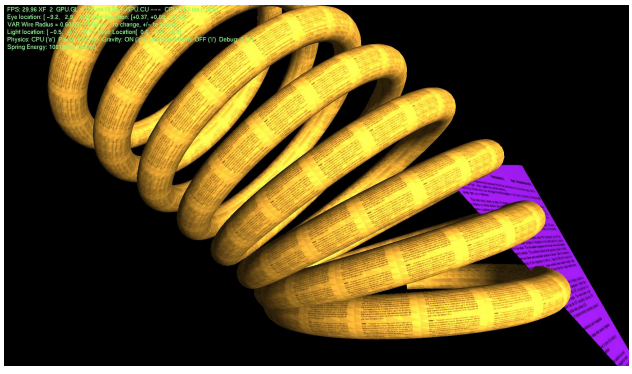
The green line starting with "Intersect" shows information about the execution of the chosen intersection routine. As discussed in class, it is this routine that will dominate performance. On that line "Bl Sz" is the number of blocks. That can be changed using the `Tab` key to find "Intersect Block Size" and then pressing `+` and `-` keys. Green text "N Blocks" shows the number of blocks. The number of blocks is determined by variable "Seg / Block" (`intersect_seg_per_block`). Green text "Bl/SM" shows the number of blocks per SM in two ways. The first value is the number of blocks running the intersect routine that *can* fit on an SM. The second number is how many are assumed running based on the number of blocks and the number of SMs. If the second number is lower than there are not enough blocks. Green text "Wp/SM" shows the number of warps per SM. That's based on the block size and the number of blocks per SM. Finally, "Bl Time" shows the number of cycles it takes to run one block. (The top green line shows timing per frame.)

On the last green line, "Shared Mem" shows how shared memory is being used to buffer the "b" segments in the intersection routine. "NONE" means that shared memory is not being used, "SYNC EXPER" is something to be used in Problem 2, "ONE ITER" means that shared memory is being loaded with "b" segments every `j` iteration (see the intersect routines), and "MULT ITERS" means that shared memory is loaded with enough data for several `j` iterations—when Problem 3 is solved correctly.

The top green line shows performance. The important items to pay attention to are "GPU.CU" and "Steps / s". "GPU.CU" shows the time per frame executing CUDA code. Use this to evaluate the performance of the intersection routines. The last item on the line (you may need to widen the window), "Steps / s", shows how many physics time steps are performed each second. If computation is fast enough it will be about 6000, otherwise it will be lower. "FPS" is the frame rate in frames per second. "XF" shows the frame update interval, an ideal number is 1. A 2 indicates that the frame is updated at half the display's refresh rate. "GPU.GL" is the time the GPU spends executiong OpenGL code. "CPU GR" is the CPU time for graphics. "CPU PH" is the CPU time for physics.

Fun Stuff: Pressing `g` will toggle gravity. The free end of the spring can be grabbed by pressing `b` and it can be moved using the arrow keys.

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 free end of the spring around. (Since the motion is



instantaneous, the spring will act like it was plucked.) Press `l` to move the light around and `e` to move the eye (which is what the arrow keys do when the program starts).

When using the arrow and other keys to move the eye, light, or ball using `Shift` will move by a 5× greater amount and using `Ctrl` will move by one $\frac{1}{5}$ the amount than the motion without either modifier.

Look at the comments in the file `hw07.cc` for documentation on other keys. For documentation see the CUDA C Programming Guide linked to the course `http://www.ece.lsu.edu/koppel/gpup/ref.html` page.

**Problem 1:** The goal of this assignment is to understand and improve the performance of the intersection routines. The performance of the intersection code is very sensitive to the block size (variable "Intersect Block Size") and to the number of `a` segments per block (variable "Seg / Block", called $m$ in our classroom analysis).

($a$) Indicate the model of GPU you were using (it's shown in the text printed when the program starts). Indicate the number of SMs (shown as MP) for that GPU.

($b$) Adjust these two variables to obtain the best performance on each intersection routine. Do these with shared memory set to NONE. Provide the following information: The settings of those variables. The initial CUDA execution time ("GPU.CU") and the best one obtained.

($c$) Based on our classroom analysis there was less data communication with a larger value of $m$. Are your results consistent with that? If not, explain why.

**Problem 2:** Set CUDA-M1 to the optimal parameters found in the previous problem. Now, cycle through the different shared memory options. The "ONE ITER" option, which caches enough data for one iteration, should be the slowest. Is it because of the syncthreads?

Modify routine `time_step_intersect_1` so that when `hi.opt_sm_option` is set to value `SMO_sync_experiment` syncthreads is called in a way similar to when `hi.opt_sm_option` is set to `SMO_one_iteration`, but without actually loading or using shared memory. Do so in a way that will determine whether syncthreads is making "ONE ITER" take longer.

Describe what you found by doing the experiment.

**Problem 3:** Modify `time_step_intersect_1` so that when `hi.opt_sm_option` is set to value `SMO_multiple_iterations` array `pos_cache` is loaded with $B$ items, where $B$ is the block size (value of `blockDim.x`). This should be done inside the `j` loop when data is needed. That is, **don't** load the cache every iteration.

Compare the performance to NONE and ONE ITER.

# 12 Fall 2015

# LSU EE 4702-1 Homework 1 Due: 14 September 2015

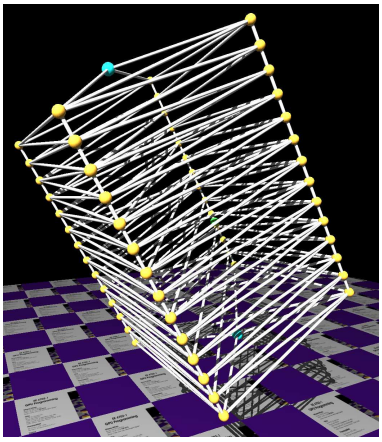
*Some of the effort for this homework assignment is in learning to use the various pieces of software, such as a text editor. Those less familiar with Linux software development procedures might seek out a more knowledgeable classmate to minimize frustration and wasted time.*



**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.

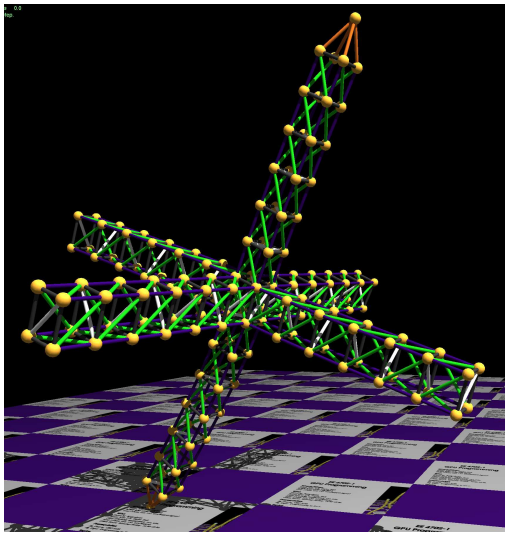
The mathematics for this problem is fairly simple: you need to find the vector pointing from `head_pos` to `tail_pos`, call that vector the *axis*. The vector `delta_unit` should be some fraction of the axis vector. Find the closest point on the axis to `surface_point`, call it $S$. Use $S$ to find `loc_x`,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

**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 $\overrightarrow{SB}$.

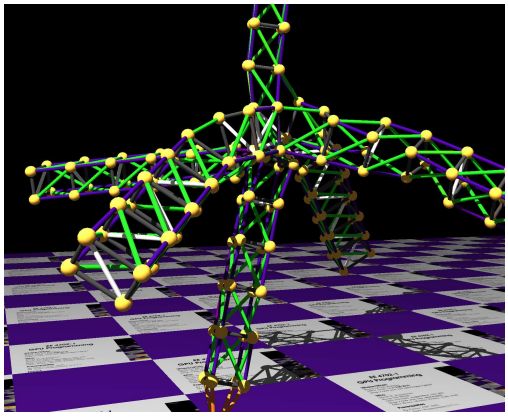
**Problem 3:** Modify `ball_setup_4` so that it adds two or four more trusses to the existing truss, these should be neatly attached to the center of the original truss and should fall on a plane normal to the original truss. See the illustration to the right.



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`.

# LSU EE 4702-1 Homework 2 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.*

**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.

**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.

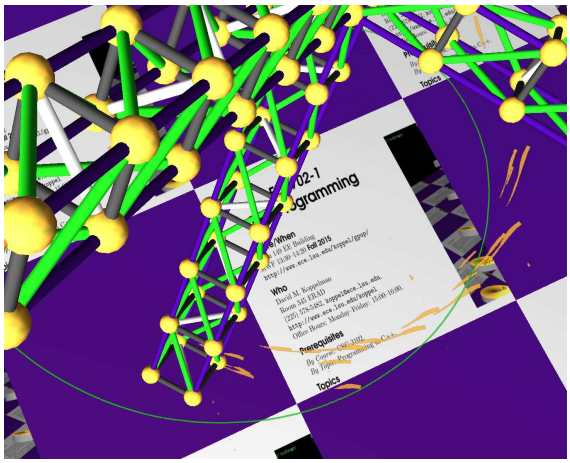
Modify `render_my_piece` and other routines so that the triangles are placed on the surface, and so that normals are set correctly. This code might use the arrays that you prepared in `make_truss`.

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).

**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.

# LSU EE 4702-1 Homework 3 Due: 12 October 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.*

The code in this package includes the friction model asked for in Homework 2. The links however are unbreakable and there's no skin. Also, shadows are deactivated by default to make the solution to this assignment easier. Notice that this starts like the code for Homework 2, except that ball at the bottom of the chain hits the platform as it swings by.

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.

*Note: There is nothing to turn in for this first problem.*

**Problem 1:** Notice how the platform is always perfectly clean. That's not realistic! In this problem modify the code so that when the balls slide across the platform they leave scuffs. The shape of a scuff should be the area of the platform touched by the ball (base the width on the intersection of the sphere and the platform) and the color should match the ball. See the illustration to the right. The ball at the base of the gyroscope is green, it leaves a neat green path. The arm balls are yellow. The balls at the arm ends striking the platform leave scuffs that start out thick and get thinner.

The code should be added to class `My_Piece_Of_The_World` and elsewhere as needed. The places in which code needs to be added start with the comment `Homework 3` and are followed by checklists of things to be done.

Scuffs are to be implemented using textures. Rather than using one large texture covering the entire platform, the textures are broken up into a grid of *overlays*. The texture corresponding to an overlay should only be initialized when the area is first scuffed, and textures should only be sent to OpenGL when the textures change. This uses vastly less data then sending a texture for the entire platform at every frame.

(*a*) Code has already been added to `time_step_cpu` to retrieve an overlay, actually a `Platform_Overlay` object, for a ball if it is colliding with the platform. Add code to this routine and elsewhere to write the appropriate texels with scuff marks. For sample code see `sample_tex_make` in the homework file. This routine creates a texture containing a big red ex. (The code in `time_step_cpu` only needs to modify the texels in `po->data`, it should not send the data to OpenGL!)

(*b*) Routine `My_Piece_Of_The_World::render()` is supposed to render the platform overlays. For each scuffed overlay it should emit primitive(s) with the corresponding texture attached. The texture application, alpha test, and blending modes need to be set so that only the scuff marks cover the underlying platform (and semi-transparently if possible). The routine has some code. Finish it, and make changes elsewhere as needed.

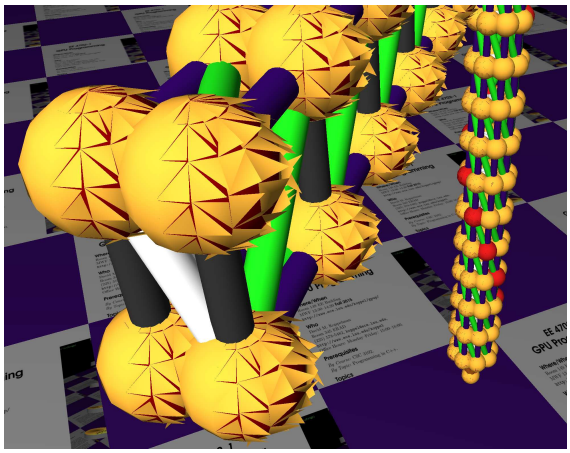
See `gpup/demo-8-textures.cc` for examples of how to do texturing, blending, and alpha testing. If you are not sure if your own texture is working you can use the texture created in `sample_tex_make`.

**Problem 2:** Answer each question below about performance aspects of texturing.

(*a*) The texel array that we send to OpenGL is an array of `floats`, because that lets us use our familiar `pColor` objects. What would be the impact of using a color representation that matched the depth of our displays, eight bits per color?

(*b*) Suppose we vary `nx` and `nz`, and choose `twid_x` and `twid_z` so that the total number of texels across the platform, `nx * twid_x`, is constant. Describe the impact on performance as `nx` and `nz` change vary from 1 to their maximum (say, 10240 for the assignment code).

# LSU EE 4702-1 Homework 4 Due: 11 November 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 start with an 8-arm gyroscope. Press the `+` and `-` keys to change the number of armds. Press digits `1` through `3` to initialize different scenes, the program starts with scene 3.

Pressing 'v' cycles between shaders.

The code in this package includes the friction model asked for in Homework 2. The links however are unbreakable and there's no skin. Also, shadows are deactivated by default to make the solution to this assignment easier.

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.

*Note: There is nothing to turn in for this first problem.*

**Problem 1:** The geometry shader `gs_main_simple` in file `hw04-shdr.cc` passes triangles through unmodified. We know that when rendering spheres and cylinders (at least the cylinders that we use for links) a triangle that's not facing the user will be behind one that is facing the user. It makes sense to discard such triangles early.

(*a*) Modify the geometry shader so that it does not emit triangles based on whether they are facing the user and the value of uniform variable `tri_cull`. If `tri_cull` is zero always emit the triangle. If it is 1, emit the triangle only if the triangle front is facing the user. If `tri_cull` is 2 emit the triangle only if the back is facing the user. Base this on the triangle coordinates, not the normals. The value of `tri_cull` can be changed by pressing `c`.

(*b*) Estimate the amount of work saved. Use the following symbols: Let $c_v$, $c_g$, and $c_f$ be the amount of work done by an invocation of the vertex, geometry, and fragment shaders respectively. For example, if we rendered one triangle covering 22 pixels the total work would be $3c_v + c_g + 22c_f$ units. For your answer let $n_t$ denote the number of triangles, and make reasonable choices for other quantities.

(*c*) Try to determine actual values for $c_v$, $c_g$, and $c_f$ by running your code. Pause the simulation to make sure the physics isn't hogging CPU. The number of vertices is shown on the green text near the top, the total number is at the end of the line. Triangle strips are used for both the balls and the links. Pressing the `+` and `-` keys when setup 3 is visible will change the number of arms on the gyroscope, use this to vary the number of balls and links.

**Problem 2:** The instance shader code rendering links shows all links the same color.

(*a*) Modify the shaders in `hw04-shdr-links.cc` so that that the fragment shader obtains the color using a value based on `glVertex_ID` rather than reading the color itself from on of its inputs. Only the vertex shader has access to `glVertex_ID`, define shader inputs and outputs as needed to send its value to the fragment shader.

Modify other shaders in this file consistently to remove the old code passing color data through the pipeline.

*Note: The original version of the problem was to use pre-defined fragment shader variable* `glVertex_ID`. *As a student pointed out, there is no such variable in the fragment shader.*

**Problem 3:** Modify `gs_main_simple` so that it can emit a second triangle that looks like it is being peeled off a moving sphere by the rushing air. The second triangle should be the same shape as the original triangle and should share one edge with the original triangle. See the illustration at the beginning of this assignment. The choice of shared edge and the angle of the triangle should be based in some way on the velocity of the ball.

# 13 Fall 2014

## LSU EE 4702-1 Homework 1 Due: 19 September 2014

*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 line of balls drop to the platform. *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. Press `1` to set up scene 1, press `2` to set up scene 2.

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.*

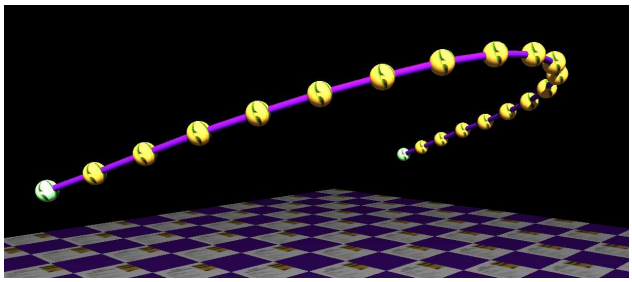
**Problem 1:** Pressing `1` sets up scene 1 (by calling `ball_setup_1`), the pendulum, and pressing `2` sets up scene 2. In an unmodified assignment setting up scene 2 only freezes the head and tail balls in place, it makes no other changes.

Modify scene 2 (look for code `ball_setup_2`) so that it places the balls evenly spaced in a straight line. The first ball (`ball[0]`) should be placed at position `first_ball` (which is declared in the routine), the last ball (`ball[chain_length-1]`) should be placed at position `last_ball`, and the other balls should be spaced evenly between them. The total number of balls is `chain_length`.

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 a vector pointing from `first_ball` to `last_ball` whose length is the distance between two adjacent balls. Look at `ball_setup_1` to see how balls are initialized.

**Problem 2:** Pressing `w` toggles a twirl option on and off. In the unmodified code that just toggles variable `opt_twirl` between `true` and `false` and changes an indicator on the upper-left area of the screen. Modify the code so that when `opt_twirl` is true forces are applied to the balls that will cause them to move around an axis defined by the first and last balls. (See the illustration to the right.)

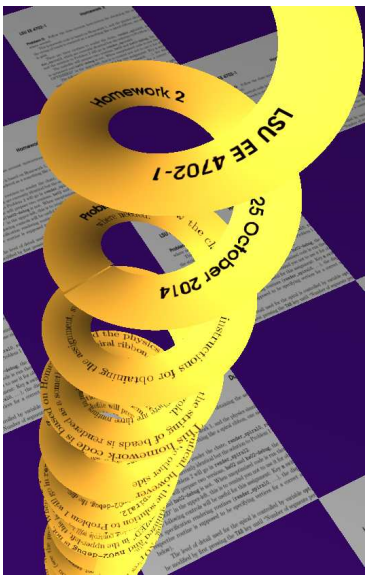


The magnitude of the force should be 1 and the direction should be orthogonal to the axis defined by the first and last ball and the shortest line from the ball to that axis. (As though the ball were embedded in a rotating cylinder with the first and last balls on the axis.) The solution to this problem should go in the `time_step_cpu` routine.

The mathematics for this problem is a little more complex, but still basic. Write an equation for the line connecting the head and tail (first and last) 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 force 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 $\overrightarrow{SB}$.

## LSU EE 4702-1 Homework 2/3 Due: 25 October 2014




**Problem 0:** Follow the class account instructions for obtaining the assignment, substitute hw02 where needed.

This homework code is based on Homework 1, and the physics simulation is identical, however the string of beads is rendered as a something like a spiral ribbon, one side is purple, the other side is gold.

There are three routines to render the chute, `render_spiral0`, `render_spiral1`, and `render_spiral2`, which are currently identical but the solution to Problem 1 will go in `render_spiral1` and the solution to Problem 2 will go in `render_spiral2`.

The makefile will prepare two versions, `hw02` and `hw02-debug`, the difference being that `hw02` is optimized and `hw02-debug` is not. When unoptimized code is run there will be a flashing "NOT OPTIMIZED" in the upper-left, this is to remind you not to use it for obtaining performance data.

The following controls will be useful for this assignment: Key `m` switches between the different vertex specification rendering routines (`render_spiral0`, ...), the display will indicate how the respective routine is *supposed to be* specifying vertices for a correct solution (see the problems below).

The level of detail used for the spiral is controlled by variable `opt_segments`. Its value can be modified by first pressing the `TAB` key until "Number of segments per spiral" appears, and then pressing `+` and `-` to adjust its value.

Check out and compile the code, and make sure that it runs correctly. The image on the upper right shows the appearance of the spiral after a correct solution to Problem 2.

**Problem 1:** Examine the code in `render_spiral1`. There you will find that individual triangles are used to render the chute.

(*a*) Modify the code so that it uses triangle strips. Be sure to do this correctly, the ribbon should be gold on one side and purple on the other and fewer vertices should be sent to the GPU.

(*b*) Measure how much of a performance difference triangle strips make. (Do this by comparing to `render_spiral0` using the `m` key.)

(*c*) Estimate the amount of data sent from CPU to GPU for the two versions of the code.

(*d*) Performance should be better with the triangle strips. How much was the improvement from `render_spiral0` to `render_spiral1` affected by:

- reduction in computation by the CPU and GPU? *Note: The original question did not include the phrase "by the CPU and GPU".*
- reduction in data sent to the GPU?
- reduction in work performed by the fragment shader (or fixed function equivalent)?

**Problem 2:** The code in `render_spiral0` and `render_spiral1` apply a texture to the spiral. The texture is a draft of this homework assignment. The code applies the texture so that the top and

bottom of the page is mapped to the inner- and outer- part of the spiral. Modify `render_spiral1` so that the width of the page is one over the chain length and as one goes down the spiral one goes down the page. (As though the page were cut into horizontal strips and those strips were connected to form a spiral.) See the illustration above.

*The problem below was originally assigned as Homework 2 Problem 3 but was later reassigned as Homework 3 Problem 1.*

**Problem 1:** In this problem use buffer objects to store vertices for one spiral section and transformation matrices to render all of them.

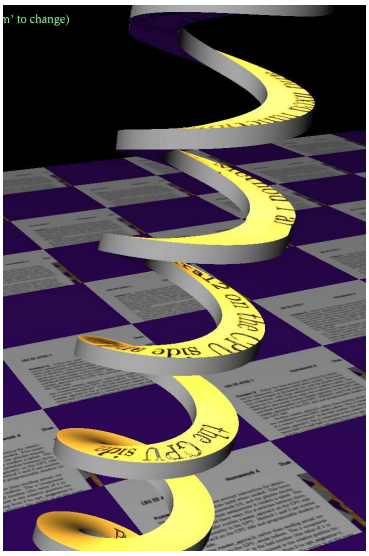
(*a*) Modify `render_spiral2` so that it uses buffer objects. (See the code in `demo-7-vtx-arrays.cc` for example of how to do that.)

A spiral section is a spiral going from one ball to the next. It is what is constructed inside of the `t` loop. Use buffer objects to store just one spiral section. Use transformation matrices to transform the spiral section in the buffer object into the ones that are needed. For your convenience a routine `make_transform(f1,f2,t1,t2)` is available which will transform a spiral between points `f1` and `f2` to a spiral between points `t1` and `t2`.

(*b*) Measure the performance improvement over the two methods used in Homework 2.

# LSU EE 4702-1 Homework 4 Due: 7 November 2014

*The solution code is in the repo in file name* `hw04-shdr-sol.cc`. *If you pull and* `make` *you can run the solution by executing* `hw04sol`. *An html version is available at* `https://www.ece.lsu.edu/koppel/gpup/2014/hw04-shdr-sol.cc.html`. *Note that the git version is always the most up to date.*

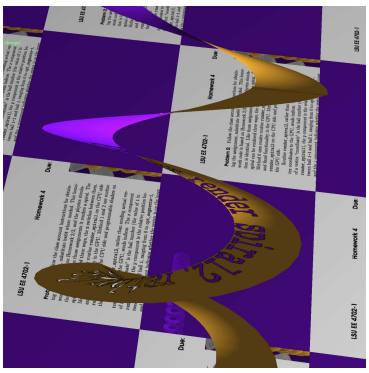


**Problem 0:** Follow the class account instructions for obtaining the assignment, substitute hw04 where needed. This homework code is based on Homework 2/3, and the physics simulation is identical. Like those assignments it renders a spiral. The spiral can be rendered three ways, the `m` switches between them. Method 0 uses render routine `render_spiral1` on the CPU side and fixed functionality in the GPU. Method 1 and 2 use routine `render_spiral2` on the CPU side and programmable shaders on the GPU side.

Routine `render_spiral2`, rather than sending actual vertex coordinates to the GPU, sends indices. The $x$ component of a vertex "coordinate" is the ball number (the value of `i` in `render_spiral1`), the $y$ component is the relative position between ball `i-1` and ball `i`, ranging from 0 to `opt_segments-1`, and the $z$ component indicates whether the vertex is on the inner or outer edge of the spiral. The code in vertex shader `vs_main` in file `hw04-shdr.cc` uses these values to compute the actual vertex coordinate and normal, and to compute the texture coordinates.

Notice that the indices sent by `render_spiral2` are not affected by the position of the balls. Therefore indices can be put in a buffer object and sent just once. (They do need to be re-sent if `opt_segments` changes.) The ball positions do need to be sent each frame, but that's much less data. The code in `render_spiral2` always uses vertex shader `vs_main` and fragment shader `fs_main`, but can activate either of two geometry shaders, `gs_main_simple` (for Method 1) and `gs_main_solution` (for Method 2). Geometry shader `gs_main_simple` makes no changes to the triangles it receives, and it should not be modified. Geometry shader `gs_main_solution` is initially the same as `gs_main_simple` but should be modified in one of the problems below.

The level of detail used for the spiral is controlled by variable `opt_segments`. Its value can be modified by first pressing the `TAB` key until "Number of segments per spiral" appears, and then pressing `+` and `-` to adjust its value. Check out and compile the code, and make sure that it runs correctly. **For this assignment** only modify file `hw04-shdr.cc`.



**Problem 1:** Notice that with Method 1 and 2 pale blue is used for both sides of the spiral. In this problem modify the fragment shader, `fs_main`, so the spiral appears like the one on the image to the right. Details are given in the subproblems. Changes to the fragment shader affect Methods 1 and 2, but not zero.

(*a*) Modify the fragment shader so that the front surface is gold and has the texture applied and so that the back surface is purple. Do this *without* using the color attribute (fragment shader input), instead rely on the material properties that are set in `render_spiral2`. Note that these material properties are uniform variables.

For the names of the predefined material property uniforms available in the fragment shader see Section 7.4.1 in the OpenGL Shading Language spec version 4.5. To determine whether a fragment is from the front or the back of a primitive see Section 7.1 in the spec.

(*b*) The surface of the spiral shows a draft this homework assignment handout, using black for the text and yellow for the background. Modify the fragment shader so that there is no spiral wherever the texture is black (or a dark color). In other words, modify it so that you can see through the spiral wherever the texture is black.

To determine if a texel is dark add up the red, green, and blue components and check if the sum is less than .05. If a texel is dark, then prevent the corresponding fragment from being written. To see how to prevent a fragment from being written see Section 6.4 in the OpenGL Shading Language spec version 4.5. (This section describes jump-like statements in the shading language.)

**Problem 2:** The spiral's appearance is a little unnatural because it is two-dimensional. Modify the shaders (except `gs_main_simple`) and surrounding code so that when Method 2 is active the spiral has some thickness. (See the illustration at the beginning of this assignment.)

Use geometry shader `gs_main_solution` to render multiple triangles as follows. Let $V_1$, $V_2$, and $V_3$ denote the vertices of a triangle at the input to the geometry shader. The geometry shader should emit $V_1V_2V_3$ (as it already does) and also $U_1U_2U_3$ where $U_i = V_i + \vec{d}$ and $\vec{d}$ is a vector of length 0.1 pointing from ball $i-1$ to $i$ (the balls that the spiral segment is between). The geometry shader should also emit triangles for an inner and outer edge which joins the two spirals. The edge should be the gray color set in `render_spiral2`. See the illustration.

The solution to this problem will require modifying all three shaders, and modifying the shaders' interface blocks (such as `Data_to_GS`). **No changes** should be necessary in file `hw04.cc`. It's okay to modify this file to help in debugging your code, but the solution itself should only be in `hw04-shdr.cc`. If you believe otherwise, please contact the instructor. Divide work appropriately between the vertex shader and geometry shader.

Here are some things to watch out for:

- Be sure to adjust the maximum number of vertices specified for the geometry shader output. If this number is too low execution will end with an error.
- When you modify an interface block (such as `Data_to_GS`) be sure to modify the other interface block with the same name (one is a shader output, the other is a shader input).

Here are some debugging tips:

- Check for shader code compilation errors when your program starts. The errors are sent to `stdout` and should appear in the shell or in gdb, depending on how you started the program.
- If execution ends with an OpenGL error, you can get a more detailed error message by turning on OpenGL debugging. To do this change `false` to `true` in `popengl_helper.ogl_debug_set(false);` near the end of `hw04.cc`.
- Three UI-controlled variables are available for debugging. They are `debug_bool.x`, `debug_bool.y`, and `debug_float`. The Booleans can be toggled with `d` and `D`. The float can be adjusted by pressing `Tab` until "Debug Float" appears and then press `+` and `-` to adjust the value.

## LSU EE 4702-1 Homework 5 Due: ≈ 12 November 2014

The following assignment is based on the code package for Homework 4. The questions are asking about the code provided for the assignment, not about code written as part of the solution. In other words, you can answer these questions without having yet solved Homework 4.

**Problem 1:** Compare the amount of data sent from CPU to GPU in Methods 1 and 2 from the Homework 4 code. (This question is based on the original code, not code written as part of the solution.)

Let $n$ denote the number of balls (`chain_length`) and $s$ the number of segments (`opt_segments`). Determine the amount of data, in bytes, sent to the GPU per frame for Method 1 and Method 2.

**Problem 2:** Consider the amount work to compute vertex coordinates and normals done by `render_spiral1` and by the vertex shader `vs_main`. The vertex shader is actually computing things multiple times that the code in `render_spiral1` computes just once. Identify such redundant computation. *Hint: It happens in two different ways, one way results in things computed twice, another way results in things computed s times.*

**Problem 3:** Explain how some of the problems above could be avoided if the CPU specified vertices using line strips and the geometry shader used lines as inputs. Of course, the geometry shader's output would be triangles.

Each point on the line would correspond to a `p` in the `t` loop in the `render_spiral1` routine.

# LSU EE 4702-1 Homework 6 Due: 24 November 2014

**Problem 0:** Follow the class account instructions for obtaining the assignment, substitute hw06 where needed. This homework code is based on Homework 4/3, and the physics simulation is identical. Like those assignments it renders a spiral. The spiral can be rendered using two routines, `render_spiral2` and `render_spiral3`; the `m` key switches between them. Routine `render_spiral2` is based on the Homework 4 solution, and initially `render_spiral3` is identical to `render_spiral2`, it will be modified for this assignment.

Routine `render_spiral2`, rather than sending actual vertex coordinates to the GPU, sends indices. The $x$ component of a vertex "coordinate" is the ball number (the value of `i` in `render_spiral1`), the $y$ component is the relative position between ball `i-1` and ball `i`, ranging from 0 to `opt_segments-1`, and the $z$ component indicates whether the vertex is on the inner or outer edge of the spiral. The code in vertex shader `vs_main` in file `hw06-shdr-triangles.cc` uses these values to compute the actual vertex coordinate and normal, and to compute the texture coordinates.

The level of detail used for the spiral is controlled by variable `opt_segments`. Its value can be modified by first pressing the `TAB` key until "Number of segments per spiral" appears, and then pressing `+` and `-` to adjust its value. Check out and compile the code, and make sure that it runs correctly.

**For this assignment** modify files `hw06.cc` and `hw06-shdr-lines.cc`. Solutions to the non-code questions can be provided in any of the following forms:

- As comments in one of the files above.
- As plain text in a separate file named `hw06.txt` in the `hw06` directory.
- In portable document format in a separate file named `hw06.pdf` in the `hw06` directory.
- Submitted on paper.
- E-mailed. If E-mailing, plain text and PDF are preferred.

**Problem 1:** The code in `render_spiral2` is based on the solution to Homework 4. As discussed in Homework 5 Problem 3, the vertex shader used for `render_spiral2` in Homework 4 wastes computation because it computes information both for inner and outer vertices but only sends one of these to the geometry shader. The problem suggests using line strips to solve that problem. The solution to Homework 5 presented some details on how this would be done, but it did not show any code. For this problem, modify `render_spiral3` and the shaders so that it avoids the waste.

For this problem you must modify code both in `render_spiral3` in file `hw06.cc` and the code in `hw06-shdr-lines.cc`, both will be collected by the TA-bot. Initially the code in `render_spiral3` is nearly identical to `render_spiral2` and `hw06-shdr-lines.cc`, except that it invokes shaders from file `hw06-shdr-lines.cc` (which you should modify). Routine `render_spiral2` invokes shader code from `hw06-shdr-triangles.cc`, which you should not modify.

Here are some things to watch out for:

- Be sure to adjust the maximum number of vertices specified for the geometry shader output. If this number is too low execution will end with an error.
- When you modify an interface block (such ad `Data_to_GS`) be sure to modify the other interface block with the same name (one is a shader output, the other is a shader input).

Here are some debugging tips:

- Check for shader code compilation errors when your program starts. The errors are sent to `stdout` and should appear in the shell or in gdb, depending on how you started the program.
- If execution ends with an OpenGL error, you can get a more detailed error message by turning on OpenGL debugging. To do this change `false` to `true` in `popengl_helper.ogl_debug_set(false);` near the end of `hw04.cc`.
- Three UI-controlled variables are available for debugging. They are `debug_bool.x`, `debug_bool.y`, and `debug_float`. The Booleans can be toggled with `d` and `D`. The float can be adjusted by pressing `Tab` until "Debug Float" appears and then press `+` and `-` to adjust the value.

**Problem 2:** Compare the performance of `render_spiral3` to `render_spiral2`. To see any difference at all the value of `opt_segments` will need to be increased, so do this if necessary.

(*a*) Show a table comparing the performance of the two methods versus `opt_segments`. In the table show CPU and GPU OpenGL time. Also indicate which GPU you are using.

(*b*) Explain how varying `opt_segments` changes the proportion of the work performed by the three shader stages (vertex, geometry, and fragment).

(*c*) In a solution to an exam problem, the computation of `sin(theta)` and `cos(theta)` was moved out of the vertex shader. Instead the pre-computed sine and cosine values would be read from a buffer object. Would this increase or decrease the relative benefit of `render_spiral3`?

**Problem 3:** When the Homework 6 code starts it prints information about the available GPUs, using a CUDA API. CUDA isn't used for anything other than printing this information. Here is some sample output:

```
GPU 0: Tesla K20c @ 0.71 GHz WITH 5119 MiB GLOBAL MEM
GPU 0: CC: 3.5  MP: 13  CC/MP: 192  TH/BL: 1024
GPU 0: SHARED: 49152 B  CONST: 65536 B  # REGS: 65536
GPU 0: L2: 1280 kiB   MEM to L2: 208.0 GB/s  SP 1760.9 GFLOPS  OP/ELT 33.86
GPU 1: GeForce GTX 780 @ 1.02 GHz WITH 3071 MiB GLOBAL MEM
GPU 1: CC: 3.5  MP: 12  CC/MP: 192  TH/BL: 1024
GPU 1: SHARED: 49152 B  CONST: 65536 B  # REGS: 65536
GPU 1: L2: 1536 kiB   MEM to L2: 288.4 GB/s  SP 2348.9 GFLOPS  OP/ELT 32.58
```

In the example above information on two GPUs is given. (My machine and all of the graphic lab computers have two GPUs.) The first GPU is a Tesla K20c and has Compute Capability 3.5. The number of streaming multiprocessors (abbreviated `MP` in the output) is 13. The maximum number of threads per block is 1024.

Suppose the time step routine (a version of `time_step_cpu`) were going to be run on the GPU.

(*a*) Based on the program's default parameters and a GPU on one of the lab machines, how many blocks should be launched and how many threads per block? (The number of blocks and number of threads per block are collectively called the *launch configuration*.) *Please* indicate which GPU and its relevant statistics when answering this question.

(*b*) Do you expect the code to run efficiently with this launch configuration? Explain.

# 14 Fall 2013

## LSU EE 4702-1 Homework 1 Due: 11 September 2013

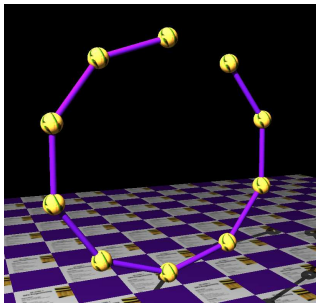
*Some of the effort for this homework assignment is in learning to use the various pieces of software, such as a text editor. Those less familiar with Linux software development procedures might seek out a more knowledgeable classmate to minimize frustration and wasted time.*

**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 line of balls drop to the platform. *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. Press `1` to set up scene 1, press `2` to set up scene 2.

Look at the comments in the file `hw1.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` sets up scene 2, in which the balls form a pendulum. Modify scene 2 (look for code `ball_setup_2`) so that it starts with the balls arranged in a circle with the balls far enough apart so that the springs are stretched (see the screenshot on the right). Try to predict how the balls will move, then compare to what they actually do, using the space bar to advance frame by frame.



**Problem 2:** An obvious problem with the code is that the balls interpenetrate (overlap in space). Fix this by modify the program so that spring forces are added for interpenetrating balls. (This should not affect the existing springs.) The changes should go in the time step routine and will require adding a loop nest comparing every pair of balls. (Comparing every pair of objects for interpenetration is time consuming when the number of objects is larger than it is here. Later in the semester less costly methods for finding interpenetration will be described.)

- Only code in the `time_step_cpu` routine needs to be modified for this problem.
- The radius of a ball is in the `radius` member of the `Ball` class. For example, `balls[i].radius` is the radius of ball `i`.

# LSU EE 4702-1 Homework 2 Due: 9 October 2013

**Problem 0:** Follow the instruction on the `http://www.ece.lsu.edu/koppel/gpup/proc.html` page for programming homework work flow, substituting hw2 where appropriate. Compile and run the homework code unmodified. It should show a chain of balls swinging next to a starkly illuminated large chocolate cone. (Cone refers to the shape, it's not a chocolate ice cream cone.)

The green thing on the left-hand side is the light, it should leave a brightly lit spot on the chocolate cone but it does not because the cone is rendered using large primitives. In this assignment that problem will be fixed, but that's the problem below. This problem is just to familiarize yourself with the code.

The code for rendering the chocolate cone and the links in the chain is in the class `Cone` near the top of the file `hw2.cc`. The routine `Cone::render` is called when it's time to render a cone. The chocolate cone is an instance named `cone_fixed`, the chain links are all a second instance named `link`.

In this assignment the `Cone` class will be rendered to fix light problem by using a larger number of primitives to tessellate the cone shape.

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. Press `1` to set up scene 1, press `2` to set up scene 2.

Look at the comments in the file `hw2.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:** Modify the member function `render_proba` so that it constructed from `10 * opt_lod` quads along the circumference and `opt_lod` quads along the length, for a total of $10L^2$ quads, where $L$ is the value of global variable `opt_lod`. (LOD is an abbreviation for level of detail.) Variable `opt_lod` can be adjusted using the keyboard by pressing the `Variables` section of the comments at the top of the code.

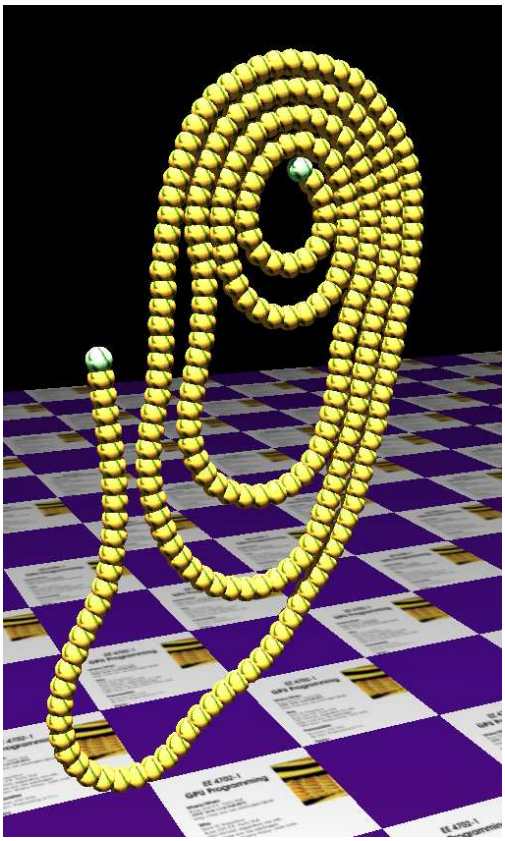
**Problem 2:** Add code to `render_probb` and elsewhere so that a buffer object will be used to render the cone.

- Declare storage for the buffer object name and other variables in the Cone class.
- The buffer object should be updated whenever the level of detail changes.
- Be careful not to leak memory. That's a common problem in assignments like this. (A memory leak is when memory is repeatedly allocated but not freed.)

Use the following resources to help solve the problem: The `demo-7-vtx-arrays.cc` example shows how to use buffer objects. The file is available several different ways, including a colorized version meant for a Web browser,
`http://www.ece.lsu.edu/koppel/gpup//code/gpup/demo-7-vtx-arrays.cc.html`. If you followed the account setup instructions you can find `demo-7-vtx.arrays.cc` in the `gpup` directory in your class account. (Executing `svn update` in that directory will update those files with the latest changes.)

The documentation for buffer objects can be found in Chapter 6 of the OpenGL 4.3 compatibility profile specification which can be found at
`http://www.ece.lsu.edu/gp/refs/glspec43.compatibility.20120806.pdf`. Commands for specifying vertices and attributes can be found in Section 10.3, however the section might be difficult to read because of the many different ways vertices can be specified.

**LSU EE 4702-1** **Homework 3** **Due: 20 November 2013**



**Problem 0:** Follow the instruction on the `http://www.ece.lsu.edu/koppel/gpup/proc.html` page for programming homework work flow, substituting hw3 where appropriate. Be sure to update the include directory in addition to checking out the homework.

Compile and run the homework code unmodified. It should initially show a string of balls in the shape of a spiral with the first and last ball fixed in space, the spiral will unravel, see the screenshot to the right.

Unless the code is run on a surprisingly fast system, the frame rate should be low, resulting in jerky movement. The physics is computed on the GPU using CUDA, the goal of this assignment is to improve the performance of the code so that animation is smooth.

For this problem, familiarize yourself with the operation of the simulation (that is, play a little). To move around the scene press `e` and then the arrow keys, PageUp, and PageDown. Press `1` to set up scene 1 (vertical string), press `2` to set up scene 2 (pendulum), press `3` to set up scene 3 (the unraveling spiral). 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. Pressing the space bar will pause and then single-step the simulation. Press `p` to resume (or to pause again).

To facilitate experimentation two Boolean variables have been provided, on CPU code `c.opt_test1` and `c.opt_test2`, on CUDA code `dc.opt_test1` and `dc.opt_test2`. Their values are shown on the second line of the display after opt_textX. To change their values press `x` or `X`.

Pressing `a` will switch the method used to run the physics (time step) routine, the method appears near the upper left (near Physics On:). Initially it will show GPU Sol, indicating the code that should be modified for this problem. The other two values are CPU and GPU Base. The GPU Base code is a copy of the code to be modified. As you work on this assignment you can compare the performance of your code, GPU Sol, to the original code, GPU Base, by switching between the two. If the CPU code was compiled without optimization then unoptimized will flash when running the CPU code. Since we are not comparing to the CPU in this problem, the CPU code can be left unoptimized.

A part of the solution involves choosing good values for the grid size (number of blocks) and block size (number of threads per block). These can be modified from the UI by pressing the `TAB` until Grid Size or Block Size appears, and then by using the `+` and `-` keys to adjust them. Gravity can also be adjusted, setting it to a low value provides something like slow motion.

The performance of the physics code is shown on the second line to the left of Time Per Time Step, it is the amount of time needed to compute a single simulation time step. The goal, of course, is to minimize this value. The first line shows other performance numbers. FPS shows the frame rate in frames per second. XF indicates the rate at which frames are updated by the code relative to how fast the device displays frames. A value of 1 is optimal. Under some circumstances the class library code may not be able to synchronize with the hardware, in which case it will update at 30 FPS. GPU.GL shows the amount of time spent on the GPU running OpenGL code per frame, it is shown in milliseconds and as a percent of time. What we do in this assignment should not directly affect this number. GPU.CU shows the amount of time running CUDA code per frame (and as a percent of total time). **Do not** use this number to compare the performance of your code

because its value depends on the number of time steps performed. CPU shows the amount of time per frame spent by the CPU on rendering. It does not include physics. Steps / s shows the number of simulation time steps performed per second. This number should be about 10000, if it's lower then the physics code is too slow to keep up. This will initially be the case for the GPU sol code, but when the assignment is finished it should be up to around 10000. When Steps / s is less than 10000 the simulation will appear to be running slowly (like slow motion) and motion may be jerky. Physics shows the amount of time spent computing physics, whether on the CPU or GPU.

When solving this assignment focus on Time Per Time Step, it can always be used to compare the performance of the different code versions (CPU, GPU Base, and GPU Sol).

The first place to look for CUDA documentation is the CUDA C Programming Guide, `http://docs.nvidia.com/cuda/cuda-c-programming-guide/`, start with Chapter 2, Programming Model, it provides a quick overview of CUDA concepts. See also Section 3.2. CUDA C is summarized in Appendix B. The link above takes you to a place providing documentation for the entire CUDA Toolkit. A complete list of CUDA runtime functions (the ones we call from CPU code) can be found in the CUDA Runtime API manual. The Programming Guide describes CUDA C, but a complete list of library functions callable from CUDA C can be found in the CUDA Math API manual.

For this assignment the only CUDA runtime calls needed are for allocating memory, and examples of those can be found in the assignment code.

In this assignment the GPU Sol kernel and related code will be modified to improve performance. Here is a summary of how the code works.

The CUDA code is located in file `hw3-sol-cuda.cu`. It consists of four kernels. At the beginning of every frame kernel `data_pack` is called. It initially does nothing but should be modified to *pack* some of the ball data. Kernel `time_step_gpu_sol_part_1` updates the balls' velocities, it is launched once per time step. (There are many time steps per frame.) Kernel `time_step_gpu_sol_part_2` updates positions (and also velocities for platform collision), it is also called once per time step. Kernel `data_unpack` is called once per frame, after the other kernels, it initially does nothing but should be modified to *unpack* data back into the `Ball` structure. These kernels are launched from the routine `frame_callback` in file `hw3.cc`.

In kernels `time_step_gpu_sol_part_1` and `time_step_gpu_sol_part_2` each thread handles one ball. There are `c.chain_length` balls, which is set to 256. As we should know, 256 is not a large number of threads for a GPU, and so we should expect the GPUs hardware to be underutilized.

The block and grid sizes are initially set to arbitrary values and these can be adjusted by using the UI, initial values can be changed by modifying routine `World::init` in `hw3.cc`.

Several variables are used by both GPU and CPU code. For convenience, they have been placed in structure `CPU_GPU_Common` (see file `hw3.cuh`), this structure is instantiated in `World`, as `c`, and is a CUDA constant, named `dc`. One member is `chain_length`, which is the number of balls. It can be accessed on the CPU as `c.chain_length` and on the GPU as `dc.chain_length`. Members may be added to this structure. The structure is copied to the GPU by routine `data_cpu_to_gpu_constants`, which is called at initialization and whenever a key is pressed.

CUDA runtime API functions (the cuda functions called by CPU code) are wrapped by a macro named `CE`. This checks the return code of the CUDA function for errors. If there is an error it will end execution with a message.

The `Makefile` is set to compile the code without CPU optimization but with CUDA optimization. If you would like to use the cuda debugger (`cuda-gdb`), you will need to modify `Makefile`. Add a `-G` to the `OPT_FLAGS` variable. (You can uncomment a line with this flag added.) Then re-build (press `F9` or type make from the command line). The code will run excruciatingly slowly, so make sure your program reaches the problematic part quickly.

Adding `-O3` to `OPT_FLAGS` will turn on CPU optimization.

In the wee hours of the 21st tnphe TA-bot will copy the following files: `hw3-sol-cuda.cu`, `hw3.cc`, and `hw3.cuh`. Limit changes to these files. If code in another file needs to be modified, contact Dr. Koppelman.

You can place text answers to questions in file `hw3-sol-cuda.cu`, either in comments or defined out. You can also E-mail or turn in paper copies.

**Problem 1:** Using the UI modify the grid and block sizes and observe the impact on performance.

(*a*) Run the GPU Base code and manually set the grid and block sizes to what you would expect to be good choices. Note the performance. Try varying these to get worse or better performance. Note which values are better, and examples of ones which are worse. Note how sensitive the program is. Does it do just as well even with fewer blocks?

(*b*) Modify the code (in routine `World::init`) so that it starts off with a good configuration. Look for comments "Helpful information" and "Put Solution Here" in this routine. (The solution to this part will have to be changed to reflect modifications made due to the solution to Problem 3. If Problem 3 isn't solved this part can still get full credit. If you prefer don't work on this part until after Problem 3.)

**Problem 2:** The CUDA code accesses the same `Ball` structure as the CPU code, which, though convenient, is not GPU-friendly because memory requests will only partially be used. Fix it. *Spoiler: It won't help performance.*

(*a*) Modify the code to use a GPU-friendly organization for `Ball::position`.

For your convenience, a variable is already present for the new storage for ball positions, `c.d_pos` (CPU name) and `dc.d_pos` (GPU name).

- You'll need to allocate device storage for the positions, do that in routine `World::init`. Device storage for `balls` is already allocated in that routine, use that code as an example.
- Modify kernel `data_pack` to reorganize the data for the GPU (reading from the `Ball` structure and writing to your new array).
- Modify `time_step_gpu_sol_part_1` and `time_step_gpu_sol_part_2` so that they read and write positions from and to your new array.
- Modify `data_unpack` so that it puts data back into the `Ball` structures.
- Optional: include the ball radius in the $w$ component of position and use it where needed.

(*b*) Measure the performance change by comparing GPU Sol (the code you modified) to GPU base. Explain why the change should not have made much of a difference in this code. (This can change depending on how the next problem is solved.) To answer this question look at the part of the code that's consuming the most execution time. It's only a few lines.

**Problem 3:** This is the problem where performance will be improved by a substantial amount. The code uses a brute-force $O(n^2)$ method (where $n$ is the number of balls) for resolving interpenetration—that we won't change. The GPU Base and Sol kernels use one thread per ball, so each of these $n$ threads has to check almost $n$ other balls for collisions. We know that $n = 256$ (the number of balls the code uses) threads is not enough to get good utilization on current NVIDIA GPUs.

Modify the code so that the `time_step_gpu_sol_part_1` kernel can use more than $n$ threads. For example, if it were launched with $2n$ threads, a pair of threads would handle one ball and each one would only need to check about $n/2$ other balls for contact. Try to write your code to use an arbitrary number of threads per ball, this can be based on a hard-coded constant (`const int thds_per_ball=4;`) or the code might compute this based on the total number of threads (this is what the spring demo does).

Be sure to update the launch configuration so that it chooses a good number of configuration for your new kernel. (If you like, you can provide a separate configuration for `time_step_gpu_sol_part_1` and the other kernels.)

The spring demo code solves a similar problem, but **unlike the spring demo** the `time_step_gpu_sol_part_1` kernel must compute both near neighbor forces and resolve penetration.

# 15 Fall 2012

## LSU EE 4702-1 Homework 1 Due: 12 September 2012

*Most 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 1:** 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 line of balls drop to the platform. *Promptly report any problems.*

Use the arrow keys, PageUp, and PageDown to move around the scene. Press lower-case 'b' and then use the same keys to move the fixed ball around. Press 'l' to move the light around and 'e' to move the eye. Look at the comments in the file `hw1.cc` for documentation on other keys. Try turning gravity on and off and changing the spring constant. *Note: There is nothing to turn in for this first problem.*

**Problem 2:** In the hw1.cc program the balls interpenetrate (overlap in space). Modify the program so that the spring forces prevent (or nearly prevent) interpenetration. Do this by making the relaxed distance of a spring connecting two balls equal to the sum of their radii. Also, the spring constant used for repulsion should be ten times that used for attraction.

- Only code in the `time_step_cpu` routine needs to be modified for this problem.
- The radius of a ball is in the `radius` member of the `Ball` class. For example, `balls[i].radius` is the radius of ball `i`.

# LSU EE 4702-1 Homework 2 Due: 10 October 2012

*Follow the* Programming Homework Work Flow *instructions on the procedures page, substituting* `hw2` *for* `hw1`.

The `hw2` code, based on `demo-8-texture`, draws a helix. Due to missing normals, the surface of the helix will not be shaded properly (see Problem 1).

The comments at the top of the file describe controls for the simulation. For this assignment controls have been added to change the amount of detail in the helix (search for "Big Circle" and "Small Circle"), to change how vertices are ordered (look at the `i` and `m` options), and to force computation and communication (look at `d` and `r`).

**Problem 1:** Run the unmodified `hw2` code and pay attention to the way the helix is lit. Notice that the shading of the helix is wrong. That's because normals have not been specified. Modify the code so that normals are correctly specified. To do this new arrays and a new buffer object will have to be added.

**Problem 2:** The use of triangle strips reduces vertex redundancy compared to individual triangles. However for the `hw2` program there is still redundancy. In particular, point `s1` (search for "Construct a Helix" and scroll down to the `j` loop) is the same as `s0` in a prior `i` iteration.

By using `glDrawElements` instead of `glDrawArrays` one can avoid this redundancy. That is, the command sends each point to the GPU once and a separate index array specifies which of those points are used, a point can be used multiple times.

(*a*) Modify the code to use `glDrawArrays` when variable `opt_use_elements` is true. (Use key `m` to change the value of `opt_use_elements`.) Some code is already in place, including code declaring and initializing the index array and the call to `glDrawElements`. However, the existing code still adds the same point to the `helix_coords` list multiple times. The index array (`indices`) is set to 0, 1, 2, ..., which makes `glDrawElements` equivalent to `glDrawArrays`.

Modify the code so that each point is added only once, making appropriate changes to `prep_indices` (which is used to set `indices`) so that the helix is drawn correctly. See Section 10.5 of OpenGL version 4.3 for use `glDrawArrays`.

(*b*) The code has a variable `opt_use_strips`. Modify the code so that when it's true the helix will be rendered using triangle strips and when it's false it will be rendered using individual triangles. *Note: This part did not appear in the original assignment.*

**Problem 3:** In this problem try to measure the impact of using `glDrawElements` to reduce the redundancy.

(*a*) Compute the amount of data sent from the CPU to the GPU in terms of the number of triangles used in the helix with and without `glDrawElements`. Use $T$ to denote the number of triangles.

(*b*) Measure the impact on performance of this difference in data size. Use the various program options to make this performance difference large. Explain which options were used and show the results.

(*c*) Try to determine if there is any difference in performance due to computation. There would be less computation using `glDrawElements` if the vertex shader were run on each vertex only once. But does that actually happen? Explain which options were used and show the results.

# LSU EE 4702-1 Homework 3 Due: 16 November 2012

*Follow the* Programming Homework Work Flow *instructions on the procedures page, substituting* `hw3` *for* `hw1`*. Also, update the include directory.*
*Use* `http://www.ece.lsu.edu/gp/refs/GLSLangSpec.4.30.6.pdf`*, the OpenGL 4.3 Shading Language spec, as a reference. The code in this assignment requires OpenGL 4.3, which should be installed on* `snow`*,* `ice`*, and* `frost`*.*

**Problem 0:** Compile and run the code unmodified, a yellow helix should be displayed. The arrow keys initially move the eye position, familiarize yourself with these keys, the others in the comments of `hw3.cc`, and the keys described in this problem.

The code is set up to run two different shader programs, called "SP Geo Shade1" and "SP Geo Shade2." The name of the active shader program is displayed on the lowest green line, pressing "s" will switch from one to the other. The two shader programs use the same vertex and fragment shader, but different geometry shaders. The problems indicate which geometry shader to modify.

For this assignment there are three user-controlled variables of interest. (See the Variables in the Keyboard Commands section of the comments in hw3.cc.) *Segs Per Helix Rev* controls the number of segments in one revolution of the helix. Larger numbers mean smaller triangles. *Segs Per Wire Rev* controls the number of segments in one revolution of the wire. Larger numbers mean smaller triangles. *Door Angle* See Problem 3.

**Problem 1:** The unmodified version of the code does everything needed to display a texture on the helix, except for sending texture coordinates. (The texture is this assignment.) Rather than generating texture coordinates on the CPU and sending them to the GPU, the goal here is to compute the texture coordinates on the GPU using a vertex shader. The vertex shader in `hw3-shder.cc` currently just copies the texture coordinates from the CPU unmodified. Since the CPU never sends texture coordinates these will hold some initial value, perhaps $(0, 0)$.

Modify the vertex shader in `hw3-shdr.cc` so that it computes texture coordinates, instead of reading them from `gl_MultiTexCoord0`.

- Display the texture so that it appears right-side up, undistorted, and is recognizable.
- Optional: The size and position of the texture should not change when number of segments per helix or wire are changes.

Note: This problem is easy. Texture coordinates can be computed in terms of the values in `helix_index`. For the optional part new uniform variables need to be defined.

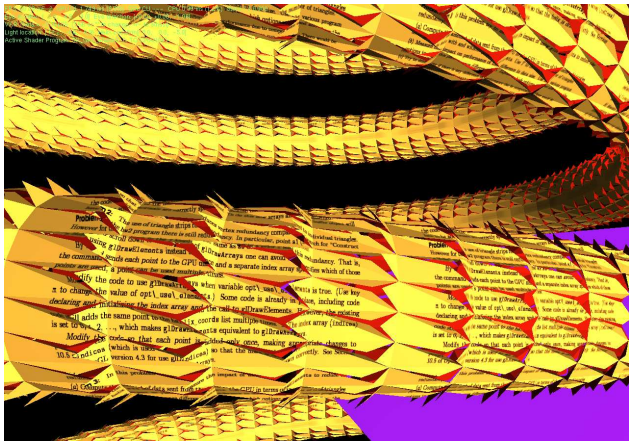
**Problem 2:** Both geometry shaders assign a variable named `type_a`, but the variable is not used. Modify `gs_main_helix` and other code so that textures are not displayed for triangles for which `type_a` is true. Do this by declaring new variables for communicating the value of `type_a` from the geometry shader to the fragment shader. Use this variable in the fragment shader to avoid doing a texture lookup.

- Declare a new variable for communicating between the geometry shader and the fragment shader. **Do not** use an existing variable, tempting though it may be.
- Set the appropriate interpolation for the new variable.
- Use the variable in the fragment shader to control whether textures are used.

Note: The fragment shader already avoids texture lookup for the back side of primitives, it also applies a reddish tint to such primitives. Make sure that reddish tint is not applied to the front of primitives for which texture lookup has been suppressed.

See Section 4.3.4 and 4.3.6 of the OpenGL 4.3 specification for details on how to declare shader input and output variables.

This problem is also fairly easy, but it's not a one-liner.



**Problem 3:** Modify geometry shader `gs_main_helix2` so that each `type_a` helix triangle is split into three triangles under control of CPU variable `opt_door_angle`, call the value of that variable $\theta$. Each new triangle consists of two vertices from the original triangle, and (initially) the original triangle center. These new triangles will rotate, the amount based on $\theta$, around an axis formed by the original edge. Though the triangles rotate, their shape should not change. The appearance might be of doors opening up, see the screen shot to the right.

More precisely, let $p_0$, $p_1$, and $p_2$ denote the vertices of the original triangle, and let $c = (p_0 + p_1 + p_2)/3$ denote the centroid, $\overrightarrow{n} = \overrightarrow{p_0p_1} \times \overrightarrow{p_0p_2}$ denote the triangle normal, and $\hat{n}$ be the normalized normal. The original triangle, $p_0p_1p_2$, is to be split into new triangles $p_0p_1c_2$, $p_0c_1p_2$, and $c_0p_1p_2$ where $c_0$ is given by

$$c_0 = q_0 + \overrightarrow{q_0c} \cos\theta + l_0\hat{n} \sin\theta,$$

where $q_0$ is the point on line $p_1p_2$ closest to $c$ and $l_0$ is the distance from $q_0$ to $c$. Points $c_1$ and $c_2$ are defined similarly.

Geometry shader `gs_main_helix2` should emit these three triangles.

- Using an appropriate method, provide a value of CPU variable `opt_door_angle` to the shader.
- Calculation of triangle coordinates should not be performed on clip-space coordinates.
- The `normal_e` output of the geometry shader should be for the respective triangle, not the wire.
- Note that the output of the geometry shader should be both clip-space coordinates (for the fixed-functionality) and eye-space coordinates (for our fragment shader).

See the OpenGL Shading Language 4.3 specification for information on the shader language, including functions for dot product, cross product, and normalization.

Here are some tips that apply to all problems:

- Shader program error messages appear when the main program starts, not when code is compiled.
- The error message line number might be off by one.
- The following shader code error message can be ignored, (but don't ignore other messages that occur with it): "warning C7547: extension GL EXT geometry shader4 not supported in profile gp5fp."

# LSU EE 4702-1 Homework 4 Due: 20 November 2012

*For the problems below refer to Chapter 2, Programming Model, and Chapter 4, Hardware Implementation, in the NVIDIA CUDA Compute Unified Device Architecture Programming Guide available locally via*
`http://www.ece.lsu.edu/gp/refs/CUDA_C_Programming_Guide.pdf`.

**Problem 1:** The kernel below is launched in a configuration of grid size of $(1024, 1, 1)$ and block size of $(256, 1, 1)$. (The components are $(x, y, z)$.)

```
__global__ void dots_iterate1() {
  int thread_count = /* Omitted so things aren't too obvious. */;
  int idx_start = threadIdx.x + blockIdx.x * blockDim.x;
  for ( int idx = idx_start; idx < array_size; idx += thread_count )
    b[idx] = v0 + v1 * a[idx]; }
```

(*a*) How many threads are there? How many blocks? How many warps?

(*b*) Suppose the array size is $5 \times 10^5 = 500\,000_{10} = 7\,a120_{16}$ elements. How many threads will perform two iterations? One iteration? Zero iterations?

(*c*) A GPU has four multiprocessors. Explain why launch configurations of four and eight blocks are each better than a launch configuration of six blocks. Use the code above as an example.

**Problem 2:** Consider the kernel code below. It is launched with a block size of 512 threads and a grid size of 64 blocks. The array has $2^{20} = 1\,048\,576$ elements.

```
__global__ void dots_iterate15()  {
  int thread_count = /* Omitted so things aren't too obvious. */;
  int tid = threadIdx.x + blockIdx.x * blockDim.x;  // Not used in code.
  int idx_start = blockIdx.x + threadIdx.x * gridDim.x;

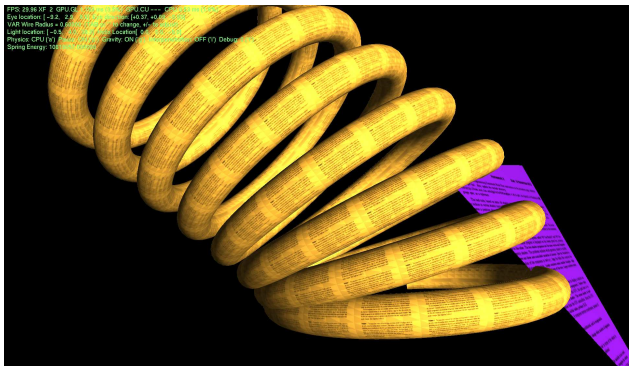
  for ( int idx = idx_start; idx < array_size; idx += thread_count )
    b[idx] = v0 + v1 * a[idx];
}
```

(*a*) The table below shows various information about selected threads in the launch described above. The first three columns should be self-explanatory. The three columns headed `idx` show the element number (value of `idx` in the code above) accessed by the respective thread in the first, second, and third iteration of the `for` loop. Each row has at least one column filled. Fill the remaining columns.

| tid | blockIdx.x | threadIdx.x | -- idx --- First Iter | -- idx --- Second Iter | -- idx --- Third Iter |
|---|---|---|---|---|---|
| 0 | | | | | |
| 1 | | | | | |
| 2 | | | | | |
| | | | 0 | | |
| | | | 1 | | |
| | | | 2 | | |
| | 1 | | 1 | | |

## LSU EE 4702-1 Homework 5 Due: 30 November 2012

*Follow the* Programming Homework Work Flow *instructions on the procedures page, substituting* `hw5` *for* `hw1`*. Also, update the include directory.*
*Use* `http://www.ece.lsu.edu/gp/refs/CUDA_C_Programming_Guide.pdf`*, the NVIDIA CUDA Compute Unified Device Architecture Programming Guide, as a reference. The code in this assignment requires OpenGL 4.3 and CUDA, which should be installed on* `snow`*,* `ice`*, and* `frost`*.*



**Problem 0:** Follow the class procedures for homework but substitute hw05 for the assignment. Compile and run the code. It should display our familiar helix, but this time it's a simulated spring, and it should be drooping under gravity, see the illustration to the right. The physics can be performed by both CPU and GPU, by default it runs on the CPU, pressing 'a' will toggle it.

The CPU code can prevent interpenetration, this option is initially off, it can be toggled by pressing 'i'. The option was turned off when the image above was captured, notice that the bottom loops of the spring interpenetrate. The code for interpenetration can be found in `time_step_cpu`, it uses a brute-force approach, comparing every segment of the helix to every other, making it very slow. The goal of this assignment is **not** to use a clever broad-phase collision detection scheme to avoid $O(n^2)$ comparisons, but instead use the GPU to do the calculations faster. (In "real life" one might use both broad-scale collision detection and a GPU to do calculations.)

**Problem 1:** The routine `time_step_cpu` contains the CPU interpenetration code, clearly identified by comments. For this problem put the interpenetration code in CUDA kernel `time_step`, in file `hw5-cuda.cu`. The `time_step` kernel is launched in a configuration with one thread per helix segment. Each thread should test "its" segment against other segments. A correct solution will be faster than the CPU, but should still be slow.

**Problem 2:** By GPU parallelism standards, the number of threads launched for the `time_step` routine is not large. In the default configuration there are 160 segments, just five warps. That will underutilize any GPU.

For this problem, add a new kernel to perform the interpenetration test. The new kernel should be launched with more threads than segments, taking advantage of more parallelism. Modify the existing `time_step` kernel so that it does not modify `helix_position` (but it still updates `helix_velocity` and the other two state variables). The new kernel should compute forces based on interpenetration tests, use that to update velocity, and then update the position.

A good solution of this problem requires the use of shared memory, atomic operations, and synchronization. That will be covered next week.

# 16 Fall 2011

## LSU EE 4702-1 Homework 1 Due: 15 September 2011

**Problem 1:** 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 the ball bouncing on the tiled platform with some prisms made of syllabi, similar to the scene shown in class. *Promptly report any problems.*

Use the arrow keys to move around the scene. Look at the comments in the file `hw1.cc` for documentation on other keys. Try turning gravity on and off. *Note: There is nothing to turn in for this first problem.*

**Problem 2:** In the code as assigned, the velocity of the ball is determined only by the acceleration of gravity (except when it hits the platform). Modify the code in `time_step_cpu_v0` to account for air viscosity when computing ball velocity, using the variable `opt_air_viscosity` for a viscosity amount. If `opt_air_viscosity` is zero there is no air resistance. The value of `opt_air_viscosity` can be changed from the user interface. See comments in `hw1.cc` under the heading Keyboard Commands and subheading Variables.

Assume that due to air viscosity a force is applied to the ball that is proportional to the ball's velocity. An approximate solution would multiply the velocity by some simple factor determined by the viscosity. At least some students should work through the calculus to determine the correct change in velocity based on the duration of the time step.

**Problem 3:** Modify the code in `hw1.cc` so that it shows an arrangement of chairs. The code must be organized in the same way as the code showing the prism tower.

Add a routine named `new_chair` that constructs and returns a single chair, similar to `new_prism`. This routine should arrange cards to form a chair using transforms. That is, **do not** position the cards by setting members `upper_left`, etc. Using transforms is doing it the hard way and in this particular case would be less efficient than setting individual coordinates. However, the point of the assignment is to learn to use transforms.

Add a second routine that calls `new_chair` and returns an arrangement of chairs, as does `new_tower` for prisms. The routine might be called `new_theater`, `new_stadium`, `new_box`, `new_deck`, etc. A third routine might be added, for example, if you've already written routines `new_chair` and `new_deck` you might add a third routine `new_stadium` that calls `new_deck`.

# 17 Fall 2026 Solutions

```
/// LSU EE 4702-1 (Fall 2026), GPU Programming
//
//
 /// Homework 1 -- SOLUTION
 //
 //  Search for "SOLUTION".
 //
 //  Assignment: https://www.ece.lsu.edu/gpup/2026/hw01.pdf
 //

#include <gp/coord.h>
#include <numbers>
#include "frame_buffer.h"

// Definitions for iVect, iCoor
#include "hw01-misc.h"

struct HW01_Data {
  HW01_Data():inited(false){};
  bool inited;
  bool tryout0, tryout1;
  int zoom;
  double t_frame_last, t_sim;
  bool paused;

};

// Global objects like this are bad style.
HW01_Data hw01_data;


void
line_draw
( pFrame_Buffer& fb, iCoor p0, iCoor p1, uint32_t co)
{
  int x0 = p0.x, y0 = p0.y;
  int x1 = p1.x, y1 = p1.y;

  const iVect delta = p1 - p0;
  const int width = fb.width_get();

  if ( abs( delta.x ) < abs( delta.y ) )
    {
      // Iterate over y
      if ( y0 > y1 ) { swap(x0,x1); swap(y0,y1); }

      float dxdy = float(delta.x) / delta.y;
      float x = x0;

      for ( int yi = y0;  yi < y1;  yi++ )
        {
          fb[ yi * width + x ] = co;
          x += dxdy;
```

```
        }
    }
  else
    {
      // Iterate over x
      if ( x0 > x1 ) { swap(x0,x1); swap(y0,y1); }

      float dydx = float(delta.y) / delta.x;
      float y = y0;

      for ( int xi = x0;  xi < x1;  xi++ )
        {
          fb[ int(y) * width + xi ] = co;
          y += dydx;
        }
    }
}

template< typename F > void
func_plot
( pFrame_Buffer& fb,
  iCoor plot_lower_left, int plot_width, int plot_height,
  float func_x0, float func_x1, F func )
{
  const int win_width [[maybe_unused]] = fb.width_get();
  const int win_height [[maybe_unused]] = fb.height_get();

  assert( plot_width > 0 && plot_height > 0 );

  //                            RRGGBB
  const uint32_t color_red [[maybe_unused]] = 0xff0000;
  const uint32_t color_deep_sky_blue(0x00bfff);

  const uint32_t color_plot [[maybe_unused]] = 0xffff00; // Gold.
  const uint32_t color_box = color_deep_sky_blue;
  const uint32_t color_background [[maybe_unused]] = 0x333333;

  iCoor p0 = plot_lower_left; // Just an abbreviation.
  iVect vx( plot_width, 0 ), vy( 0, plot_height );
  iCoor p1 = p0 + vx + vy;

  line_draw( fb, p0, p0 + vy, color_box );
  line_draw( fb, p0, p0 + vx, color_box );
  line_draw( fb, p0 + vy, p1, color_box );
  line_draw( fb, p0 + vx, p1, color_box );

  float func_y0 [[maybe_unused]] = func(func_x0);
  float func_y1 [[maybe_unused]] = func(func_x1);

  /// Homework 1: Put solutions to all problems in this routine.

  /// SOLUTION -- Part (a)
  //
  //  Write every pixel in plot area with color_background.
```

```
//
for ( int i=p0.x+1; i<p1.x; i++ )
  for ( int j=p0.y+1; j<p1.y; j++ )
    fb[ i + j * win_width ] = color_background;


/// SOLUTION -- Part (b)
//
//  Plot the function.
//
// Compute a delta_x (or dx/di). Used to find the change in x for
// a change of 1 in y.
//
float dfx = ( func_x1 - func_x0 ) / plot_width;
//
// Find the minimum and maximum value of y ( func(x) ) over the plot range.
//
float min_y = func(func_x0);
float max_y = min_y;
for ( int i=0; i<plot_width; i++ )
  {
    float x = func_x0 + dfx * i;
    float y = func(x);
    if ( y < min_y ) min_y = y;
    if ( y > max_y ) max_y = y;
  }
//
// Compute the range of y (func(x)) values plus a 5% margin at the
// top and bottom ..
//
float delta_y_tight = max_y - min_y;
float margin = delta_y_tight * 0.05;
float delta_y = delta_y_tight + 2 * margin;
//
// .. and use these to adjust the maximum and minimum y ..
//
max_y += margin;
min_y -= margin;
//
// .. and finally compute a scale factor for transforming y into
// a pixel vertical coordinate.
//
float scale = plot_height / delta_y;
//
// Plot the function.
//
for ( int i=0; i<plot_width; i++ )
  {
    // Coordinates to plot in the function's space.
    //
    float x = func_x0 + dfx * i;
    //
    // Function value at this i value.
    //
```

```
      float y = func(x);
      //
      // Function value at next i value.
      //
      float y_next = func(x+dfx);

      // Compute coordinates to pixel space.
      //
      int ix = p0.x + i;
      int iy0 = p0.y + ( y - min_y ) * scale;
      int iy1 = p0.y + ( y_next - min_y ) * scale;
      //
      int iylo = min(iy0,iy1);
      int iyhi = max(iy0,iy1);
      //
      // Draw a line from low to high y value.
      //
      for ( int iy = iylo; iy <= iyhi; iy++ )
        fb[ ix + iy * win_width ] = color_plot;
      //
      // Note: Though this results in a continuous plot, it might be
      // misleading because the minimum and maximum values of func(x)
      // for x in [x,x+dfx) might be lower or higher than iylo and
      // iyhi.
    }
}



void kbd_input_handle(pFrame_Buffer& fb, HW01_Data& hw01_data);


void
render_hw01(pFrame_Buffer& fb, HW01_Data& hw01_data)
{
  // Window Size, in Pixels
  //
  const int win_width = fb.width_get();
  const int win_height = fb.height_get();

  // Act on keys that have been pressed, perhaps by adjusting variables.
  kbd_input_handle(fb,hw01_data);

  /// Coordinates Used for Placement of Other Objects
  //
  const int sq_slen = min(win_width,win_height) * .45;
  const int sq_x0 = sq_slen * 0.1,   sq_y1 = win_height - 75;
  const int sq_x1 = sq_x0 + sq_slen, sq_y0 = sq_y1 - sq_slen;
  //
  // One vertex of the square is at (sq_x0,sq_y0) and the opposite
  // vertex is at (sq_x1,sq_y1).

  //                                              RRGGBB
  const uint32_t color_red [[maybe_unused]] = 0xff0000;
```

```
const uint32_t color_blue [[maybe_unused]] =    0xff;
const uint32_t color_green [[maybe_unused]] = 0xff00;

/// Demo Code: Draw Sine Waves -- No need to modify this.
//
int num_waves = 4;
float sine_ht = sq_slen/float(num_waves);
for ( float
        yb = sq_y1 - sine_ht/2,  freq = 4 * M_PI / ( win_width - 40 - sq_x1 );
      yb > sq_y0;
      yb -= sine_ht,  freq *= 2 )
  for ( int x = sq_x1 + 20;  x < win_width - 20;  x++ )
    {
      int y = yb + sin( x * freq ) * sine_ht * .45;
      fb[ y * win_width + x ] = 0xff00;
    }

/// Demo Code: Draw Lines Radiating From a Point -- No need to modify.
//
//  This code calls the line_draw routine.
//
int n_pts = 12 * 4;
float c_r1 = sq_slen/6;  // Inner radius
float c_r2 = sq_slen/2;  // Outer radius.
pCoor c_ctr( c_r2 + 20, c_r2 + 20 ); // Center of "circle"
float delta_theta = 2 * numbers::pi / n_pts;
for ( int i=0; i<n_pts; i++ )
  {
    pVect v( cosf(i*delta_theta), sinf(i*delta_theta), 0 );
    pCoor p1 = c_ctr + c_r1 * v;
    pCoor p2 = c_ctr + c_r2 * v;
    line_draw(fb, p1, p2, 0xffffff );
  }

fb.fb_stats_off(); // Turn off "redundancy" statistics.


/// Draw misc waves.

const bool simple_demo = false;
if ( simple_demo )
  {
    iCoor plot_ll( 20, 20 );
    float p_wd = 400, p_ht = 200;
    float fx0 = -2 * numbers::pi, fx1 = numbers::pi * 7;
    func_plot( fb, plot_ll, p_wd, p_ht,  fx0, fx1, sinf );
  }

float alphap = 0.5, alphan = 1;
// https://arxiv.org/abs/2411.13010
auto xIELU = [&](float x)
{ return x > 0 ? alphap * x * x + 0.5 * x
    : alphan * ( exp(x) - 1 ) - alphan * x + 0.5*x; };
float s0 = 0.7;
```

```
iCoor p00(c_ctr.x - sq_slen*s0*0.5, c_ctr.y-sq_slen*s0*0.5);
func_plot( fb, p00, sq_slen*s0, sq_slen*s0, -3, 3, xIELU);

float f_carrier = 17;
float f_modmod = 1;
float t_sim = hw01_data.t_sim;
float f_mod = 2 + sinf(t_sim*f_modmod);

auto fm = [&](float t) { return sinf( 4*sinf(t*f_mod) + t*f_carrier ); };
auto am = [&](float t) { return (1.2 + sinf(t*f_mod) ) * sinf(t*f_carrier); };

float plot_width = sq_slen * 0.7;
int plot_gap = sq_slen / 8;
int plot_stride = plot_width + plot_gap;
int plot1_ht = sq_slen * .4;
int plot1_y1 = sq_y1 - ( sq_slen - plot1_ht ) / 2;
int plot1_y0 = plot1_y1 - plot1_ht;
float fx0 = hw01_data.t_sim * 0.1 - 10;
float fxwid = numbers::pi;
int plot2_ht = sq_slen * 0.75;
int plot2_y0 = c_ctr.y - plot2_ht / 2;

for ( int x0 = sq_x1 + plot_gap;  x0 + plot_stride < win_width;
      x0 += plot_stride )
  {
    iCoor p10(x0,plot1_y0);
    iCoor p20(x0,plot2_y0);
    func_plot( fb, p10, plot_width, plot1_ht, fx0, fx0+fxwid, fm );
    func_plot( fb, p20, plot_width, plot2_ht, fx0, fx0+fxwid, am );
    fxwid *= 2;
  }
//
// See 2024 Homework 1 for a description of iCoor.

fb.fb_stats_off(); // Turn off "redundancy" statistics.

const int zoom = hw01_data.zoom;

/// Copy/Zoom Frame Buffer
///

// Compute coordinates of the center of the big box.
//
const int x_mid = ( sq_x0 + sq_x1 ) / 2;
const int y_mid = ( sq_y0 + sq_y1 ) / 2;

// Compute 1/2 the length of the mouse box.
//
const int src_hwid = sq_slen / ( zoom * 2 );

// Coordinates of mouse pointer and circle center.
//
const int mouse_x = fb.mouse_x;
```

```
  const int mouse_y = fb.mouse_y;

  // Iterate using relative coordinates.
  //
  // Note: x=0,y=0  is at the center of the big box and mouse box.
  //
  for ( int y = -src_hwid;  y < src_hwid;  y++ )
    for ( int x = -src_hwid;  x < src_hwid;  x++ )
      {
        // Read the pixel from the mouse area.
        //
        uint32_t pixel = fb[ mouse_x + x + ( mouse_y + y ) * win_width ];

        // Compute the index of the lower-left part of the big box that
        // the pixel above will be written to.
        //
        int fb_idx = x_mid + x * zoom + ( y_mid + y * zoom ) * win_width;

        // Write the pixel zoom * zoom times.
        //
        for ( int yy = 0;  yy < zoom;  yy++ )
          for ( int xx = 0;  xx < zoom;  xx++ )
            fb[ fb_idx + xx + yy * win_width ] = pixel;
      }

  /// Box Around Mouse Pointer and Zoom Area
  //
  //  Use our line drawing routines to draw the box.
  const iCoor mouse_p(fb.mouse_x,fb.mouse_y);
  const int src_wid = src_hwid * 2;
  iVect src_dx(src_wid,0), src_dy(0,src_wid);
  iCoor src_ll = mouse_p - iVect(src_hwid,src_hwid);  // Lower Left
  iCoor src_ur = mouse_p + iVect(src_hwid,src_hwid);  // Upper Right

  uint32_t color_src_box = 0x8888ff; // Light Blue
  uint32_t color_dst_box = 0x8888ff; // Light Blue

  line_draw( fb, src_ll, src_ll + src_dx, color_src_box );
  line_draw( fb, src_ll, src_ll + src_dy, color_src_box );
  line_draw( fb, src_ur, src_ur - src_dx, color_src_box );
  line_draw( fb, src_ur, src_ur - src_dy, color_src_box );

  iCoor dst_ll( sq_x0, sq_y0 ), dst_ur( sq_x1, sq_y1 );
  iVect dst_dx( sq_slen, 0 ), dst_dy( 0, sq_slen );

  line_draw( fb, dst_ll, dst_ll + dst_dx, color_dst_box );
  line_draw( fb, dst_ll, dst_ll + dst_dy, color_dst_box );
  line_draw( fb, dst_ur, dst_ur - dst_dx, color_dst_box );
  line_draw( fb, dst_ur, dst_ur - dst_dy, color_dst_box );
}

void
kbd_input_handle(pFrame_Buffer& fb, HW01_Data& hw01_data)
{
```

```
  if ( !hw01_data.inited )
    {
      hw01_data.inited = true;
      hw01_data.tryout0 = false;
      hw01_data.tryout1 = false;
      hw01_data.zoom = 4;
      hw01_data.t_frame_last = time_wall_fp();
      hw01_data.t_sim = 0;
      hw01_data.paused = false;
    }

  switch ( fb.keyboard_key ) {
#if 0
  case FB_KEY_RIGHT: case FB_KEY_LEFT:
    {
      const int inc = fb.keyboard_key == FB_KEY_LEFT ? -kbd_inc : kbd_inc;
      break;
    }
  case FB_KEY_UP: case FB_KEY_DOWN:
    {
      const int inc = fb.keyboard_key == FB_KEY_DOWN ? -kbd_inc : kbd_inc;
      break;
    }
#endif

    case 'p': case 'P': hw01_data.paused = !hw01_data.paused; break;

  case 'y':
    hw01_data.tryout0 = !hw01_data.tryout0; break;
  case 'Y':
    hw01_data.tryout1 = !hw01_data.tryout1; break;

  case 'u': case 'U': if ( hw01_data.zoom > 1 ) hw01_data.zoom--; break;
  case 'z': case 'Z': hw01_data.zoom++; break;

  default:
    break;
  };

  const double t_now = time_wall_fp();
  if ( ! hw01_data.paused )
    hw01_data.t_sim += t_now - hw01_data.t_frame_last;

  hw01_data.t_frame_last = t_now;


  fb.fbprintf("Tryout 0,1:  %s,%s ('y','Y')  %s ('p')  "
              "Zoom: %d ('u','z') \n",
              hw01_data.tryout0 ? "TRUE " : "FALSE",
              hw01_data.tryout1 ? "TRUE " : "FALSE",
              hw01_data.paused ? "PAUSED " : "RUNNING",
              hw01_data.zoom);
}
```

```
int
main(int argc, char **argv)
{
  pFrame_Buffer demo_frame_buffer(argc,argv);

  demo_frame_buffer.show
    ( [&](pFrame_Buffer& fb){ render_hw01(fb,hw01_data); } );
  return 0;
}
```

```
/// # LSU EE 4702-X (Fall 2026), GPU Programming
//
/// ## Homework 2 -- SOLUTION
//
//  Assignment: https://www.ece.lsu.edu/gpup/2026/hw02.pdf
//
/// ## Modified this file only.
//
/// ## Solution and other Work In:  World::hw02_prob1 and World::hw02_prob2
//
// Based on the Links Project Code in gp/proj-base/v-links
//

/// # Purpose
//
//   Demonstrate simulation of point masses connected by springs.

/// # What Code Does

// Simulates balls connected by springs over a platform. Balls and
// springs can be initialized in different arrangements (called
// scenes). Currently scene 1 is a tree-like arrangement. The platform
// consists of tiles, some are purple-tinted mirrors (showing a
// reflection of the ball), the others show the course syllabus.


/// # Keyboard Commands
//
/// ## Object (Eye, Light, Ball) Location or Push
//   Arrows (←,→,↑,↓) Page Up, Page Down
//        Move object or push ball, depending on mode.
//        Shift + KEY: motion is 5x faster.
//        Ctrl + KEY : motion is 5x slower.
//   'e': Move eye.
//   'l': Move light.
//   'b': Move head (first) ball. (Change position but not velocity.)
//   'B': Push head ball. (Add velocity.)
//
/// ## Eye Direction and GUI Options
//
//   'Home', 'End', 'Delete', 'Insert'
//         Turn the eye direction. Home should rotate eye direction
//         up, End should rotate eye down, Delete should rotate eye
//         left, Insert should rotate eye right. The eye direction
//         vector is displayed in the upper left.
//  'C-='  (Ctrl =) Increase green text size.
//  'C--'  (Ctrl -) Decrease green text size.
//  'F12'  Write screenshot to PNG file.

/// ## Simulation Execution Options
//  (Also see variables below.)
//
//  'p'    Pause simulation. (Press again to resume.)
//  ' '    (Space bar.) Advance simulation by 1/30 second.
//  'S- '  (Shift Space bar.) Advance simulation by one time step.

/// ## Performance Tuning, Graphics Effects, and Debugging Options
//
//  'y'    Toggle value of opt_tryout1. Intended for experiments and debugging.
//  'Y'    Toggle value of opt_tryout2. Intended for experiments and debugging.
//  'x'    Toggle value of opt_tryout3. Intended for experiments and debugging.
//  'n'    Toggle use of surface normals for spheres.
//  'o'    Toggle shadow effect.
//  'r'    Toggle mirror effect.
```

```
//  '!'    Toggle rendering of spheres (balls).
//  '@'    Toggle rendering of links.
//  '#'    Toggle rendering of platform.


/// Scene and Simulated System Options
//  (Also see variables below.)
//
//  'w'    Twirl balls around axis formed by head and tail. (Prob 2 soln).
//  '1'    Set up scene 1.
//  '2'    Set up scene 2.
//  '3'    Set up scene 3.
//  '4'    Set up scene 4.
//  'h'    Freeze position of first (head) ball. (Press again to release.)
//  't'    Freeze position of last (tail) ball. (Press again to release.)
//  's'    Stop balls.
//  'g'    Turn gravity on and off.

/// Variables
//   Selected program variables can be modified using the keyboard.
//   Use "Tab" to cycle through the variable to be modified, the
//   name of the variable is displayed next to "VAR" on the bottom
//   line of green text.

//  'Tab' Cycle to next variable.
//  '`'   Cycle to previous variable.
//  '+'   Increase variable value.
//  '-'   Decrease variable value.
//
//  VAR Spring Constant - Set spring constant.
//  VAR Time Step Duration - Set physics time step.
//  VAR Air Resistance - Set air resistance.
//  VAR Light Intensity - The light intensity.
//  VAR Gravity - Gravitational acceleration. (Turn on/off using 'g'.)

#define MAIN_INCLUDE
#undef VBUFFER_STRICT
//  #define VBUFFER_STRICT_WATCH 0xd10d270000000018
#include <vhelper.h>

#include <vstroke.h>
#include <gp/coord.h>
#include <gp/pstring.h>
#include <gp/misc.h>

#include <vutil-texture.h>
#include <vutil-pipeline.h>
#include "shapes.h"

#include <cuda_runtime.h>

#include <gp/colors.h>

#include <gp/util-containers.h>
#include <gp/coord-containers.h>
#include <gp/cuda-gpuinfo.h>
#include "links.cuh"

#include "links-shdr-common.h"

#include <numbers>


///
/// Main Data Structures
///
//
```

```
// class World: All data about scene.


class World;

enum GPU_Physics_Method { GP_cpu, GP_cuda, GP_ENUM_SIZE };
const char* const gpu_physics_method_str[] = { "CPU", "CUDA" };
const char *sh_names[] = { "PLAIN", "SET 1", "SET 2" };


// Object Holding Ball State
//
class Ball {
public:
  Ball() { init(); }
  Ball(pCoor pos){ init(); position = pos; constants_update(); }
  Ball(pCoor pos, const Ball &bd)
  { *this = bd; position = pos; constants_update(); }

  void init()
    {
      velocity = pVect(0,0,0);
      omega = pVect(0,0,0);
      radius = 1;
      mass = 0;
      mass_inv = 1;
      density = 1.00746;
      fdt_to_do = 0;
      locked = false;
      color = color_lsu_spirit_gold;
      specularity = 0;
      contact = false;
      orientation_set(pVect(0,1,0),0);
      world_time_expire = 3600.0 * 24 * 365 * 1000;
    }

  pCoor position;
  pVect velocity;

  // Orientation of ball. Both orientation and omatrix updated by time
  // step code.
  pQuat orientation;
  pMatrix3x3p omatrix;
  pVect omega;                    // Spin rate and axis.
  double world_time_expire;

  int idx; // Position in balls_pos_rad;

  union {
    float4 m_r_fdt;
    struct {
      float mass;
      float radius;
      float fdt_to_do; // Radius / moment of inertia.
      float _;
    }; };

  float mass_min; // Mass below which simulation is unstable.
  float radius_sq, radius_inv;  // Pre-computed based on radius.
  float density;
  float mass_inv;

  bool locked;

  pVect force;
  pVect torque;
  pColor color;
```

```
  float specularity;            // Amount of shininess, in [0,1].
  bool contact;                 // When true, ball rendered in gray.
  float spring_constant_sum;    // Used to compute minimum mass.

  void orientation_set(pNorm dir, float angle)
    { orientation_set(pQuat(dir,angle)); }
  void orientation_set(pQuat ori)
    {
      orientation = ori;
      omatrix = pMatrix3x3_Rotation(orientation);
    }
  void constants_update();
  pVect point_rot_vel(pNorm tact_dir);
  pVect point_rot_vel(pVect tact_dir);
  void apply_tan_force(pNorm tact_dir, pNorm force_dir, double force_mag);
  void apply_tan_force(pNorm tact_dir, pVect force);
  void push(pVect amt);
  void translate(pVect amt);
  void stop();
  void freeze();
};


void
Ball::constants_update()
{
  assert( radius );
  radius_sq = radius * radius;
  if ( mass >= 0 )
    {
      mass = 4.0 / 3 * numbers::pi * radius_sq * radius * density;
      mass_inv = 1 / mass;
    }
  radius_inv = 1 / radius;
  // FYI, notice simplifications:
  //   const float mo_inertia = 0.4 * mass * radius_sq;
  //   fdt_to_do = radius / mo_inertia;
  fdt_to_do = radius_inv * 2.5 * mass_inv;
}

pVect
Ball::point_rot_vel(pNorm direction)
{
  /// Return velocity of point on surface of ball.
  //
  return radius * cross( omega, direction );
}

pVect
Ball::point_rot_vel(pVect rel_pos)
{
  /// Return velocity of point relative to center.
  //
  return cross( omega, rel_pos );
}

void
Ball::apply_tan_force(pNorm tact_dir, pVect force)
{
  torque += cross(tact_dir, force);
}

void
Ball::apply_tan_force(pNorm tact_dir, pNorm force_dir, double force_mag)
{
  apply_tan_force(tact_dir, force_mag * force_dir);
}
```

```
struct Link_Args {
  pColor natural_color = color_red;
  float distance_relaxed = 0;
  float spring_constant = 0;
  float link_stiffness = 0;
  float radius = 0;
};

class Link {
public:
  Link(Ball *b1, Ball *b2, Link_Args args = {}):ball1(b1),ball2(b2),
     snapped(false)
  { init(args); }
  Link(Link *link, pVect cb1p, pVect cb2p, float drp):
    ball1(link->ball1), ball2(link->ball2), cb1(cb1p), cb2(cb2p),
    is_surface_connection(true), is_renderable(false),
    is_simulatable(true), distance_relaxed(drp), spring_constant(0),
    radius(0),snapped(false){link->is_simulatable = false;}

  Link(Link *link, pVect cb1p, pVect cb2p, Link_Args args = {} ):
    ball1(link->ball1), ball2(link->ball2), cb1(cb1p), cb2(cb2p),
    is_surface_connection(true), is_renderable(false),
    is_simulatable(true),
    distance_relaxed(args.distance_relaxed),
    spring_constant(args.spring_constant),
    radius(0),snapped(false){link->is_simulatable = false;}

  void init(Link_Args& args)
  {
    assert( args.distance_relaxed == 0 );
    natural_color = args.natural_color;
    spring_constant = args.spring_constant;
    radius = args.radius;
    init();
  }
  void init()
    {
      assert( ball1->radius > 0 );
      assert( ball2->radius > 0 );
      is_simulatable = true;
      is_renderable = true;
      pNorm n12(ball1->position,ball2->position);
      const float rad_sum = ball1->radius + ball2->radius;
      is_surface_connection = n12.magnitude >= rad_sum;

      pMatrix3x3 b1rot = ball1->omatrix;
      pMatrix3x3 b2rot = ball2->omatrix;

      pVect b1_y = b1rot * pVect(0,1,0);
      pVect b1_x = b1rot * pVect(1,0,0);
      bool b1_dir_is_y = fabs(dot(b1_y,n12)) < 0.999;
      pVect b1_dir_loc = b1_dir_is_y ? b1_y : b1_x;

      con_x = pNorm(cross(b1_dir_loc,n12));
      con_y = cross(n12,con_x);
      b1_dir = mult_MTV( b1rot, con_y );
      b2_dir = mult_MTV( b2rot, con_y );

      if ( !is_surface_connection )
        {
          distance_relaxed = n12.magnitude;
          cb1 = cb2 = pVect(0,0,0);
          is_renderable = false;
          return;
        }
```

```
      distance_relaxed = n12.magnitude - rad_sum;
      cb1 = ball1->radius * mult_MTV( b1rot,  n12 );
      cb2 = ball2->radius * mult_MTV( b2rot, -n12 );
    }
  int serial;
  int idx;  // Position in lis_pos1, etc.
  Ball* const ball1;
  Ball* const ball2;
  union {
    int2 ball_idx;
    struct { int ball1_idx, ball2_idx; };
  };
  pVect cb1, cb2, con_x, con_y, b1_dir, b2_dir;
  bool is_surface_connection;
  bool is_renderable, is_simulatable;
  union {
    float2 dr_sc;
    struct {
      float distance_relaxed;
      float spring_constant;
    };
  };
  float radius;
  bool snapped;
  pColor natural_color;
  pColor color;

  pVect spring_force_12;
  pVect torque1,torque2;

};

class Surface {
public:
  Ball *b0, *b1, *b2;
  pColor color = color_blue;
  pTCoor tc0, tc1, tc2;
  union {
    int4 b_idx;
    struct { int b0_idx, b1_idx, b2_idx, _; };
  };
  Surface* operator ->() { return this; }
};

// Declare containers for Balls and Links. The containers are derived
// from std::vector and use overloaded operator += for push_back. (See
// include/util-containers.h.)
//
typedef pVectorI<Link> Links;
typedef pVectorI<Ball> Balls;


enum Shader_Option { SO_Plain, SO_Instances, SO_True, SO_ENUM_SIZE };

class World {
public:
  World(pVulkan_Helper &fb, int argc, char **argv)
    :vh(fb), ff_state(fb.qs), raytrace(fb.qs),
     frame_timer(vh.frame_timer), transform(fb.qs),
     shapes(ff_state), sphere(ff_state),
     opt_tryout1(fb.qs.opt_tryout1),opt_tryout2(fb.qs.opt_tryout2),
     opt_tryout3(fb.qs.opt_tryout3)
  {init(argc,argv);}
  void init(int argc, char **argv);
  void init_graphics();
  void run();
  void frame_callback();
```

```
void render(vk::CommandBuffer& cb);
void render_objects(vk::CommandBuffer& cb, Render_Option render_option);
void objects_erase();
void cb_keyboard();
void modelview_update();

pVulkan_Helper& vh;
VFixed_Function_State_Manager ff_state;
VRaytrace raytrace;
pFrame_Timer& frame_timer;
VTransform transform;
Shapes shapes;
Sphere sphere;

bool opt_want_raytrace;

pVariable_Control variable_control;
double world_time;
double last_frame_wall_time;
double energy;
float opt_time_step_duration;
int time_step_count;
float opt_gravity_accel;      // Value chosen by user.
pVect gravity_accel;          // Set to zero when opt_gravity is false;
bool opt_gravity;
bool opt_head_lock, opt_tail_lock;
bool &opt_tryout1, &opt_tryout2, &opt_tryout3;  // For ad-hoc experiments.
float opt_tryoutf;
bool opt_sphere_true;
VBufferV<Shdr_Uni_Common> buf_uni_common;

// Tiled platform for ball.
//
float platform_xmin, platform_xmax, platform_zmin, platform_zmax;
float platform_pi_xwidth_inv;
VVertex_Buffer_Set bset_platform_tex, bset_platform_mir;
VPipeline pipe_platform_mir, pipe_platform_tex;
RT_Geo_Package rt_geo_platform_tex, rt_geo_platform_mir;

vk::Sampler sampler;
VTexture texid_hw;
VTexture texid_syl;
VTexture texid_emacs;

void platform_update();
bool platform_collision_possible(pCoor pos);

pCoor light_location;
float opt_light_intensity;
enum { MI_Eye, MI_Light, MI_Ball, MI_Ball_V, MI_COUNT } opt_move_item;
bool opt_pause;
bool opt_single_frame;      // Simulate for one frame.
int opt_single_time_step;  // Simulate for one time step.

VBufferV<Uni_Lighting> uni_light, uni_light_ambient, uni_light_all;
VBufferV<Uni_Lighting> uni_light_reflected;
VBufferV<Uni_Lighting> *puni_light_curr;

pCoor eye_location;
pVect eye_direction;
pMatrix transform_projection;
pMatrix modelview;

pVect adj_vector;
double adj_t_prev;
double adj_t_stop;
double adj_duration;
```

```
uint opt_shader;

float opt_background_shade; // 0 black, 1 white.

bool opt_mirror;
int opt_mirror_depth;
bool opt_shadows;
bool opt_shadow_volumes;      // Make shadow volumes visible.
bool opt_normal_sphere;
int opt_hide_stuff;

int last_setup; // Last scene set up.

void ball_setup_1(bool big);  // Silly Tree
void ball_setup_2(); // Hanging thing.
void ball_setup_3(); // Corona virus particle.
void ball_setup_4(); // Wheel.
void ball_setup_5(); // Top.
void ball_setup_6(); // Wheel with webbing between spokes.
void ball_setup_7(bool rand);
void setup_at_end();
void time_step_cpu(double);
void energy_compute();
void balls_stop();
void balls_freeze();
void balls_translate(pVect amt, int idx);
void balls_translate(pVect amt);
void balls_push(pVect amt, int idx);
void balls_push(pVect amt);
void balls_twirl();
void lock_update();

Ball *make_marker(pCoor pos, pColor col, float radius = 0.2 );
Balls balls_marker; // Written to balls right after setup.

float opt_spring_constant, opt_spring_damp;
float opt_friction_dynamic;
float opt_air_resistance;
float distance_relaxed;
int chain_length;
Balls balls;
Ball *head_ball, *tail_ball;
Links links;
pColor color_link;
vector<Surface> surfaces;
int opt_sphere_slices;
float opt_omega;  // For disk and top.

Links link_new
( Ball *ball1, Ball *ball2,
  float stiffness = 0.2, float spring_constant = 0);
Links link_new
( Ball *ball1, Ball *ball2, Link_Args largs );

/// CUDA Stuff

int opt_physics_method;
GPU_Info gpu_info;
cudaEvent_t frame_start_ce, frame_stop_ce;
int opt_block_size;
bool gpu_const_data_stale;
bool cpu_phys_data_stale;

void data_cpu_to_gpu_constants();
void data_cpu_to_gpu_dynamic();
```

```
  void data_gpu_to_cpu_dynamic();
  CPU_GPU_Common c;  // c is for common.  See links.cuh.

  void render_link_start();
  void render_link_gather(Link *link);
  void render_link_render(vk::CommandBuffer& cb, bool shadow_volumes = false);
  void render_link_render_sv(vk::CommandBuffer& cb)
  { render_link_render(cb,true); };

  VPipeline pipe_set_2, pipe_set_2_sv;
  double world_time_link_update;

  VCompute comp_links;
  VBuffer<int> buf_links_as_idx; // Indices for links' acceleration structures.
  VBuffer<vec2> buf_links_tex_coor;
  VBuffer<vec4> buf_links_vertex_g;
  VBuffer<vec4> buf_links_normal_g;
  RT_Geo_Package rt_geo_links;

  int opt_sides, opt_segments;

  vector< VBufferVV<vec4>* > lis_bufs;
  VBufferVV<vec4> lis_pos1, lis_pos2;
  VBufferVV<vec4> lis_v1, lis_v2;
  VBufferVV<vec4> lis_b1_ydir, lis_b2_ydir, lis_tex_rad;

  VVertex_Buffer_Set bset_surfaces;
  VPipeline pipe_surfaces;
  RT_Geo_Package rt_geo_surfaces;
  void render_surfaces(vk::CommandBuffer& cb);

  // 2026 Homework 2

  void spring_make(pCoor axis_top, pCoor axis_bottom, float r, int n_balls);
  void spring_make2(pMatrix m, int n_balls);
  void hw02_prob2
  (pCoor ring_top, pCoor ring_center, pVect helix_vx, int n_balls );
  void hw02_prob1(pCoor axis_top, pCoor axis_bottom, float r, int n_balls);

};

enum { OH_Links = 1, OH_Platform = 2, OH_Sphere = 4 };


 ///
 /// Material for 2026 Homework 2 is Below ↓↓↓↓
 ///

void
World::spring_make(pCoor axis_top, pCoor axis_bottom, float r, int n_balls)
{
  // Routine to construct leftmost spring in Scene 7.

  // 2026 Homework 2: Can inspect, but no need to modify, this routine.

  float delta_theta = numbers::pi / 8;
  float ball_rad = r * delta_theta / 5;

  pVect vaxis(axis_top,axis_bottom);
  pNorm az(vaxis);
  pVect vz = vaxis / n_balls;
  pNorm ax = az.orthogonal();
  pVect ay = cross(az,ax);
  pVect vx = r * ax;
  pVect vy = r * ay;

  for ( int i=0; i<n_balls; i++ )
```

```
    {
      float theta = i * delta_theta;
      pCoor point_on_axis = axis_top + i * vz;
      pVect vector_from_axis_to_helix = cosf(theta) * vx + sinf(theta) * vy;
      pCoor pos = point_on_axis + vector_from_axis_to_helix;
      Ball* const ball = new Ball;
      ball->position = pos;
      ball->locked = false;
      ball->velocity = pVect(0,0,0);
      ball->radius = ball_rad;
      ball->contact = false;
      ball->specularity = 0.5;
      if ( i ) links += link_new(balls.back(),ball,10);
      balls += ball;
    }
}

void
World::hw02_prob1(pCoor axis_top, pCoor axis_bottom, float r, int n_balls)
{
  pMatrix m;

  /// SOLUTION - Problem 1
  //  First, get rid of placeholder m.set_identity();

  /// 2026 Homework 2 Problem 1:
  //
  //  Put solution in this routine.
  //  Can inspect but don't modify spring_make2 (except changes
  //    needed to understand how spring_make2 works.)
  //
  //  [✔] Compute m so that spring appears as described in assignment.


  /// Examples of Transfromation Matrices
  //
#if 0
  // Scale by sx along x axis, etc.
  float sx=1, sy=1, sz=1;
  pMatrix_Scale scale(sx,sy,sz);
  //
  // Rotate so local x axis now points in direction ax, etc.
  pNorm global_ax, global_ay, global_az;
  pMatrix_Cols rot_to_global(global_ax, global_ay, global_az);
  // Rotate by 1.23 radians around global_ax.
  pMatrix_Rotation rot( global_ax, 1.23 );
  //
  // Translate so that local origin (0,0,0) is moved to global_origin.
  pCoor global_origin;
  pMatrix_Translate to_global( global_origin );
#endif

  /// SOLUTION -- Problem 1
  //

  /// Construct local axes.
  //
  //  The z-axis is the cylinder axis. Computing it is easy since
  //  two points on the cylinder axis are given.
  //
  pVect vaxis(axis_top,axis_bottom);
  //
  //  The problem wording describes the axis as going from axis_top
  //  to axis_bottom. That kind of makes the helix upside down if
  //  axis_top.y > axis_bottom.y.
  //
  pNorm az(vaxis);
  //
```

```
  // Compute an axis orthogonal to az.
  //
  pNorm ax = az.orthogonal();
  //
  // Study Tip: Be able to compute orthogonal axis without member function.
  //
  // That leaves the y axis, which can be computed from the other two:
  //
  pVect ay = cross(az,ax);
  //
  // Use the local axes to construct a rotation matrix.
  //
  pMatrix_Cols rotate(ax,ay,az);

  // Scale the x and y axes so helix radius is r, and scale the z axis
  // so the length of the cylinder axis is the distance from
  // axis_bottom to axis_top.
  //
  pMatrix_Scale scale_rz(r,r,az.magnitude);
  //
  // Note: An alternative to the scale matrix is scaling the local axes
  // ax, ay, az.

  // Construct a translation matrix to move the local origin to
  // axis_top.
  //
  pMatrix_Translate to_top( axis_top );
  //
  // Why not axis_bottom? Because the problem statement described the
  // axis as from axis_top to axis_bottom, so the helix starts at
  // axis_top and the origin must be moved to axis_top. If one instead
  // used axis_bottom in the pMatrix_Translate instantiation the helix
  // would be in the wrong place and that would be very obvious.

  // Finally, compute m by multiplying the three together.
  //
  m = to_top * rotate * scale_rz;

  spring_make2( m, n_balls );
}


void
World::spring_make2(pMatrix m, int n_balls)
{
  // Routine to construct center spring in Scene 7.

  // 2026 Homework 2:
  //
  // Feel free to inspect this routine.
  // Should *not* modify this routine as part of solution.
  // It's okay to make changes to understand how this works ..
  // .. but no changes should make the solution to Problem 1 easier.

  float delta_theta = numbers::pi / 8;

  pNorm scale = m * pVect(1,0,0);
  float r = scale.magnitude;
  float ball_rad = r * delta_theta / 5;
  float delta_z = 1.0 / n_balls;

  pMatrix_Rotation rot( pVect(0,0,1), delta_theta );
  pMatrix_Translate trans( pVect(0,0,delta_z) );
  pMatrix local_space_delta_pos = rot * trans;

  pMatrix minv = m.inverse();
  pMatrix global_space_delta_pos = m * local_space_delta_pos * minv;
```

```
  pCoor pos_i = m * pCoor(1,0,0);

  for ( int i=0; i<n_balls; i++ )
    {
      Ball* const ball = new Ball;
      ball->position = pos_i;

      // Compute position of next ball.
      //
      pos_i = global_space_delta_pos * pos_i;

      ball->locked = false;
      ball->velocity = pVect(0,0,0);
      ball->radius = ball_rad;
      ball->contact = false;
      ball->color = color_purple;
      ball->specularity = 0.5;
      if ( i ) links += link_new(balls.back(),ball,10);
      balls += ball;
    }
}

void
World::hw02_prob2
( pCoor ring_top, pCoor ring_center, pVect helix_vx, int n_balls )
{
  // Construct a helix wound around a ring.

  // Prepare axes describing the ring.
  //
  float ring_delta_theta = 2 * numbers::pi / n_balls;
  pVect ring_vy = pVect( ring_center, ring_top );
  pNorm ring_ay = ring_vy;
  pNorm ring_ax = cross( ring_ay, helix_vx );
  float ring_r = ring_ay.magnitude;

  // Get index of the next ball (first of ring), for use in linking first
  // and last balls.
  //
  size_t idx_first = balls.size();

  Link_Args link_args{
    .spring_constant = opt_spring_constant * 15,  .link_stiffness = 10 };

  // Problem 2: There should be no need to modify the code above.
  // But code can be modified if it helps.

  //
  /// Problem 2: Add code here and inside the loop.
  //

  // [ ] When drawing helix compute ball radius from helix radius and Δ angle.
  float ball_rad = ring_r * ring_delta_theta / 5;


  /// SOLUTION -- Problem 2
  //
  // Compute helix delta theta so that:
  //  - There are about 9 balls per rotation.
  //  - The number of rotations is an integer.
  //
  float balls_per_rot_target = 9;
  int n_rots = n_balls / balls_per_rot_target;
  float balls_per_rot = float(n_balls) / n_rots;
  float helix_delta_theta = 2 * numbers::pi / balls_per_rot;
  //
  // The solution uses two methods to draw the toroidal helix. In the
  // primary solution a helix_ax and helix_ay are computed. An
```

```
// alternative solution, selected when opt_tryout3 is true, does not
// need helix_ay.
//
// To draw helix using the primary solution we need local axes. Axis
// helix_ax is orthogonal to the ring and so that it is the same for
// all positions along the ring, and so it is computed before the i
// loop. Axis helix_ay is not the same along the ring. One possible
// helix_ay is the vector from the ring center to the point on the
// ring. That is what the code inside computes.
//
// Compute local x-axis and extract helix radius from given helix_vx.
//
pNorm helix_ax(helix_vx);
float helix_r = helix_ax.magnitude;
//
// Note: helix_ax and helix_r are done once outside the loop.
//
// Compute ball radius based on helix radius and delta theta.
//
ball_rad = helix_r * helix_delta_theta / 5;

for ( int i=0; i<n_balls; i++ )  // <--- The Loop
  {
    Ball* const ball = new Ball;
    float ring_theta = i * ring_delta_theta;

    /// SOLUTION -- Problem 2
    //
    // Removed the unused computation of pos (was above).
    //
    // Compute helix theta.
    //
    float helix_theta = i * helix_delta_theta;
    //
    // For the primary solution compute helix_ay, ring_pos, and pos:
    //
    pVect helix_ay = cosf(ring_theta) * ring_ax + sinf(ring_theta) * ring_ay;
    //
    // Remember that helix_ax does not change with i, but helix_ay does.
    //
    // Using helix_ay compute point on the ring (circle) ..
    //
    pCoor ring_pos = ring_center + ring_r * helix_ay;
    //
    // .. and use that to compute a point on the toroidal helix.
    //
    pCoor pos = ring_pos
      + cosf(helix_theta) * helix_vx
      + helix_r * sinf(helix_theta) * helix_ay;
    //
    // That's it for the primary solution. The alternative solution
    // appears below.

    if ( opt_tryout3 )
      {
        // Alternative solution in which point on toroidal helix
        // is found without using helix_ay.
        //
        // See Math stack exchange:
        // https://math.stackexchange.com/questions/324527/do-these-equations-create-a-helix-wrapped-into-a-torus

        float rr = ring_r + helix_r * cosf(helix_theta);
        pos = ring_center
          + rr * cosf(ring_theta) * ring_ax
          + rr * sinf(ring_theta) * ring_ay
          + helix_r * sinf(helix_theta) * helix_ax;
        //
        // This looks a little more compact than the primary solution
```

```
          // but does it use less computation?
        }

      // Code below does not need to be modified.
      //
      // But feel free to experiment. The key sequence "C-x v =" is your friend.
      //
      ball->position = pos;
      ball->locked = false;
      ball->velocity = pVect(0,0,0);
      ball->radius = ball_rad;
      ball->contact = false;
      ball->specularity = 0.5;
      if ( i ) links += link_new(balls.back(),ball,link_args);
      balls += ball;
    }

  // Code below does not need to be modified.

  // Connect first and last balls.
  //
  links += link_new(balls.back(),balls[idx_first],link_args);
}

 ///
 /// Material for 2026 Homework 2 is Above ↑↑↑↑
 ///


void
World::init_graphics()
{
  vh.want_raytrace().init();
  vh.opt_record_every_time = true;
  vh.display_cb_set([&](){frame_callback();});
  vh.cbs_cmd_record.push_back( [&](vk::CommandBuffer& cb){ render(cb); });
  sphere.ppuni_light = &puni_light_curr;
  sphere.transform = &transform;

  opt_want_raytrace = true;

  if ( vh.have_raytrace )
    {
      raytrace.init();
      for ( auto b: { &bset_platform_mir, &bset_platform_tex, &bset_surfaces } )
        b->usage_set
          ( vk::BufferUsageFlagBits::eVertexBuffer
            | vk::BufferUsageFlagBits::eStorageBuffer );
      buf_links_as_idx.init
        ( vh.qs, vk::BufferUsageFlagBits::eStorageBuffer, 0,
          vk::MemoryPropertyFlagBits::eDeviceLocal );
      buf_links_tex_coor.init
        ( vh.qs, vk::BufferUsageFlagBits::eStorageBuffer, 0,
          vk::MemoryPropertyFlagBits::eDeviceLocal );
      buf_links_vertex_g.init
        ( vh.qs, vk::BufferUsageFlagBits::eStorageBuffer, 0,
          vk::MemoryPropertyFlagBits::eDeviceLocal );
      buf_links_normal_g.init
        ( vh.qs, vk::BufferUsageFlagBits::eStorageBuffer, 0,
          vk::MemoryPropertyFlagBits::eDeviceLocal );
    }

  buf_uni_common.init(vh.qs,vk::BufferUsageFlagBits::eUniformBuffer);
  sphere.buf_uni_common = buf_uni_common;

  lis_bufs =
    { &lis_pos1, &lis_pos2,
      &lis_v1, &lis_v2,
```

```
    &lis_b1_ydir, &lis_b2_ydir,
    &lis_tex_rad };

for ( auto b: lis_bufs )
  b->init(vh.dev_phys, vh.dev, vk::BufferUsageFlagBits::eStorageBuffer );

///
/// Graphical Model Initialization
///

opt_head_lock = false;
opt_tail_lock = false;
opt_tryout1 = opt_tryout2 = opt_tryout3 = false;
opt_normal_sphere = true;
opt_sphere_true = true;
opt_tryoutf = 1;
variable_control.insert(opt_tryoutf,"Tryout F");

opt_background_shade = 0;
variable_control
  .insert_linear(opt_background_shade,"Background Lightness",.1)
  .set_max(1).set_min(0);

eye_location = pCoor(19.9,14.4,-40.7);
eye_direction = pVect(-0.42,-0.09,0.9);

platform_xmin = -40; platform_xmax = 40;
platform_zmin = -40; platform_zmax = 40;

texid_syl.init
  ( vh.qs, P_Image_Read( vh.gp_root_get() / "vulkan" / "gpup.png", 255 ) );
texid_emacs.init
  ( vh.qs, P_Image_Read
    ( vh.gp_root_get() / "proj-base" / "v-links" / "mult.png", -1) );
texid_hw.init
  ( vh.qs, P_Image_Read
    ( vh.gp_root_get() / "proj-base" / "v-links" / "hw05-assign.png", 255) );
sampler = vh.qs.dev.createSampler
  ( { {},
      vk::Filter::eLinear,  // magFilter
      vk::Filter::eLinear,  // minFilter
      vk::SamplerMipmapMode::eLinear,
      // Also: eRepeat, eMirroredRepeat, eClampToEdge, etc.
      vk::SamplerAddressMode::eRepeat,
      vk::SamplerAddressMode::eRepeat,
      vk::SamplerAddressMode::eRepeat,
      0.0f,  // mipLodBias
      true, // anisotropyEnable,
      16.0f, // maxAnisotropy
      false, vk::CompareOp::eNever,  // compareEnable, compareOp
      0.0f, VK_LOD_CLAMP_NONE,  // min and max LOD
      vk::BorderColor::eFloatOpaqueBlack,
      false // unnormalizedCoordinates
      } );

sphere.texture_set(sampler,texid_emacs);

opt_light_intensity = 140;
light_location = pCoor(platform_xmax,70,platform_zmin);
uni_light.init(vh.qs,vk::BufferUsageFlagBits::eUniformBuffer);
uni_light_all.init(vh.qs,vk::BufferUsageFlagBits::eUniformBuffer);
uni_light_reflected.init(vh.qs,vk::BufferUsageFlagBits::eUniformBuffer);
uni_light_ambient.init(vh.qs,vk::BufferUsageFlagBits::eUniformBuffer);

variable_control.insert(opt_light_intensity,"Light Intensity");

color_link = color_light_gray;
color_link = color_spring_green;
```

```
  opt_move_item = MI_Eye;
  opt_pause = false;
  opt_single_time_step = 0;
  opt_single_frame = false;

  platform_update();
  modelview_update();

  adj_t_stop = 0;
  adj_duration = 0.25;

  opt_sides = 20;
  opt_segments = 16;
  variable_control.insert(opt_sides,"Link Sides");
  variable_control.insert(opt_segments,"Link Segments",5,1);
  opt_hide_stuff = 0;

  if ( vh.have_raytrace )
    {
      raytrace.buf_light_set(uni_light_all);
    }
}

void
World::run()
{
  vh.message_loop_spin();

  uni_light.destroy();
  uni_light_ambient.destroy();
  uni_light_all.destroy();
  uni_light_reflected.destroy();
  transform.destroy();
  buf_uni_common.destroy();

  for ( auto b: lis_bufs ) b->destroy();

  pipe_platform_tex.destroy(); pipe_platform_mir.destroy();
  bset_platform_tex.destroy(); bset_platform_mir.destroy();
  pipe_set_2.destroy();
  pipe_set_2_sv.destroy();

  texid_syl.destroy();
  texid_emacs.destroy();
  texid_hw.destroy();
  vh.dev.destroySampler(sampler);
  shapes.destroy();
  sphere.destroy();

  bset_surfaces.destroy();
  pipe_surfaces.destroy();
  rt_geo_surfaces.destroy();

  buf_links_as_idx.destroy();
  buf_links_tex_coor.destroy();
  buf_links_normal_g.destroy();
  buf_links_vertex_g.destroy();
  rt_geo_platform_tex.destroy();
  rt_geo_platform_mir.destroy();
  rt_geo_links.destroy();
  comp_links.destroy();
  raytrace.destroy();

  vh.finish();
}

void
```

```
World::render_surfaces(vk::CommandBuffer& cb)
{
  if ( !pipe_surfaces )
    pipe_surfaces
      .init( vh.qs )
      .ds_follow( puni_light_curr )
      .ds_follow( transform )
      .ds_uniform_use( "BIND_UNI_COMMON", buf_uni_common )
      .topology_set( vk::PrimitiveTopology::eTriangleList )
      .create();

  bset_surfaces.reset( pipe_surfaces );

  for ( Surface& s: surfaces )
    {
      Ball &b0 = *s.b0, &b1 = *s.b1, &b2 = *s.b2;
      pNorm snorm = cross(b0.position,b1.position,b2.position);
      bset_surfaces << b0.position << b1.position << b2.position;
      bset_surfaces << s.color << s.color << s.color;
      bset_surfaces << snorm << snorm << snorm;
    }

  bset_surfaces.to_dev();

  if ( vh.have_raytrace && !rt_geo_surfaces )
    rt_geo_surfaces.init( raytrace );

  if ( vh.raytrace_active() )
    {
      rt_geo_surfaces.bind( bset_surfaces, RT_Geometry_Triangles );
      return;
    }

  pipe_surfaces.record_draw( cb, bset_surfaces );
}

void
World::platform_update()
{
  const float tile_count = 19;
  const float ep = 1.00001;
  const float delta_x = ( platform_xmax - platform_xmin ) / tile_count * ep;
  const float zdelta = ( platform_zmax - platform_zmin ) / tile_count * ep;

  const float trmin = 0.05;
  const float trmax = 0.7;
  const float tsmin = 0;
  const float tsmax = 0.4;
  pColor color_platform( color_white * 0.9 );

  if ( !pipe_platform_mir )
    pipe_platform_mir
      .init( ff_state )
      .ds_set( color_lsu_spirit_purple )
      .ds_follow( puni_light_curr )
      .ds_follow( transform )
      .topology_set( vk::PrimitiveTopology::eTriangleList )
      .create();
  bset_platform_mir.reset( pipe_platform_mir ).tolerate_tcoords();

  if ( !pipe_platform_tex )
    pipe_platform_tex
      .init( ff_state )
      .ds_set_material(color_platform)
      .ds_follow( puni_light_curr )
      .ds_follow( transform )
      .topology_set( vk::PrimitiveTopology::eTriangleList )
      .ds_use( sampler, texid_syl )
```

```
      .create();
  bset_platform_tex.reset( pipe_platform_tex );

  const pVect platform_normal(0,1,0);
  pTCoor t00(trmin,tsmin), t01(trmin,tsmax), t10(trmax,tsmin), t11(trmax,tsmax);
  bool even = true;

  for ( int i = 0; i < tile_count; i++ )
    {
      const float x0 = platform_xmin + i * delta_x;
      const float x1 = x0 + delta_x;
      const float y = 0;
      for ( float z = platform_zmin; z < platform_zmax; z += zdelta )
        {
          auto& bset = even ? bset_platform_tex : bset_platform_mir;
          const float z1 = z+zdelta;
          pCoor p00(x0,y,z), p01(x0,y,z1), p10(x1,y,z), p11(x1,y,z1);

          bset << t11 << platform_normal << p00;
          bset << t10 << platform_normal << p01;
          bset << t00 << platform_normal << p11;

          bset << t11 << platform_normal << p00;
          bset << t00 << platform_normal << p11;
          bset << t01 << platform_normal << p10;

          even = !even;
        }
    }

  bset_platform_mir.to_dev();
  bset_platform_tex.to_dev();

  if ( vh.have_raytrace && !rt_geo_platform_mir )
    {
      pColor mirror_color(color_lsu_spirit_purple);
      mirror_color.a = -0.5;
      rt_geo_platform_mir
        .init( raytrace )
        .color_front_set( mirror_color )
        .bind( bset_platform_mir, RT_Geometry_Triangles );
      rt_geo_platform_tex
        .init( raytrace )
        .color_front_set( color_platform )
        .bind( VR_BUF_IDX_sampler, sampler, texid_syl )
        .bind( bset_platform_tex, RT_Geometry_Triangles );
    }
}

void
World::modelview_update()
{
  pMatrix_Translate center_eye(-eye_location);
  pMatrix_Rotation rotate_eye(eye_direction,pVect(0,0,-1));
  modelview = rotate_eye * center_eye;
}

void
World::render_objects(vk::CommandBuffer& cb, Render_Option option)
{
  //  const float shininess_ball = 5;
  pColor spec_color(1,1,1);
  //  pColor spec_light = color_white * opt_light_intensity;

  const bool hide_links = opt_hide_stuff & OH_Links;
  const bool hide_platform = opt_hide_stuff & OH_Platform;
  const bool hide_sphere = opt_hide_stuff & OH_Sphere;
```

```
  sphere.init(opt_sphere_slices);

  if ( option == RO_Shadow_Volumes )
    {
      if ( !hide_sphere )
        {
          sphere.light_pos = light_location;
          sphere.render_bunch_gather_sv(world_time);
          for ( Ball *ball: balls )
            sphere.render_shadow_volume(cb,ball->radius,ball->position);

          sphere.render_bunch_render_sv(cb);
        }

      if ( !hide_links )
        {
          render_link_start();
          for ( Link *link: links )
            if ( link->is_renderable ) render_link_gather(link);
          render_link_render_sv(cb);
        }
    }
  else
    {
      sphere.rt_geo_sphere.rt_wanted_set(!hide_sphere);
      if ( !hide_sphere )
        {
          sphere.render_bunch_gather(world_time);
          for ( Ball *ball: balls )
            {
              if ( ball->contact )
                sphere.color = color_gray;
              else if ( ball->mass > 0 && ball->mass < ball->mass_min )
                sphere.color = color_red;
              else if ( ball->mass > 0 && ball->locked )
                sphere.color = color_pale_green;
              else
                sphere.color = ball->color;

              sphere.color.a = ball->specularity ? -ball->specularity : 1;
              sphere.render
                (cb, ball->radius,ball->position,
                 pMatrix_Rotation(ball->orientation));
            }

          sphere.render_bunch_render(cb, opt_sphere_true);
        }

      rt_geo_links.rt_wanted_set(!hide_links);
      if ( !hide_links )
        {
          render_link_start();
          for ( Link *link: links )
            if ( link->is_renderable ) render_link_gather(link);
          render_link_render(cb);
        }
    }

  rt_geo_platform_tex.rt_wanted_set(!hide_platform);
  rt_geo_platform_mir.rt_wanted_set(!hide_platform);

  if ( option != RO_Shadow_Volumes )
    render_surfaces(cb);

  if ( vh.raytrace_active() ) return;
  if ( hide_platform ) return;

  if ( option == RO_Shadow_Volumes ) return;
```

```
  //
  // Render Platform
  //
  pipe_platform_tex.record_draw(cb,bset_platform_tex);
}

void
World::render(vk::CommandBuffer& cb)
{
  const bool do_raytrace = vh.raytrace_active();

  /// Emit a Graphical Representation of Simulation State
  //

  // Understanding of the code below not required for introductory
  // lectures.

  // That said, much of the complexity of the code is to show
  // the ball shadow and reflection.


  const int win_width = vh.get_width();
  const int win_height = vh.get_height();
  const float aspect = float(win_width) / win_height;
  const float n_dist = 0.01;
  const float xr = .8 * n_dist;

  transform.eye_from_global_set( modelview );
  transform.clip_from_eye_set
    ( pMatrix_Frustum(-xr,xr,-xr/aspect,xr/aspect,n_dist,5000) );
  // pMatrix_Frustum: left, right, bottom, top, near, far

  float fb_dark = opt_background_shade;

  vh.clearValues[0].color =
    vk::ClearColorValue( array<float, 4>( { {fb_dark,fb_dark,fb_dark,0.5} } ) );

  /// Lighting
  //
  auto& light_all = uni_light_all->cgl_LightSource[0];

  light_all.position = transform.eye_from_global * light_location;

  light_all.constantAttenuation = 0.5;
  light_all.linearAttenuation = 1.0;
  light_all.quadraticAttenuation = 0;
  //
  // See generic_lighting or OGL documentation for details on how
  // these attenuation coefficients are used to compute lighted color.

  // Illumination that is independent of distance to light and angles.
  //
  uni_light_all->cgl_LightModel.ambient = 0.0 * color_white;

  // Illumination that is a function of distance BUT NOT angle to light.
  //
  light_all.ambient = 0.3 * opt_light_intensity * color_white;

  // Illumination that is a function of distance and angle to light.
  //
  light_all.diffuse = opt_light_intensity * color_white;

  // Illumination that is a function of distance and angle to light and eye.
  //
  light_all.specular = opt_light_intensity * color_white;

  uni_light_reflected = uni_light_all.get_val();
```

```
  uni_light_reflected->cgl_LightSource[0].position =
    transform.eye_from_global * pMatrix_Scale(1,-1,1) * light_location;

  uni_light_ambient = uni_light_all.get_val();
  auto& light_amb = uni_light_ambient->cgl_LightSource[0];
  light_amb.diffuse = color_black;
  light_amb.specular = color_black;

  uni_light = uni_light_all.get_val();
  uni_light->cgl_LightModel.ambient = color_black;
  uni_light->cgl_LightSource[0].ambient = color_black;


  uni_light.to_dev();
  uni_light_ambient.to_dev();
  uni_light_all.to_dev();
  uni_light_reflected.to_dev();

  puni_light_curr = &uni_light_all;

  buf_uni_common->tryout =
    ivec4(opt_tryout1,opt_tryout2,opt_tryout3,opt_normal_sphere);
  buf_uni_common->tryoutf = vec4(opt_tryoutf, 0, 0, 0);
  buf_uni_common->light_pos_g = light_location;
  if ( do_raytrace )
    {
      raytrace.buf_uni_common->opt_tryout = buf_uni_common->tryout;
      raytrace.buf_uni_common->opt_tryoutf = buf_uni_common->tryoutf;
      raytrace.buf_uni_common->opt_shadows = opt_shadows;
      raytrace.buf_uni_common->opt_mirror = opt_mirror;
      raytrace.buf_light_set(uni_light_all);
    }

  const double time_now = time_wall_fp();
  const bool blink_visible = int64_t(time_now*3) & 1;
# define BLINK(txt,pad) ( blink_visible ? txt : pad )

  vh.fbprintf("%s\n",frame_timer.frame_rate_text_get());

  vh.fbprintf
    ("Raytrace: %s ('R')   Compiled: %s\n",
     !vh.have_raytrace ? "NA" :
     opt_want_raytrace ? ( raytrace.rt_ready ? "ON " : "???" ) : "OFF",
#ifdef __OPTIMIZE__
     "WITH OPTIMIZATION"
#else
     BLINK("WITHOUT OPTIMIZATION","")
#endif
     );

  vh.fbprintf
    ("Energy: %.0f J  Time Step: %8d  World Time: %11.6f  %s\n",
     energy, time_step_count, world_time,
     opt_pause ? BLINK("PAUSED, 'p' to unpause, SPC or S-SPC to step.","") :
     "Press 'p' to pause."
     );

  vh.fbprintf
    ("Eye loc: [%5.1f, %5.1f, %5.1f]  "
     "Eye dir: [%+.2f, %+.2f, %+.2f] "
     "Light loc: [%5.1f, %5.1f, %5.1f]\n",
     eye_location.x, eye_location.y, eye_location.z,
     eye_direction.x, eye_direction.y, eye_direction.z,
     light_location.x, light_location.y, light_location.z);

  Ball& ball = *balls[0];

  vh.fbprintf
```

```
  ("Head Ball Pos  [%5.1f,%5.1f,%5.1f] Vel [%+5.1f,%+5.1f,%+5.1f]\n",
   ball.position.x,ball.position.y,ball.position.z,
   ball.velocity.x,ball.velocity.y,ball.velocity.z );

vh.fbprintf
  ("%4zd Balls  %4zd Links  Physics: %s ('a' to change)  \n",
   balls.size(), links.size(), gpu_physics_method_str[opt_physics_method]);

vh.fbprintf
  ("Hide: %c%c%c ('!@#')  Effect: %c%c ('or')  Sphere: %s  ('z')  "
   "Tryout 1:%s ('y')  2:%s ('Y')  3:%s ('x')  "
   "Normals: %s ('n')\n",
   opt_hide_stuff & OH_Sphere ? 'S' : '_',
   opt_hide_stuff & OH_Links ? 'L' : '_',
   opt_hide_stuff & OH_Platform ? 'P' : '_',
   opt_shadows ? 'S' : '_',
   opt_mirror ? 'M' : '_',
   opt_sphere_true ? "TRUE" : "TRI" ,
   opt_tryout1 ? BLINK("ON ","   ") : "OFF",
   opt_tryout2 ? BLINK("ON ","   ") : "OFF",
   opt_tryout3 ? BLINK("ON ","   ") : "OFF",
   opt_normal_sphere ? "SPHERE" : "TRI");

pVariable_Control_Elt* const cvar = variable_control.current;
vh.fbprintf("VAR %s = %.5f  (TAB or SH-TAB to change, +/- to adjust)\n",
                    cvar->name,cvar->get_val());

// Render Marker for Light Source
//
shapes.record_tetrahedron(cb,transform,light_location,0.15);

//
// Render ball reflection.  (Will be blended with dark tiles.)
//

const bool hide_platform = opt_hide_stuff & OH_Platform;

if ( do_raytrace )
  {
    raytrace.tform_update(transform);
    render_objects(cb,RO_Normally);
    raytrace.buf_uni_common.to_dev();

    return;
  }

buf_uni_common.to_dev();

vk::ColorComponentFlags clr_all
  ( vk::ColorComponentFlagBits::eR | vk::ColorComponentFlagBits::eG |
    vk::ColorComponentFlagBits::eB | vk::ColorComponentFlagBits::eA );

if ( !hide_platform )
  {
    // Save current state of fixed-functionality state.
    //
    ff_state.push();

    if ( opt_mirror )
      {
        /// Mirror Effect
        //
        //  - Step 1:  Write location of mirrors into stencil buffer.
        //
        //  - Step 2a: Prepare a matrix that transforms to mirrored location.
        //  - Step 2b: Set up blending to blend mirrored objects with mirrors.
        //
        //  - Step 3:  Render all objects except mirrors.
```

```
    ///   Step 1: Write location of mirrors into stencil buffer.
    //
    //    Write stencil value 2 at location of dark (mirrored) tiles.
    //

    vk::StencilOpState s_op_apply
      ( vk::StencilOp::eReplace, vk::StencilOp::eKeep,
        vk::StencilOp::eKeep, vk::CompareOp::eNever, 2, 2, 2 );
    // sfail, spass, dfail, compare;  cmask, wmask, ref

    ff_state.pipelineDepthStencilStateCreateInfo_get()
      .setStencilTestEnable(true)
      .setFront(s_op_apply).setBack(s_op_apply);

    // Render mirrored tiles (but only to write stencil buffer).
    //
    pipe_platform_mir.record_draw(cb,bset_platform_mir);

    cb.nextSubpass(vk::SubpassContents::eInline); vh.qs.subpass++;

    // Prepare to write only stenciled locations.
    //
    vk::StencilOpState s_op_use
      ( vk::StencilOp::eKeep, vk::StencilOp::eKeep,
        vk::StencilOp::eKeep, vk::CompareOp::eEqual, 2, 2, 2 );
    // sfail, spass, dfail, compare;  cmask, wmask, ref
    //
    // Stencil test passes if stencil value = 2.
    // Don't change stencil buffer.

    ff_state.pipelineDepthStencilStateCreateInfo_get()
      .setFront(s_op_use).setBack(s_op_use);

    ///   Step 2a: Prepare a matrix that transforms to mirrored location.
    //    Use a transform that reflects objects to other side of platform.
    //
    const pMatrix prev = transform.eye_from_global;
    const pMatrix_Scale reflect(1,-1,1);
    transform.eye_from_global = prev * reflect;

    // Reflected front face should still be treated as the front face.
    //
    ff_state.pipelineRasterizationStateCreateInfo_get()
      .setFrontFace(vk::FrontFace::eClockwise);

    puni_light_curr = &uni_light_reflected;

    /// Step 3: Render all objects.

    render_objects(cb,RO_Mirrored);

    transform.eye_from_global = prev;

    ff_state.pipelineRasterizationStateCreateInfo_get()
      .setFrontFace(vk::FrontFace::eCounterClockwise);
  }

// blend operation, blend factors, blend constants
ff_state.pipelineColorBlendStateCreateInfo_get()
  .setBlendConstants( array<float,4>{.5,.5,.5,.5} );

ff_state.pipelineColorBlendAttachmentState_get() =
  vk::PipelineColorBlendAttachmentState
  ( true,                    // blendEnable
    vk::BlendFactor::eConstantAlpha,  // srcColorBlendFactor
    vk::BlendFactor::eConstantAlpha,  // dstColorBlendFactor
    vk::BlendOp::eAdd,       // colorBlendOp
```

```
        vk::BlendFactor::eOne,   // srcAlphaBlendFactor
        vk::BlendFactor::eZero,  // dstAlphaBlendFactor
        vk::BlendOp::eAdd,       // alphaBlendOp
        clr_all      // colorWriteMask
        );

    puni_light_curr = &uni_light_all;;
    pipe_platform_mir.record_draw(cb,bset_platform_mir);

    ff_state.pop();
  }

if ( !opt_shadows )
  {
    // Render.
    //
    render_objects(cb,RO_Normally);
  }
else
  {
    //
    // First pass, render using only ambient light.
    //
    puni_light_curr = &uni_light_ambient;

    // Send balls, tiles, and platform to Vulkan
    //
    render_objects(cb,RO_Normally);

    //
    // Second pass, find un-shadowed areas and add on light0.
    //


    /// Write Shadow Volumes
    //
    vk::ClearAttachment clr_att
      ( vk::ImageAspectFlagBits::eStencil, 0,
        vk::ClearDepthStencilValue(0,0) );
    vk::ClearRect clr_rect( vk::Rect2D({}, vh.s_extent), 0, 1 );
    cb.clearAttachments( clr_att, clr_rect );

    ff_state.push();

    vk::StencilOpState s_op_apply_f
      ( vk::StencilOp::eKeep, vk::StencilOp::eIncrementAndWrap,
        vk::StencilOp::eKeep, vk::CompareOp::eAlways, -1, -1, 0 );
    vk::StencilOpState s_op_apply_b
      ( vk::StencilOp::eKeep, vk::StencilOp::eDecrementAndWrap,
        vk::StencilOp::eKeep, vk::CompareOp::eAlways, -1, -1, 0 );
    // sfail, spass, dfail, compare; c-mask, w-mask, ref
    // ref is -1 (all 1's) because we want to operate on all bits in
    // stencil value.

    ff_state.pipelineDepthStencilStateCreateInfo_get()
      .setStencilTestEnable(true)
      .setDepthWriteEnable(false)
      .setFront(s_op_apply_f).setBack(s_op_apply_b);

    if ( !opt_shadow_volumes )
      {
        ff_state.pipelineColorBlendAttachmentState_get()
          .setColorWriteMask( {} );
      }
    else
      {
        ff_state.pipelineColorBlendAttachmentState_get() =
          vk::PipelineColorBlendAttachmentState
```

```
            ( true,                   // blendEnable
              vk::BlendFactor::eConstantAlpha,  // srcColorBlendFactor
              vk::BlendFactor::eConstantAlpha,  // dstColorBlendFactor
              vk::BlendOp::eAdd,       // colorBlendOp
              vk::BlendFactor::eOne,   // srcAlphaBlendFactor
              vk::BlendFactor::eZero,  // dstAlphaBlendFactor
              vk::BlendOp::eAdd,       // alphaBlendOp
              clr_all      // colorWriteMask
              );

          ff_state.pipelineColorBlendStateCreateInfo_get()
            .setBlendConstants( array<float,4>{.5,.5,.5,.5} );

          ff_state.pipelineDepthStencilStateCreateInfo_get()
            .setDepthWriteEnable(true);
        }

      // Write stencil with shadow locations based on shadow volumes
      // cast by light0 (light_location).  Shadowed locations will
      // have a positive stencil value.
      //
      render_objects(cb,RO_Shadow_Volumes);

      cb.nextSubpass(vk::SubpassContents::eInline); vh.qs.subpass++;

      // Use stencil test to prevent writes to shadowed areas.
      //
      vk::StencilOpState s_op_apply_fb
        ( vk::StencilOp::eKeep, vk::StencilOp::eKeep,
          vk::StencilOp::eKeep, vk::CompareOp::eEqual, -1, -1, 0 );
      // sfail, spass, dfail, compare;  c-mask, w-mask, ref
      //
      // Stencil test passes if stencil value == 0.

      ff_state.pipelineDepthStencilStateCreateInfo_get()
        .setFront(s_op_apply_fb).setBack(s_op_apply_fb)
        .setDepthCompareOp(vk::CompareOp::eEqual);

      // Allow pixels to be re-written.
      //
      vk::ColorComponentFlags cmask_all
        ( vk::ColorComponentFlagBits::eR | vk::ColorComponentFlagBits::eG |
          vk::ColorComponentFlagBits::eB | vk::ColorComponentFlagBits::eA );

      ff_state.pipelineColorBlendAttachmentState_get() =
        vk::PipelineColorBlendAttachmentState
        ( true,                   // blendEnable
          vk::BlendFactor::eOne,  // srcColorBlendFactor
          vk::BlendFactor::eOne,  // dstColorBlendFactor
          vk::BlendOp::eAdd,       // colorBlendOp
          vk::BlendFactor::eOne,   // srcAlphaBlendFactor
          vk::BlendFactor::eOne,  // dstAlphaBlendFactor
          vk::BlendOp::eAdd,       // alphaBlendOp
          cmask_all      // colorWriteMask
          );

      puni_light_curr = &uni_light;

      // Render.
      //
      render_objects(cb,RO_Normally);

      ff_state.pop();
    }

  puni_light_curr = &uni_light_all;
}
```

```
void
World::cb_keyboard()
{
  const int key = vh.keyboard_key_get();
  if ( !key ) return;
  pVect adjustment(0,0,0);
  pVect user_rot_axis(0,0,0);
  const bool kb_mod_s = vh.keyboard_shift;
  const bool kb_mod_c = vh.keyboard_control;
  const float move_amt = kb_mod_s ? 2.0 : kb_mod_c ? 0.08 : 0.4;

  switch ( key ) {
  case FB_KEY_LEFT: adjustment.x = -move_amt; break;
  case FB_KEY_RIGHT: adjustment.x = move_amt; break;
  case FB_KEY_PAGE_UP: adjustment.y = move_amt; break;
  case FB_KEY_PAGE_DOWN: adjustment.y = -move_amt; break;
  case FB_KEY_DOWN: adjustment.z = move_amt; break;
  case FB_KEY_UP: adjustment.z = -move_amt; break;
  case FB_KEY_DELETE: user_rot_axis.y = 1; break;
  case FB_KEY_INSERT: user_rot_axis.y =  -1; break;
  case FB_KEY_HOME:
    if ( opt_move_item == MI_Light ) adjustment.x = 1; else user_rot_axis.x = 1;
    break;
  case FB_KEY_END: user_rot_axis.x = -1; break;
  case '1': ball_setup_1(kb_mod_c); break;
  case '2': ball_setup_2(); break;
  case '3': ball_setup_3(); break;
  case '4': ball_setup_4(); break;
  case '5': ball_setup_5(); break;
  case '6': ball_setup_6(); break;
  case '7': ball_setup_7(!kb_mod_c); break;
  case 'a':
    opt_physics_method++;
    if ( opt_physics_method == GP_ENUM_SIZE ) opt_physics_method = GP_cpu;
    break;
  case 'b': opt_move_item = MI_Ball; break;
  case 'B': opt_move_item = MI_Ball_V; break;
  case 'e': case 'E': opt_move_item = MI_Eye; break;
  case 'g': case 'G': opt_gravity = !opt_gravity; break;
  case 'h': case 'H': opt_head_lock = !opt_head_lock; break;
  case '!': opt_hide_stuff ^= OH_Sphere; break;
  case '@': opt_hide_stuff ^= OH_Links; break;
  case '#': opt_hide_stuff ^= OH_Platform; break;
  case 't': case 'T': opt_tail_lock = !opt_tail_lock; break;
  case 'l': case 'L': opt_move_item = MI_Light; break;
  case 'r': opt_mirror = !opt_mirror; break;
  case 'R':
    opt_want_raytrace = !opt_want_raytrace;
    world_time_link_update = 0;
    break;
  case 'o': opt_shadows = !opt_shadows; break;
  case 'O':
    opt_shadow_volumes = !opt_shadow_volumes;
    if ( opt_shadow_volumes ) opt_shadows = true;
    break;
  case 'n': case 'N': opt_normal_sphere = !opt_normal_sphere; break;
  case 'p': case 'P': opt_pause = !opt_pause; break;
  case 's': case 'S': balls_stop(); break;
  case 'v': case 'V': opt_shader++;
    if ( opt_shader >= sizeof(sh_names)/sizeof(sh_names[0]) ) opt_shader = 0;
    break;
  case 'w': case 'W': balls_twirl(); break;
  case 'x': opt_tryout3 = !opt_tryout3; break;
  case 'y': opt_tryout1 = !opt_tryout1; break;
  case 'Y': opt_tryout2 = !opt_tryout2; break;
  case 'z': opt_sphere_true = !opt_sphere_true; break;
  case ' ':
```

```
    if ( kb_mod_s && kb_mod_c ) opt_single_time_step = 10;
    else if ( kb_mod_s ) opt_single_time_step = 100;
    else if ( kb_mod_c ) opt_single_time_step = 1; else opt_single_frame = true;
    opt_pause = true;
    break;
  case FB_KEY_TAB:
    if ( !kb_mod_s ) { variable_control.switch_var_right(); break; }
  case 96: variable_control.switch_var_left(); break;
  case '-':case '_': variable_control.adjust_lower(); break;
  case '+':case '=': variable_control.adjust_higher(); break;
  default: printf("Unknown key, %d\n",key); break;
  }

  gravity_accel.y = opt_gravity ? -opt_gravity_accel : 0;

  opt_mirror_depth = raytrace.recursion_max_set( opt_mirror_depth + 1 ) - 1;
  lock_update();
  sphere.bunch_invalidate();

  // Update eye_direction based on keyboard command.
  //
  if ( user_rot_axis.x || user_rot_axis.y )
    {
      pMatrix_Rotation rotall(eye_direction,pVect(0,0,-1));
      user_rot_axis *= invert(rotall);
      eye_direction *= pMatrix_Rotation(user_rot_axis, numbers::pi * 0.03);
      modelview_update();
    }

  // Update eye_location based on keyboard command.
  //
  if ( adjustment.x || adjustment.y || adjustment.z )
    {
      const double angle =
        fabs(eye_direction.y) > 0.99
        ? 0 : atan2(eye_direction.x,-eye_direction.z);
      pMatrix_Rotation rotall(pVect(0,1,0),-angle);
      adjustment *= rotall;

      switch ( opt_move_item ){
      case MI_Ball:
        adj_vector = adjustment;
        if ( adj_t_stop == 0 )
          adj_t_prev = adj_t_stop = world_time;
        adj_t_stop += adj_duration;
        break;
      case MI_Ball_V: balls_push(adjustment,0); break;
      case MI_Light:
        if ( key == FB_KEY_HOME )
          {
            pVect l_to_b(light_location,balls[0]->position);
            adjustment = 0.25 * l_to_b;
          }
        light_location += adjustment;
        break;
      case MI_Eye: eye_location += adjustment; break;
      default: break;
      }
      modelview_update();
    }

}



void
World::init(int argc, char **argv)
```

```
{
  chain_length = 14;

  opt_time_step_duration = 0.00003;
  variable_control.insert(opt_time_step_duration,"Time Step Duration");

  distance_relaxed = 15.0 / chain_length;
  opt_spring_constant = 500;
  variable_control.insert(opt_spring_constant,"Spring Constant");

  opt_spring_damp = 0.7;
  variable_control.insert(opt_spring_damp,"Spring Damping").set_max(1);

  opt_friction_dynamic = 0.35;
  variable_control.insert(opt_friction_dynamic,"Dynamic Friction");

  opt_omega = 5;
  variable_control.insert(opt_omega,"Initial Rotation Rate");

  opt_gravity_accel = 9.8;
  opt_gravity = true;
  gravity_accel = pVect(0,-opt_gravity_accel,0);
  variable_control.insert(opt_gravity_accel,"Gravity");

  opt_air_resistance = 0.001;
  variable_control.insert(opt_air_resistance,"Air Resistance");

  world_time = 0;
  time_step_count = 0;
  last_frame_wall_time = time_wall_fp();
  frame_timer.work_unit_set("Steps / s");

  world_time_link_update = 0;

  opt_mirror = true;
  opt_mirror_depth = 2;
  variable_control.insert(opt_mirror_depth,"Mirror Depth",1,0);
  opt_shadows = true;
  opt_shadow_volumes = false;
  opt_shader = SO_Plain;

  /// Init CUDA

  // Get information about GPU and its ability to run CUDA.
  //
  int device_count;
  cudaGetDeviceCount(&device_count); // Get number of GPUs.
  if ( device_count == 0 )
    {
      fprintf(stderr,"No GPU found, exiting.\n");
      exit(1);
    }

  // Print information about the available GPUs.
  //
  gpu_info_print();

  // Use course library routine to choose a good GPU for CUDA.
  //
  int dev = gpu_choose_index();
  CE(cudaSetDevice(dev));
  printf("Using GPU %d\n",dev);

  gpu_info.get_gpu_info(dev);
  cuda_setup(&gpu_info);

  // Print information about time_step routine.
  //
```

```
printf("\nCUDA Routine Resource Usage:\n");

int block_size_max = 4096;

for ( int i=0; i<gpu_info.num_kernels; i++ )
  {
    printf("For %s:\n", gpu_info.ki[i].name);
    printf("  %6zd shared, %zd const, %zd loc, %d regs; "
           "%d max threads per block.\n",
           gpu_info.ki[i].cfa.sharedSizeBytes,
           gpu_info.ki[i].cfa.constSizeBytes,
           gpu_info.ki[i].cfa.localSizeBytes,
           gpu_info.ki[i].cfa.numRegs,
           gpu_info.ki[i].cfa.maxThreadsPerBlock);
    set_min(block_size_max,gpu_info.ki[i].cfa.maxThreadsPerBlock);
  }

opt_physics_method = GP_cuda;
gpu_const_data_stale = true;
cpu_phys_data_stale = false;

CE( cudaEventCreate(&frame_start_ce) );
CE( cudaEventCreate(&frame_stop_ce) );

opt_block_size = 32;

variable_control.insert_power_of_2
  (opt_block_size, "Block Size", 32, block_size_max);

c.alloc_n_balls = 0;
c.alloc_n_links = 0;

init_graphics();

opt_sphere_slices = 35;
sphere.opt_texture = true;
sphere.init(opt_sphere_slices);
variable_control.insert(opt_sphere_slices,"Sphere Slices");

const char* const links_shader_code_path = "links-shdr-links.cc";

const auto bvec_vs = vk::ShaderStageFlagBits::eVertex;
const auto bvec_vgs =
  vk::ShaderStageFlagBits::eVertex | vk::ShaderStageFlagBits::eGeometry;

pipe_set_2
  .init( ff_state )
  .ds_follow( transform )
  .ds_follow( puni_light_curr )
  .ds_uniform_use( "BIND_UNI_COMMON", buf_uni_common )
  .ds_set_material( color_link, color_salmon )
  .ds_use( sampler, texid_hw )
  .ds_storage_follow( "BIND_LINKS_POS1", lis_pos1, bvec_vs )
  .ds_storage_follow( "BIND_LINKS_POS2", lis_pos2, bvec_vs )
  .ds_storage_follow( "BIND_LINKS_V1", lis_v1, bvec_vs )
  .ds_storage_follow( "BIND_LINKS_V2", lis_v2, bvec_vs )
  .ds_storage_follow( "BIND_LINKS_B1_YDIR", lis_b1_ydir, bvec_vs )
  .ds_storage_follow( "BIND_LINKS_B2_YDIR", lis_b2_ydir, bvec_vs )
  .ds_storage_follow( "BIND_LINKS_TEX_RAD", lis_tex_rad, bvec_vgs )
  .shader_inputs_info_set( )
  .shader_code_set
  (links_shader_code_path,
   "vs_main_2(); ",     // Name of vertex shader main routine.
   "gs_main_2();",      // Name of geometry shader main routine.
   "fs_main();"         // Name of fragment shader main routine.
   )
  .topology_set( vk::PrimitiveTopology::eLineStrip )
  .create();
```

```
  if ( vh.have_raytrace )
    comp_links
      .init( vh.qs )
      .ds_uniform_use( "BIND_UNI_COMMON", buf_uni_common )
      .ds_uniform_use( "BIND_TRANSFORM", raytrace.buf_uni_transform )
      .ds_storage_follow( "BIND_LINKS_POS1", lis_pos1 )
      .ds_storage_follow( "BIND_LINKS_POS2", lis_pos2 )
      .ds_storage_follow( "BIND_LINKS_V1", lis_v1 )
      .ds_storage_follow( "BIND_LINKS_V2", lis_v2 )
      .ds_storage_follow( "BIND_LINKS_B1_YDIR", lis_b1_ydir )
      .ds_storage_follow( "BIND_LINKS_B2_YDIR", lis_b2_ydir )
      .ds_storage_follow( "BIND_LINKS_TEX_RAD", lis_tex_rad )
      .ds_storage_follow( "BIND_LINKS_AS_IDX", buf_links_as_idx )
      .ds_storage_follow( "BIND_LINKS_TEX_COOR", buf_links_tex_coor )
      .ds_storage_follow( "BIND_LINKS_VERTEX_G", buf_links_vertex_g )
      .ds_storage_follow( "BIND_LINKS_NORMAL_G", buf_links_normal_g )
      .shader_code_set(links_shader_code_path, "comp_main(); ")
      .create();

  pipe_set_2_sv
    .init( ff_state ).ds_use( uni_light )
    .ds_follow( transform )
    .ds_uniform_use( "BIND_UNI_COMMON", buf_uni_common )
    .ds_storage_follow( "BIND_LINKS_POS1", lis_pos1 )
    .ds_storage_follow( "BIND_LINKS_POS2", lis_pos2 )
    .ds_storage_follow( "BIND_LINKS_V1", lis_v1 )
    .ds_storage_follow( "BIND_LINKS_V2", lis_v2 )
    .ds_storage_follow( "BIND_LINKS_B1_YDIR", lis_b1_ydir )
    .ds_storage_follow( "BIND_LINKS_B2_YDIR", lis_b2_ydir )
    .ds_storage_follow( "BIND_LINKS_TEX_RAD", lis_tex_rad )
    .shader_inputs_info_set( )
    .shader_code_set
    (links_shader_code_path,
     "vs_main_2_sv(); ",     // Name of vertex shader main routine.
     "gs_main_2_sv();",      // Name of geometry shader main routine.
     "fs_main_sv();"         // Name of fragment shader main routine.
     )
    .topology_set( vk::PrimitiveTopology::eLineStrip )
    .create();

  ball_setup_7(true);
  return;
  // For benchmarking.
  opt_pause = true;
  opt_hide_stuff = OH_Links | OH_Platform;
  opt_shadows = false;
  opt_mirror = false;
}

Ball*
World::make_marker(pCoor position, pColor color, float radius )
{
  Ball* const ball = new Ball;
  ball->position = position;
  ball->locked = true;
  ball->velocity = pVect(0,0,0);
  ball->radius = radius;
  ball->mass = -1;
  ball->contact = false;
  ball->color = color;
  balls_marker.push_back(ball);
  return ball;
}

void
World::lock_update()
{
```

```
  // This routine called when options like opt_head_lock might have
  // changed.

  // Update locked status.
  //
  if ( head_ball ) head_ball->locked = opt_head_lock;
  if ( tail_ball ) tail_ball->locked = opt_tail_lock;

  // Re-compute minimum mass needed for stability.
  //
  for ( Ball *ball: balls ) ball->spring_constant_sum = 0;
  const double dtis = pow( opt_time_step_duration, 2 );
  for ( Link *link: links )
    {
      Ball* const b1 = link->ball1;
      Ball* const b2 = link->ball2;
      b1->spring_constant_sum += opt_spring_constant;
      b2->spring_constant_sum += opt_spring_constant;
    }
  for ( Ball *ball: balls )
    {
      ball->mass_min = ball->spring_constant_sum * dtis;
      ball->constants_update();
    }
  gpu_const_data_stale = true;
}

void
World::balls_twirl()
{
  if ( !head_ball || !tail_ball ) return;

  pNorm axis(head_ball->position, tail_ball->position);
  const float omega = 0.1 * opt_omega;

  for ( Ball *ball: balls )
    {
      pVect b_to_top(ball->position,head_ball->position);
      const float dist_along_axis = dot(b_to_top,axis);
      const float lsq = b_to_top.mag_sq() - dist_along_axis * dist_along_axis;
      ball->omega += omega * axis;
      if ( lsq <= 1e-5 ) continue;
      const float dist_to_axis = sqrt(lsq);
      pNorm rot_dir = cross(b_to_top,axis);
      ball->velocity += omega * dist_to_axis * rot_dir;
    }
}

void
World::objects_erase()
{
  balls.erase();
  links.erase();
  surfaces.clear();
}

Links
World::link_new
(Ball *ball1, Ball *ball2, float stiffness, float spring_constant_p)
{
  Link_Args largs
    { .spring_constant = spring_constant_p, .link_stiffness = stiffness };
  return link_new(ball1,ball2,largs);
}

Links
World::link_new
```

```
(Ball *ball1, Ball *ball2, Link_Args largs )
{
  const float spring_constant_p = largs.spring_constant;
  const float stiffness = largs.link_stiffness;
  Links links_rv;
  assert( ball1->radius > 0 );
  assert( ball2->radius > 0 );

  const float spring_constant_base = spring_constant_p ?: opt_spring_constant;

  ball1->constants_update();
  ball2->constants_update();
  float mass_ref = 0.04 / 3 * numbers::pi;
  float mass_min = ball1->locked ? ball2->mass :
    ball2->locked ? ball1->mass : min(ball1->mass,ball2->mass);
  float spring_constant = spring_constant_base * mass_min / mass_ref;

  Link* const rlink = new Link(ball1,ball2);
  rlink->radius = largs.radius;
  links_rv += rlink;

  pNorm n12(ball1->position,ball2->position);
  const float rad_sum = ball1->radius + ball2->radius;
  pMatrix3x3 b1rot = ball1->omatrix;
  pMatrix3x3 b2rot = ball2->omatrix;
  pCoor ctr = ball1->position
    + ( ball1->radius + 0.5 * ( n12.magnitude - rad_sum ) ) * n12;

  pVect con_x = rlink->con_x;
  pVect con_y = rlink->con_y;

  const float lrad = stiffness * ball1->radius;

  const int n_springs = 4;
  const double delta_theta = 2 * numbers::pi / n_springs;

  for ( int i=0; i<n_springs; i++ )
    {
      const double theta = i * delta_theta;
      pVect convec = lrad * ( cos(theta) * con_x + sin(theta) * con_y );
      pCoor con = ctr + convec;
      pVect cb1 = mult_MTV( b1rot, con - ball1->position );
      pVect cb2 = mult_MTV( b2rot, con - ball2->position );
      links_rv +=
        new Link(rlink, cb1, cb2,
                 { .distance_relaxed=0, .spring_constant=spring_constant } );
    }
  return links_rv;
}

///
/// CUDA Code
///

float4 tof4(pCoor c)
{
  float4 f4;
  f4.x = c.x;
  f4.y = c.y;
  f4.z = c.z;
  f4.w = c.w;
  return f4;
}

float4 tof4(pVect c)
{
  float4 f4;
```

```
  f4.x = c.x;
  f4.y = c.y;
  f4.z = c.z;
  f4.w = 0;
  return f4;
}

pCoor topc(float4 c)
{
  pCoor f4;
  f4.x = c.x;
  f4.y = c.y;
  f4.z = c.z;
  f4.w = c.w;
  return f4;
}

void
World::data_cpu_to_gpu_constants()
{
  c.platform_xmin = platform_xmin;
  c.platform_xmax = platform_xmax;
  c.platform_zmax = platform_zmax;
  c.platform_zmin = platform_zmin;

  c.n_balls = balls.size();
  c.n_links = links.size();
  c.n_surfaces = surfaces.size();
  c.opt_tryout1 = opt_tryout1;
  c.opt_tryout2 = opt_tryout2;
  c.opt_tryout3 = opt_tryout3;
  c.opt_tryoutf = opt_tryoutf;
  c.opt_spring_constant = opt_spring_constant;
  c.opt_spring_damp = opt_spring_damp;
  c.opt_air_resistance = opt_air_resistance;
  c.opt_friction_dynamic = opt_friction_dynamic;
  c.opt_head_lock = opt_head_lock;
  c.opt_tail_lock = opt_tail_lock;
  c.gravity_accel = gravity_accel;

  data_cpu_to_gpu_common(&c);
}


void
World::data_cpu_to_gpu_dynamic()
{
  const int n_balls = balls.size();
  const int n_links = links.size();
  const int n_surfaces = surfaces.size();
  const bool need_alloc =
    n_balls > c.alloc_n_balls || n_links > c.alloc_n_links
    || n_surfaces > c.alloc_n_surfaces;
  const bool need_init = c.alloc_n_balls == 0;

#define ALLOC_MOVE_ZERO(n_balls,balls,memb,do_move,do_zero)                \
  {                                                                        \
    const int size_elt_bytes = sizeof(c.balls.memb[0]);                    \
    const int size_bytes = n_balls * size_elt_bytes;                       \
    if ( need_alloc )                                                      \
      {                                                                    \
        if ( !need_init )                                                  \
          {                                                                \
            CE( cudaFree( c.balls.memb ) );                                \
            free(c.h_##balls.memb);                                        \
          }                                                                \
        CE( cudaMalloc( &c.balls.memb, size_bytes ) );                     \
```

```
        c.h_##balls.memb = (decltype(c.balls.memb)) malloc( size_bytes );     \
      }                                                                       \
    if ( do_move )                                                            \
      {                                                                       \
        for ( int i=0; i<n_balls; i++ )                                       \
          c.h_##balls.memb[i] = balls[i]->memb;                               \
        CE( cudaMemcpy                                                        \
            ( c.balls.memb, c.h_##balls.memb,                                 \
              size_bytes, cudaMemcpyDefault ) );                              \
      }                                                                       \
    else if ( do_zero )                                                       \
      {                                                                       \
        CE( cudaMemset( c.balls.memb, 0, size_bytes ) );                      \
      }                                                                       \
  }

#define MOVEc(num,ptr,memb) \
  ALLOC_MOVE_ZERO(num,ptr,memb,gpu_const_data_stale,0)
#define MOVE(num,ptr,memb) ALLOC_MOVE_ZERO(num,ptr,memb,1,0)
#define ALLOC(num,ptr,memb) ALLOC_MOVE_ZERO(num,ptr,memb,0,0)
#define ZERO(num,ptr,memb) ALLOC_MOVE_ZERO(num,ptr,memb,0,1)
  MOVEc(n_balls,balls,position);
  MOVEc(n_balls,balls,velocity);
  ZERO(n_balls,balls,force);

  MOVEc(n_balls,balls,orientation);
  MOVEc(n_balls,balls,omatrix);
  MOVEc(n_balls,balls,omega);
  ZERO(n_balls,balls,torque);
  MOVEc(n_balls,balls,m_r_fdt);
  MOVEc(n_balls,balls,locked);

  MOVEc(n_links,links,ball_idx);
  MOVEc(n_links,links,cb1);
  MOVEc(n_links,links,cb2);
  MOVEc(n_links,links,dr_sc);
  MOVEc(n_links,links,is_simulatable);
  MOVEc(n_links,links,is_surface_connection);

  MOVEc(n_surfaces,surfaces,b_idx);

#undef MOVE
#undef MOVEc
#undef ALLOC
#undef ZERO
#undef ALLOC_MOVE
  if ( need_alloc )
    {
      c.alloc_n_balls = n_balls;
      c.alloc_n_links = n_links;
      c.alloc_n_surfaces = n_surfaces;
      data_cpu_to_gpu_common(&c);
    }
  gpu_const_data_stale = false;
}

void
World::data_gpu_to_cpu_dynamic()
{
  const int n_balls = balls.size();

#define MOVE(n_balls,balls,memb)                                              \
  {                                                                           \
    const int size_elt_fr_bytes = sizeof(c.balls.memb[0]);                    \
    const int size_bytes = n_balls * size_elt_fr_bytes;                       \
    CE( cudaMemcpy                                                            \
        ( c.h_##balls.memb, c.balls.memb, size_bytes,                         \
```

```
          cudaMemcpyDeviceToHost ) );                                    \
    for ( int i=0; i<n_balls; i++ )                                      \
      balls[i]->memb = c.h_##balls.memb[i];                              \
  }

  MOVE(n_balls,balls,position);
  MOVE(n_balls,balls,velocity);
  MOVE(n_balls,balls,orientation);
  MOVE(n_balls,balls,omatrix);
  MOVE(n_balls,balls,omega);

#undef MOVE

}


///
/// Physical Simulation Code
///

 /// Initialize Simulation
//

void
World::ball_setup_1(bool big)
{
  // Arrange balls to form sort of tree.

  last_setup = 1;

  double ball_rad = 1;  // Warning: Re-assigned, changes behavior of nb.

  pCoor first_pos(17.2,ball_rad,-20.2);

  pVect dir_up(0.1,3,0.1);
  pVect dir_z(3*distance_relaxed,0,0);
  pVect dir_x(0,0,3*distance_relaxed);

  // Remove objects from the simulated objects lists, balls and links.
  // The delete operator is used on objects in the lists.
  //
  objects_erase();

  auto nb =
    [&] (pCoor pos) {
      Ball* const ball = new Ball;
      ball->position = pos;
      ball->locked = false;
      ball->velocity = pVect(0,0,0);
      ball->radius = ball_rad;
      ball->contact = false;
      balls += ball;
      return ball;
      };

  tail_ball = nb(first_pos);
  tail_ball->specularity = 0.5;

  pCoor last_pos = tail_ball->position;

  const int n_tiers = big ? 32 : 8;
  const double shrink_final = 0.17;
  const double shrink = pow(shrink_final,1.0/n_tiers);

  Balls tail_balls;
  for ( int i=1; i<n_tiers; i++ )
    {
```

```
      ball_rad *= shrink;
      Ball* const ball = nb(last_pos + (i==1?1.0:ball_rad) * dir_up);
      ball->specularity = 0.5;
      tail_balls += ball;
      links += link_new(balls[balls-2],ball,0.8);
      last_pos = ball->position;
    }

  head_ball = balls[balls-1];

  pNorm dir_zn(dir_z);
  pNorm dir_xn(dir_x);
  for ( Ball *ball: tail_balls )
    {
      if ( ball == head_ball ) break;
      pNorm to_top(ball->position,head_ball->position);
      const float r = to_top.magnitude;
      int num_hairs = max(4,int(7*r));
      ball_rad = ball->radius / 4;
      const float delta_theta = 2 * numbers::pi / num_hairs;
      for ( int i=0; i<num_hairs; i++ )
        {
          const float theta = i * delta_theta;
          pCoor pos = ball->position +
            r * cos(theta) * dir_xn + r * sin(theta) * dir_zn;
          Ball* const ballr = nb(pos);
          links += link_new(ballr,ball);
        }
    }

  opt_head_lock = false;
  opt_tail_lock = true;
  setup_at_end();
}

void
World::ball_setup_2()
{
  // Arrange balls to form a suspended thing.

  last_setup = 2;

  pCoor first_pos(17.2,14.8f,-20.2);

  pVect dir_dn(0.5,first_pos.y/8,0.5);
  pVect dir_z(3*distance_relaxed,0,0);
  pVect dir_x(0,0,3*distance_relaxed);

  // Remove objects from the simulated objects lists, balls and links.
  // The delete operator is used on objects in the lists.
  //
  objects_erase();

  float ball_rad = 0.3;  // Warning: Re-assigned, changes behavior of nb.

  auto nb =
    [&] (pCoor pos) {
      Ball* const ball = new Ball;
      ball->position = pos;
      ball->locked = false;
      ball->velocity = pVect(0,0,0);
      ball->radius = ball_rad;
      ball->contact = false;
      ball->specularity = 0.5;
      balls += ball;
      return ball;
      };
```

```
  head_ball = nb(first_pos);
  for ( int j: {-1,1} )
    for ( int i: {-1,0,1} )
      {
        Ball* const ball = nb(first_pos - dir_dn + i * dir_x + j * dir_z );
        if ( i != 0 ) links += link_new(head_ball,ball);
        if ( i != -1 ) links += link_new(balls[balls-2],ball);
        if ( j == 1 && i != 0 )  links += link_new(balls[balls-4],ball);
      }

  Ball* const tail_start = nb(first_pos - dir_dn);
  links += link_new(balls[balls-6],tail_start);
  links += link_new(balls[balls-3],tail_start);
  pCoor last_pos = tail_start->position;
  Balls tail_balls;
  for ( int i=1; i<5; i++ )
    {
      Ball* const ball = nb(last_pos - 0.5 * dir_dn);
      tail_balls += ball;
      links += link_new(balls[balls-2],ball);
      last_pos = ball->position;
    }

  tail_ball = balls[balls-1];

  pNorm dir_zn(dir_z);
  pNorm dir_xn(dir_x);
  int num_hairs = 20;
  ball_rad = 0.1;
  for ( Ball *ball: tail_balls )
    {
      const float delta_theta = 2 * numbers::pi / num_hairs;
      for ( int i=0; i<num_hairs; i++ )
        {
          const float theta = i * delta_theta;
          const float r = ball->radius * 10;
          pCoor pos = ball->position +
            r * cos(theta) * dir_xn + r * sin(theta) * dir_zn;
          Ball* const ballr = nb(pos);
          links += link_new(ballr,ball);
        }
    }

  opt_head_lock = true;    // Head ball will be frozen in space.
  opt_tail_lock = false;   // Tail ball can move freely.
  setup_at_end();
}

void
World::ball_setup_3()
{
  // Arrange links to form what I would eventually learn to be a
  // corona virus.

  const bool sarcov = true;

  last_setup = 3;
  objects_erase();

  pCoor first_pos(17.2,14.8f,-10.2);

  float ball_radius = 1;

  auto nb =
    [&] (pCoor pos) {
      Ball* const ball = new Ball;
      ball->position = pos;
      ball->locked = false;
```

```
      ball->velocity = pVect(0,0,0);
      ball->radius = ball_radius;
      ball->contact = false;
      balls += ball;
      return ball;
      };

  head_ball = nb(first_pos);
  head_ball->specularity = 0.5;
  ball_radius = ball_radius * ( sarcov ? 0.08 : .1 );
  const int n_spikes = sarcov ? 100 : 500;

  for ( int i=0; i<n_spikes; i++ )
    {
      pNorm dir(0.5-drand48(),0.5-drand48(),0.5-drand48());
      const float len = head_ball->radius * ( sarcov ? 1.5 : 3.0 );
      Ball* const ball = nb(first_pos + len * dir);
      links += link_new(ball,head_ball,2);
    }

  tail_ball = balls[1];

  opt_head_lock = false;    // Head ball will be frozen in space.
  opt_tail_lock = false;   // Tail ball can move freely.

  setup_at_end();
}


// Some day use <random>
pVect vec_rand() {return pVect(drand48(),drand48(),drand48());}
float frand(float range){ return drand48() * range; }
float frand(float low, float high){ return low + frand(high-low); }
int irand(int range){ return random()%range; }
int irand(int low, int high){ return low + irand(high-low); }
pVect vec_randc() {return pVect(frand(-1,1),frand(-1,1),frand(-1,1));}

void
World::ball_setup_4()
{
  // Render a wheel (disk of balls with one in the center).

  last_setup = 4;
  objects_erase();

  pCoor center_pos = pCoor(17.2,14.8f,-10.2) + 2*vec_rand();

  const int n_balls = 4 * ( 4 + random()%4 );
  float ring_outer_radius = 5 + 2 * drand48();
  float ball_radius = 0.2 * 2 * numbers::pi * ring_outer_radius / n_balls;

  pNorm c_to_eye(center_pos,eye_location);
  pNorm az = c_to_eye + 0.5 * vec_rand() + pVect(0,0.5,0);

  pNorm ax = fabs(az.x) < fabs(az.y)
    ? pVect(0,az.z,-az.y) : pVect(az.z,0,-az.x);
  pVect ay(az,ax);

  // Default settings for balls.
  //
  Ball ball_default;
  ball_default.specularity = 0.5;
  ball_default.velocity = pVect(0,0,0);
  ball_default.radius = ball_radius;
  ball_default.color = color_light_slate_gray;

  // Scale axes by radius so that they can be used to find points on circle.
```

```
  //
  pVect vx = ring_outer_radius * ax;
  pVect vy = ring_outer_radius * ay;

  const float d_theta = 2 * numbers::pi / n_balls;

  // Place balls in a circle.
  //
  for ( int i=0; i<n_balls; i++ )
    {
      const float theta = d_theta * i;
      pCoor pos = center_pos + vx * cosf(theta) + vy * sinf(theta);

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

      // Set color of first ball to blue, others to white.
      ball->color = i == 0 ? color_blue : color_white;
    }

  const float link_stiffness = 10;

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

  // Place a ball in the center and connect it to all the other balls.
  if ( true )
    {
      Ball* const center_ball =
        balls.emplace_back( new Ball(center_pos,ball_default) );

      for ( int i=0; i<n_balls; i++ )
        links += link_new( center_ball, balls[i], 2 );
    }

  pNorm spin_axis = az;
  pCoor c_of_mass = center_pos;

  for ( Ball* ball: balls )
    {
      pVect cm_to_ball( c_of_mass, ball->position );

      ball->velocity = opt_omega * cross( spin_axis, cm_to_ball );
      ball->omega = opt_omega * spin_axis;
    }

  tail_ball = balls.back();
  head_ball = balls.front();

  opt_head_lock = false; // If true, head ball locked in place.
  opt_tail_lock = false;

  setup_at_end();
}

void
World::ball_setup_5()
{
  // Construct a top. (Top, as in a child's toy.)

  last_setup = 5;
  objects_erase();

  pCoor center_pos(17.2,14.8f,-10.2);
```

```
const int n_balls = 20;
const float crad = 5;

const float ball_radius = 0.3 * 2 * numbers::pi * crad / n_balls;

// Default settings for balls.
//
Ball ball_default;
ball_default.specularity = 0.5;
ball_default.velocity = pVect(0,0,0);
ball_default.radius = ball_radius;
ball_default.color = pColor(0xffbbff); // Plum 1. That's the color's name.

const float d_theta = 2 * numbers::pi / n_balls;

pVect vz = pVect(1,6,1) - 2*vec_rand();
pNorm ax( vz.y, -vz.x, 0 );
pNorm ay = cross(vz,ax);
pVect vx = crad * ax;
pVect vy = crad * ay;

pNorm spin_axis = vz;
float omega = opt_omega;
opt_spring_damp = .4;

// Construct Top's Main Ring
//
for ( int i=0; i<n_balls; i++ )
  {
    const float theta = d_theta * i;
    pCoor pos = center_pos + vx * cosf(theta) + vy * sinf(theta);
    Ball* ball = balls.emplace_back( new Ball(pos,ball_default) );

    // Find the coordinate on the spin axis closest to the new ball.
    //
    pCoor axis_cp =
      center_pos + spin_axis * dot( spin_axis, pVect(center_pos,pos) );
    //
    // Use this coordinate to compute a vector giving the ball's
    // motion when the top is spinning around axis spin_axis.
    //
    pVect cp_to_p(axis_cp,pos);
    ball->velocity = omega * cross(spin_axis,cp_to_p);
  }

// Make one of the balls on the top's main ring a different color.
balls[0]->color = color_blue;

// Connect the balls on the main ring with links.
//
for ( int i=0; i<n_balls; i++ )
  links += link_new( balls[i], balls[(i+1)%n_balls], 2 );

// Complete the top with balls above and below the ring, and one in
// the center.
//
pNorm az(vz);
pCoor top_pos = center_pos + az * crad * 1.5;
pCoor bot_pos = center_pos - az * crad * 1.5;

head_ball = balls.emplace_back( new Ball(top_pos,ball_default) );
tail_ball = balls.emplace_back( new Ball(bot_pos,ball_default) );
Ball* ctr_ball = balls.emplace_back( new Ball(center_pos,ball_default) );

for ( int i=0; i<n_balls; i += 2 )
  {
    links += link_new( head_ball, balls[i], 2 );
    links += link_new( tail_ball, balls[i], 2 );
```

```
    }

  for ( int i=0; i<n_balls; i++ )
    links += link_new( ctr_ball, balls[i], 2 );

  for ( auto b: balls ) b->omega = omega * spin_axis;

  opt_head_lock = false;    // Head ball will be frozen in space.
  opt_tail_lock = false;   // Tail ball can move freely.

  setup_at_end();
}

void
World::ball_setup_6()
{
  // Render what will become a parachute, probably.

  last_setup = 6;
  objects_erase();

  pCoor center_pos = pCoor(17.2,14.8f,-10.2) + 2*vec_rand();

  const int n_balls = 4 * ( 4 + random()%4 );
  float ring_outer_radius = 5 + 2 * drand48();
  float ring_inner_radius = ring_outer_radius * ( 0.1 + drand48() * 0.4 );
  float ball_radius = 0.2 * 2 * numbers::pi * ring_outer_radius / n_balls;

  pNorm c_to_eye(center_pos,eye_location);
  pNorm az = c_to_eye + 0.5 * vec_rand() + pVect(0,0.5,0);

  pNorm ax = fabs(az.x) < fabs(az.y)
    ? pVect(0,az.z,-az.y) : pVect(az.z,0,-az.x);
  pVect ay(az,ax);

  // Default settings for balls.
  //
  Ball ballo_default;
  ballo_default.specularity = 0.5;
  ballo_default.velocity = pVect(0,0,0);
  ballo_default.radius = ball_radius;
  ballo_default.color = color_light_slate_gray;
  Ball balli_default = ballo_default;
  balli_default.radius = ballo_default.radius * 0.6;

  const float d_theta = 2 * numbers::pi / n_balls;

  const float link_stiffness = 10;

  vector<Ball*> balls_outer, balls_inner;

  // Place balls in a circle.
  //
  for ( int i=0; i<n_balls; i++ )
    {
      const float theta = d_theta * i;
      pVect pos_n = cosf(theta) * ax + sinf(theta) * ay;
      pCoor pos_o = center_pos + ring_outer_radius * pos_n;
      pCoor pos_i = center_pos + ring_inner_radius * pos_n;

      // Construct new ball and add it to the ball list.
      //
      Ball* ball_o = balls_outer.emplace_back( new Ball(pos_o,ballo_default) );
      Ball* ball_i = balls_inner.emplace_back( new Ball(pos_i,balli_default) );
      Link *link =
        new Link( ball_o, ball_i,
                  { .spring_constant = opt_spring_constant/200 } );
      links.push_back(link);
```

```
      // Set color of first ball to blue, others to white.
      ball_i->color = ball_o->color =
        i == 0 ? color_blue :
        theta <= numbers::pi/2 ? color_lsu_spirit_gold :
        color_white;
    }

  // Connect each ball to its neighbor with a link.
  //
  for ( auto bc: { &balls_outer, &balls_inner } )
    for ( int i=0; i<n_balls; i++ )
      {
        balls.push_back( bc[0][i] );
        links += link_new( bc[0][i], bc[0][(i+1)%n_balls], link_stiffness );
      }

  for ( int i=0; i<n_balls; i++ )
    {
      const int j = ( i + 1 ) % n_balls;
      Ball* b00 = balls_inner[i];
      Ball* b01 = balls_outer[i];
      Ball* b10 = balls_inner[j];
      Ball* b11 = balls_outer[j];
      surfaces.emplace_back(b00,b01,b11,color_light_gray);
      surfaces.emplace_back(b10,b00,b11,color_light_gray);
    }

  pNorm spin_axis = az;
  pCoor c_of_mass = center_pos;
  float omega_init = opt_omega * 0.1;

  if ( 0 )
  for ( Ball* ball: balls )
    {
      pVect cm_to_ball( c_of_mass, ball->position );

      ball->velocity = omega_init * cross( spin_axis, cm_to_ball );
      ball->omega = omega_init * spin_axis;
    }

  tail_ball = balls.back();
  head_ball = balls.front();

  opt_head_lock = false; // If true, head ball locked in place.
  opt_tail_lock = false;

  setup_at_end();
}

void
World::ball_setup_7( bool randomize = true )
{
  //
  last_setup = 7;

  // Remove objects from the simulated objects lists, balls and links.
  // The delete operator is used on objects in the lists.
  //
  objects_erase();

  int n_balls = 70 + ( randomize ? irand(-20,20) : 0 );
  pCoor axis_bottom(17.2,5,-10.2);
  float r = randomize ? frand(3,5) : 4;
  pNorm dir = randomize ? 0.3 * vec_randc() + pVect(0,1,0) : pVect(.2,1,.4);
  pCoor axis_top = axis_bottom + 14 * dir;

  spring_make(axis_top,axis_bottom,r,n_balls);
```

```
  head_ball = balls[0];
  tail_ball = balls.back();

  // Don't look straight up (or straight down).
  pNorm eye_x( -eye_direction.z, 0, eye_direction.x );

  pVect spacing = 3*r * eye_x;

  pCoor axis_top2 = axis_top + spacing;
  pCoor axis_bottom2 = axis_bottom + spacing;

  hw02_prob1(axis_top2, axis_bottom2, r, n_balls );


  pCoor helix_pos_start = axis_top2 + spacing;
  pCoor ring_center = helix_pos_start - ( randomize ? frand(3,5) : 4 ) * dir;
  float r_helix = r / ( randomize ? frand(3,5) : 4.0f );
  pVect to_eye( ring_center, eye_location );
  pVect de = cross(dir,to_eye);
  pVect helix_vx = r_helix * pNorm( cross(de,dir) );
  assert( dot( helix_vx, to_eye ) > 0 );
  hw02_prob2( helix_pos_start, ring_center, helix_vx,
              250 + ( randomize ? irand(-50,50) : 0 ) );

  opt_head_lock = true;
  opt_tail_lock = false;

  setup_at_end();
}



void
World::setup_at_end()
{
  balls.take(balls_marker);
  lock_update();
  time_step_count = 0;
  world_time = 0;
  world_time_link_update = -1;
  for ( uint i=0; i<balls.size(); i++ ) balls[i]->idx = i;
  for ( Link *l : links )
    { l->ball1_idx = l->ball1->idx; l->ball2_idx = l->ball2->idx; }
  for ( auto& s: surfaces )
    { s.b0_idx = s.b0->idx;  s.b1_idx = s.b1->idx;  s.b2_idx = s.b2->idx; }
}


 /// Advance Simulation State by delta_t Seconds
//
void
World::time_step_cpu(double delta_t)
{
  const int n_balls = balls;
  const int n_links = links;

  // Smoothly move ball in response to user input.
  //
  if ( adj_t_stop )
    {
      const double dt = min(world_time,adj_t_stop) - adj_t_prev;
      pVect adj = dt/adj_duration * adj_vector;
      balls_translate(adj,0);
      adj_t_prev = world_time;
      if ( world_time >= adj_t_stop ) adj_t_stop = 0;
    }
```

```
#pragma omp parallel for
  for ( int bi=0; bi<n_balls; bi++ )
    {
      Ball* const ball = balls[bi];
      assert( ball->fdt_to_do );
      ball->force = ball->mass * gravity_accel;
      ball->torque = pVect(0,0,0);
    }

  const int n_surfaces = surfaces.size();
  for ( int i=0; i<n_surfaces; i++ )
    {
      Surface& s = surfaces[i];
      Ball& b0 = *s.b0;
      Ball& b1 = *s.b1;
      Ball& b2 = *s.b2;
      pNorm snorm = cross(b0.position,b1.position,b2.position);
      pVect vel = (1.0/3) * ( b0.velocity + b1.velocity + b2.velocity );
      float speed = dot(snorm,vel);
      float fmag =
        -speed * fabs(speed) * snorm.magnitude * 0.5f * opt_air_resistance;
      pVect f = fmag * snorm;
      b0.force += f;
      b1.force += f;
      b2.force += f;
    }

#pragma omp parallel for
  for ( int i=0; i<n_links; i++ )
    {
      Link* const link = links[i];
      if ( !link->is_simulatable ) continue;

      //
      // Spring Force from Neighbor Balls
      //

      Ball* const ball1 = link->ball1;
      Ball* const ball2 = link->ball2;

      /// Notes:
      //
      // link->cb1
      //   Coordinate where link touches ball1, in ball1's local space.
      //
      // ball1->omatrix
      //   Ball's orientation transformation ..
      //   .. which transforms vectors in ball1's local space to object space.
      //

      // Find position and velocity of the point where the link touches
      // the surface of ball 1 ...
      //
      pVect dir1 = ball1->omatrix * link->cb1;
      pCoor pos1 = ball1->position + dir1;
      pVect vel1 = ball1->velocity + ball1->point_rot_vel(dir1);

      // ... and ball 2.
      //
      pVect dir2 = ball2->omatrix * link->cb2;
      pCoor pos2 = ball2->position + dir2;
      pVect vel2 = ball2->velocity + ball2->point_rot_vel(dir2);

      // Construct a normalized (Unit) Vector from ball to neighbor
      // based on link connection points and ball centers.
      //
      pNorm link_dir(pos1,pos2);
      pNorm c_to_c(ball1->position,ball2->position);
```

```
      const float link_length = link_dir.magnitude;

      // Compute the speed of ball's end of link towards neighbor's end of link.
      //
      pVect delta_v = vel2 - vel1;
      float delta_s = dot( delta_v, link_dir );

      // Compute by how much the spring is stretched (positive value)
      // or compressed (negative value).
      //
      const float spring_stretch = link_length - link->distance_relaxed;

      // Determine whether spring is gaining energy (whether its length
      // is getting further from its relaxed length).
      //
      const bool gaining_e = ( delta_s > 0.0 ) == ( spring_stretch > 0 );

      const float blend_length_factor = 0.1; // Just a guess.
      const float blend_s = blend_length_factor * c_to_c.magnitude;
      const float blend = min( 1.0f, fabs(delta_s) / blend_s );

      // Use a smaller spring constant when spring is loosing energy,
      // a quick and dirty way of simulating energy loss due to spring
      // friction.
      //
      const float sc_base = link->spring_constant ?: opt_spring_constant;
      const float spring_constant =
        gaining_e ? sc_base :
        sc_base * ( 1.0f - blend + blend * opt_spring_damp );

      const float force_mag = spring_constant * spring_stretch;
      pVect spring_force_12 = force_mag * link_dir;

      // Apply forces affecting linear momentum.
      //
      link->spring_force_12 = spring_force_12;

      if ( ! link->is_surface_connection ) continue;

      // Apply torque.  Why doesn't the ball radius matter?
      //
      link->torque1 = cross(pNorm(dir1), spring_force_12);
      link->torque2 = cross(pNorm(dir2), spring_force_12);
    }

  // Note: Because two links can reference the same ball this should
  // not be done in parallel.
  for ( Link *link: links )
    {
      if ( !link->is_simulatable ) continue;
      Ball* const ball1 = link->ball1;
      Ball* const ball2 = link->ball2;
      ball1->force += link->spring_force_12;
      ball2->force -= link->spring_force_12;
      ball1->torque += link->torque1;
      ball2->torque -= link->torque2;
    }

  ///
  /// Update Position of Each Ball
  ///

#pragma omp parallel for
  for ( int bi=0; bi<n_balls; bi++ )
    {
      Ball* const ball = balls[bi];
      if ( ball->locked )
```

```
  {
    ball->velocity = pVect(0,0,0);
    ball->omega = pVect(0,0,0);
    continue;
  }

// Update Velocity
//
// This code assumes that force on ball is constant over time
// step. This is clearly wrong when balls are moving with
// respect to each other because the springs are changing
// length. This inaccuracy will make the simulation unstable
// when spring constant is large for the time step.
//
const float mass = max( ball->mass, ball->mass_min );

pVect delta_v = ( ball->force / mass ) * delta_t;
const float r = ball->radius;

if ( platform_collision_possible(ball->position)
     && fabs(ball->position.y) < r )
  {
    const bool above = ball->position.y >= 0;
    const float spring_constant_plat =
      ball->velocity.y < 0 == above ? 100000 : 50000;
    const float fric_coefficient = opt_friction_dynamic;
    const float force_away = ( above ? 1 : -1 )
      * ( r - fabs(ball->position.y) ) * spring_constant_plat;
    const float delta_v_away = force_away / mass * delta_t;
    const float fric_force_mag = fric_coefficient * force_away;
    pNorm surface_v(ball->velocity.x,0,ball->velocity.z);
    const float delta_v_surf = fric_force_mag / mass * delta_t;

    if ( delta_v_surf > surface_v.magnitude )
      {
        // Ignoring other forces?
        delta_v =
          pVect(-ball->velocity.x,delta_v.y,-ball->velocity.z);
      }
    else
      {
        delta_v -= delta_v_surf * surface_v;
      }
    delta_v.y += delta_v_away;
  }

ball->velocity += delta_v;

// Air Resistance
//
const pNorm ball_dir(ball->velocity);
const float delta_s_air_raw =
  opt_air_resistance * ball_dir.mag_sq * delta_t;
const float delta_s_air = min( ball_dir.magnitude, delta_s_air_raw );
ball->velocity -= delta_s_air * ball_dir;

// Update Position
//
// Assume that velocity is constant.
//
ball->position += ball->velocity * delta_t;

ball->omega += delta_t * ball->fdt_to_do * ball->torque;

pNorm axis(ball->omega);

// Update Orientation
//
```

```
      // If ball isn't spinning fast skip expensive rotation.
      //
      if ( axis.mag_sq < 0.000001 ) continue;

      // Update ball orientation.
      //
      ball->orientation_set
        ( pQuat(axis,delta_t * axis.magnitude) * ball->orientation );
    }
}

void
World::energy_compute()
{
  const int n_balls = balls;
  const int n_links = links;

  energy = 0;

#pragma omp parallel for reduction(+:energy)
  for ( int bi=0; bi<n_balls; bi++ )
    {
      Ball* const ball = balls[bi];
      if ( ball->locked ) continue;

      // Kinetic Energy
      energy += 0.5 * ball->velocity.mag_sq() * ball->mass;

      // Rotational Energy
      energy += 0.2 * ball->radius_sq * ball->mass * ball->omega.mag_sq();

      // Potential Energy
      if ( opt_gravity )
        energy += ball->position.y * ball->mass * opt_gravity_accel;
    }

#pragma omp parallel for reduction(+:energy)
  for ( int i=0; i<n_links; i++ )
    {
      Link* const link = links[i];
      if ( !link->is_simulatable ) continue;

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

      // Find position and velocity of the point where the link touches
      // the surface of ball 1 ...
      //
      pVect dir1 = ball1->omatrix * link->cb1;
      pCoor pos1 = ball1->position + dir1;

      // ... and ball 2.
      //
      pVect dir2 = ball2->omatrix * link->cb2;
      pCoor pos2 = ball2->position + dir2;

      // Construct a normalized (Unit) Vector from ball to neighbor
      // based on link connection points and ball centers.
      //
      pNorm link_dir(pos1,pos2);
      const float link_length = link_dir.magnitude;
      const float spring_stretch = link_length - link->distance_relaxed;
      const float sc_base = link->spring_constant ?: opt_spring_constant;

      energy += 0.5 * spring_stretch * spring_stretch * sc_base;
```

```
    }
}

bool
World::platform_collision_possible(pCoor pos)
{
  // Assuming no motion in x or z axes.
  //
  return pos.x >= platform_xmin && pos.x <= platform_xmax
    && pos.z >= platform_zmin && pos.z <= platform_zmax;
}

 /// External Modifications to State
//
//   These allow the user to play with state while simulation
//   running.

// Move the ball.
//
void Ball::translate(pVect amt) {position += amt;}

// Add velocity to the ball.
//
void Ball::push(pVect amt) {velocity += amt;}

// Set the velocity to zero.
//
void Ball::stop() {velocity = pVect(0,0,0); }

// Set the velocity and rotation (not yet supported) to zero.
//
void Ball::freeze() {velocity = pVect(0,0,0); }



void World::balls_translate(pVect amt,int b){head_ball->translate(amt);}
void World::balls_push(pVect amt,int b){head_ball->push(amt);}
void World::balls_translate(pVect amt)
{ for ( Ball *ball: balls ) ball->translate(amt); }
void World::balls_push(pVect amt)
{ for ( Ball *ball: balls ) ball->push(amt); }
void World::balls_stop()
{ for ( Ball *ball: balls ) ball->stop(); }
void World::balls_freeze(){balls_stop();}


void
World::render_link_start()
{
  if ( world_time_link_update == world_time ) return;

  // Empty the containers in preparation for a new pass.
  for ( auto b: lis_bufs ) b->clear();
}

void
World::render_link_gather(Link *link)
{
  // Place data needed to render link (the argument to this function)
  // in arrays. Just before rendering those arrays will be placed in
  // buffer objects.


  // If we've already collected links, just return.  This happens
  // when doing multiple passes to render shadows and reflections.
  //
  if ( world_time_link_update == world_time ) return;
```

```
  Ball *const ball1 = link->ball1;
  Ball *const ball2 = link->ball2;

  // Direction of link relative to center off ball.
  //
  pVect dir1 = ball1->omatrix * link->cb1;

  // Place on surface of ball where link connects.
  //
  pCoor pos1 = ball1->position + dir1;

  // Direction and connection location for ball 2.
  //
  pVect dir2 = ball2->omatrix * link->cb2;
  pCoor pos2 = ball2->position + dir2;

  pVect p1p2(pos1,pos2);
  pNorm p1p2n(p1p2);

  // Radius of link.
  //
  const float rad = link->radius ?: ball1->radius * 0.3;

  // Compute scale factors for texture.
  const float tex_aspect_ratio = 8.5 / 11;  // Yes, inches. Can't remember mm.
  const float page_width_o = 2.5;  // Width of texture in object-space coords.
  const float tex_t_scale = link->distance_relaxed / page_width_o;
  const float tex_angle_scale = rad * tex_aspect_ratio / page_width_o;

  pNorm dirn1(dir1);
  pNorm dirn2(dir2);

  // Vectors used to describe the cubic Hermite curve.
  //
  pVect4 v1 = p1p2n.magnitude * dirn1;
  pVect4 v2 = p1p2n.magnitude * dirn2;

  // Convert link's local x and y axes to global coordinates.
  //
  pVect4 b1_ydir = ball1->omatrix * link->b1_dir;
  pVect4 b2_ydir = ball2->omatrix * link->b2_dir;


  // Place data needed to render links in lists. Note that
  // the += operator appends the right-hand side to the container
  // on the left-hand side.
  //
  link->idx = lis_pos1.size();
  lis_pos1 << pos1;
  lis_pos2 << pos2;
  lis_v1 << v1;
  lis_v2 << v2;
  lis_b1_ydir << b1_ydir;
  lis_b2_ydir << b2_ydir;
  lis_tex_rad << vec4( tex_t_scale, tex_angle_scale, rad, 0);
}



void
World::render_link_render(vk::CommandBuffer& cb, bool shadow_volumes)
{
  // Perform a rendering pass to render all the links collected
  // by the routine render_link_2.


  // Check whether this is the first time we are rendering this data.
```

```
//
const bool first_render =
  world_time_link_update != world_time
  || opt_segments != buf_uni_common->links_n_segs
  || opt_sides != buf_uni_common->sides;
world_time_link_update = world_time;
const size_t n_instances = lis_pos1.size();
const bool do_rt = vh.raytrace_active();
const int segments = shadow_volumes ? max(2,opt_segments/4) : opt_segments;
const int segments_sv = max(2,opt_segments/4);

if ( !n_instances ) return;

if ( first_render )
  {
    for ( auto b: lis_bufs ) b->to_dev();
    buf_uni_common->sides = opt_sides;
    buf_uni_common->links_n_segs_sv = segments_sv;
    buf_uni_common->delta_tee_sv = 1.0f / segments_sv;
    buf_uni_common->links_n_segs = opt_segments;
    buf_uni_common->delta_tee = 1.0f / opt_segments;
  }

if ( do_rt && first_render )
  {
    const int n_tri_p_segment = 2 * opt_sides;
    const int n_tri_p_link = n_tri_p_segment * opt_segments;
    const int n_tri_p_pass = n_tri_p_link * n_instances;

    const int n_vtx_p_segment = 2 * ( opt_sides + 1 );
    const int n_vtx_p_link = n_vtx_p_segment * opt_segments;
    const int n_vtx_p_pass = n_vtx_p_link * n_instances;

    if ( !rt_geo_links )
      rt_geo_links
        .init( raytrace )
        .color_set( color_link, color_salmon )
        .bind( VR_BUF_IDX_sampler, sampler, texid_hw )
        .grouping_set( RT_Geometry_Indexed );

    buf_links_as_idx.assure(3*n_tri_p_pass);
    buf_links_tex_coor.assure(n_vtx_p_pass);
    buf_links_vertex_g.assure(n_vtx_p_pass);
    buf_links_normal_g.assure(n_vtx_p_pass);

    rt_geo_links
      .bind( VR_BUF_IDX_indices, buf_links_as_idx, 3*n_tri_p_pass )
      .bind( VR_BUF_IDX_pos, buf_links_vertex_g, n_vtx_p_pass )
      .bind( VR_BUF_IDX_normal, buf_links_normal_g )
      .bind( VR_BUF_IDX_tcoor, buf_links_tex_coor );
  }

if ( do_rt )
  {
    QSet& qs = vh.qs;
    buf_uni_common.to_dev();
    qs.cb.begin( { vk::CommandBufferUsageFlagBits::eOneTimeSubmit } );
    comp_links.record_dispatch(qs.cb,opt_segments,n_instances);
    qs.cb.end();
    qs.q.submit( vk::SubmitInfo{ 0, nullptr, nullptr, 1, &qs.cb }, nullptr );
    qs.q.waitIdle();
    return;
  }


VPipeline& pipe = shadow_volumes ? pipe_set_2_sv : pipe_set_2;

pipe.record_draw_instanced(cb, segments+1, n_instances);
```

```
}


void
World::frame_callback()
{
  // This routine called whenever window needs to be updated, but
  // before commands are recorded.

  // Get any waiting keyboard commands.
  //
  cb_keyboard();

  if ( vh.have_raytrace ) vh.opt_want_raytrace_now = opt_want_raytrace;

  // This is a hack for 2023 HW 2.
  while ( !balls.empty() && balls.back()->world_time_expire <= world_time )
    balls.pop_back();
  //
  // Balls must be last in list and not part of a link.

  const double time_now = time_wall_fp();
  const int time_step_count_start = time_step_count;
  const int nmp = gpu_info.cuda_prop.multiProcessorCount;
  int bl_p_sm_max = 1024;
  for ( int i=0; i<gpu_info.num_kernels; i++ )
    set_min
      (bl_p_sm_max,
       gpu_info.get_max_active_blocks_per_mp(i, opt_block_size, 0) );
  const int grid_size = bl_p_sm_max * nmp;

  const bool opt_cuda = opt_physics_method != GP_cpu;
  if ( opt_cuda )
    {
      CE( cudaEventRecord(frame_start_ce,0) );
      data_cpu_to_gpu_constants();
      data_cpu_to_gpu_dynamic();
    }

  if ( !opt_pause || opt_single_frame || opt_single_time_step )
    {
      /// Advance simulation state.

      // Amount of time since the user saw the last frame.
      //
      const double wall_delta_t = time_now - last_frame_wall_time;

      // Compute amount by which to advance simulation state for this frame.
      //
      const double duration =
        vh.video_frame_duration() ?:
        opt_single_time_step ? opt_single_time_step * opt_time_step_duration :
        opt_single_frame ? 1/60.0 :
        wall_delta_t;

      const double world_time_target = world_time + duration;
      const double wall_time_limit = time_now + 0.05;

      while ( world_time < world_time_target )
        {
          if ( opt_cuda )
            {
              launch_time_step(opt_time_step_duration,grid_size,opt_block_size);
            }
          else
            {
              time_step_cpu(opt_time_step_duration);
            }
```

```
          time_step_count++;
          world_time += opt_time_step_duration;
          const double time_right_now = time_wall_fp();
          if ( time_right_now > wall_time_limit ) break;
        }

      // Reset these, just in case they were set.
      //
      opt_single_frame = false;
      opt_single_time_step = 0;
    }

  if ( opt_cuda )
    {
      data_gpu_to_cpu_dynamic();
      CE( cudaEventRecord(frame_stop_ce,0) );
    }

  frame_timer.phys_end();
  frame_timer.work_amt_set(time_step_count-time_step_count_start);

  energy_compute();

  float cuda_time = 0;
  if ( opt_cuda )
    {
      CE(cudaEventSynchronize(frame_stop_ce));
      CE(cudaEventElapsedTime(&cuda_time,frame_start_ce,frame_stop_ce));
    }
  frame_timer.cuda_frame_time_set(cuda_time);

  last_frame_wall_time = time_now;

}

int
main(int argv, char **argc)
{
  pVulkan_Helper pvulkan_helper(argv,argc);
  World world(pvulkan_helper,argv,argc);

  world.run();
}
```

```
/// LSU EE 4702-X (Fall 2026), GPU Programming
//
 /// Homework 3 -- SOLUTION
 //
 //  Assignment: https://www.ece.lsu.edu/gpup/2026/hw03.pdf
 //
 /// Modified this file only.
 //
 //  Search for SOLUTION to find the places where the solution
 //  appears.
 //
 // Based on cpu-only/demo-06-rend-pipe.cc.

//
 /// Rendering Pipeline
//

#include <ranges>
#include <gp/coord.h>
#include <gp/colors.h>
#include "frame_buffer.h"

 ///
 /// Our_3D: Low-Level 3D Graphics Library.
 ///

enum TopologyE { Topology_Individual, Topology_Strip, Topology_Ring,
  Topology_ENUM_SIZE };

typedef int Topology;

class Our_3D {
public:
  Our_3D(pFrame_Buffer& fbp)
    :frame_buffer(fbp)
  {
    vtx_colors = nullptr;
    vtx_coors = nullptr;
    vtx_normals = nullptr;
    topology = Topology_Individual;
    opt_tri_normals_always = false;
  };

  Our_3D& transform_eye_from_object_set(const pMatrix& m)
  {
    eye_from_object = m;
    return *this;
  }
  Our_3D& transform_clip_from_eye_set(const pMatrix& m)
  {
    clip_from_eye = m;
    return *this;
  }
```

```
Our_3D& light_location_set(const pCoor c)
{
  light_location = c;
  return *this;
}
Our_3D& light_color_set(const pColor c)
{
  light_color = c;
  return *this;
}

Our_3D& topology_set( Topology topo = Topology_Individual )
{
  topology = topo;
  return *this;
}

/// SOLUTION -- Problem 2
//
//  Provide a way of specifying a number of vertices. This number
//  is used for Topology_Ring and ignored for other topologies.
//
Our_3D& topology_set( Topology topo, size_t n_vtx )
{
  topology = topo;
  ring_n_elt = n_vtx;
  return *this;
}


Our_3D& vtx_normals_set()
{
  vtx_normals = nullptr;
  return *this;
}

Our_3D& vtx_normals_set(vector<pVect4>& c)
{
  vtx_normals = c.data();
  return *this;
}

Our_3D& vtx_coors_set(vector<pCoor>& c)
{
  vtx_coors = &c;
  return *this;
}

Our_3D& vtx_colors_set(vector<pColor>& c)
{
  vtx_colors = &c;
  return *this;
}
```

```
  Our_3D& rendering_stats_reset()
  {
    stats_n_triangles = stats_n_vertices = stats_n_fragments = 0;
    return *this;
  }

  Our_3D& draw_rasterization();

  pFrame_Buffer& frame_buffer;
  pMatrix eye_from_object, clip_from_eye;
  pCoor light_location;
  pColor light_color;
  vector<pCoor> *vtx_coors;
  pVect4 *vtx_normals;
  vector<pColor> *vtx_colors;
  Topology topology;

  int topology_n_vertices;
  int64_t stats_n_triangles, stats_n_vertices, stats_n_fragments;

  bool opt_tri_normals_always;
  bool opt_tryout1, opt_tryout2;

  /// SOLUTION -- Problem 2
  //
  size_t ring_n_elt;
};


 ///
 ///   World: An application.
 ///
class World : public World_Base {
public:
  World(pFrame_Buffer& fb):World_Base(fb),gc(fb){}
  Our_3D gc;

  // Called each time a frame needs to be rendered.
  virtual void render(bool first_time)
  {
    if ( first_time ) init_scene();
    render_scene();
  }

  // Called Once
  void init_scene();

  // Called for each frame.
  void render_scene();

  pCoor eye_location, light_location;
  pColor light_color;
  pCoor *adj_location;
  pVect eye_direction;
```

```
  bool opt_tryout1,opt_tryout2;

  Topology opt_hw3_topology;

};

void
World::init_scene()
{
  eye_location = pCoor(-1.2,.5,10.2);
  light_location = pCoor(-2.2,1.5,10.2);
  eye_direction = pVect(.1,0,-1);
  adj_location = &eye_location;
  opt_tryout1 = opt_tryout2 = false;
  gc.opt_tryout1 = opt_tryout1;
  gc.opt_tryout2 = opt_tryout2;
  light_color = color_white * 100;

  opt_hw3_topology = Topology_Strip;
}

void
World::render_scene()
{
  pFrame_Buffer& frame_buffer = gc.frame_buffer;

  switch ( frame_buffer.keyboard_key ) {
  case FB_KEY_LEFT: adj_location->x -= 0.1; break;
  case FB_KEY_RIGHT: adj_location->x += 0.1; break;
  case FB_KEY_PAGE_UP: adj_location->y += 0.1; break;
  case FB_KEY_PAGE_DOWN: adj_location->y -= 0.1; break;
  case FB_KEY_UP: adj_location->z -= 0.1; break;
  case FB_KEY_DOWN: adj_location->z += 0.1; break;

  case '1':
    opt_hw3_topology++;
    if ( opt_hw3_topology >= Topology_ENUM_SIZE ) opt_hw3_topology = 0;
    break;
  case '!':
    if ( opt_hw3_topology == 0 ) opt_hw3_topology = Topology_ENUM_SIZE;
    opt_hw3_topology--;
    break;

  case 'e': case 'E': adj_location = &eye_location; break;
  case 'l': case 'L': adj_location = &light_location; break;
  case 'v': case 'V': gc.opt_tri_normals_always = !gc.opt_tri_normals_always;
    break;
  case 'y': gc.opt_tryout1 = opt_tryout1 = !opt_tryout1; break;
  case 'Y': gc.opt_tryout2 = opt_tryout2 = !opt_tryout2; break;
  case '+': case '=':
    if ( !frame_buffer.keyboard_control ) light_color *= 1.1;
    break;
  case '-':
    if ( !frame_buffer.keyboard_control ) light_color *= 1/1.1;
```

```
  break;

}

gc.frame_buffer.fbprintf
  ( "Light Location [ %.2f, %.2f, %.2f ],  Light Intensity %.2f ('+-')  "
    "Eye Location [ %.2f, %.2f, %.2f ]\n",
    light_location.x, light_location.y, light_location.z,
    light_color.r,
    eye_location.x, eye_location.y, eye_location.z );
frame_buffer.fbprintf
  ("HW3 Topo: %s ('1')  Normals: %s ('v')  "
   "Tryout1 %s ('y')  Tryout2 %s ('Y')\n",
   opt_hw3_topology == Topology_Individual ? "I-TRIS" :
   opt_hw3_topology == Topology_Strip ? "STRIP" :
   opt_hw3_topology == Topology_Ring ? "RING " : "?????",
   gc.opt_tri_normals_always ? "TRIANGLE" : "VERTEX  ",
   opt_tryout1 ? "ON " : "OFF", opt_tryout2 ? "ON " : "OFF");

gc.light_location_set( light_location ).light_color_set( light_color );

///
/// Prepare Transformations
///

/// Specifications of User's Monitor
//
//  First, the width of the user's monitor in object space coordinates.
//
const float width_m = 1.6;  // Wow, that's a big monitor! (m is for meter.)
const float qn = 1;         // Distance from eye to monitor along z axis.
//
//  To determine the height of the user's monitor we need to know the
//  aspect ratio of the window we are painting.
//
const uint win_width = frame_buffer.width_get();
const uint win_height = frame_buffer.height_get();
//
//  win_width and win_height are pixel-space coordinates.
//
const float aspect = 1.0 * win_width / win_height;
//
//  Using the aspect ratio and the width we can compute the height.
//
const float height_m = width_m / aspect;
//
//  width_m and height_m are in object-space coordinates.

// Compute transformation from Object Space to Eye Space ..
//
pMatrix_Translate center_eye(-eye_location);
pMatrix_Rotation rotate_eye(eye_direction,pVect(0,0,-1));
pMatrix eye_from_object = rotate_eye * center_eye;
//
```

```
// .. and give the transformation to Our3D:
//
gc.transform_eye_from_object_set(eye_from_object);

// Compute transformation from Eye Space to Clip Space ..
//
pMatrix_Frustum clip_from_eye
  ( -width_m/2, width_m/2,  -height_m/2, height_m/2,  qn, 5000 );
//   l          r           b            t            n   f
//   The letters l,r,b,t,n,f are from a frustum transform definition.
//
// .. and give the transformation to Our3D:
//
gc.transform_clip_from_eye_set(clip_from_eye);


///
/// Prepare Triangles
///

// List of coordinates in object space and list of colors.
//
vector<pCoor> coors_os;
vector<pColor> colors;

// Insert First Triangle
//
// Insert coordinates of a triangle into coordinate list,
// and insert its color into a color list.
//
coors_os << pCoor( 0, 0, 0 ) << pCoor( 9, 6, -9 ) << pCoor( 0, 7, -5 );
colors << color_white << color_white << color_white;

// Insert Second Triangle
//
// Add coordinates and color of another triangle
//
coors_os << pCoor(7,4,-2) << pCoor(-3,5,-9) << pCoor(9,2,-2);
colors << color_blue << color_blue << color_blue;

// Insert Two More Triangles
//
// Add a square consisting of a gold and multi-colored triangle.
//
colors << color_lsu_spirit_gold << color_lsu_spirit_gold
       << color_lsu_spirit_gold;
coors_os << pCoor(-7,0,-2) << pCoor(-7,2,-2) << pCoor(-5,0,-2);

colors   << color_red      << color_blue     << color_green;
coors_os << pCoor(-7,2,-2) << pCoor(-5,2,-2) << pCoor(-5,0,-2);

// Insert a triangle at light location, facing user. Light is top point.
pVect light_to_eye( light_location, eye_location );
float l_sz = 0.1;
```

```
pVect lax = cross( pVect( 0, 1, 0 ), light_to_eye ).normal() * l_sz;
pVect lay = cross( light_to_eye, lax ).normal() * l_sz;
coors_os << light_location
         << light_location - lax - lay << light_location + lax - lay;
colors << color_yellow << color_yellow << color_yellow;

///
/// Send Triangles to Our3D for Rendering
///
//
gc.vtx_normals_set();            // Tell Our3D to compute triangle normals.
gc.topology_set(Topology_Individual);  // Use individual triangle grouping.
gc.vtx_coors_set(coors_os);      // Coordinates to render.
gc.vtx_colors_set(colors);       // Colors to render.
gc.draw_rasterization();         // Tell Our3D to render now.

///
/// Render a Helix Tube
///
//  A helix tube is a cylinder with its axis following a helix.
//  Could be called a helixial cylinder.

// First remove the coordinates and colors inserted above.
//
coors_os.clear(); colors.clear();
gc.rendering_stats_reset();

// Container for vertex normals.
vector<pVect4> normals_os;

// Specifications of Helix Tube
//
pCoor helix_start(-4,-1,-3);

//
/// Compute vertices of a ring.
//
// The ring center is at the origin, and its normal is along z axis.
//
int n_ring_vtx = 20;  // Number of vertices in ring.
float delta_theta = 2 * numbers::pi / n_ring_vtx;
float tube_radius = 0.75;  // Tube radius is the ring radius.

vector<pCoor> coors_ring;
vector<pVect4> norms_ring;
vector<pColor> colors_ring;

// Local coordinates of ring.
pVect gx(1,0,0), gy(0,1,0), gz(0,0,1);

for ( int i=0; i<n_ring_vtx; i++ )
  {
    float theta = i * delta_theta + numbers::pi/2;
    pVect4 n = gx * cosf(theta) + gy * sinf(theta);
```

```
    pCoor c = tube_radius * n;
    coors_ring << c;
    norms_ring << n;
    colors_ring << ( i ? color_cyan : color_yellow );
  }

// For convenience, append a copy of first vertex, so that the first
// and last vertex in the lists are the same.
//
if ( opt_hw3_topology != Topology_Ring )
  {
    coors_ring << coors_ring[0];
    norms_ring << norms_ring[0];
    colors_ring << colors_ring.back();
  }

//
/// Compute Vertices of Helix Tube
//
int n_ssegs = 80;
float s_delta_eta = 7 * numbers::pi / n_ssegs;
float s_ht = 3;
float s_wd = 10;
float s_delta_x = s_wd / n_ssegs;
int n_rev = 10;
float delta_rho = n_rev * 2 * numbers::pi / n_ssegs;

vector <pCoor> tr_co_prev;
vector <pVect4> tr_n_prev;
pNorm helix_ax = gz, helix_ay = gy, helix_az = gx;
for ( int i=0; i<=n_ssegs; i++ )
  {
    // Compute coordinates of vertices at position i along the helix.
    //
    // This is done by computing transformation matrices and using
    // them to transform coordinates in coors_ring and normals_ring.
    //
    // The transformed coordinates and matrices are placed in
    // tr_co_curr and tr_n_curr.
    //
    // The transformed coordinates and matrices from the previous
    // iteration are in tr_co_prev and tr_n_prev.
    //
    // Finally, insert coordinates, normals, and colors into the
    // containers passed to Our_3D: coors_os, normals_os, colors.
    //
    // The order in which coordinates, normals, and colors are
    // inserted into coors_os, normals_os, and colors is determined
    // by opt_hw3_topology, which is set to the topology that Our_3D
    // expects.


    float eta = i * s_delta_eta;
    pVect tube_ay = cosf(eta) * helix_ax + sinf(eta) * helix_ay;
```

```
// Coordinate of point on the helix.
pCoor tube_axis = helix_start + s_ht * tube_ay + i * s_delta_x * helix_az;

// Vector pointing in direction of helix at this point. (The
// derivative with respect to i of the parametric formula for
// tube_axis.)
//
pNorm tube_az =
  -s_ht * s_delta_eta * sinf(eta) * helix_ax
  + s_ht * s_delta_eta * cosf(eta) * helix_ay
  + s_delta_x * helix_az;
pNorm tube_ax = cross(tube_ay,tube_az);

// Rotate ring to point on helix.
pMatrix_Cols srot(tube_ax,tube_ay,tube_az);

// Twist ring (so that yellow strip forms a spiral).
pMatrix_Rotation arot( gz, i * delta_rho );

pMatrix rot = srot * arot;
pMatrix_Translate to_helix(tube_axis);
pMatrix m = to_helix * rot;

// Compute coordinates and normals of ring at this point on the helix.
//
vector <pCoor> tr_co_curr;  // Transformed Ring Coordinate CURRent.
vector <pVect4> tr_n_curr;  // Transformed Ring Normal CURRent.
for ( auto& c: coors_ring ) tr_co_curr << m * c;
for ( auto& n: norms_ring ) tr_n_curr << pVect4(rot * n);

switch ( opt_hw3_topology ) {

case Topology_Ring:
  // Vertex ordering used for Problem 2.
  //
  // Study but don't modify code in this case.

  // For the ring topology, just append the new coordinates and
  // normals to the containers. The colors don't change along
  // the helix, so just append the initial list.
  //
  // Note that for the ring topology each vertex appears once.

  // DO NOT modify code below.
  for ( auto c: tr_co_curr ) coors_os << c;
  for ( auto n: tr_n_curr ) normals_os << n;
  for ( auto c: colors_ring ) colors << c;
  // DO NOT modify code above.
  break;

case Topology_Strip:
  // DO NOT modify code for his case.
```

```
    // Insert vertices in triangle-strip order.
    //
    // tr_co_curr holds coordinates of upper ring.
    // tr_co_prev holds coordinates of lower ring.

    // Note: Loops don't execute if tr_co_prev or tr_n_prev empty.
    for ( auto [co_prev,co_curr]: views::zip( tr_co_prev, tr_co_curr ) )
      coors_os << co_prev << co_curr;

    for ( auto [ n_prev, n_curr ]: views::zip( tr_n_prev, tr_n_curr ) )
      normals_os << n_prev << n_curr;

    for ( auto [co, _]: views::zip(colors_ring,tr_n_prev) )
      colors << co << co;

    break;

  case Topology_Individual:

    // Problem 1:
    //
    // [✔] Insert vertices for individual triangle ordering.
    // [✔] Most changes should be in this case.
    //
    // The unmodified code inserts vertices in triangle-strip order.

    for ( ssize_t j = 0;  j < ssize_t(tr_co_prev.size())-1;  j++ )
      {
        // Note: j current position on the ring.
        size_t jn = j + 1;  // Compute next position on the ring.
        pColor& color = colors_ring[j];  // For convenience.

        // Problem 1 solution goes here.

        if ( !opt_tryout1 )
          {
            /// SOLUTION -- Problem 1
            //
            //  Use current vertex (j) and next vertex (jn).
            //
            //  Insert two triangles instead of one.
            //
            coors_os << tr_co_prev[j]  << tr_co_curr[j] << tr_co_prev[jn];
            coors_os << tr_co_prev[jn] << tr_co_curr[j] << tr_co_curr[jn];

            normals_os << tr_n_prev[j]  << tr_n_curr[j] << tr_n_prev[jn];
            normals_os << tr_n_prev[jn] << tr_n_curr[j] << tr_n_curr[jn];

            // Just use the color from vertex j. Our_3D will use
            // this color for the entire triangle for
            // triangle-strip ordering, so do the same for
            // individual triangles.
            //
            colors << color << color << color << color << color << color;
```

```
            }
          else
            {
              /// Original (unsolved) Code
              //
              // The code below is for a triangle strip order.
              // [ ] Remove code below when you get started.
              // Note: Left code to make it easy to compare.
              //
              coors_os   << tr_co_prev[j] << tr_co_curr[j];
              normals_os << tr_n_prev[j]  << tr_n_curr[j];
              colors << color << color;
            }

        }

      break;
    }

    // Note: Lines below not needed for the ring topology.
    swap(tr_co_prev,tr_co_curr); swap(tr_n_prev,tr_n_curr);
  }

///
/// Send Triangles to Our3D for Rendering
///
//

// Problem 2: Modify some code below.

// Set topology (vertex ordering) that Our_3D should use.
//
/// SOLUTION -- Problem 2
//
//  Specify number of vertices in ring. This is needed for the
//  ring topology and harmless in others.
//
gc.topology_set(opt_hw3_topology,coors_ring.size());

// Provide a pointer to the coordinates, normals, and colors to be used.
gc.vtx_coors_set(coors_os);
gc.vtx_normals_set(normals_os);
gc.vtx_colors_set(colors);

// Tell Our_3D to now rendering using the vertices and topologies
// specified above.
//
gc.draw_rasterization();

// Print a helpful message to the green text.
// Feel free to modify if it might help.
gc.frame_buffer.fbprintf
  ("Helix Tube: %ld vertices, %ld triangles, %ld fragments.\n",
   gc.stats_n_vertices, gc.stats_n_triangles, gc.stats_n_fragments);
```

```
}

Our_3D&
Our_3D::draw_rasterization()
{
  ///
  /// Prepare Transformations
  ///

  // Specifications of User's Monitor
  //
  const uint win_width = frame_buffer.width_get();
  const uint win_height = frame_buffer.height_get();
  //
  // win_width and win_height are pixel-space coordinates.

  // Transform from Clip to Pixel
  //
  pMatrix_Translate recenter(pVect(1,1,0));
  pMatrix_Scale scale( win_width/2, win_height/2, 1 );
  pMatrix window_from_clip = scale * recenter;
  pMatrix ws_from_os = window_from_clip * clip_from_eye * eye_from_object;

  ///
  /// Rasterize Triangle and Write Frame Buffer
  ///

  // Get pointer to the depth (z) buffer.
  //
  float* const zbuffer = frame_buffer.buffer_depth_get();
  //
  // The depth buffer uses the same index as the frame buffer.

  auto& coors_os = *vtx_coors;
  auto& colors = *vtx_colors;

  const bool topo_tris [[maybe_unused]] = topology == Topology_Individual;
  const bool topo_strip [[maybe_unused]] = topology == Topology_Strip;
  const bool topo_ring [[maybe_unused]] = topology == Topology_Ring;

  size_t start = 0;

  /// SOLUTION -- Problem 2 -- Plan
  //
  //  Consider the way vertices v0, v1,..., v299 form triangles when
  //  ring_n_elt = 100 with 300 vertices, and pardon my non-standard
  //  Roman numerals:
  //
  //  v200   v201   v202    v298   v299   v200
  //  +------+------+ ...   +------+------+
  //  |\CCII |\CCIV |       |\     |\ CCC |
  //  |  \   |  \   |       |  \   |  \   |
  //  |    \ |    \ |       |    \ |    \ |
```

```
//  | CCI \|CCIII\|       |     \|CCIC \|
//  |      |      |       |      |      |
//  v100   v101   v102    v198   v199   v100
//  +------+------+ ...   +------+------+
//  |\  II |\  IV |       |\ CIIC|\  CC |
//  |  \   |  \   |       |  \   |  \   |
//  |    \ |    \ |       |    \ |    \ |
//  | I   \| III \|       |CIIIC\| CIC \|
//  +------+------+ ...   +------+------+
//  v0     v1     v2      v98    v99    v0
//
//  In the i loop below when i=0, we emit Triangle I (v0,v1,v100),
//  when i=1, we emit Triangle II (v1,v101,v100). When i=2 we emit
//  Triangle III, and when i=3 we emit Triangle IV, and so on.
//
//  To do this the number of iterations in the i loop is 2(V-n),
//  where V is the number of vertices (coors_os.size()) and n is the
//  number of vertices in a ring (value of ring_n_elt). See the
//  initialization of variable stop_ring below.
//
//  In an i loop iteration indices i0, i1, and i2, are used for the
//  three vertices of the triangle being rendered. For strips and
//  individual triangles. they are set to i0=i, i1=i+1, i2=i+2, but
//  for rings they are not sequential.
//
//  Consider Triangle III. That's emitted for i=2. The index i0 of
//  v1 is i0=i/2=1, the index i1 of v2 is i1=i/2+1=2, and the index
//  i2 of v101 is i2=i/2+100=101. The indices i0 and i1 are computed
//  differently for Triangle IV. A special case is wrap-around, seen
//  in Triangle CIC, where v99+1 wraps around to v0. For details see
//  the solution code in the i loop.
//
//  Another special case is the color. Both triangles in a pair must
//  be the same color and it must be possible to set each pair to a
//  different color. To achieve this the color for triangles I and
//  II are both retrieved using index i/2, not index i0. Had i0 been
//  used the second triangle of a pair would be a different color.
//  It would have been convenient to use i2 as the index of the
//  color since i2 is the same for both triangles of a pair, but
//  that coloring would not match the way they are done for a
//  triangle strip here.
//
//  BTW, The correct way to represent 199 in Roman numerals is
//  CXCIX. The Romans *did* think in powers of 10 (a mile is 1000
//  paces) and for that reason each power of 10 in a Roman numeral
//  is grouped separately: C + XC + IX. Maybe some Roman did see
//  that it could be written as I * C + (IX) * X + IX * I but just
//  could not convince anyone that it was better to write numbers
//  that way. Or maybe an Indian beat him (or her) to it.


/// SOLUTION -- Problem 2
//
//  Set increment to 1 for ring ..
```

```
  //
  size_t inc = topo_tris ? 3 : 1;
  //
  //  .. and use most vertices twice for a ring.
  //
  size_t stop_ring = 2 * ( coors_os.size() - ring_n_elt ) - 1;
  size_t stop = topo_ring ? stop_ring : coors_os.size() - 3;

  stats_n_vertices += coors_os.size();

  ///
  /// Main Rendering Loop
  ///

#pragma omp parallel for reduction(+:stats_n_triangles,stats_n_fragments)
  for ( size_t i=start; i<=stop; i += inc )
    {
      // Problem 2: Some of solution goes below.

      // Compute indices of the next triangle.
      //
      size_t i0 = i,  i1 = i+1,  i2 = i+2;
      //
      // In unmodified code works for strips and individual triangles,
      // but not rings.

      /// SOLUTION -- Problem 2
      //
      //  Set indices as described in the Plan above.
      //
      if ( topo_ring )
        {
          // If true, we are at the lower-left triangle of the pair.
          // See diagram in Plan, above.
          //
          bool triLL = ( i & 1 ) == 0;

          size_t ir0 = i >> 1;
          size_t ir1m = ir0 + 1;  // Value before checking for wrap-around.
          size_t ir1 = ir1m % ring_n_elt  ?  ir1m  :  ir1m - ring_n_elt;

          i0 = triLL ?  ir0  :  ir1;
          i1 = triLL ?  ir1  :  ir1 + ring_n_elt;
          i2 = ir0 + ring_n_elt;
        }

      // Get next triangle's object space coordinates ..
      //
      pCoor o0 = coors_os[i0],  o1 = coors_os[i1],  o2 = coors_os[i2];
      //
      // .. and transform them to window space.
      //
      pCoor_Homogenized_Keep_w w0( ws_from_os * o0 );
      pCoor_Homogenized_Keep_w w1( ws_from_os * o1 );
```

```
pCoor_Homogenized_Keep_w w2( ws_from_os * o2 );

// Get colors.
//
pColor c0 = colors[i0], c1 = colors[i1], c2 = colors[i2];

/// SOLUTION -- Problem 2
//
if ( topo_ring ) c0 = colors[i>>1];

stats_n_triangles++;

// Compute triangle normal (for lighting).
//
pNorm tn = cross(o0,o1,o2);
//
// May not be needed.

// If true, use normals from vtx_normals container. If false,
// use triangle normals.
//
const bool want_normals = !opt_tri_normals_always && vtx_normals;
vector<pVect4> np;
if ( want_normals ) for ( auto ix: {i0,i1,i2} ) np << vtx_normals[ix];

pVect w20(w2,w0), w21(w2,w1), w01(w0,w1);
const float db0 =
  0.5/ min( max(fabs(w20.x),fabs(w20.y)), max(fabs(w01.x),fabs(w01.y)) );
const float db1 = 0.5/max(fabs(w21.x),fabs(w21.y));

// Quick and dirty method of skipping huge triangles.
if ( db0 * db1 < 1e-7 ) continue;

// Used for computing perspective-correct barycentric coordinates.
const float w0ow2 = w0.w/w2.w, w0ow1 = w0.w/w1.w;
const float w1ow2 = w1.w/w2.w, w1ow0 = w1.w/w0.w;

// Rasterize the Triangle.
// Iterate over triangle using barycentric coordinates.
//
for ( float b0=0; b0<=1; b0 += db0 )
  for ( float b1=0; b1<=1-b0; b1 += db1 )
    {
      const float b2 = 1 - b0 - b1;

      // Each iteration operates on one fragment of triangle.

      // Window-Space Coordinates of Fragment
      //
      pCoor wf = b0 * w0 + b1 * w1 + b2 * w2;

      stats_n_fragments++;

      // Scissor (discard) fragment if it's outside the window.
```

```
            //
            if ( uint(wf.x) >= win_width || uint(wf.y) >= win_height ) continue;

            // Compute the index into the frame and depth buffers.
            //
            const size_t idx = wf.x + int(wf.y) * win_width;

            // Depth (Z) Test
            //
            float zinv = wf.z;
            if ( zinv < 0 || zinv > 1 ) continue;
            if ( zbuffer[ idx ] < zinv ) continue;

            zbuffer[ idx ] = zinv;

            // Compute perspective-correct barycentric coordinates.
            //
            float bc0 = b0 / ( b2 * w0ow2 + b1 * w0ow1 + b0 );
            float bc1 = b1 / ( b2 * w1ow2 + b0 * w1ow0 + b1 );
            float bc2 = 1 - bc0 - bc1;

            // Find object-space coordinate of fragment.
            //
            pCoor of =     bc0 * o0 + bc1 * o1 + bc2 * o2;

            // Blend color of three vertices together.
            //
            pColor color = topo_tris ? bc0 * c0 + bc1 * c1 + bc2 * c2 : c0;

            // If provided, blend vertex normals, otherwise use triangle normal.
            //
            pVect4 n = want_normals ? bc0*np[0] + bc1*np[1] + bc2*np[2] : tn;

            // Compute lighted color.
            //
            pNorm f_to_l(of,light_location);
            float phase = fabs(dot(n,f_to_l));
            pColor lighted_color = phase / f_to_l.mag_sq * light_color * color;

            //
            // Write the frame (color) buffer with the lighted color
            //
            frame_buffer[ idx ] = lighted_color.int_rgb();
          }
      }
    return *this;
}


int
main(int argc, char **argv)
{
  pFrame_Buffer demo_frame_buffer(argc,argv);
  WRender render( demo_frame_buffer );
```

```
  World our_world( demo_frame_buffer );
  WRENDER_INSERT(render, our_world);
  render.run();
  return 0;
}
```

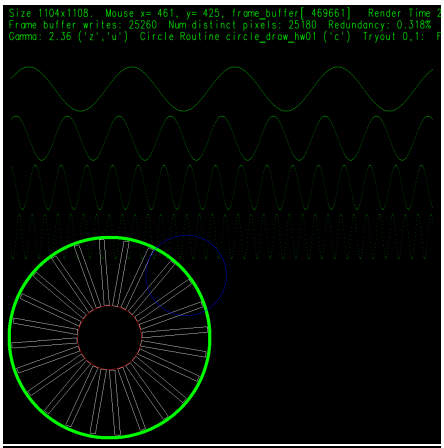
# 18 Fall 2025 Solutions

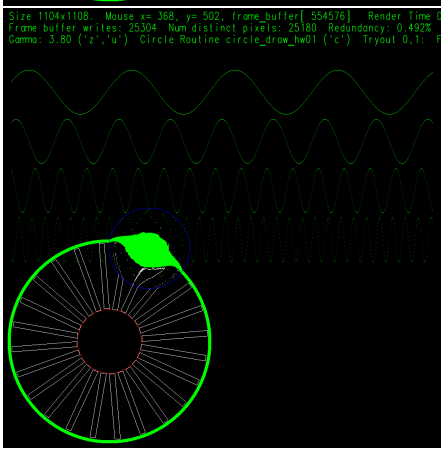
**LSU EE 4702-1** **Homework 1** **Due: 9 September 2025**

Solution *Formatted 16:43, 18 October 2025*

*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/2025/hw01.cc.html`. *The solution code is in the repository and at* `https://www.ece.lsu.edu/koppel/gpup/2025/hw01-sol.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 some sine waves and some radially arranged white rectangles with an inner red circle and an outer green ring (which is just another name for a thick green circle). See the screenshot to the upper right. dots. In the lower image, taken from a correct solution. Though it can't be seen in the screenshots, the render time is much lower. Also, there is a blue circle inside of which the image appears distorted.




### Display of Performance-Related Data

The top green text line shows performance-related and other information. `Size` refers to the size of the window. `Mouse` refers to the coordinates of the mouse pointer. Coordinate $(0, 0)$ is at the **lower** left of the window. Text `frame_buffer[N]` shows the index of the frame buffer corresponding to the point under the mouse pointer. (In the assignment file `frame_buffer` is abbreviated to `fb`, for convenience.)

`Render Time` and `Potential Frame Rate` show the CPU time needed to write the frame buffer. They can be ignored for this assignment.

### General 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 `hw01.png` or `hw01-debug.png`.

The value of two Boolean debug-support variables, `tryout0` and `tryout1`, are shown in the green text, pressing `y` toggles `tryout0` (between `true` and `false`) and pressing `Y` toggles `tryout1`. The variables are available in routine `render_hw01` and `circle_draw_hw01`, use them to try things out.

### Problem-Specific User Interface

The circle can be drawn by two routines `circle_draw_hw01` (where part of the solution is to be placed) and `circle_draw_parametric` (which is there for reference). Pressing `c` will toggle between two routines. The routine currently being used is shown in the last line of green text to the right of `Circle Routine`.

Pressing `z` will increase the distortion factor, $\gamma$, and pressing `u` will decrease it. Distortion, which should appear in the blue circle, won't be seen until Problem 2 is solved correctly.

### Code Generation and Debug Support

The compiler generates an optimized version of the code, `hw01`, and a debug-able version of the code, `hw01-debug`. The `hw01-debug` version is compiled with optimization turned off, which makes it easier to debug. When needed, you are strongly encouraged to run `hw01-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. These are `tryout0` and `tryout1`. You can use these variables in your code (for example, `if ( tryout1 ) { x += 5; }`) to help debug, to help familiarize yourself with how the code works, or to experiment with new ideas. Keys `y` and`Y` to toggle their values, which are shown in the green text at the label `Tryout 0,1:`.

### Homework Code Overview

For this assignment the code in routines `hw01_render` and `circle_draw` will be modified. The code in `hw01_render` writes the frame buffer with sine waves and a circle of rectangles radiating from a point. The code for the sine waves and circle of rectangles is there for references, but it does not need to be modified for this assignment.

### Resources

A good reference for C++ is `https://en.cppreference.com/w/`. See Homework 1 assigned in the past few semesters for similar problems. In particular, in 2023 Homework 1 Problem 2 part of the frame buffer around the mouse pointer was to be copied to another part of the frame buffer. Something like this copying is also done in Problem 2 in this assignment.

### 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:** Routine `circle_draw_parametric` draws a circle (ring) into the frame buffer:

```
void circle_draw_parametric( pFrame_Buffer& fb, HW01_Data& hw01_data,
  int ctr_x, int ctr_y, int r_outer, int r_inner, uint32_t co)
{
  int win_width = fb.width_get();
  float delta_theta = asinf(1.0f/r_outer);
  for ( float theta = 0; theta < 2*numbers::pi; theta += delta_theta )
    {
      float cth = cosf(theta),       sth = sinf(theta);
      for ( int r = r_inner; r<=r_outer; r++ )
        {
          int x = ctr_x + r * cth,   y = ctr_y + r * sth;
          fb[ y * win_width + x ] = co;
        }}}
```

The center of the circle is at pixel coordinate `ctr_x`, `ctr_y`, the inner radius is `r_inner` and the outer radius is `r_outer`. Pixels of the frame buffer from the inner and outer radius are set to color `co`, or at least they should, but some pixels are skipped. The routine is based on a parametric description of a circle: $P(\theta) = C + \begin{bmatrix} r\cos\theta \\ r\sin\theta \end{bmatrix}$.

The routine above is not the best way of drawing a circle in pixel space (which is what we are using) because it calls trig functions (sine and cosine) many times (and that's computationally costly), because it doesn't write every pixel, and because it writes some pixels multiple times. (If `delta_theta` is made small enough it would write every pixel at least once, but it would be writing many more pixels multiple times.)

The fundamental problem with the routine above is that we are iterating over $\theta$ and $r$, a better approach is to iterate over pixel coordinates. In the unmodified assignment the code in `circle_draw_hw01` does iterate over pixel coordinates:

```
void circle_draw_hw01 ( pFrame_Buffer& fb, HW01_Data& hw01_data,
  int ctr_x, int ctr_y, int r_outer, int r_inner, uint32_t co)
{
  const int win_width [[maybe_unused]] = fb.width_get();
  const int win_height [[maybe_unused]] = fb.height_get();
  /// WARNING: Ridiculously inefficient.
  for ( int x = 0; x < win_width; x++ )
    for ( int y = 0; y < win_height; y++ )
      {
        int dx = x - ctr_x,   dy = y - ctr_y;
        int dsq = dx * dx + dy * dy;
        if ( dsq > r_outer * r_outer ) continue;
        if ( dsq < r_inner * r_inner ) continue;
        fb[ y * win_width + x ] = co;
      }
```

The code iterates over every pixel *in the frame buffer*, and checks whether that pixel is on the circle (between the inner and outer radii). On the plus side this code won't skip any pixels, it won't write the same pixel twice, it doesn't call any trigonometry functions, and it even avoids computing a square root. But all of those positives are outweighed by the fact that it checks every single pixel **in the frame buffer**!

Modify the code so that it iterates over fewer pixels. Consider solutions that keep the two-level loop nest, but in which the loop bounds are chosen so that the body is executed fewer times.

Consider the following ideas:

*Idea 1:* Modify the loop bounds so that `x` and `y` iterate over a *bounding box* that contains the circle. (A bounding box of a circle is the smallest possible box containing the circle. Any smaller and a part of the circle would be outside the box.)

*Idea 2:* Compute the limits on the `y` loop as a function of `x`. (Note that Idea 2 replaces or modifies the Idea 1 code.)

*Idea 3:* Have `x` and `y` iterate over only one quadrant of the circle, but have the loop body write four pixels (one write for the quadrant corresponding to `x` and `y`, and the other three writes for the other three quadrants.)

Solution on next page.

Solution code appears below.

```
void circle_draw_hw01 ( pFrame_Buffer& fb, HW01_Data& hw01_data,
  int ctr_x, int ctr_y,  int r_outer, int r_inner, uint32_t co)
{
  const int win_width [[maybe_unused]] = fb.width_get();
  const int win_height [[maybe_unused]] = fb.height_get();

  // Pre-compute the squares of the inner and outer radii.
  //
  int r_o_sq = r_outer * r_outer;
  int r_i_sq = r_inner * r_inner;

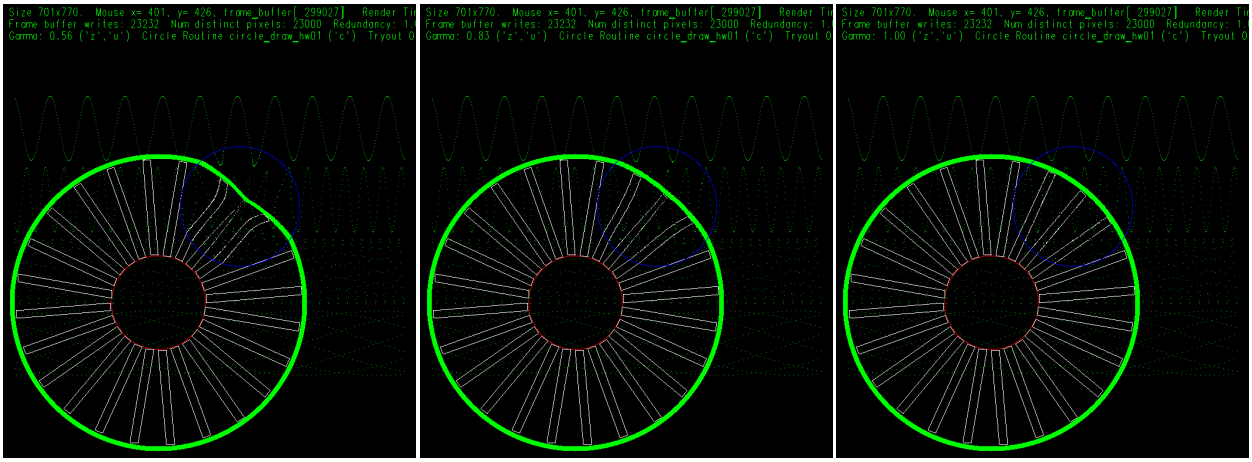
  // The term "ring" refers to the area from r_inner to r_outer.
  //
  // Technically ring refers to the parts of the circle at
  // (ctr_x,ctr_y) of radius r_outer which are not also parts of the
  // circle at (ctr_x,ctr_y) of radius r_inner.

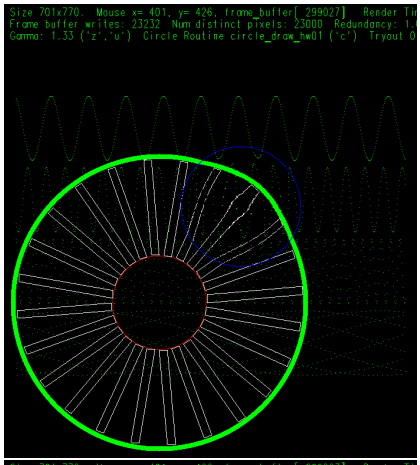
  // Compute frame buffer array index of center of circle.
  //
  int idx_ctr = ctr_y * win_width + ctr_x;

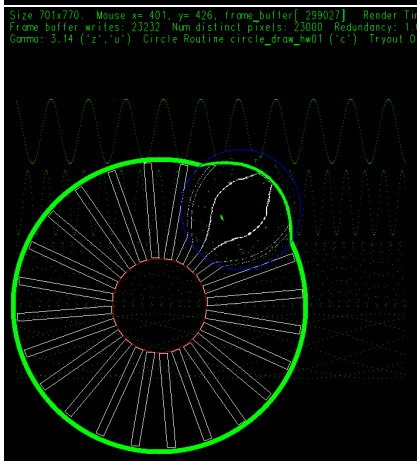
  // Idea 1 (and 3): Iterate over circle-local coordinates, ly and lx.
  // Center of circle is at lx=0, ly=0.
  //
  for ( int ly = 0;  ly <= r_outer;  ly++ )
    {
      // Idea 2: (with x and y swapped)
      // Compute range of x values over the ring at ly.
      //
      int lx0 = sqrtf( max( 0, r_i_sq - ly * ly ) );
      int lx1 = sqrtf( max( 0, r_o_sq - ly * ly ) );

      // Compute frame buffer array index distance from center of
      // circle to the pixel at (lx=0, ly ):
      //
      int ly_ww = ly * win_width;

      for ( int lx = lx0;  lx <= lx1;  lx++ )
        {
          // Idea 3: Draw a pixel in each of four quadrants of the circle.
          //
          fb[ idx_ctr + ly_ww + lx ] = co;
          fb[ idx_ctr + ly_ww - lx ] = co;
          fb[ idx_ctr - ly_ww + lx ] = co;
          fb[ idx_ctr - ly_ww - lx ] = co;
        }
    }
}
```







**Problem 2:** A blue circle is drawn near the end of routine `render_hw01` in a call to `circle_draw`. Modify the code after this point so that the portion of the frame buffer inside the blue circle (but not the blue circle itself) is distorted as follows:

Let $C$ denote the center of the circle (in the code the center is in variable `ctr`) and $R$ its radius (`mag_r` in the code). Let $P = \begin{bmatrix} P_x \\ P_y \end{bmatrix}$ denote the coordinates of a pixel inside the circle.

Let $v = C - P = \begin{bmatrix} C_x - P_x \\ C_y - P_y \end{bmatrix}$ denote the vector from $C$ to $P$. In vector notation $P = C + v$. Let $d$ denote the length of $v$, which in this case is the distance from $C$ to $P$. (If you don't know how to compute the distance, look it up.)

For this problem define *distorted distance* $d' = R\left(\frac{d}{R}\right)^{\gamma}$, where $\gamma$ is a distortion factor. Note that for $\gamma = 1$ we just have $d' = d$. Given a distorted distance we can define an alternative pixel coordinate $P' = C + \frac{d'}{d}v$.

Modify the code in `hw01_render` so that every pixel $P$ in the blue circle is written with the frame buffer contents at $P'$ as defined above. In the code variable `ctr` is the circle center (see how it's used in the call to `circle_draw`). Variable `mag_r` is $R$, and `gamma` is $\gamma$. Variable `P` is $P$; your code can just use `x` and `y`. In the code variable `v` is $v$; your code can just use `dx` and `dy`. The code computes $d^2$, but not $d$. Using the user interface, the value of `gamma` can be modified by pressing `z` and `u`, and its value is shown on the third line of green text.

Some things to watch out for: most of the variables are declared `int`. To work correctly $d'$ must be computed using floating point operands. Let `xp` and `yp` denote the coordinates of $P'$. If these are used to compute a frame buffer index, `fb[ yp * win_width + xp]`, make sure that `yp` is an integer, or cast it to an integer otherwise `yp * win_width` won't give the correct result.

Finally, and importantly remember that the pixel being read (at coordinate $P'$) must be from

the frame buffer *before* it was modified by the code that fills the blue circle. So code like `fb[ y * win_width + x] = fb[ int(yp) * win_width + xp];` won't work correctly because the pixel being read, `fb[ int(yp) * win_width + xp];`, might have already been written. So, the area inside the blue circle first needs to be copied to a buffer area and that buffer area needs to be used when writing inside the blue circle. Before writing any pixels in the blue circle first copy them to array `fb_dup`. Array `fb_dup` is of size $(2(R+1))^2$, and so it has enough space for the blue pixels. It's okay of a bounding box around the blue circle is copied into `fb_dup`.

Once the blue circle are is copied to `fb_dup` one can get the distortion effect by writing `fb[ int(yp) * win_width + xp] = fb_dup[ b_y * b_width + b_x]`, where `b_width` is the width of the area copied **(which can't be larger than** $2(R+1)$**)** and `b_y` and `b_x` are relative to a corner of the copied area. See 2023 Homework 1 Problem 2 and its solution for more on how to use such a buffer area (called `fb_area_dup`) in that problem.

An excerpt from the solution appears below. The complete solution is in the repository in file `hw01-sol.cc`.

```
int idx_ctr = ctr.y * win_width + ctr.x;
int mag_r_sq = mag_r * mag_r;
int mag_2r = 2 * mag_r;

// Copy area under blue circle to fb_dup.
//
for ( int x = -mag_r; x < mag_r; x++ )
  for ( int y = -mag_r; y < mag_r; y++ )
    fb_dup[ ( y + mag_r ) * mag_2r + x + mag_r ]
      = fb[ ( ctr.y + y ) * win_width + ctr.x + x ];
//
// This also copies pixels outside the circle. We don't need the
// pixels outside the circle, but we copy them anyway because it
// might take more time to determine whether a pixel is in the
// circle than to copy it.

const float mag_r_inv = 1.0 / mag_r;

// [✓] Change x and y bounds so they iterate over a blue circle bounding box.
//     Actually, iterate over one quadrant of circle in local coordinates.
//
for ( int lx = 0; lx < mag_r; lx++ )
  for ( int ly = 0; ly < mag_r; ly++ )
    {
      // Skip this iteration if we are not inside the circle.
      //
      int len_sq = lx*lx + ly*ly;
      if ( len_sq > mag_r_sq ) break;

      // Compute distance from pixel to circle center ..
      //
      float d = powf( len_sq, 0.5 );
      //
      // .. and the distorted distance.
      //
      float d_prime = powf( d * mag_r_inv, gamma ) * mag_r;
```

```
  // Scale lx and ly based on the distorted distance.
  //
  int lx_src = d ? lx * d_prime / d : 0;
  int ly_src = d ? ly * d_prime / d : 0;
  //
  // Note: because d_prime is a float lx is converted (promoted)
  // to a float before performing the multiplication lx * d_prime.

  // For each quadrant, read the pixel at lx_src, ly_src from fb_dup,
  // and write it to fp.
  //
  fb[ idx_ctr + ly * win_width + lx ] =
    fb_dup[ (mag_r+ly_src) * mag_2r + lx_src + mag_r];
  fb[ idx_ctr + ly * win_width - lx ] =
    fb_dup[ (mag_r+ly_src) * mag_2r - lx_src + mag_r];
  fb[ idx_ctr - ly * win_width - lx ] =
    fb_dup[ (mag_r-ly_src) * mag_2r - lx_src + mag_r];
  fb[ idx_ctr - ly * win_width + lx ] =
    fb_dup[ (mag_r-ly_src) * mag_2r + lx_src + mag_r];
  //
  // It would not be correct to both read and write from fb, since
  // the pixel read might be an already distorted pixel.
}
```

```
/// LSU EE 4702-1 (Fall 2025), GPU Programming
//
//
 /// Homework 1 -- SOLUTION
 //
 //  Assignment: https://www.ece.lsu.edu/gpup/2025/hw01.pdf
 //
 //  This code is based on the solution to 2023 Homework 1.
 //  For the solution to 2023 Homework 1 look
 //  at file ../../2023/hw01/hw01-sol.cc
 //  or visit https://www.ece.lsu.edu/koppel/gpup/2023/hw01-sol.cc.html
 //

#include <gp/coord.h>
#include <numbers>
#include "frame_buffer.h"

// Definitions for iVect, iCoor
#include "hw01-misc.h"

template<class T>
class safe_vector : public vector<T> {
public:
  T& operator [] ( size_t idx )
  {
    return vector<T>::at(idx);
  }
};

struct HW01_Data {
  HW01_Data():inited(false){};
  bool inited;
  bool tryout0, tryout1;
  // Note: Safe vector defined in hw01-misc.h
  safe_vector<uint32_t> fb_area_dup;

  bool opt_circle_hw01;

  float gamma;
  int mag_r;
};


void
circle_draw_hw01
( pFrame_Buffer& fb, HW01_Data& hw01_data,
  int ctr_x, int ctr_y,
  int r_outer, int r_inner, uint32_t co)
{
  const int win_width [[maybe_unused]] = fb.width_get();
  const int win_height [[maybe_unused]] = fb.height_get();

  /// SOLUTION -- Problem 1
```

```
  // Pre-compute the squares of the inner and outer radii.
  //
  int r_o_sq = r_outer * r_outer;
  int r_i_sq = r_inner * r_inner;

  // The term "ring" refers to the area from r_inner to r_outer.
  //
  // Technically ring refers to the parts of the circle at
  // (ctr_x,ctr_y) of radius r_outer which are not also parts of the
  // circle at (ctr_x,ctr_y) of radius r_inner.

  // Compute frame buffer array index of center of circle.
  //
  int idx_ctr = ctr_y * win_width + ctr_x;

  // Idea 1 (and 3): Iterate over circle-local coordinates, ly and lx.
  // Center of circle is at lx=0, ly=0.
  //
  for ( int ly = 0;  ly <= r_outer;  ly++ )
    {
      // Idea 2: (with x and y swapped)
      // Compute range of x values over the ring at ly.
      //
      int lx0 = sqrtf( max( 0, r_i_sq - ly * ly ) );
      int lx1 = sqrtf( max( 0, r_o_sq - ly * ly ) );

      // Compute frame buffer array index distance from center of
      // circle to the pixel at (lx=0, ly ):
      //
      int ly_ww = ly * win_width;

      for ( int lx = lx0;  lx <= lx1;  lx++ )
        {
          // Idea 3: Draw a pixel in each of four quadrants of the circle.
          //
          fb[ idx_ctr + ly_ww + lx ] = co;
          fb[ idx_ctr + ly_ww - lx ] = co;
          fb[ idx_ctr - ly_ww + lx ] = co;
          fb[ idx_ctr - ly_ww - lx ] = co;
        }
    }
}

void
circle_draw_parametric
( pFrame_Buffer& fb, HW01_Data& hw01_data,
  int ctr_x, int ctr_y,
  int r_outer, int r_inner, uint32_t co)
{
  const int win_width = fb.width_get();

  /// This routine DOES NOT have to be modified.
  //
```

```
  //  It can be modified to experiment or to help in understanding
  //  the code, but no problems in 2025 Homework 1 require that this
  //  routine be modified.

  // Choose delta_theta so that a loop will write every pixel along
  // the outer edge.
  float delta_theta = asinf(1.0f/r_outer);
  for ( float theta = 0; theta < 2*numbers::pi; theta += delta_theta )
    {
      float cth = cosf(theta),       sth = sinf(theta);
      for ( int r = r_inner; r<=r_outer; r++ )
        {
          int x = ctr_x + r * cth,   y = ctr_y + r * sth;
          fb[ y * win_width + x ] = co;
        }
    }
}

void circle_draw
( pFrame_Buffer& fb, HW01_Data& hw01_data,
  int ctr_x, int ctr_y,
  int r_outer, int r_inner, uint32_t co)
{
  if ( hw01_data.opt_circle_hw01 )
    circle_draw_hw01(fb,hw01_data,ctr_x,ctr_y,r_outer, r_inner,co);
  else
    circle_draw_parametric(fb,hw01_data,ctr_x,ctr_y,r_outer, r_inner,co);
}


void
line_draw
( pFrame_Buffer& fb, iCoor p0, iCoor p1, uint32_t co)
{
  int x0 = p0.x, y0 = p0.y;
  int x1 = p1.x, y1 = p1.y;

  const iVect delta = p1 - p0;
  const int width = fb.width_get();

  if ( abs( delta.x ) < abs( delta.y ) )
    {
      // Iterate over y
      if ( y0 > y1 ) { swap(x0,x1); swap(y0,y1); }

      float dxdy = float(delta.x) / delta.y;
      float x = x0;

      for ( int yi = y0;  yi < y1;  yi++ )
        {
          fb[ yi * width + x ] = co;
          x += dxdy;
        }
    }
```

```
  else
    {
      // Iterate over x
      if ( x0 > x1 ) { swap(x0,x1); swap(y0,y1); }

      float dydx = float(delta.y) / delta.x;
      float y = y0;

      for ( int xi = x0;  xi < x1;  xi++ )
        {
          fb[ int(y) * width + xi ] = co;
          y += dydx;
        }
    }
}


void
parallel_lines_draw
( pFrame_Buffer& fb, iCoor p0, iCoor pb, iVect v, uint32_t co)
{
  line_draw( fb, p0, p0 + v, co );
  line_draw( fb, pb, pb + v, co );
}

void
parallelogram_draw
( pFrame_Buffer& fb, iCoor p00, iVect vx,  iVect vy, uint32_t color )
{
  parallel_lines_draw(fb, p00, p00+vy, vx, color);
  parallel_lines_draw(fb, p00, p00+vx, vy, color);
}

void
aa_rectangle_draw
( pFrame_Buffer& fb, iCoor p00, int wd, int ht, uint32_t color )
{
  // Window Size, in Pixels
  //
  const int win_width [[maybe_unused]] = fb.width_get();
  const int win_height [[maybe_unused]] = fb.height_get();

  iVect vx(wd,0);
  iVect vy(0,ht);
  iCoor p10 = p00 + vx;
  iCoor p01 = p00 + vy;
  iCoor p11 = p10 + vy;

  // Draw vertical lines.
  line_draw( fb, p00, p01, color );
  line_draw( fb, p10, p11, color );

  // Draw horizontal lines.
  line_draw( fb, p00, p10, color );
```

```
  line_draw( fb, p01, p11, color );

}

void kbd_input_handle(pFrame_Buffer& fb, HW01_Data& hw01_data);


void
render_hw01(pFrame_Buffer& fb, HW01_Data& hw01_data)
{
  // Window Size, in Pixels
  //
  const int win_width = fb.width_get();
  const int win_height = fb.height_get();

  // Adjust size and position of mag window based on keyboard input.
  kbd_input_handle(fb,hw01_data);

  /// Coordinates Used for Placement of Other Objects
  //
  const int sq_slen = min(win_width,win_height) * .45;
  const int sq_x0 = sq_slen * 0.1,   sq_y0 = win_height - 150;
  const int sq_x1 = sq_x0 + sq_slen, sq_y1 = sq_y0 - sq_slen;
  //
  // One vertex of the square is at (sq_x0,sq_y0) and the opposite
  // vertex is at (sq_x1,sq_y1).

  //                            RRGGBB
  const uint32_t color_red = 0xff0000;
  const uint32_t color_blue =    0xff;
  const uint32_t color_green = 0xff00;

  /// Demo Code: Draw Sine Waves  -- No need to modify this.
  //
  int num_waves = 4;
  float plot_ht = sq_slen/float(num_waves);
  for ( float
          yb = sq_y0 - plot_ht/2,  freq = 4 * M_PI / ( win_width - 40 - sq_x1 );
        yb > sq_y1;
        yb -= plot_ht,  freq *= 2 )
    for ( int x = 20;  x < win_width - 20;  x++ )
      {
        int y = yb + sin( x * freq ) * plot_ht * .45;
        fb[ y * win_width + x ] = 0xff00;
      }

  /// Demo Code: Draw Lines Radiating From a Point -- No need to modify.
  //
  //  This code calls the line_draw routine.
  //
  int n_rects = 24;
  int n_pts = n_rects * 3;
  float c_r1 = sq_slen/6;  // Inner radius
  float c_r2 = sq_slen/2;  // Outer radius.
```

```
pCoor c_ctr( c_r2 + 20, c_r2 + 20 ); // Center of "circle"
float delta_theta = 2 * M_PI / n_pts;
vector<pVect> vecs;

circle_draw(fb, hw01_data, c_ctr.x, c_ctr.y, c_r2+9, c_r2+1, color_green);
circle_draw(fb, hw01_data, c_ctr.x, c_ctr.y, c_r1-1, c_r1-2, color_red);

for ( int i=0; i<n_pts; i++ )
  vecs.push_back( pVect( cosf(i*delta_theta), sinf(i*delta_theta), 0 ) );
for ( int i=0; i<n_pts; i+=3 )
  {
    pVect v1 = vecs[i+1] * ( c_r2 - c_r1 );
    pVect v2 = c_r1 * ( vecs[i+2] - vecs[i] );
    pCoor p1 = c_ctr + c_r1 * vecs[i];
    parallelogram_draw(fb, p1, v1, v2, 0xffffff );
  }

fb.fb_stats_off(); // Turn off "redundancy" statistics.


/// Homework 1 Problem 2

// Coordinates of mouse pointer and circle center.
//
const iCoor ctr(fb.mouse_x,fb.mouse_y);
//
// See 2024 Homework 1 for a description of iCoor.

// Radius of distorted area.
//
const int mag_r = hw01_data.mag_r;          // Radius of distorted area.

// Distortion factor.
//
const float gamma [[maybe_unused]] = hw01_data.gamma;
//
// Gamma can be modified in the UI by pressing z and u.

// Storage for a copy of the frame buffer in and around the blue circle.
//
auto& fb_dup [[maybe_unused]] = hw01_data.fb_area_dup;

// Draw a blue circle.
//
circle_draw(fb, hw01_data,  ctr.x, ctr.y,  mag_r+2, mag_r+1, color_blue);

/// Put Problem 2 Solution Below (and possibly above, but mostly below).
//

// [✔] Copy area inside the box to fb_dup.
// [✔] Compute d' (see handout)
// [✔] Use d' to compute coordinates of part of frame buffer (actually fb_dup)
//     to read from.
// [y] Read from those coordinates and write to (x,y).
```

```
/// SOLUTION -- Problem 2

int idx_ctr = ctr.y * win_width + ctr.x;
int mag_r_sq = mag_r * mag_r;
int mag_2r = 2 * mag_r;

// Copy area under blue circle to fb_dup.
//
for ( int x = -mag_r; x < mag_r; x++ )
  for ( int y = -mag_r; y < mag_r; y++ )
    fb_dup[ ( y + mag_r ) * mag_2r + x + mag_r ]
      = fb[ ( ctr.y + y ) * win_width + ctr.x + x ];
//
// This also copies pixels outside the circle. We don't need the
// pixels outside the circle, but we copy them anyway because it
// might take more time to determine whether a pixel is in the
// circle than to copy it.

const float mag_r_inv = 1.0 / mag_r;

// [✔] Change x and y bounds so they iterate over a blue circle bounding box.
//     Actually, iterate over one quadrant of circle in local coordinates.
//
for ( int lx = 0; lx < mag_r; lx++ )
  for ( int ly = 0; ly < mag_r; ly++ )
    {
      // Skip this iteration if we are not inside the circle.
      //
      int len_sq = lx*lx + ly*ly;
      if ( len_sq > mag_r_sq ) break;

      // Compute distance from pixel to circle center ..
      //
      float d = powf( len_sq, 0.5 );
      //
      // .. and the distorted distance.
      //
      float d_prime = powf( d * mag_r_inv, gamma ) * mag_r;

      // Scale lx and ly based on the distorted distance.
      //
      int lx_src = d ? lx * d_prime / d : 0;
      int ly_src = d ? ly * d_prime / d : 0;
      //
      // Note: because d_prime is a float lx is converted (promoted)
      // to a float before performing the multiplication lx * d_prime.

      // For each quadrant, read the pixel at lx_src, ly_src from fb_dup,
      // and write it to fp.
      //
      fb[ idx_ctr + ly * win_width + lx ] =
        fb_dup[ (mag_r+ly_src) * mag_2r + lx_src + mag_r];
      fb[ idx_ctr + ly * win_width - lx ] =
```

```
          fb_dup[ (mag_r+ly_src) * mag_2r - lx_src + mag_r];
        fb[ idx_ctr - ly * win_width - lx ] =
          fb_dup[ (mag_r-ly_src) * mag_2r - lx_src + mag_r];
        fb[ idx_ctr - ly * win_width + lx ] =
          fb_dup[ (mag_r-ly_src) * mag_2r + lx_src + mag_r];
        //
        // It would not be correct to both read and write from fb, since
        // the pixel read might be an already distorted pixel.
      }

}

void
kbd_input_handle(pFrame_Buffer& fb, HW01_Data& hw01_data)
{
  if ( !hw01_data.inited )
    {
      hw01_data.inited = true;
      hw01_data.gamma = 1;
      hw01_data.mag_r = 100;
      hw01_data.opt_circle_hw01 = true;
      int mag_2r = (hw01_data.mag_r+1) * 2;

      hw01_data.tryout0 = false;
      hw01_data.tryout1 = false;

      hw01_data.fb_area_dup.resize( mag_2r * mag_2r );
    }

  switch ( fb.keyboard_key ) {
#if 0
  case FB_KEY_RIGHT: case FB_KEY_LEFT:
    {
      const int inc = fb.keyboard_key == FB_KEY_LEFT ? -kbd_inc : kbd_inc;
      break;
    }
  case FB_KEY_UP: case FB_KEY_DOWN:
    {
      const int inc = fb.keyboard_key == FB_KEY_DOWN ? -kbd_inc : kbd_inc;
      break;
    }
#endif

  case 'c': case 'C':
    hw01_data.opt_circle_hw01 = !hw01_data.opt_circle_hw01; break;

  case 'u': hw01_data.gamma /= 1.1; break;
  case 'U': hw01_data.gamma /= 1.5; break;
  case 'z': hw01_data.gamma *= 1.1; break;
  case 'Z': hw01_data.gamma *= 1.5; break;

  case 'y':
    hw01_data.tryout0 = !hw01_data.tryout0; break;
  case 'Y':
```

```
    hw01_data.tryout1 = !hw01_data.tryout1; break;
  default:
    break;
  };

  fb.fbprintf("Gamma: %.2f ('z','u')  Circle Routine %s ('c')  "
              "Tryout 0,1:  %s,%s ('y','Y')\n",
              hw01_data.gamma,
              hw01_data.opt_circle_hw01
              ? "circle_draw_hw01" : "circle_draw_parametric",
              hw01_data.tryout0 ? "TRUE " : "FALSE",
              hw01_data.tryout1 ? "TRUE " : "FALSE");
}

int
main(int argc, char **argv)
{
  pFrame_Buffer demo_frame_buffer(argc,argv);

  HW01_Data hw01_data;
  demo_frame_buffer.show
    ( [&](pFrame_Buffer& fb){ render_hw01(fb,hw01_data); } );
  return 0;
}
```

```
/// # LSU EE 4702-X (Fall 2025), GPU Programming
//
 /// ## Homework 2 -- SOLUTION
 //
 //  Assignment: https://www.ece.lsu.edu/gpup/2025/hw02.pdf
 //
 /// ## Modified this file only.
 //
 /// ## Solution In:  World::ball_setup_7()
 //
 // Based on the Links Project Code
 //


/// # Purpose
//
//   Demonstrate simulation of point masses connected by springs.


/// # What Code Does

// Simulates balls connected by springs over a platform. Balls and
// springs can be initialized in different arrangements (called
// scenes). Currently scene 1 is a tree-like arrangement. The platform
// consists of tiles, some are purple-tinted mirrors (showing a
// reflection of the ball), the others show the course syllabus.


///  # Keyboard Commands
 //
 /// ## Object (Eye, Light, Ball) Location or Push
 //   Arrows (←,→,↑,↓) Page Up, Page Down
 //        Move object or push ball, depending on mode.
 //        Shift + KEY: motion is 5x faster.
 //        Ctrl + KEY : motion is 5x slower.
 //   'e': Move eye.
 //   'l': Move light.
 //   'b': Move head (first) ball. (Change position but not velocity.)
 //   'B': Push head ball. (Add velocity.)
 //
 /// ## Eye Direction and GUI Options
 //
 //   'Home', 'End', 'Delete', 'Insert'
 //         Turn the eye direction. Home should rotate eye direction
 //         up, End should rotate eye down, Delete should rotate eye
 //         left, Insert should rotate eye right. The eye direction
 //         vector is displayed in the upper left.
 //  'C-='  (Ctrl =) Increase green text size.
 //  'C--'  (Ctrl -) Decrease green text size.
 //  'F12'  Write screenshot to PNG file.
```

```
/// Simulation Execution Options
//  (Also see variables below.)
//
//  'p'    Pause simulation. (Press again to resume.)
//  ' '    (Space bar.) Advance simulation by 1/30 second.
//  'S- '  (Shift Space bar.) Advance simulation by one time step.

/// Performance Tuning, Graphics Effects, and Debugging Options
//
//  'y'    Toggle value of opt_tryout1. Intended for experiments and debugging.
//  'Y'    Toggle value of opt_tryout2. Intended for experiments and debugging.
//  'n'    Toggle use of surface normals for spheres.
//  'o'    Toggle shadow effect.
//  'r'    Toggle mirror effect.
//  '!'    Toggle rendering of spheres (balls).
//  '@'    Toggle rendering of links.
//  '#'    Toggle rendering of platform.


/// Scene and Simulated System Options
//  (Also see variables below.)
//
//  'w'    Twirl balls around axis formed by head and tail. (Prob 2 soln).
//  '1'    Set up scene 1.
//  '2'    Set up scene 2.
//  '3'    Set up scene 3.
//  '4'    Set up scene 4.
//  'h'    Freeze position of first (head) ball. (Press again to release.)
//  't'    Freeze position of last (tail) ball. (Press again to release.)
//  's'    Stop balls.
//  'g'    Turn gravity on and off.

/// Variables
//   Selected program variables can be modified using the keyboard.
//   Use "Tab" to cycle through the variable to be modified, the
//   name of the variable is displayed next to "VAR" on the bottom
//   line of green text.

//  'Tab' Cycle to next variable.
//  '`'   Cycle to previous variable.
//  '+'   Increase variable value.
//  '-'   Decrease variable value.
//
//  VAR Spring Constant - Set spring constant.
//  VAR Time Step Duration - Set physics time step.
//  VAR Air Resistance - Set air resistance.
//  VAR Light Intensity - The light intensity.
//  VAR Gravity - Gravitational acceleration. (Turn on/off using 'g'.)

#define MAIN_INCLUDE
#undef VBUFFER_STRICT
//  #define VBUFFER_STRICT_WATCH 0xd10d270000000018
#include <vhelper.h>
```

```
#include <vstroke.h>
#include <gp/coord.h>
#include <gp/pstring.h>
#include <gp/misc.h>

#include <vutil-texture.h>
#include <vutil-pipeline.h>
#include "shapes.h"

#include <cuda_runtime.h>

#include <gp/colors.h>

#include <gp/util-containers.h>
#include <gp/coord-containers.h>
#include <gp/cuda-gpuinfo.h>
#include "links.cuh"

#include "links-shdr-common.h"

#include <numbers>
#include <ranges>
#include <algorithm>

///
/// Main Data Structures
///
//
// class World: All data about scene.


class World;

enum GPU_Physics_Method { GP_cpu, GP_cuda, GP_ENUM_SIZE };
const char* const gpu_physics_method_str[] = { "CPU", "CUDA" };
const char *sh_names[] = { "PLAIN", "SET 1", "SET 2" };


// Object Holding Ball State
//
class Ball {
public:
  Ball() { init(); }
  Ball(pCoor pos){ init(); position = pos; constants_update(); }
  Ball(pCoor pos, const Ball &bd)
  { *this = bd; position = pos; constants_update(); }

  void init()
    {
      velocity = pVect(0,0,0);
      omega = pVect(0,0,0);
      radius = 1;
      mass = 0;
      mass_inv = 1;
```

```
    density = 1.00746;
    fdt_to_do = 0;
    locked = false;
    color = color_lsu_spirit_gold;
    specularity = 0;
    contact = false;
    orientation_set(pVect(0,1,0),0);
    world_time_expire = 3600.0 * 24 * 365 * 1000;
  }

pCoor position;
pVect velocity;
pQuat orientation;
pMatrix3x3p omatrix;
pVect omega;                  // Spin rate and axis.
double world_time_expire;

int idx; // Position in balls_pos_rad;

// 2025 Homework 2
int id;

union {
  float4 m_r_fdt;
  struct {
    float mass;
    float radius;
    float fdt_to_do; // Radius / moment of inertia.
    float _;
  }; };

float mass_min; // Mass below which simulation is unstable.
float radius_sq, radius_inv;  // Pre-computed based on radius.
float density;
float mass_inv;

bool locked;

pVect force;
pVect torque;
pColor color;
float specularity;            // Amount of shininess, in [0,1].
bool contact;                 // When true, ball rendered in gray.
float spring_constant_sum;    // Used to compute minimum mass.

void orientation_set(pNorm dir, float angle)
  { orientation_set(pQuat(dir,angle)); }
void orientation_set(pQuat ori)
  {
    orientation = ori;
    omatrix = pMatrix3x3_Rotation(orientation);
  }
void constants_update();
pVect point_rot_vel(pNorm tact_dir);
```

```
  pVect point_rot_vel(pVect tact_dir);
  void apply_tan_force(pNorm tact_dir, pNorm force_dir, double force_mag);
  void apply_tan_force(pNorm tact_dir, pVect force);
  void push(pVect amt);
  void translate(pVect amt);
  void stop();
  void freeze();
};


void
Ball::constants_update()
{
  assert( radius );
  radius_sq = radius * radius;
  if ( mass >= 0 )
    {
      mass = 4.0 / 3 * numbers::pi * radius_sq * radius * density;
      mass_inv = 1 / mass;
    }
  radius_inv = 1 / radius;
  // FYI, notice simplifications:
  //   const float mo_inertia = 0.4 * mass * radius_sq;
  //   fdt_to_do = radius / mo_inertia;
  fdt_to_do = radius_inv * 2.5 * mass_inv;
}

pVect
Ball::point_rot_vel(pNorm direction)
{
  /// Return velocity of point on surface of ball.
  //
  return radius * cross( omega, direction );
}

pVect
Ball::point_rot_vel(pVect rel_pos)
{
  /// Return velocity of point relative to center.
  //
  return cross( omega, rel_pos );
}

void
Ball::apply_tan_force(pNorm tact_dir, pVect force)
{
  torque += cross(tact_dir, force);
}

void
Ball::apply_tan_force(pNorm tact_dir, pNorm force_dir, double force_mag)
{
  apply_tan_force(tact_dir, force_mag * force_dir);
```

```
}

struct Link_Args {
  pColor natural_color = color_red;
  float distance_relaxed = 0;
  float spring_constant = 0;
  float link_stiffness = 0;
  float radius = 0;
};

class Link {
public:
  Link(Ball *b1, Ball *b2, Link_Args args = {}):ball1(b1),ball2(b2),
     snapped(false)
  { init(args); }
  Link(Link *link, pVect cb1p, pVect cb2p, float drp):
    ball1(link->ball1), ball2(link->ball2), cb1(cb1p), cb2(cb2p),
    is_surface_connection(true), is_renderable(false),
    is_simulatable(true), distance_relaxed(drp), spring_constant(0),
    radius(0),snapped(false){link->is_simulatable = false;}

  Link(Link *link, pVect cb1p, pVect cb2p, Link_Args args = {} ):
    ball1(link->ball1), ball2(link->ball2), cb1(cb1p), cb2(cb2p),
    is_surface_connection(true), is_renderable(false),
    is_simulatable(true),
    distance_relaxed(args.distance_relaxed),
    spring_constant(args.spring_constant),
    radius(0),snapped(false){link->is_simulatable = false;}

  void init(Link_Args& args)
  {
    natural_color = args.natural_color;
    spring_constant = args.spring_constant;
    radius = args.radius;
    assert( ball1->radius > 0 );
    assert( ball2->radius > 0 );
    is_simulatable = true;
    is_renderable = true;
    pNorm n12(ball1->position,ball2->position);
    const float rad_sum = ball1->radius + ball2->radius;
    is_surface_connection = n12.magnitude >= rad_sum;

    pMatrix3x3 b1rot = ball1->omatrix;
    pMatrix3x3 b2rot = ball2->omatrix;

    pVect b1_y = b1rot * pVect(0,1,0);
    pVect b1_x = b1rot * pVect(1,0,0);
    bool b1_dir_is_y = fabs(dot(b1_y,n12)) < 0.999;
    pVect b1_dir_loc = b1_dir_is_y ? b1_y : b1_x;

    con_x = pNorm(cross(b1_dir_loc,n12));
    con_y = cross(n12,con_x);
    b1_dir = mult_MTV( b1rot, con_y );
    b2_dir = mult_MTV( b2rot, con_y );
```

```
    if ( !is_surface_connection )
      {
        distance_relaxed = n12.magnitude;
        cb1 = cb2 = pVect(0,0,0);
        is_renderable = false;
        return;
      }

    distance_relaxed = n12.magnitude - rad_sum;
    cb1 = ball1->radius * mult_MTV( b1rot,  n12 );
    cb2 = ball2->radius * mult_MTV( b2rot, -n12 );
  }
  int serial;
  int idx;  // Position in lis_pos1, etc.
  Ball* const ball1;
  Ball* const ball2;
  union {
    int2 ball_idx;
    struct { int ball1_idx, ball2_idx; };
  };
  pVect cb1, cb2, con_x, con_y, b1_dir, b2_dir;
  bool is_surface_connection;
  bool is_renderable, is_simulatable;
  union {
    float2 dr_sc;
    struct {
      float distance_relaxed;
      float spring_constant;
    };
  };
  float radius;
  bool snapped;
  pColor natural_color;
  pColor color;

  pVect spring_force_12;
  pVect torque1,torque2;

};

class Surface {
public:
  Ball *b0, *b1, *b2;
  pColor color = color_blue;
  pTCoor tc0, tc1, tc2;
  union {
    int4 b_idx;
    struct { int b0_idx, b1_idx, b2_idx, _; };
  };
  Surface* operator ->() { return this; }
};

// Declare containers for Balls and Links. The containers are derived
```

```
// from std::vector and use overloaded operator += for push_back. (See
// include/util-containers.h.)
//
typedef pVectorI<Link> Links;
typedef pVectorI<Ball> Balls;


enum Shader_Option { SO_Plain, SO_Instances, SO_True, SO_ENUM_SIZE };

class World {
public:
  World(pVulkan_Helper &fb, int argc, char **argv)
    :vh(fb), ff_state(fb.qs), raytrace(fb.qs),
     frame_timer(vh.frame_timer), transform(fb.qs),
     shapes(ff_state), sphere(ff_state),
     opt_tryout1(fb.qs.opt_tryout1),opt_tryout2(fb.qs.opt_tryout2)
  {init(argc,argv);}
  void init(int argc, char **argv);
  void init_graphics();
  void run();
  void frame_callback();
  void render(vk::CommandBuffer& cb);
  void render_objects(vk::CommandBuffer& cb, Render_Option render_option);
  void objects_erase();
  void cb_keyboard();
  void modelview_update();

  pVulkan_Helper& vh;
  VFixed_Function_State_Manager ff_state;
  VRaytrace raytrace;
  pFrame_Timer& frame_timer;
  VTransform transform;
  Shapes shapes;
  Sphere sphere;

  bool opt_want_raytrace;

  pVariable_Control variable_control;
  double world_time;
  double last_frame_wall_time;
  float opt_time_step_duration;
  int time_step_count;
  float opt_gravity_accel;      // Value chosen by user.
  pVect gravity_accel;          // Set to zero when opt_gravity is false;
  bool opt_gravity;
  bool opt_head_lock, opt_tail_lock;
  bool &opt_tryout1, &opt_tryout2;  // For ad-hoc experiments.
  float opt_tryoutf;
  bool opt_sphere_true;
  VBufferV<Shdr_Uni_Common> buf_uni_common;

  // Tiled platform for ball.
  //
  float platform_xmin, platform_xmax, platform_zmin, platform_zmax;
```

```
float platform_pi_xwidth_inv;
VVertex_Buffer_Set bset_platform_tex, bset_platform_mir;
VPipeline pipe_platform_mir, pipe_platform_tex;
RT_Geo_Package rt_geo_platform_tex, rt_geo_platform_mir;

vk::Sampler sampler;
VTexture texid_hw;
VTexture texid_syl;
VTexture texid_emacs;

void platform_update();
bool platform_collision_possible(pCoor pos);

pCoor light_location;
float opt_light_intensity;
enum { MI_Eye, MI_Light, MI_Ball, MI_Ball_V, MI_COUNT } opt_move_item;
bool opt_pause;
bool opt_single_frame;      // Simulate for one frame.
int opt_single_time_step;  // Simulate for one time step.

VBufferV<Uni_Lighting> uni_light, uni_light_ambient, uni_light_all;
VBufferV<Uni_Lighting> uni_light_reflected;
VBufferV<Uni_Lighting> *puni_light_curr;

pCoor eye_location;
pVect eye_direction;
pMatrix transform_projection;
pMatrix modelview;

pVect adj_vector;
double adj_t_prev;
double adj_t_stop;
double adj_duration;

uint opt_shader;

bool opt_mirror;
int opt_mirror_depth;
bool opt_shadows;
bool opt_shadow_volumes;      // Make shadow volumes visible.
bool opt_normal_sphere;
int opt_hide_stuff;

int last_setup; // Last scene set up.

void ball_setup_1(bool big);  // Silly Tree
void ball_setup_2(); // Hanging thing.
void ball_setup_3(); // Corona virus particle.
void ball_setup_4(); // Wheel.
void ball_setup_5(); // Top.
void ball_setup_6(); // Wheel with webbing between spokes.
void ball_setup_7();
void setup_at_end();
```

```
void time_step_cpu(double);
void balls_stop();
void balls_freeze();
void balls_translate(pVect amt, int idx);
void balls_translate(pVect amt);
void balls_push(pVect amt, int idx);
void balls_push(pVect amt);
void balls_twirl();
void lock_update();

Ball *make_marker(pCoor pos, pColor col, float radius = 0.2 );
Balls balls_marker; // Written to balls right after setup.

float opt_spring_constant, opt_spring_damp;
float opt_friction_dynamic;
float opt_air_resistance;
float distance_relaxed;
int chain_length;
Balls balls;
Ball *head_ball, *tail_ball;
Links links;
vector<Surface> surfaces;
int opt_sphere_slices;
float opt_omega;  // For disk and top.

Links link_new
( Ball *ball1, Ball *ball2,
  float stiffness = 0.2, float spring_constant = 0);
Links link_new
( Ball *ball1, Ball *ball2, Link_Args largs );

/// CUDA Stuff

int opt_physics_method;
GPU_Info gpu_info;
cudaEvent_t frame_start_ce, frame_stop_ce;
int opt_block_size;
bool gpu_const_data_stale;
bool cpu_phys_data_stale;

void data_cpu_to_gpu_constants();
void data_cpu_to_gpu_dynamic();
void data_gpu_to_cpu_dynamic();
CPU_GPU_Common c;  // c is for common.  See links.cuh.

void render_link_start();
void render_link_gather(Link *link);
void render_link_render(vk::CommandBuffer& cb, bool shadow_volumes = false);
void render_link_render_sv(vk::CommandBuffer& cb)
{ render_link_render(cb,true); };

VPipeline pipe_set_2, pipe_set_2_sv;
double world_time_link_update;
```

```
  VCompute comp_links;
  VBuffer<int> buf_links_as_idx; // Indices for links' acceleration structures.
  VBuffer<vec2> buf_links_tex_coor;
  VBuffer<vec4> buf_links_vertex_g;
  VBuffer<vec4> buf_links_normal_g;
  RT_Geo_Package rt_geo_links;

  int opt_sides, opt_segments;

  vector< VBufferVV<vec4>* > lis_bufs;
  VBufferVV<vec4> lis_pos1, lis_pos2;
  VBufferVV<vec4> lis_v1, lis_v2;
  VBufferVV<vec4> lis_b1_ydir, lis_b2_ydir, lis_tex_rad;

  VVertex_Buffer_Set bset_surfaces;
  VPipeline pipe_surfaces;
  RT_Geo_Package rt_geo_surfaces;
  void render_surfaces(vk::CommandBuffer& cb);

  /// 2025 Homework 2
  int n_coors_bad; // Number of vertices that are wrong distance from center.
  int show_n; // 0: just pentagon, 1: half of dodecahedron, 2: all.

};

enum { OH_Links = 1, OH_Platform = 2, OH_Sphere = 4 };

void
World::init_graphics()
{
  vh.want_raytrace().init();
  vh.opt_record_every_time = true;
  vh.display_cb_set([&](){frame_callback();});
  vh.cbs_cmd_record.push_back( [&](vk::CommandBuffer& cb){ render(cb); });
  sphere.ppuni_light = &puni_light_curr;
  sphere.transform = &transform;

  opt_want_raytrace = true;

  if ( vh.have_raytrace )
    {
      raytrace.init();
      for ( auto b: { &bset_platform_mir, &bset_platform_tex, &bset_surfaces } )
        b->usage_set
          ( vk::BufferUsageFlagBits::eVertexBuffer
            | vk::BufferUsageFlagBits::eStorageBuffer );
      buf_links_as_idx.init
        ( vh.qs, vk::BufferUsageFlagBits::eStorageBuffer, 0,
          vk::MemoryPropertyFlagBits::eDeviceLocal );
      buf_links_tex_coor.init
        ( vh.qs, vk::BufferUsageFlagBits::eStorageBuffer, 0,
          vk::MemoryPropertyFlagBits::eDeviceLocal );
      buf_links_vertex_g.init
```

```
      ( vh.qs, vk::BufferUsageFlagBits::eStorageBuffer, 0,
        vk::MemoryPropertyFlagBits::eDeviceLocal );
    buf_links_normal_g.init
      ( vh.qs, vk::BufferUsageFlagBits::eStorageBuffer, 0,
        vk::MemoryPropertyFlagBits::eDeviceLocal );
  }

buf_uni_common.init(vh.qs,vk::BufferUsageFlagBits::eUniformBuffer);
sphere.buf_uni_common = buf_uni_common;

lis_bufs =
  { &lis_pos1, &lis_pos2,
    &lis_v1, &lis_v2,
    &lis_b1_ydir, &lis_b2_ydir,
    &lis_tex_rad };

for ( auto b: lis_bufs )
  b->init(vh.dev_phys, vh.dev, vk::BufferUsageFlagBits::eStorageBuffer );

///
/// Graphical Model Initialization
///

opt_head_lock = false;
opt_tail_lock = false;
opt_tryout1 = opt_tryout2 = false;
opt_normal_sphere = true;
opt_sphere_true = true;
opt_tryoutf = 0;
variable_control.insert_linear(opt_tryoutf,"Tryout F",0.1);

eye_location = pCoor(24.2,11.6,-38.7);
eye_direction = pVect(-0.42,-0.09,0.9);

platform_xmin = -40; platform_xmax = 40;
platform_zmin = -40; platform_zmax = 40;

texid_syl.init
  ( vh.qs, P_Image_Read( vh.gp_root_get() / "vulkan" / "gpup.png", 255 ) );
texid_emacs.init( vh.qs, P_Image_Read("mult.png",-1) );
texid_hw.init( vh.qs, P_Image_Read("hw05-assign.png",255) );
sampler = vh.qs.dev.createSampler
  ( { {},
      vk::Filter::eLinear,  // magFilter
      vk::Filter::eLinear,  // minFilter
      vk::SamplerMipmapMode::eLinear,
      // Also: eRepeat, eMirroredRepeat, eClampToEdge, etc.
      vk::SamplerAddressMode::eRepeat,
      vk::SamplerAddressMode::eRepeat,
      vk::SamplerAddressMode::eRepeat,
      0.0f,  // mipLodBias
      true, // anisotropyEnable,
      16.0f, // maxAnisotropy
      false, vk::CompareOp::eNever,  // compareEnable, compareOp
```

```
        0.0f, VK_LOD_CLAMP_NONE,  // min and max LOD
        vk::BorderColor::eFloatOpaqueBlack,
        false // unnormalizedCoordinates
        } );

  sphere.texture_set(sampler,texid_emacs);

  opt_light_intensity = 50;
  light_location = pCoor(platform_xmax,platform_xmax,platform_zmin);
  uni_light.init(vh.qs,vk::BufferUsageFlagBits::eUniformBuffer);
  uni_light_all.init(vh.qs,vk::BufferUsageFlagBits::eUniformBuffer);
  uni_light_reflected.init(vh.qs,vk::BufferUsageFlagBits::eUniformBuffer);
  uni_light_ambient.init(vh.qs,vk::BufferUsageFlagBits::eUniformBuffer);

  variable_control.insert(opt_light_intensity,"Light Intensity");

  opt_move_item = MI_Eye;
  opt_pause = false;
  opt_single_time_step = 0;
  opt_single_frame = false;

  platform_update();
  modelview_update();

  adj_t_stop = 0;
  adj_duration = 0.25;

  opt_sides = 20;
  opt_segments = 16;
  variable_control.insert(opt_sides,"Link Sides");
  variable_control.insert(opt_segments,"Link Segments",5,1);
  opt_hide_stuff = 0;

  show_n = 2;

  if ( vh.have_raytrace )
    {
      raytrace.buf_light_set(uni_light_all);
    }
}

void
World::run()
{
  vh.message_loop_spin();

  uni_light.destroy();
  uni_light_ambient.destroy();
  uni_light_all.destroy();
  uni_light_reflected.destroy();
  transform.destroy();
  buf_uni_common.destroy();

  for ( auto b: lis_bufs ) b->destroy();
```

```
  pipe_platform_tex.destroy(); pipe_platform_mir.destroy();
  bset_platform_tex.destroy(); bset_platform_mir.destroy();
  pipe_set_2.destroy();
  pipe_set_2_sv.destroy();

  texid_syl.destroy();
  texid_emacs.destroy();
  texid_hw.destroy();
  vh.dev.destroySampler(sampler);
  shapes.destroy();
  sphere.destroy();

  bset_surfaces.destroy();
  pipe_surfaces.destroy();
  rt_geo_surfaces.destroy();

  buf_links_as_idx.destroy();
  buf_links_tex_coor.destroy();
  buf_links_normal_g.destroy();
  buf_links_vertex_g.destroy();
  rt_geo_platform_tex.destroy();
  rt_geo_platform_mir.destroy();
  rt_geo_links.destroy();
  comp_links.destroy();
  raytrace.destroy();

  vh.finish();
}

void
World::render_surfaces(vk::CommandBuffer& cb)
{
  if ( !pipe_surfaces )
    pipe_surfaces
      .init( vh.qs )
      .ds_follow( puni_light_curr )
      .ds_follow( transform )
      .ds_uniform_use( "BIND_UNI_COMMON", buf_uni_common )
      .topology_set( vk::PrimitiveTopology::eTriangleList )
      .create();

  bset_surfaces.reset( pipe_surfaces );

  for ( Surface& s: surfaces )
    {
      Ball &b0 = *s.b0, &b1 = *s.b1, &b2 = *s.b2;
      pNorm snorm = cross(b0.position,b1.position,b2.position);
      bset_surfaces << b0.position << b1.position << b2.position;
      bset_surfaces << s.color << s.color << s.color;
      bset_surfaces << snorm << snorm << snorm;
    }

  bset_surfaces.to_dev();
```

```
  if ( vh.have_raytrace && !rt_geo_surfaces )
    rt_geo_surfaces.init( raytrace );

  if ( vh.raytrace_active() )
    {
      rt_geo_surfaces.bind( bset_surfaces, RT_Geometry_Triangles );
      return;
    }

  pipe_surfaces.record_draw( cb, bset_surfaces );
}

void
World::platform_update()
{
  const float tile_count = 19;
  const float ep = 1.00001;
  const float delta_x = ( platform_xmax - platform_xmin ) / tile_count * ep;
  const float zdelta = ( platform_zmax - platform_zmin ) / tile_count * ep;

  const float trmin = 0.05;
  const float trmax = 0.7;
  const float tsmin = 0;
  const float tsmax = 0.4;
  pColor color_platform( color_white * 0.9 );

  if ( !pipe_platform_mir )
    pipe_platform_mir
      .init( ff_state )
      .ds_set( color_lsu_spirit_purple )
      .ds_follow( puni_light_curr )
      .ds_follow( transform )
      .topology_set( vk::PrimitiveTopology::eTriangleList )
      .create();
  bset_platform_mir.reset( pipe_platform_mir ).tolerate_tcoords();

  if ( !pipe_platform_tex )
    pipe_platform_tex
      .init( ff_state )
      .ds_set_material(color_platform)
      .ds_follow( puni_light_curr )
      .ds_follow( transform )
      .topology_set( vk::PrimitiveTopology::eTriangleList )
      .ds_use( sampler, texid_syl )
      .create();
  bset_platform_tex.reset( pipe_platform_tex );

  const pVect platform_normal(0,1,0);
  pTCoor t00(trmin,tsmin), t01(trmin,tsmax), t10(trmax,tsmin), t11(trmax,tsmax);
  bool even = true;

  for ( int i = 0; i < tile_count; i++ )
    {
```

```
      const float x0 = platform_xmin + i * delta_x;
      const float x1 = x0 + delta_x;
      const float y = 0;
      for ( float z = platform_zmin; z < platform_zmax; z += zdelta )
        {
          auto& bset = even ? bset_platform_tex : bset_platform_mir;
          const float z1 = z+zdelta;
          pCoor p00(x0,y,z), p01(x0,y,z1), p10(x1,y,z), p11(x1,y,z1);

          bset << t11 << platform_normal << p00;
          bset << t10 << platform_normal << p01;
          bset << t00 << platform_normal << p11;

          bset << t11 << platform_normal << p00;
          bset << t00 << platform_normal << p11;
          bset << t01 << platform_normal << p10;

          even = !even;
        }
    }

  bset_platform_mir.to_dev();
  bset_platform_tex.to_dev();

  if ( vh.have_raytrace && !rt_geo_platform_mir )
    {
      pColor mirror_color(color_lsu_spirit_purple);
      mirror_color.a = -0.5;
      rt_geo_platform_mir
        .init( raytrace )
        .color_front_set( mirror_color )
        .bind( bset_platform_mir, RT_Geometry_Triangles );
      rt_geo_platform_tex
        .init( raytrace )
        .color_front_set( color_platform )
        .bind( VR_BUF_IDX_sampler, sampler, texid_syl )
        .bind( bset_platform_tex, RT_Geometry_Triangles );
    }
}

void
World::modelview_update()
{
  pMatrix_Translate center_eye(-eye_location);
  pMatrix_Rotation rotate_eye(eye_direction,pVect(0,0,-1));
  modelview = rotate_eye * center_eye;
}

void
World::render_objects(vk::CommandBuffer& cb, Render_Option option)
{
  //  const float shininess_ball = 5;
  pColor spec_color(1,1,1);
  //  pColor spec_light = color_white * opt_light_intensity;
```

```
const bool hide_links = opt_hide_stuff & OH_Links;
const bool hide_platform = opt_hide_stuff & OH_Platform;
const bool hide_sphere = opt_hide_stuff & OH_Sphere;

sphere.init(opt_sphere_slices);

if ( option == RO_Shadow_Volumes )
  {
    if ( !hide_sphere )
      {
        sphere.light_pos = light_location;
        sphere.render_bunch_gather_sv(world_time);
        for ( Ball *ball: balls )
          sphere.render_shadow_volume(cb,ball->radius,ball->position);

        sphere.render_bunch_render_sv(cb);
      }

    if ( !hide_links )
      {
        render_link_start();
        for ( Link *link: links )
          if ( link->is_renderable ) render_link_gather(link);
        render_link_render_sv(cb);
      }
  }
else
  {
    sphere.rt_geo_sphere.rt_wanted_set(!hide_sphere);
    if ( !hide_sphere )
      {
        sphere.render_bunch_gather(world_time);
        for ( Ball *ball: balls )
          {
            if ( ball->contact )
              sphere.color = color_gray;
            else if ( ball->mass > 0 && ball->mass < ball->mass_min )
              sphere.color = color_red;
            else if ( ball->mass > 0 && ball->locked )
              sphere.color = color_pale_green;
            else
              sphere.color = ball->color;

            sphere.color.a = ball->specularity ? -ball->specularity : 1;
            sphere.render
              (cb, ball->radius,ball->position,
               pMatrix_Rotation(ball->orientation));
          }

        sphere.render_bunch_render(cb, opt_sphere_true);
      }

    rt_geo_links.rt_wanted_set(!hide_links);
```

```
      if ( !hide_links )
        {
          render_link_start();
          for ( Link *link: links )
            if ( link->is_renderable ) render_link_gather(link);
          render_link_render(cb);
        }
    }

  rt_geo_platform_tex.rt_wanted_set(!hide_platform);
  rt_geo_platform_mir.rt_wanted_set(!hide_platform);

  if ( option != RO_Shadow_Volumes )
    render_surfaces(cb);

  if ( vh.raytrace_active() ) return;
  if ( hide_platform ) return;

  if ( option == RO_Shadow_Volumes ) return;

  //
  // Render Platform
  //
  pipe_platform_tex.record_draw(cb,bset_platform_tex);
}

void
World::render(vk::CommandBuffer& cb)
{
  const bool do_raytrace = vh.raytrace_active();

  /// Emit a Graphical Representation of Simulation State
  //

  // Understanding of the code below not required for introductory
  // lectures.

  // That said, much of the complexity of the code is to show
  // the ball shadow and reflection.


  const int win_width = vh.get_width();
  const int win_height = vh.get_height();
  const float aspect = float(win_width) / win_height;
  const float n_dist = 0.01;
  const float xr = .8 * n_dist;

  transform.eye_from_global_set( modelview );
  transform.clip_from_eye_set
    ( pMatrix_Frustum(-xr,xr,-xr/aspect,xr/aspect,n_dist,5000) );
  // pMatrix_Frustum: left, right, bottom, top, near, far

  vh.clearValues[0].color =
    vk::ClearColorValue( array<float, 4>( { { 0, 0, 0, 0.5 } } ) );
```

```
  /// Lighting
  //
  auto& light_all = uni_light_all->cgl_LightSource[0];

  light_all.position = transform.eye_from_global * light_location;

  light_all.constantAttenuation = 0.5;
  light_all.linearAttenuation = 1.0;
  light_all.quadraticAttenuation = 0;

  pColor ambient_color(0x555555);

  uni_light_all->cgl_LightModel.ambient = ambient_color;

  light_all.diffuse = opt_light_intensity * color_white;
  light_all.ambient = color_black;
  light_all.specular = opt_light_intensity * color_white;

  uni_light_reflected = uni_light_all.get_val();
  uni_light_reflected->cgl_LightSource[0].position =
    transform.eye_from_global * pMatrix_Scale(1,-1,1) * light_location;

  uni_light_ambient = uni_light_all.get_val();
  auto& light_amb = uni_light_ambient->cgl_LightSource[0];
  light_amb.diffuse = color_black;
  light_amb.specular = color_black;

  uni_light = uni_light_all.get_val();
  uni_light->cgl_LightModel.ambient = color_black;

  uni_light.to_dev();
  uni_light_ambient.to_dev();
  uni_light_all.to_dev();
  uni_light_reflected.to_dev();

  puni_light_curr = &uni_light_all;

  buf_uni_common->tryout =
    ivec4(opt_tryout1,opt_tryout2,opt_normal_sphere,false);
  buf_uni_common->tryoutf = vec4(opt_tryoutf, 0, 0, 0);
  buf_uni_common->light_pos_g = light_location;
  if ( do_raytrace )
    {
      raytrace.buf_uni_common->opt_tryout = buf_uni_common->tryout;
      raytrace.buf_uni_common->opt_tryoutf = buf_uni_common->tryoutf;
      raytrace.buf_uni_common->opt_shadows = opt_shadows;
      raytrace.buf_uni_common->opt_mirror = opt_mirror;
      raytrace.buf_light_set(uni_light_all);
    }

  const double time_now = time_wall_fp();
  const bool blink_visible = int64_t(time_now*3) & 1;
# define BLINK(txt,pad) ( blink_visible ? txt : pad )
```

```
  vh.fbprintf("%s\n",frame_timer.frame_rate_text_get());

  vh.fbprintf
    ("Raytrace: %s ('R')   Compiled: %s\n",
     !vh.have_raytrace ? "NA" :
     opt_want_raytrace ? ( raytrace.rt_ready ? "ON " : "???" ) : "OFF",
#ifdef __OPTIMIZE__
     "WITH OPTIMIZATION"
#else
     BLINK("WITHOUT OPTIMIZATION","")
#endif
     );

  vh.fbprintf
    ("Time Step: %8d  World Time: %11.6f  %s\n",
     time_step_count, world_time,
     opt_pause ? BLINK("PAUSED, 'p' to unpause, SPC or S-SPC to step.","") :
     "Press 'p' to pause."
     );

  vh.fbprintf
    ("Eye location: [%5.1f, %5.1f, %5.1f]  "
     "Eye direction: [%+.2f, %+.2f, %+.2f]\n",
     eye_location.x, eye_location.y, eye_location.z,
     eye_direction.x, eye_direction.y, eye_direction.z);

  Ball& ball = *balls[0];

  vh.fbprintf
    ("Head Ball Pos  [%5.1f,%5.1f,%5.1f] Vel [%+5.1f,%+5.1f,%+5.1f]\n",
     ball.position.x,ball.position.y,ball.position.z,
     ball.velocity.x,ball.velocity.y,ball.velocity.z );

  vh.fbprintf
    ("Hide: %c%c%c ('!@#')  Effect: %c%c ('or')  Sphere: %s  ('z')  "
     "Tryout 1: %s  ('y')  Tryout 2: %s  ('Y')  Normals: %s ('n')\n",
     opt_hide_stuff & OH_Sphere ? 'S' : '_',
     opt_hide_stuff & OH_Links ? 'L' : '_',
     opt_hide_stuff & OH_Platform ? 'P' : '_',
     opt_shadows ? 'S' : '_',
     opt_mirror ? 'M' : '_',
     opt_sphere_true ? "TRUE" : "TRI" ,
     opt_tryout1 ? BLINK("ON ","   ") : "OFF",
     opt_tryout2 ? BLINK("ON ","   ") : "OFF",
     opt_normal_sphere ? "SPHERE" : "TRI");

  size_t num_vis_links = ranges::count( links, true, &Link::is_renderable );
  pStringF bad_coords("%2d Bad Coords", n_coors_bad);
  vh.fbprintf
    ("Show %s ('789')  "
     "%4zd Balls  %4zd Links   %s\n",
     last_setup != 7 ? "             " :
     show_n == 0 ? BLINK("PENTAGON ONLY", "             ") :
```

```
    show_n == 1 ? BLINK("HALF DODEC   ", "             ") :
    show_n == 2 ? "EVERYTHING   " : "",
    balls.size(), num_vis_links,
    last_setup == 7 && n_coors_bad
    ? BLINK(bad_coords.s,"             ") : bad_coords.s );

  pVariable_Control_Elt* const cvar = variable_control.current;
  vh.fbprintf("VAR %s = %.5f  (TAB or SH-TAB to change, +/- to adjust)\n",
                      cvar->name,cvar->get_val());

  // Render Marker for Light Source
  //
  shapes.record_tetrahedron(cb,transform,light_location,0.15);

  //
  // Render ball reflection.  (Will be blended with dark tiles.)
  //

  const bool hide_platform = opt_hide_stuff & OH_Platform;

  if ( do_raytrace )
    {
      raytrace.tform_update(transform);
      render_objects(cb,RO_Normally);
      raytrace.buf_uni_common.to_dev();

      return;
    }

  buf_uni_common.to_dev();

  vk::ColorComponentFlags clr_all
    ( vk::ColorComponentFlagBits::eR | vk::ColorComponentFlagBits::eG |
      vk::ColorComponentFlagBits::eB | vk::ColorComponentFlagBits::eA );

  if ( !hide_platform )
    {
      // Save current state of fixed-functionality state.
      //
      ff_state.push();

      if ( opt_mirror )
        {
          /// Mirror Effect
          //
          //  - Step 1:  Write location of mirrors into stencil buffer.
          //
          //  - Step 2a: Prepare a matrix that transforms to mirrored location.
          //  - Step 2b: Set up blending to blend mirrored objects with mirrors.
          //
          //  - Step 3:  Render all objects except mirrors.

          ///  Step 1: Write location of mirrors into stencil buffer.
          //
```

```
  //   Write stencil value 2 at location of dark (mirrored) tiles.
  //

  vk::StencilOpState s_op_apply
    ( vk::StencilOp::eReplace, vk::StencilOp::eKeep,
      vk::StencilOp::eKeep, vk::CompareOp::eNever, 2, 2, 2 );
  // sfail, spass, dfail, compare;  cmask, wmask, ref

  ff_state.pipelineDepthStencilStateCreateInfo_get()
    .setStencilTestEnable(true)
    .setFront(s_op_apply).setBack(s_op_apply);

  // Render mirrored tiles (but only to write stencil buffer).
  //
  pipe_platform_mir.record_draw(cb,bset_platform_mir);

  cb.nextSubpass(vk::SubpassContents::eInline); vh.qs.subpass++;

  // Prepare to write only stenciled locations.
  //
  vk::StencilOpState s_op_use
    ( vk::StencilOp::eKeep, vk::StencilOp::eKeep,
      vk::StencilOp::eKeep, vk::CompareOp::eEqual, 2, 2, 2 );
  // sfail, spass, dfail, compare;  cmask, wmask, ref
  //
  // Stencil test passes if stencil value = 2.
  // Don't change stencil buffer.

  ff_state.pipelineDepthStencilStateCreateInfo_get()
    .setFront(s_op_use).setBack(s_op_use);

  ///   Step 2a: Prepare a matrix that transforms to mirrored location.
  //    Use a transform that reflects objects to other side of platform.
  //
  const pMatrix prev = transform.eye_from_global;
  const pMatrix_Scale reflect(1,-1,1);
  transform.eye_from_global = prev * reflect;

  // Reflected front face should still be treated as the front face.
  //
  ff_state.pipelineRasterizationStateCreateInfo_get()
    .setFrontFace(vk::FrontFace::eClockwise);

  puni_light_curr = &uni_light_reflected;

  /// Step 3: Render all objects.

  render_objects(cb,RO_Mirrored);

  transform.eye_from_global = prev;

  ff_state.pipelineRasterizationStateCreateInfo_get()
    .setFrontFace(vk::FrontFace::eCounterClockwise);
}
```

```
    // blend operation, blend factors, blend constants
    ff_state.pipelineColorBlendStateCreateInfo_get()
      .setBlendConstants( array<float,4>{.5,.5,.5,.5} );

    ff_state.pipelineColorBlendAttachmentState_get() =
      vk::PipelineColorBlendAttachmentState
      ( true,                    // blendEnable
        vk::BlendFactor::eConstantAlpha,  // srcColorBlendFactor
        vk::BlendFactor::eConstantAlpha,  // dstColorBlendFactor
        vk::BlendOp::eAdd,       // colorBlendOp
        vk::BlendFactor::eOne,   // srcAlphaBlendFactor
        vk::BlendFactor::eZero,  // dstAlphaBlendFactor
        vk::BlendOp::eAdd,       // alphaBlendOp
        clr_all      // colorWriteMask
        );

    puni_light_curr = &uni_light_all;;
    pipe_platform_mir.record_draw(cb,bset_platform_mir);

    ff_state.pop();
  }

if ( !opt_shadows )
  {
    // Render.
    //
    render_objects(cb,RO_Normally);
  }
else
  {
    //
    // First pass, render using only ambient light.
    //
    puni_light_curr = &uni_light_ambient;

    // Send balls, tiles, and platform to opengl.
    //
    render_objects(cb,RO_Normally);

    //
    // Second pass, find un-shadowed areas and add on light0.
    //


    /// Write Shadow Volumes
    //
    vk::ClearAttachment clr_att
      ( vk::ImageAspectFlagBits::eStencil, 0,
        vk::ClearDepthStencilValue(0,0) );
    vk::ClearRect clr_rect( vk::Rect2D({}, vh.s_extent), 0, 1 );
    cb.clearAttachments( clr_att, clr_rect );

    ff_state.push();
```

```
vk::StencilOpState s_op_apply_f
  ( vk::StencilOp::eKeep, vk::StencilOp::eIncrementAndWrap,
    vk::StencilOp::eKeep, vk::CompareOp::eAlways, -1, -1, 0 );
vk::StencilOpState s_op_apply_b
  ( vk::StencilOp::eKeep, vk::StencilOp::eDecrementAndWrap,
    vk::StencilOp::eKeep, vk::CompareOp::eAlways, -1, -1, 0 );
// sfail, spass, dfail, compare; c-mask, w-mask, ref
// ref is -1 (all 1's) because we want to operate on all bits in
// stencil value.

ff_state.pipelineDepthStencilStateCreateInfo_get()
  .setStencilTestEnable(true)
  .setDepthWriteEnable(false)
  .setFront(s_op_apply_f).setBack(s_op_apply_b);

if ( !opt_shadow_volumes )
  {
    ff_state.pipelineColorBlendAttachmentState_get()
      .setColorWriteMask( {} );
  }
else
  {
    ff_state.pipelineColorBlendAttachmentState_get() =
      vk::PipelineColorBlendAttachmentState
      ( true,                   // blendEnable
        vk::BlendFactor::eConstantAlpha,  // srcColorBlendFactor
        vk::BlendFactor::eConstantAlpha,  // dstColorBlendFactor
        vk::BlendOp::eAdd,        // colorBlendOp
        vk::BlendFactor::eOne,    // srcAlphaBlendFactor
        vk::BlendFactor::eZero,   // dstAlphaBlendFactor
        vk::BlendOp::eAdd,        // alphaBlendOp
        clr_all      // colorWriteMask
        );

    ff_state.pipelineColorBlendStateCreateInfo_get()
      .setBlendConstants( array<float,4>{.5,.5,.5,.5} );

    ff_state.pipelineDepthStencilStateCreateInfo_get()
      .setDepthWriteEnable(true);
  }

// Write stencil with shadow locations based on shadow volumes
// cast by light0 (light_location).  Shadowed locations will
// have a positive stencil value.
//
render_objects(cb,RO_Shadow_Volumes);

cb.nextSubpass(vk::SubpassContents::eInline); vh.qs.subpass++;

// Use stencil test to prevent writes to shadowed areas.
//
vk::StencilOpState s_op_apply_fb
  ( vk::StencilOp::eKeep, vk::StencilOp::eKeep,
```

```
          vk::StencilOp::eKeep, vk::CompareOp::eEqual, -1, -1, 0 );
      // sfail, spass, dfail, compare;  c-mask, w-mask, ref
      //
      // Stencil test passes if stencil value == 0.

      ff_state.pipelineDepthStencilStateCreateInfo_get()
        .setFront(s_op_apply_fb).setBack(s_op_apply_fb)
        .setDepthCompareOp(vk::CompareOp::eEqual);

      // Allow pixels to be re-written.
      //
      vk::ColorComponentFlags cmask_all
        ( vk::ColorComponentFlagBits::eR | vk::ColorComponentFlagBits::eG |
          vk::ColorComponentFlagBits::eB | vk::ColorComponentFlagBits::eA );

      ff_state.pipelineColorBlendAttachmentState_get() =
        vk::PipelineColorBlendAttachmentState
        ( true,                   // blendEnable
          vk::BlendFactor::eOne,  // srcColorBlendFactor
          vk::BlendFactor::eOne,  // dstColorBlendFactor
          vk::BlendOp::eAdd,       // colorBlendOp
          vk::BlendFactor::eOne,   // srcAlphaBlendFactor
          vk::BlendFactor::eOne,  // dstAlphaBlendFactor
          vk::BlendOp::eAdd,       // alphaBlendOp
          cmask_all      // colorWriteMask
          );

      puni_light_curr = &uni_light;

      // Render.
      //
      render_objects(cb,RO_Normally);

      ff_state.pop();
    }

  puni_light_curr = &uni_light_all;
}


void
World::cb_keyboard()
{
  const int key = vh.keyboard_key_get();
  if ( !key ) return;
  pVect adjustment(0,0,0);
  pVect user_rot_axis(0,0,0);
  const bool kb_mod_s = vh.keyboard_shift;
  const bool kb_mod_c = vh.keyboard_control;
  const float move_amt = kb_mod_s ? 2.0 : kb_mod_c ? 0.08 : 0.4;

  switch ( key ) {
  case FB_KEY_LEFT: adjustment.x = -move_amt; break;
  case FB_KEY_RIGHT: adjustment.x = move_amt; break;
```

```
  case FB_KEY_PAGE_UP: adjustment.y = move_amt; break;
  case FB_KEY_PAGE_DOWN: adjustment.y = -move_amt; break;
  case FB_KEY_DOWN: adjustment.z = move_amt; break;
  case FB_KEY_UP: adjustment.z = -move_amt; break;
  case FB_KEY_DELETE: user_rot_axis.y = 1; break;
  case FB_KEY_INSERT: user_rot_axis.y =  -1; break;
  case FB_KEY_HOME:
    if ( opt_move_item == MI_Light ) adjustment.x = 1; else user_rot_axis.x = 1;
    break;
  case FB_KEY_END: user_rot_axis.x = -1; break;
  case '1': ball_setup_1(kb_mod_c); break;
  case '2': ball_setup_2(); break;
  case '3': ball_setup_3(); break;
  case '4': ball_setup_4(); break;
  case '5': ball_setup_5(); break;
  case '6': ball_setup_6(); break;
  case '7': show_n=2;ball_setup_7(); break;
  case '8': show_n=0;ball_setup_7(); break;
  case '9': show_n=1;ball_setup_7(); break;
  case 'a':
    opt_physics_method++;
    if ( opt_physics_method == GP_ENUM_SIZE ) opt_physics_method = GP_cpu;
    break;
  case 'b': opt_move_item = MI_Ball; break;
  case 'B': opt_move_item = MI_Ball_V; break;
  case 'e': case 'E': opt_move_item = MI_Eye; break;
  case 'g': case 'G': opt_gravity = !opt_gravity; break;
  case 'h': case 'H': opt_head_lock = !opt_head_lock; break;
  case '!': opt_hide_stuff ^= OH_Sphere; break;
  case '@': opt_hide_stuff ^= OH_Links; break;
  case '#': opt_hide_stuff ^= OH_Platform; break;
  case 't': case 'T': opt_tail_lock = !opt_tail_lock; break;
  case 'l': case 'L': opt_move_item = MI_Light; break;
  case 'r': opt_mirror = !opt_mirror; break;
  case 'R':
    opt_want_raytrace = !opt_want_raytrace;
    world_time_link_update = 0;
    break;
  case 'o': opt_shadows = !opt_shadows; break;
  case 'O':
    opt_shadow_volumes = !opt_shadow_volumes;
    if ( opt_shadow_volumes ) opt_shadows = true;
    break;
  case 'n': case 'N': opt_normal_sphere = !opt_normal_sphere; break;
  case 'p': case 'P': opt_pause = !opt_pause; break;
  case 's': case 'S': balls_stop(); break;
  case 'v': case 'V': opt_shader++;
    if ( opt_shader >= sizeof(sh_names)/sizeof(sh_names[0]) ) opt_shader = 0;
    break;
  case 'w': case 'W': balls_twirl(); break;
  case 'y': opt_tryout1 = !opt_tryout1; break;
  case 'Y': opt_tryout2 = !opt_tryout2; break;
  case 'z': opt_sphere_true = !opt_sphere_true; break;
  case ' ':
```

```
  if ( kb_mod_s && kb_mod_c ) opt_single_time_step = 10;
  else if ( kb_mod_s ) opt_single_time_step = 100;
  else if ( kb_mod_c ) opt_single_time_step = 1; else opt_single_frame = true;
  opt_pause = true;
  break;
case FB_KEY_TAB:
  if ( !kb_mod_s ) { variable_control.switch_var_right(); break; }
case 96: variable_control.switch_var_left(); break;
case '-':case '_': variable_control.adjust_lower(); break;
case '+':case '=': variable_control.adjust_higher(); break;
default: printf("Unknown key, %d\n",key); break;
}

gravity_accel.y = opt_gravity ? -opt_gravity_accel : 0;

opt_mirror_depth = raytrace.recursion_max_set( opt_mirror_depth + 1 ) - 1;
lock_update();
sphere.bunch_invalidate();

// Update eye_direction based on keyboard command.
//
if ( user_rot_axis.x || user_rot_axis.y )
  {
    pMatrix_Rotation rotall(eye_direction,pVect(0,0,-1));
    user_rot_axis *= invert(rotall);
    eye_direction *= pMatrix_Rotation(user_rot_axis, numbers::pi * 0.03);
    modelview_update();
  }

// Update eye_location based on keyboard command.
//
if ( adjustment.x || adjustment.y || adjustment.z )
  {
    const double angle =
      fabs(eye_direction.y) > 0.99
      ? 0 : atan2(eye_direction.x,-eye_direction.z);
    pMatrix_Rotation rotall(pVect(0,1,0),-angle);
    adjustment *= rotall;

    switch ( opt_move_item ){
    case MI_Ball:
      adj_vector = adjustment;
      if ( adj_t_stop == 0 )
        adj_t_prev = adj_t_stop = world_time;
      adj_t_stop += adj_duration;
      break;
    case MI_Ball_V: balls_push(adjustment,0); break;
    case MI_Light:
      if ( key == FB_KEY_HOME )
        {
          pVect l_to_b(light_location,balls[0]->position);
          adjustment = 0.25 * l_to_b;
        }
      light_location += adjustment;
```

```
        break;
      case MI_Eye: eye_location += adjustment; break;
      default: break;
      }
      modelview_update();
    }

}




void
World::init(int argc, char **argv)
{
  chain_length = 14;

  opt_time_step_duration = 0.00003;
  variable_control.insert(opt_time_step_duration,"Time Step Duration");

  distance_relaxed = 15.0 / chain_length;
  opt_spring_constant = 500;
  variable_control.insert(opt_spring_constant,"Spring Constant");

  opt_spring_damp = 0.7;
  variable_control.insert(opt_spring_damp,"Spring Damping").set_max(1);

  opt_friction_dynamic = 0.2;
  variable_control.insert(opt_friction_dynamic,"Dynamic Friction");

  opt_omega = 10;
  variable_control.insert(opt_omega,"Initial Rotation Rate");

  opt_gravity_accel = 9.8;
  opt_gravity = true;
  gravity_accel = pVect(0,-opt_gravity_accel,0);
  variable_control.insert(opt_gravity_accel,"Gravity");

  opt_air_resistance = 0.01;
  variable_control.insert(opt_air_resistance,"Air Resistance");

  world_time = 0;
  time_step_count = 0;
  last_frame_wall_time = time_wall_fp();
  frame_timer.work_unit_set("Steps / s");

  world_time_link_update = 0;

  opt_mirror = true;
  opt_mirror_depth = 2;
  variable_control.insert(opt_mirror_depth,"Mirror Depth",1,0);
  opt_shadows = true;
  opt_shadow_volumes = false;
  opt_shader = SO_Plain;
```

```
/// Init CUDA

// Get information about GPU and its ability to run CUDA.
//
int device_count;
cudaGetDeviceCount(&device_count); // Get number of GPUs.
if ( device_count == 0 )
  {
    fprintf(stderr,"No GPU found, exiting.\n");
    exit(1);
  }

// Print information about the available GPUs.
//
gpu_info_print();

// Use course library routine to choose a good GPU for CUDA.
//
int dev = gpu_choose_index();
CE(cudaSetDevice(dev));
printf("Using GPU %d\n",dev);

gpu_info.get_gpu_info(dev);
cuda_setup(&gpu_info);

// Print information about time_step routine.
//
printf("\nCUDA Routine Resource Usage:\n");

int block_size_max = 4096;

for ( int i=0; i<gpu_info.num_kernels; i++ )
  {
    printf("For %s:\n", gpu_info.ki[i].name);
    printf("  %6zd shared, %zd const, %zd loc, %d regs; "
           "%d max threads per block.\n",
           gpu_info.ki[i].cfa.sharedSizeBytes,
           gpu_info.ki[i].cfa.constSizeBytes,
           gpu_info.ki[i].cfa.localSizeBytes,
           gpu_info.ki[i].cfa.numRegs,
           gpu_info.ki[i].cfa.maxThreadsPerBlock);
    set_min(block_size_max,gpu_info.ki[i].cfa.maxThreadsPerBlock);
  }

opt_physics_method = GP_cuda;
gpu_const_data_stale = true;
cpu_phys_data_stale = false;

CE( cudaEventCreate(&frame_start_ce) );
CE( cudaEventCreate(&frame_stop_ce) );

opt_block_size = 32;
```

```
variable_control.insert_power_of_2
  (opt_block_size, "Block Size", 32, block_size_max);

c.alloc_n_balls = 0;
c.alloc_n_links = 0;

init_graphics();

opt_sphere_slices = 35;
sphere.opt_texture = true;
sphere.init(opt_sphere_slices);
variable_control.insert(opt_sphere_slices,"Sphere Slices");

const char* const links_shader_code_path = "links-shdr-links.cc";

const auto bvec_vs = vk::ShaderStageFlagBits::eVertex;
const auto bvec_vgs =
  vk::ShaderStageFlagBits::eVertex | vk::ShaderStageFlagBits::eGeometry;

pipe_set_2
  .init( ff_state )
  .ds_follow( transform )
  .ds_follow( puni_light_curr )
  .ds_uniform_use( "BIND_UNI_COMMON", buf_uni_common )
  .ds_set_material( color_dim_gray, color_salmon )
  .ds_use( sampler, texid_hw )
  .ds_storage_follow( "BIND_LINKS_POS1", lis_pos1, bvec_vs )
  .ds_storage_follow( "BIND_LINKS_POS2", lis_pos2, bvec_vs )
  .ds_storage_follow( "BIND_LINKS_V1", lis_v1, bvec_vs )
  .ds_storage_follow( "BIND_LINKS_V2", lis_v2, bvec_vs )
  .ds_storage_follow( "BIND_LINKS_B1_YDIR", lis_b1_ydir, bvec_vs )
  .ds_storage_follow( "BIND_LINKS_B2_YDIR", lis_b2_ydir, bvec_vs )
  .ds_storage_follow( "BIND_LINKS_TEX_RAD", lis_tex_rad, bvec_vgs )
  .shader_inputs_info_set( )
  .shader_code_set
  (links_shader_code_path,
   "vs_main_2(); ",     // Name of vertex shader main routine.
   "gs_main_2();",      // Name of geometry shader main routine.
   "fs_main();"         // Name of fragment shader main routine.
   )
  .topology_set( vk::PrimitiveTopology::eLineStrip )
  .create();

if ( vh.have_raytrace )
  comp_links
    .init( vh.qs )
    .ds_uniform_use( "BIND_UNI_COMMON", buf_uni_common )
    .ds_uniform_use( "BIND_TRANSFORM", raytrace.buf_uni_transform )
    .ds_storage_follow( "BIND_LINKS_POS1", lis_pos1 )
    .ds_storage_follow( "BIND_LINKS_POS2", lis_pos2 )
    .ds_storage_follow( "BIND_LINKS_V1", lis_v1 )
    .ds_storage_follow( "BIND_LINKS_V2", lis_v2 )
    .ds_storage_follow( "BIND_LINKS_B1_YDIR", lis_b1_ydir )
    .ds_storage_follow( "BIND_LINKS_B2_YDIR", lis_b2_ydir )
```

```
      .ds_storage_follow( "BIND_LINKS_TEX_RAD", lis_tex_rad )
      .ds_storage_follow( "BIND_LINKS_AS_IDX", buf_links_as_idx )
      .ds_storage_follow( "BIND_LINKS_TEX_COOR", buf_links_tex_coor )
      .ds_storage_follow( "BIND_LINKS_VERTEX_G", buf_links_vertex_g )
      .ds_storage_follow( "BIND_LINKS_NORMAL_G", buf_links_normal_g )
      .shader_code_set(links_shader_code_path, "comp_main(); ")
      .create();

  pipe_set_2_sv
    .init( ff_state ).ds_use( uni_light )
    .ds_follow( transform )
    .ds_uniform_use( "BIND_UNI_COMMON", buf_uni_common )
    .ds_storage_follow( "BIND_LINKS_POS1", lis_pos1 )
    .ds_storage_follow( "BIND_LINKS_POS2", lis_pos2 )
    .ds_storage_follow( "BIND_LINKS_V1", lis_v1 )
    .ds_storage_follow( "BIND_LINKS_V2", lis_v2 )
    .ds_storage_follow( "BIND_LINKS_B1_YDIR", lis_b1_ydir )
    .ds_storage_follow( "BIND_LINKS_B2_YDIR", lis_b2_ydir )
    .ds_storage_follow( "BIND_LINKS_TEX_RAD", lis_tex_rad )
    .shader_inputs_info_set( )
    .shader_code_set
    (links_shader_code_path,
     "vs_main_2_sv(); ",     // Name of vertex shader main routine.
     "gs_main_2_sv();",      // Name of geometry shader main routine.
     "fs_main_sv();"         // Name of fragment shader main routine.
     )
    .topology_set( vk::PrimitiveTopology::eLineStrip )
    .create();

  ball_setup_7();
  return;
  // For benchmarking.
  opt_pause = true;
  opt_hide_stuff = OH_Links | OH_Platform;
  opt_shadows = false;
  opt_mirror = false;
}

Ball*
World::make_marker(pCoor position, pColor color, float radius )
{
  Ball* const ball = new Ball;
  ball->position = position;
  ball->locked = true;
  ball->velocity = pVect(0,0,0);
  ball->radius = radius;
  ball->mass = -1;
  ball->contact = false;
  ball->color = color;
  balls_marker.push_back(ball);
  return ball;
}

void
```

```
World::lock_update()
{
  // This routine called when options like opt_head_lock might have
  // changed.

  // Update locked status.
  //
  if ( head_ball ) head_ball->locked = opt_head_lock;
  if ( tail_ball ) tail_ball->locked = opt_tail_lock;

  // Re-compute minimum mass needed for stability.
  //
  for ( Ball *ball: balls ) ball->spring_constant_sum = 0;
  const double dtis = pow( opt_time_step_duration, 2 );
  for ( Link *link: links )
    {
      Ball* const b1 = link->ball1;
      Ball* const b2 = link->ball2;
      b1->spring_constant_sum += opt_spring_constant;
      b2->spring_constant_sum += opt_spring_constant;
    }
  for ( Ball *ball: balls )
    {
      ball->mass_min = ball->spring_constant_sum * dtis;
      ball->constants_update();
    }
  gpu_const_data_stale = true;
}

void
World::balls_twirl()
{
  if ( !head_ball || !tail_ball ) return;

  pNorm axis(head_ball->position, tail_ball->position);
  const float omega = 0.1 * opt_omega;

  for ( Ball *ball: balls )
    {
      pVect b_to_top(ball->position,head_ball->position);
      const float dist_along_axis = dot(b_to_top,axis);
      const float lsq = b_to_top.mag_sq() - dist_along_axis * dist_along_axis;
      ball->omega += omega * axis;
      if ( lsq <= 1e-5 ) continue;
      const float dist_to_axis = sqrt(lsq);
      pNorm rot_dir = cross(b_to_top,axis);
      ball->velocity += omega * dist_to_axis * rot_dir;
    }
}

void
World::objects_erase()
{
  balls.erase();
```

```
  links.erase();
  surfaces.clear();
}

Links
World::link_new
(Ball *ball1, Ball *ball2, float stiffness, float spring_constant_p)
{
  Link_Args largs
    { .spring_constant = spring_constant_p, .link_stiffness = stiffness };
  return link_new(ball1,ball2,largs);
}

Links
World::link_new
(Ball *ball1, Ball *ball2, Link_Args largs )
{
  const float spring_constant_p = largs.spring_constant;
  const float stiffness = largs.link_stiffness;
  Links links_rv;
  assert( ball1->radius > 0 );
  assert( ball2->radius > 0 );
  const float spring_constant = spring_constant_p ?: opt_spring_constant;

  Link* const rlink = new Link(ball1,ball2);
  rlink->radius = largs.radius;
  links_rv += rlink;

  pNorm n12(ball1->position,ball2->position);
  const float rad_sum = ball1->radius + ball2->radius;
  pMatrix3x3 b1rot = ball1->omatrix;
  pMatrix3x3 b2rot = ball2->omatrix;
  pCoor ctr = ball1->position
    + ( ball1->radius + 0.5 * ( n12.magnitude - rad_sum ) ) * n12;

  pVect con_x = rlink->con_x;
  pVect con_y = rlink->con_y;

  const float lrad = stiffness * ball1->radius;

  const int n_springs = 4;
  const double delta_theta = 2 * numbers::pi / n_springs;

  for ( int i=0; i<n_springs; i++ )
    {
      const double theta = i * delta_theta;
      pVect convec = lrad * ( cos(theta) * con_x + sin(theta) * con_y );
      pCoor con = ctr + convec;
      pVect cb1 = mult_MTV( b1rot, con - ball1->position );
      pVect cb2 = mult_MTV( b2rot, con - ball2->position );
      links_rv +=
        new Link(rlink, cb1, cb2,
                 { .distance_relaxed=0, .spring_constant=spring_constant } );
    }
```

```
  return links_rv;
}

///
/// CUDA Code
///

float4 tof4(pCoor c)
{
  float4 f4;
  f4.x = c.x;
  f4.y = c.y;
  f4.z = c.z;
  f4.w = c.w;
  return f4;
}

float4 tof4(pVect c)
{
  float4 f4;
  f4.x = c.x;
  f4.y = c.y;
  f4.z = c.z;
  f4.w = 0;
  return f4;
}

pCoor topc(float4 c)
{
  pCoor f4;
  f4.x = c.x;
  f4.y = c.y;
  f4.z = c.z;
  f4.w = c.w;
  return f4;
}

void
World::data_cpu_to_gpu_constants()
{
  c.platform_xmin = platform_xmin;
  c.platform_xmax = platform_xmax;
  c.platform_zmax = platform_zmax;
  c.platform_zmin = platform_zmin;

  c.n_balls = balls.size();
  c.n_links = links.size();
  c.n_surfaces = surfaces.size();
  c.opt_tryout1 = opt_tryout1;
  c.opt_tryout2 = opt_tryout2;
  c.opt_tryoutf = opt_tryoutf;
  c.opt_spring_constant = opt_spring_constant;
  c.opt_spring_damp = opt_spring_damp;
  c.opt_air_resistance = opt_air_resistance;
```

```
  c.opt_friction_dynamic = opt_friction_dynamic;
  c.opt_head_lock = opt_head_lock;
  c.opt_tail_lock = opt_tail_lock;
  c.gravity_accel = gravity_accel;

  data_cpu_to_gpu_common(&c);
}


void
World::data_cpu_to_gpu_dynamic()
{
  const int n_balls = balls.size();
  const int n_links = links.size();
  const int n_surfaces = surfaces.size();
  const bool need_alloc =
    n_balls > c.alloc_n_balls || n_links > c.alloc_n_links
    || n_surfaces > c.alloc_n_surfaces;
  const bool need_init = c.alloc_n_balls == 0;

#define ALLOC_MOVE_ZERO(n_balls,balls,memb,do_move,do_zero)                \
  {                                                                        \
    const int size_elt_bytes = sizeof(c.balls.memb[0]);                    \
    const int size_bytes = n_balls * size_elt_bytes;                       \
    if ( need_alloc )                                                      \
      {                                                                    \
        if ( !need_init )                                                  \
          {                                                                \
            CE( cudaFree( c.balls.memb ) );                                \
            free(c.h_##balls.memb);                                        \
          }                                                                \
        CE( cudaMalloc( &c.balls.memb, size_bytes ) );                     \
        c.h_##balls.memb = (decltype(c.balls.memb)) malloc( size_bytes );  \
      }                                                                    \
    if ( do_move )                                                         \
      {                                                                    \
        for ( int i=0; i<n_balls; i++ )                                    \
          c.h_##balls.memb[i] = balls[i]->memb;                            \
        CE( cudaMemcpy                                                     \
            ( c.balls.memb, c.h_##balls.memb,                              \
              size_bytes, cudaMemcpyDefault ) );                           \
      }                                                                    \
    else if ( do_zero )                                                    \
      {                                                                    \
        CE( cudaMemset( c.balls.memb, 0, size_bytes ) );                   \
      }                                                                    \
  }

#define MOVEc(num,ptr,memb) \
  ALLOC_MOVE_ZERO(num,ptr,memb,gpu_const_data_stale,0)
#define MOVE(num,ptr,memb) ALLOC_MOVE_ZERO(num,ptr,memb,1,0)
#define ALLOC(num,ptr,memb) ALLOC_MOVE_ZERO(num,ptr,memb,0,0)
#define ZERO(num,ptr,memb) ALLOC_MOVE_ZERO(num,ptr,memb,0,1)
  MOVEc(n_balls,balls,position);
```

```
  MOVEc(n_balls,balls,velocity);
  ZERO(n_balls,balls,force);

  MOVEc(n_balls,balls,orientation);
  MOVEc(n_balls,balls,omatrix);
  MOVEc(n_balls,balls,omega);
  ZERO(n_balls,balls,torque);
  MOVEc(n_balls,balls,m_r_fdt);
  MOVEc(n_balls,balls,locked);

  MOVEc(n_links,links,ball_idx);
  MOVEc(n_links,links,cb1);
  MOVEc(n_links,links,cb2);
  MOVEc(n_links,links,dr_sc);
  MOVEc(n_links,links,is_simulatable);
  MOVEc(n_links,links,is_surface_connection);

  MOVEc(n_surfaces,surfaces,b_idx);

#undef MOVE
#undef MOVEc
#undef ALLOC
#undef ZERO
#undef ALLOC_MOVE
  if ( need_alloc )
    {
      c.alloc_n_balls = n_balls;
      c.alloc_n_links = n_links;
      c.alloc_n_surfaces = n_surfaces;
      data_cpu_to_gpu_common(&c);
    }
  gpu_const_data_stale = false;
}

void
World::data_gpu_to_cpu_dynamic()
{
  const int n_balls = balls.size();

#define MOVE(n_balls,balls,memb)                                        \
  {                                                                     \
    const int size_elt_fr_bytes = sizeof(c.balls.memb[0]);              \
    const int size_bytes = n_balls * size_elt_fr_bytes;                 \
    CE( cudaMemcpy                                                      \
        ( c.h_##balls.memb, c.balls.memb, size_bytes,                   \
          cudaMemcpyDeviceToHost ) );                                   \
    for ( int i=0; i<n_balls; i++ )                                     \
      balls[i]->memb = c.h_##balls.memb[i];                             \
  }

  MOVE(n_balls,balls,position);
  MOVE(n_balls,balls,velocity);
  MOVE(n_balls,balls,orientation);
  MOVE(n_balls,balls,omatrix);
```

```
  MOVE(n_balls,balls,omega);

#undef MOVE

}


///
/// Physical Simulation Code
///

 /// Initialize Simulation
//

void
World::ball_setup_1(bool big)
{
  // Arrange balls to form sort of tree.

  last_setup = 1;

  double ball_rad = 1;  // Warning: Re-assigned, changes behavior of nb.

  pCoor first_pos(17.2,ball_rad,-20.2);

  pVect dir_up(0.1,3,0.1);
  pVect dir_z(3*distance_relaxed,0,0);
  pVect dir_x(0,0,3*distance_relaxed);

  // Remove objects from the simulated objects lists, balls and links.
  // The delete operator is used on objects in the lists.
  //
  objects_erase();

  auto nb =
    [&] (pCoor pos) {
      Ball* const ball = new Ball;
      ball->position = pos;
      ball->locked = false;
      ball->velocity = pVect(0,0,0);
      ball->radius = ball_rad;
      ball->contact = false;
      balls += ball;
      return ball;
      };

  tail_ball = nb(first_pos);
  tail_ball->specularity = 0.5;

  pCoor last_pos = tail_ball->position;

  const int n_tiers = big ? 32 : 8;
  const double shrink_final = 0.17;
  const double shrink = pow(shrink_final,1.0/n_tiers);
```

```
  Balls tail_balls;
  for ( int i=1; i<n_tiers; i++ )
    {
      ball_rad *= shrink;
      Ball* const ball = nb(last_pos + (i==1?1.0:ball_rad) * dir_up);
      ball->specularity = 0.5;
      tail_balls += ball;
      links += link_new(balls[balls-2],ball,0.8);
      last_pos = ball->position;
    }

  head_ball = balls[balls-1];

  pNorm dir_zn(dir_z);
  pNorm dir_xn(dir_x);
  for ( Ball *ball: tail_balls )
    {
      if ( ball == head_ball ) break;
      pNorm to_top(ball->position,head_ball->position);
      const float r = to_top.magnitude;
      int num_hairs = max(4,int(7*r));
      ball_rad = ball->radius / 4;
      const float delta_theta = 2 * numbers::pi / num_hairs;
      for ( int i=0; i<num_hairs; i++ )
        {
          const float theta = i * delta_theta;
          pCoor pos = ball->position +
            r * cos(theta) * dir_xn + r * sin(theta) * dir_zn;
          Ball* const ballr = nb(pos);
          links += link_new(ballr,ball);
        }
    }

  opt_head_lock = false;
  opt_tail_lock = true;
  setup_at_end();
}

void
World::ball_setup_2()
{
  // Arrange balls to form a suspended thing.

  last_setup = 2;

  pCoor first_pos(17.2,14.8f,-20.2);

  pVect dir_dn(0.5,first_pos.y/8,0.5);
  pVect dir_z(3*distance_relaxed,0,0);
  pVect dir_x(0,0,3*distance_relaxed);

  // Remove objects from the simulated objects lists, balls and links.
  // The delete operator is used on objects in the lists.
```

```
//
objects_erase();

float ball_rad = 0.3;  // Warning: Re-assigned, changes behavior of nb.

auto nb =
  [&] (pCoor pos) {
    Ball* const ball = new Ball;
    ball->position = pos;
    ball->locked = false;
    ball->velocity = pVect(0,0,0);
    ball->radius = ball_rad;
    ball->contact = false;
    ball->specularity = 0.5;
    balls += ball;
    return ball;
    };

head_ball = nb(first_pos);
for ( int j: {-1,1} )
  for ( int i: {-1,0,1} )
    {
      Ball* const ball = nb(first_pos - dir_dn + i * dir_x + j * dir_z );
      if ( i != 0 ) links += link_new(head_ball,ball);
      if ( i != -1 ) links += link_new(balls[balls-2],ball);
      if ( j == 1 && i != 0 )  links += link_new(balls[balls-4],ball);
    }

Ball* const tail_start = nb(first_pos - dir_dn);
links += link_new(balls[balls-6],tail_start);
links += link_new(balls[balls-3],tail_start);
pCoor last_pos = tail_start->position;
Balls tail_balls;
for ( int i=1; i<5; i++ )
  {
    Ball* const ball = nb(last_pos - 0.5 * dir_dn);
    tail_balls += ball;
    links += link_new(balls[balls-2],ball);
    last_pos = ball->position;
  }

tail_ball = balls[balls-1];

pNorm dir_zn(dir_z);
pNorm dir_xn(dir_x);
int num_hairs = 20;
ball_rad = 0.1;
for ( Ball *ball: tail_balls )
  {
    const float delta_theta = 2 * numbers::pi / num_hairs;
    for ( int i=0; i<num_hairs; i++ )
      {
        const float theta = i * delta_theta;
        const float r = ball->radius * 10;
```

```
          pCoor pos = ball->position +
            r * cos(theta) * dir_xn + r * sin(theta) * dir_zn;
          Ball* const ballr = nb(pos);
          links += link_new(ballr,ball);
        }
    }

  opt_head_lock = true;    // Head ball will be frozen in space.
  opt_tail_lock = false;   // Tail ball can move freely.
  setup_at_end();
}

void
World::ball_setup_3()
{
  // Arrange links to form what I would eventually learn to be a
  // corona virus.

  const bool sarcov = true;

  last_setup = 3;
  objects_erase();

  pCoor first_pos(17.2,14.8f,-10.2);

  float ball_radius = 1;

  auto nb =
    [&] (pCoor pos) {
      Ball* const ball = new Ball;
      ball->position = pos;
      ball->locked = false;
      ball->velocity = pVect(0,0,0);
      ball->radius = ball_radius;
      ball->contact = false;
      balls += ball;
      return ball;
      };

  head_ball = nb(first_pos);
  head_ball->specularity = 0.5;
  ball_radius = ball_radius * ( sarcov ? 0.08 : .1 );
  const int n_spikes = sarcov ? 100 : 500;

  for ( int i=0; i<n_spikes; i++ )
    {
      pNorm dir(0.5-drand48(),0.5-drand48(),0.5-drand48());
      const float len = head_ball->radius * ( sarcov ? 1.5 : 3.0 );
      Ball* const ball = nb(first_pos + len * dir);
      links += link_new(ball,head_ball,2);
    }

  tail_ball = balls[1];
```

```
  opt_head_lock = false;    // Head ball will be frozen in space.
  opt_tail_lock = false;   // Tail ball can move freely.

  setup_at_end();
}

pVect vec_rand() {return pVect(drand48(),drand48(),drand48());}

void
World::ball_setup_4()
{
  // Render a wheel (disk of balls with one in the center).

  last_setup = 4;
  objects_erase();

  pCoor center_pos = pCoor(17.2,14.8f,-10.2) + 2*vec_rand();

  const int n_balls = 4 * ( 4 + random()%4 );
  float ring_outer_radius = 5 + 2 * drand48();
  float ball_radius = 0.2 * 2 * numbers::pi * ring_outer_radius / n_balls;

  pNorm c_to_eye(center_pos,eye_location);
  pNorm az = c_to_eye + 0.5 * vec_rand() + pVect(0,0.5,0);

  pNorm ax = fabs(az.x) < fabs(az.y)
    ? pVect(0,az.z,-az.y) : pVect(az.z,0,-az.x);
  pVect ay(az,ax);

  // Default settings for balls.
  //
  Ball ball_default;
  ball_default.specularity = 0.5;
  ball_default.velocity = pVect(0,0,0);
  ball_default.radius = ball_radius;
  ball_default.color = color_light_slate_gray;

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

  const float d_theta = 2 * numbers::pi / n_balls;

  // Place balls in a circle.
  //
  for ( int i=0; i<n_balls; i++ )
    {
      const float theta = d_theta * i;
      pCoor pos = center_pos + vx * cosf(theta) + vy * sinf(theta);

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

```
      // Set color of first ball to blue, others to white.
      ball->color = i == 0 ? color_blue : color_white;
    }

  const float link_stiffness = 10;

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

  // Place a ball in the center and connect it to all the other balls.
  if ( true )
    {
      Ball* const center_ball =
        balls.emplace_back( new Ball(center_pos,ball_default) );

      for ( int i=0; i<n_balls; i++ )
        links += link_new( center_ball, balls[i], 2 );
    }

  pNorm spin_axis = az;
  pCoor c_of_mass = center_pos;

  for ( Ball* ball: balls )
    {
      pVect cm_to_ball( c_of_mass, ball->position );

      ball->velocity = opt_omega * cross( spin_axis, cm_to_ball );
      ball->omega = opt_omega * spin_axis;
    }

  tail_ball = balls.back();
  head_ball = balls.front();

  opt_head_lock = false; // If true, head ball locked in place.
  opt_tail_lock = false;

  setup_at_end();
}

void
World::ball_setup_5()
{
  // Construct a top. (Top, as in a child's toy.)

  last_setup = 5;
  objects_erase();

  pCoor center_pos(17.2,14.8f,-10.2);

  const int n_balls = 20;
  const float crad = 5;
```

```
const float ball_radius = 0.3 * 2 * numbers::pi * crad / n_balls;

// Default settings for balls.
//
Ball ball_default;
ball_default.specularity = 0.5;
ball_default.velocity = pVect(0,0,0);
ball_default.radius = ball_radius;
ball_default.color = pColor(0xffbbff); // Plum 1. That's the color's name.

const float d_theta = 2 * numbers::pi / n_balls;

pVect vz = pVect(1,6,1) - 2*vec_rand();
pNorm ax( vz.y, -vz.x, 0 );
pNorm ay = cross(vz,ax);
pVect vx = crad * ax;
pVect vy = crad * ay;

pNorm spin_axis = vz;
float omega = opt_omega;
opt_spring_damp = .4;

// Construct Top's Main Ring
//
for ( int i=0; i<n_balls; i++ )
  {
    const float theta = d_theta * i;
    pCoor pos = center_pos + vx * cosf(theta) + vy * sinf(theta);
    Ball* ball = balls.emplace_back( new Ball(pos,ball_default) );

    // Find the coordinate on the spin axis closest to the new ball.
    //
    pCoor axis_cp =
      center_pos + spin_axis * dot( spin_axis, pVect(center_pos,pos) );
    //
    // Use this coordinate to compute a vector giving the ball's
    // motion when the top is spinning around axis spin_axis.
    //
    pVect cp_to_p(axis_cp,pos);
    ball->velocity = omega * cross(spin_axis,cp_to_p);
  }

// Make one of the balls on the top's main ring a different color.
balls[0]->color = color_blue;

// Connect the balls on the main ring with links.
//
for ( int i=0; i<n_balls; i++ )
  links += link_new( balls[i], balls[(i+1)%n_balls], 2 );

// Complete the top with balls above and below the ring, and one in
// the center.
//
```

```
  pNorm az(vz);
  pCoor top_pos = center_pos + az * crad * 1.5;
  pCoor bot_pos = center_pos - az * crad * 1.5;

  head_ball = balls.emplace_back( new Ball(top_pos,ball_default) );
  tail_ball = balls.emplace_back( new Ball(bot_pos,ball_default) );
  Ball* ctr_ball = balls.emplace_back( new Ball(center_pos,ball_default) );

  for ( int i=0; i<n_balls; i += 2 )
    {
      links += link_new( head_ball, balls[i], 2 );
      links += link_new( tail_ball, balls[i], 2 );
    }

  for ( int i=0; i<n_balls; i++ )
    links += link_new( ctr_ball, balls[i], 2 );

  for ( auto b: balls ) b->omega = omega * spin_axis;

  opt_head_lock = false;    // Head ball will be frozen in space.
  opt_tail_lock = false;   // Tail ball can move freely.

  setup_at_end();
}

void
World::ball_setup_6()
{
  // Render what will become a parachute, probably.

  last_setup = 6;
  objects_erase();

  pCoor center_pos = pCoor(17.2,14.8f,-10.2) + 2*vec_rand();

  const int n_balls = 4 * ( 4 + random()%4 );
  float ring_outer_radius = 5 + 2 * drand48();
  float ring_inner_radius = ring_outer_radius * ( 0.1 + drand48() * 0.4 );
  float ball_radius = 0.2 * 2 * numbers::pi * ring_outer_radius / n_balls;

  pNorm c_to_eye(center_pos,eye_location);
  pNorm az = c_to_eye + 0.5 * vec_rand() + pVect(0,0.5,0);

  pNorm ax = fabs(az.x) < fabs(az.y)
    ? pVect(0,az.z,-az.y) : pVect(az.z,0,-az.x);
  pVect ay(az,ax);

  // Default settings for balls.
  //
  Ball ballo_default;
  ballo_default.specularity = 0.5;
  ballo_default.velocity = pVect(0,0,0);
  ballo_default.radius = ball_radius;
  ballo_default.color = color_light_slate_gray;
```

```
  Ball balli_default = ballo_default;
  balli_default.radius = ballo_default.radius * 0.6;

  const float d_theta = 2 * numbers::pi / n_balls;

  const float link_stiffness = 10;

  vector<Ball*> balls_outer, balls_inner;

  // Place balls in a circle.
  //
  for ( int i=0; i<n_balls; i++ )
    {
      const float theta = d_theta * i;
      pVect pos_n = cosf(theta) * ax + sinf(theta) * ay;
      pCoor pos_o = center_pos + ring_outer_radius * pos_n;
      pCoor pos_i = center_pos + ring_inner_radius * pos_n;

      // Construct new ball and add it to the ball list.
      //
      Ball* ball_o = balls_outer.emplace_back( new Ball(pos_o,ballo_default) );
      Ball* ball_i = balls_inner.emplace_back( new Ball(pos_i,balli_default) );
      Link *link =
        new Link( ball_o, ball_i,
                  { .spring_constant = opt_spring_constant/200 } );
      links.push_back(link);

      // Set color of first ball to blue, others to white.
      ball_i->color = ball_o->color =
        i == 0 ? color_blue :
        theta <= numbers::pi/2 ? color_lsu_spirit_gold :
        color_white;
    }

  // Connect each ball to its neighbor with a link.
  //
  for ( auto bc: { &balls_outer, &balls_inner } )
    for ( int i=0; i<n_balls; i++ )
      {
        balls.push_back( bc[0][i] );
        links += link_new( bc[0][i], bc[0][(i+1)%n_balls], link_stiffness );
      }

  for ( int i=0; i<n_balls; i++ )
    {
      const int j = ( i + 1 ) % n_balls;
      Ball* b00 = balls_inner[i];
      Ball* b01 = balls_outer[i];
      Ball* b10 = balls_inner[j];
      Ball* b11 = balls_outer[j];
      surfaces.emplace_back(b00,b01,b11,color_light_gray);
      surfaces.emplace_back(b10,b00,b11,color_light_gray);
    }
```

```
  pNorm spin_axis = az;
  pCoor c_of_mass = center_pos;
  float omega_init = opt_omega * 0.1;

  if ( 0 )
  for ( Ball* ball: balls )
    {
      pVect cm_to_ball( c_of_mass, ball->position );

      ball->velocity = omega_init * cross( spin_axis, cm_to_ball );
      ball->omega = omega_init * spin_axis;
    }

  tail_ball = balls.back();
  head_ball = balls.front();

  opt_head_lock = false; // If true, head ball locked in place.
  opt_tail_lock = false;

  setup_at_end();
}

void
World::ball_setup_7()
{
  last_setup = 7;
  objects_erase();

  /// 2025 Homework 2
  //
  //  Put solution in this routine.

  /// Randomly Choose Dodecahedron Position, Size, Orientation
  //
  //  It's okay to modify this to facilitate debugging, but
  //  after that restore it's original functionality.
  //
  pCoor center_pentagon_0 = pCoor(17.2,14.8f,-10.2) + 2*vec_rand();
  pNorm c_to_eye(center_pentagon_0,eye_location);

  pNorm az = c_to_eye + 0.5 * vec_rand() + pVect(0,0.5,0);
  pNorm ax = fabs(az.x) < fabs(az.y)
    ? pVect(0,az.z,-az.y) : pVect(az.z,0,-az.x);
  pVect ay(az,ax);

  const int n_sides = 5;
  const float ball_radius = 0.5;


  // Edge Length
  //
  // Length of dodecahedron and pentagon edges.
  //
  const float len_edge = 2 + 4 * drand48();
```

```
// Distance from pentagon center to pentagon vertex.
//
const float rc_pentagon = len_edge / ( 2 * sinf(numbers::pi/5) );

// Circumscribed Sphere Radius
//
// Radius of the smallest sphere that can surround the dodecahedron.
// The dodecahedron vertices will be on the surface of this sphere.
//
const float rc = len_edge * numbers::phi * numbers::sqrt3 / 2;
//
// Note: numbers::phi is the Golden Ratio: ( 1 + sqrt(5) ) / 2.

// Inscribed Sphere Radius
//
// Radius of the largest sphere that will fit inside the dodecahedron.
// The surface of the sphere will touch the center of each pentagon.
//
float ri = len_edge * sqrtf( sqrt(5) * pow(numbers::phi,5) / 20 );

// Dodecahedron Center
//
// This is also the center of the circumscribed and inscribed
// spheres.
//
pCoor center = center_pentagon_0 + ri * az;

// Dihedral Angle
//
// Angle between adjacent pentagons.
//
float theta_dihedral [[maybe_unused]] = 2 * atanf( numbers::phi );


// Default settings for balls.
//
Ball ball_default;
ball_default.specularity = 0.5;
ball_default.velocity = pVect(0,0,0);
ball_default.radius = ball_radius;
ball_default.color = color_green;

// Dodecahedron Vertex Coordinates -- Including Duplicates
//
// The number of items in coors must be a multiple of 5 ..
// .. and must include duplicates.
//
vector< pCoor > coors;
//
// First pentagon:  coors[0] through coors[4].
// Second pentagon: coors[5] through coors[9].
// And so on.
//
```

```
  // DO NOT try to remove or avoid duplicate vertices in coors.
  //
  // A dodecahedron has 20 vertices. Each vertex is shared by three
  // faces (pentagons), so in a correct solution coors should have:
  //  60  =  20 * 3  =  12 * 5  elements.


  ///
  /// Compute Coordinates of First Pentagon
  ///

  pCoor pos_prev [[maybe_unused]] = center_pentagon_0 + rc_pentagon * ax;

  const float d_theta = 2 * numbers::pi / n_sides;

  /// SOLUTION -- Problem 1
  //
  //  Prepare transformation matrix m1 that rotates d_theta radians
  //  around the pentagon center.
  //
  pMatrix_Translate from_pc( pVect( center_pentagon_0, pCoor(0,0,0) ) );
  pMatrix_Translate to_pc( pVect( pCoor(0,0,0), center_pentagon_0 ) );
  pMatrix_Rotation rotp( az, d_theta );
  pMatrix mp = to_pc * rotp * from_pc;
  //
  //  Note that for rotation to work the axis of rotation, az in this case,
  //  must pass through the origin, which is why the transformation
  //  has translations to and from the pentagon center. It would also
  //  be correct to translate to the dodecahedron center in this case.

  for ( int i=0; i<5; i++ )
    {
#if 1
      coors << pos_prev;
      pos_prev = mp * pos_prev;
#else
      const float theta = d_theta * i;
      pVect dir = cosf(theta) * ax + sinf(theta) * ay;
      pCoor pos = center_pentagon_0 + rc_pentagon * dir;
      coors << pos;
#endif
    }

  // Don't modify this.
  const int n_coors_pentagon = coors.size();

  ///
  /// Compute Coordinates of the Next Five Pentagons
  ///

  for ( int i=0; i<5; i++ )
    {
      // Two adjacent vertices from the first pentagon.
      pCoor p0 [[maybe_unused]] = coors[i];
```

```
    pCoor p1 [[maybe_unused]] = coors[(i+1)%5];

    /// SOLUTION -- Problem 2
    //
    //  Compute the coordinates of a new pentagon obtained
    //  by rotating the original pentagon (coors[0] through coors[4])
    //  by theta_dihedral radians around edge p0p1.
    //
    //  Start by constructing the transformation matrix, m.
    //
    pMatrix_Translate to_p0( p0 );
    pMatrix_Rotation rot2( pNorm(p0,p1), theta_dihedral );
    pMatrix_Translate to_Origin( -p0 );
    pMatrix m = to_p0 * rot2 * to_Origin;
    //
    //  Note: The translation can be to any point on the line
    //  formed by p0,p1. Point p0 is convenient.
    //
    //  Next, use m to insert 5 new vertices based on the original
    //  pentagon.
    //
    for ( int j=0; j<5; j++ ) coors << m * coors[j];
  }

// Don't modify this either.
const int n_coors_demi = coors.size();

///
/// Compute Coordinates of Last Six Pentagons
///

/// 2025 HW 2 Problem 3
//
//  Compute a correct transformation matrix, ms;
//
/// SOLUTION -- Problem 3
//
//  Rotate the coordinates obtained above by π radians around ay.
//  In this case the translation needs to be to the dodecahedron
//  center.
//
pMatrix_Translate from_ctr( -center );
pMatrix_Translate to_ctr( center );
pMatrix_Rotation rot3( ay, numbers::pi );
pMatrix ms = to_ctr * rot3 * from_ctr;

for ( int i=0; i<n_coors_demi; i++ ) coors << ms * coors[i];

//
/// SOLUTION Appears Above


/// Code Below DOES NOT Need to be Modified
```

```
// If desired, just show the first pentagon or the first half of the
// dodecahedron.
//
if ( show_n == 0 ) coors.resize( n_coors_pentagon );
else if ( show_n == 1 ) coors.resize( n_coors_demi );

// Count the number of vertices that are the wrong distance from the center.
//
n_coors_bad = ranges::count_if
  ( coors, [&](auto c){ return fabs( pDistance( center, c ) - rc ) > 1e-4; });
//
// This number is shown in the UI near "Bad Coords".

// Number of coordinates, including duplicates.
//
const int n_coors = coors.size();
```

### Identify Duplicated Coordinates

```
// Sort coordinates by x component to efficiently find duplicates.
//
vector<int> balls_idx(n_coors);
for ( int i: views::iota(0,n_coors) ) balls_idx[i] = i;
ranges::sort( balls_idx, ranges::greater(), [&](int i){return coors[i].x; } );

vector<Ball*> all_balls(n_coors,nullptr);
const float tiny_dist = 1e-4;
const float tiny_dist_sq = tiny_dist * tiny_dist;
for ( int ii: views::iota( 0, n_coors ) )
  {
    int i = balls_idx[ii];
    if ( all_balls[i] ) continue;

    // Haven't seen this coordinate yet ..
    //
    pCoor p = coors[i];
    //
    // .. so create a ball.
    //
    Ball* original = new Ball( p, ball_default );
    original->id = i;
    all_balls[i] = original;
    balls << original;

    // Look for duplicates of p.
    //
    for ( int jj: views::iota( ii+1, n_coors ) )
      {
        int j = balls_idx[jj];
        pCoor pj = coors[j];

        // If the x components are this far apart then no subsequent
        // member of balls_idx can be within tiny_dist of p.
        //
```

```
        if ( fabs( p.x - pj.x ) > tiny_dist ) break;

        // We can't rule out closeness by just looking at x so we'll
        // need to do a bit more work.
        //
        if ( pDistance_sq(p,pj) >= tiny_dist_sq ) continue;

        // It looks like pj is a duplicate of p.

        assert( !all_balls[j] );
        all_balls[j] = original;
      }
  }

// Create links between each pair of pentagon vertices.
//
map<int,bool> link_done;
for ( int p = 0;  p < n_coors;  p += 5 )
  for ( int v = 0;  v < 5;  v++ )
    {
      Ball* b0 = all_balls[ p + v ];
      Ball* b1 = all_balls[ p + (v+1)%5 ];
      int key = min(b0->id,b1->id)*n_coors + max(b0->id,b1->id);
      if ( link_done.contains(key) ) continue;
      link_done[key] = true;
      links += link_new( b0, b1, 2 );
    }

// Set all balls in the first half to purple.
//
for ( auto b: views::take( all_balls, n_coors_demi ) )
  b->color = color_lsu_spirit_purple;

// Set the balls in the initial pentagon to gold.
//
for ( auto b: views::take( all_balls, 5 ) )
  b->color = color_lsu_spirit_gold;

// Choose the head and tail balls.
//
// The UI can be used to lock the head or tail ball in place ('h'
// and 't') and to spin around the axis formed by the head and tail
// ball ('w').
//
head_ball = all_balls[0];

// Set the tail ball to the ball most distant from the head ball.
//
tail_ball = head_ball;
for ( auto b: balls )
  if ( pDistance_sq(head_ball->position,b->position)
       > pDistance_sq(head_ball->position,tail_ball->position) )
    tail_ball = b;
```

```
  setup_at_end();
}

void
World::setup_at_end()
{
  balls.take(balls_marker);
  lock_update();
  time_step_count = 0;
  world_time = 0;
  world_time_link_update = -1;
  for ( uint i=0; i<balls.size(); i++ ) balls[i]->idx = i;
  for ( Link *l : links )
    { l->ball1_idx = l->ball1->idx; l->ball2_idx = l->ball2->idx; }
  for ( auto& s: surfaces )
    { s.b0_idx = s.b0->idx;  s.b1_idx = s.b1->idx;  s.b2_idx = s.b2->idx; }
}


 /// Advance Simulation State by delta_t Seconds
//
void
World::time_step_cpu(double delta_t)
{
  const int n_balls = balls;
  const int n_links = links;

  // Smoothly move ball in response to user input.
  //
  if ( adj_t_stop )
    {
      const double dt = min(world_time,adj_t_stop) - adj_t_prev;
      pVect adj = dt/adj_duration * adj_vector;
      balls_translate(adj,0);
      adj_t_prev = world_time;
      if ( world_time >= adj_t_stop ) adj_t_stop = 0;
    }

#pragma omp parallel for
  for ( int bi=0; bi<n_balls; bi++ )
    {
      Ball* const ball = balls[bi];
      assert( ball->fdt_to_do );
      ball->force = ball->mass * gravity_accel;
      ball->torque = pVect(0,0,0);
    }

  const int n_surfaces = surfaces.size();
  for ( int i=0; i<n_surfaces; i++ )
    {
      Surface& s = surfaces[i];
      Ball& b0 = *s.b0;
      Ball& b1 = *s.b1;
      Ball& b2 = *s.b2;
```

```
      pNorm snorm = cross(b0.position,b1.position,b2.position);
      pVect vel = (1.0/3) * ( b0.velocity + b1.velocity + b2.velocity );
      float speed = dot(snorm,vel);
      float fmag =
        -speed * fabs(speed) * snorm.magnitude * 0.5f * opt_air_resistance;
      pVect f = fmag * snorm;
      b0.force += f;
      b1.force += f;
      b2.force += f;
    }

#pragma omp parallel for
  for ( int i=0; i<n_links; i++ )
    {
      Link* const link = links[i];
      if ( !link->is_simulatable ) continue;

      //
      // Spring Force from Neighbor Balls
      //

      Ball* const ball1 = link->ball1;
      Ball* const ball2 = link->ball2;

      /// Notes:
      //
      // link->cb1
      //   Coordinate where link touches ball1, in ball1's local space.
      //
      // ball1->omatrix
      //   Ball's orientation transformation ..
      //   .. which transforms vectors in ball1's local space to object space.
      //

      // Find position and velocity of the point where the link touches
      // the surface of ball 1 ...
      //
      pVect dir1 = ball1->omatrix * link->cb1;
      pCoor pos1 = ball1->position + dir1;
      pVect vel1 = ball1->velocity + ball1->point_rot_vel(dir1);

      // ... and ball 2.
      //
      pVect dir2 = ball2->omatrix * link->cb2;
      pCoor pos2 = ball2->position + dir2;
      pVect vel2 = ball2->velocity + ball2->point_rot_vel(dir2);

      // Construct a normalized (Unit) Vector from ball to neighbor
      // based on link connection points and ball centers.
      //
      pNorm link_dir(pos1,pos2);
      pNorm c_to_c(ball1->position,ball2->position);

      const float link_length = link_dir.magnitude;
```

```
    // Compute the speed of ball's end of link towards neighbor's end of link.
    //
    pVect delta_v = vel2 - vel1;
    float delta_s = dot( delta_v, link_dir );

    // Compute by how much the spring is stretched (positive value)
    // or compressed (negative value).
    //
    const float spring_stretch = link_length - link->distance_relaxed;

    // Determine whether spring is gaining energy (whether its length
    // is getting further from its relaxed length).
    //
    const bool gaining_e = ( delta_s > 0.0 ) == ( spring_stretch > 0 );

    const float blend_length_factor = 0.1; // Just a guess.
    const float blend_s = blend_length_factor * c_to_c.magnitude;
    const float blend = min( 1.0f, fabs(delta_s) / blend_s );

    // Use a smaller spring constant when spring is loosing energy,
    // a quick and dirty way of simulating energy loss due to spring
    // friction.
    //
    const float sc_base = link->spring_constant ?: opt_spring_constant;
    const float spring_constant =
      gaining_e ? sc_base :
      sc_base * ( 1.0f - blend + blend * opt_spring_damp );

    const float force_mag = spring_constant * spring_stretch;
    pVect spring_force_12 = force_mag * link_dir;

    // Apply forces affecting linear momentum.
    //
    link->spring_force_12 = spring_force_12;

    if ( ! link->is_surface_connection ) continue;

    // Apply torque.
    //
    link->torque1 = cross(pNorm(dir1), spring_force_12);
    link->torque2 = cross(pNorm(dir2), spring_force_12);
  }

// Note: Because two links can reference the same ball this should
// not be done in parallel.
for ( Link *link: links )
  {
    if ( !link->is_simulatable ) continue;
    Ball* const ball1 = link->ball1;
    Ball* const ball2 = link->ball2;
    ball1->force += link->spring_force_12;
    ball2->force -= link->spring_force_12;
    ball1->torque += link->torque1;
```

```
      ball2->torque -= link->torque2;
    }

  ///
  /// Update Position of Each Ball
  ///

#pragma omp parallel for
  for ( int bi=0; bi<n_balls; bi++ )
    {
      Ball* const ball = balls[bi];
      if ( ball->locked )
        {
          ball->velocity = pVect(0,0,0);
          ball->omega = pVect(0,0,0);
          continue;
        }

      // Update Velocity
      //
      // This code assumes that force on ball is constant over time
      // step. This is clearly wrong when balls are moving with
      // respect to each other because the springs are changing
      // length. This inaccuracy will make the simulation unstable
      // when spring constant is large for the time step.
      //
      const float mass = max( ball->mass, ball->mass_min );

      pVect delta_v = ( ball->force / mass ) * delta_t;
      const float r = ball->radius;

      if ( platform_collision_possible(ball->position)
           && fabs(ball->position.y) < r )
        {
          const bool above = ball->position.y >= 0;
          const float spring_constant_plat =
            ball->velocity.y < 0 == above ? 100000 : 50000;
          const float fric_coefficient = opt_friction_dynamic;
          const float force_away = ( above ? 1 : -1 )
            * ( r - fabs(ball->position.y) ) * spring_constant_plat;
          const float delta_v_away = force_away / mass * delta_t;
          const float fric_force_mag = fric_coefficient * force_away;
          pNorm surface_v(ball->velocity.x,0,ball->velocity.z);
          const float delta_v_surf = fric_force_mag / mass * delta_t;

          if ( delta_v_surf > surface_v.magnitude )
            {
              // Ignoring other forces?
              delta_v =
                pVect(-ball->velocity.x,delta_v.y,-ball->velocity.z);
            }
          else
            {
              delta_v -= delta_v_surf * surface_v;
```

```
            }
          delta_v.y += delta_v_away;
        }

      ball->velocity += delta_v;

      // Air Resistance
      //
      const pNorm ball_dir(ball->velocity);
      const float delta_s_air_raw =
        opt_air_resistance * ball_dir.mag_sq * delta_t;
      const float delta_s_air = min( ball_dir.magnitude, delta_s_air_raw );
      ball->velocity -= delta_s_air * ball_dir;

      // Update Position
      //
      // Assume that velocity is constant.
      //
      ball->position += ball->velocity * delta_t;

      ball->omega += delta_t * ball->fdt_to_do * ball->torque;

      pNorm axis(ball->omega);

      // Update Orientation
      //
      // If ball isn't spinning fast skip expensive rotation.
      //
      if ( axis.mag_sq < 0.000001 ) continue;

      // Update ball orientation.
      //
      ball->orientation_set
        ( pQuat(axis,delta_t * axis.magnitude) * ball->orientation );
    }
}

bool
World::platform_collision_possible(pCoor pos)
{
  // Assuming no motion in x or z axes.
  //
  return pos.x >= platform_xmin && pos.x <= platform_xmax
    && pos.z >= platform_zmin && pos.z <= platform_zmax;
}

 /// External Modifications to State
//
//   These allow the user to play with state while simulation
//   running.

// Move the ball.
//
void Ball::translate(pVect amt) {position += amt;}
```

```
// Add velocity to the ball.
//
void Ball::push(pVect amt) {velocity += amt;}

// Set the velocity to zero.
//
void Ball::stop() {velocity = pVect(0,0,0); }

// Set the velocity and rotation (not yet supported) to zero.
//
void Ball::freeze() {velocity = pVect(0,0,0); }



void World::balls_translate(pVect amt,int b){head_ball->translate(amt);}
void World::balls_push(pVect amt,int b){head_ball->push(amt);}
void World::balls_translate(pVect amt)
{ for ( Ball *ball: balls ) ball->translate(amt); }
void World::balls_push(pVect amt)
{ for ( Ball *ball: balls ) ball->push(amt); }
void World::balls_stop()
{ for ( Ball *ball: balls ) ball->stop(); }
void World::balls_freeze(){balls_stop();}


void
World::render_link_start()
{
  if ( world_time_link_update == world_time ) return;

  // Empty the containers in preparation for a new pass.
  for ( auto b: lis_bufs ) b->clear();
}

void
World::render_link_gather(Link *link)
{
  // Place data needed to render link (the argument to this function)
  // in arrays. Just before rendering those arrays will be placed in
  // buffer objects.


  // If we've already collected links, just return.  This happens
  // when doing multiple passes to render shadows and reflections.
  //
  if ( world_time_link_update == world_time ) return;

  Ball *const ball1 = link->ball1;
  Ball *const ball2 = link->ball2;

  // Direction of link relative to center off ball.
  //
```

```
  pVect dir1 = ball1->omatrix * link->cb1;

  // Place on surface of ball where link connects.
  //
  pCoor pos1 = ball1->position + dir1;

  // Direction and connection location for ball 2.
  //
  pVect dir2 = ball2->omatrix * link->cb2;
  pCoor pos2 = ball2->position + dir2;

  pVect p1p2(pos1,pos2);
  pNorm p1p2n(p1p2);

  // Radius of link.
  //
  const float rad = link->radius ?: ball1->radius * 0.3;

  // Compute scale factors for texture.
  const float tex_aspect_ratio = 8.5 / 11;  // Yes, inches. Can't remember mm.
  const float page_width_o = 2.5;  // Width of texture in object-space coords.
  const float tex_t_scale = link->distance_relaxed / page_width_o;
  const float tex_angle_scale = rad * tex_aspect_ratio / page_width_o;

  pNorm dirn1(dir1);
  pNorm dirn2(dir2);

  // Vectors used to describe the cubic Hermite curve.
  //
  pVect4 v1 = p1p2n.magnitude * dirn1;
  pVect4 v2 = p1p2n.magnitude * dirn2;

  // Convert link's local x and y axes to global coordinates.
  //
  pVect4 b1_ydir = ball1->omatrix * link->b1_dir;
  pVect4 b2_ydir = ball2->omatrix * link->b2_dir;


  // Place data needed to render links in lists. Note that
  // the += operator appends the right-hand side to the container
  // on the left-hand side.
  //
  link->idx = lis_pos1.size();
  lis_pos1 << pos1;
  lis_pos2 << pos2;
  lis_v1 << v1;
  lis_v2 << v2;
  lis_b1_ydir << b1_ydir;
  lis_b2_ydir << b2_ydir;
  lis_tex_rad << vec4( tex_t_scale, tex_angle_scale, rad, 0);
}
```

```
void
World::render_link_render(vk::CommandBuffer& cb, bool shadow_volumes)
{
  // Perform a rendering pass to render all the links collected
  // by the routine render_link_2.


  // Check whether this is the first time we are rendering this data.
  //
  const bool first_render =
    world_time_link_update != world_time
    || opt_segments != buf_uni_common->links_n_segs
    || opt_sides != buf_uni_common->sides;
  world_time_link_update = world_time;
  const size_t n_instances = lis_pos1.size();
  const bool do_rt = vh.raytrace_active();
  const int segments = shadow_volumes ? max(2,opt_segments/4) : opt_segments;
  const int segments_sv = max(2,opt_segments/4);

  if ( !n_instances ) return;

  if ( first_render )
    {
      for ( auto b: lis_bufs ) b->to_dev();
      buf_uni_common->sides = opt_sides;
      buf_uni_common->links_n_segs_sv = segments_sv;
      buf_uni_common->delta_tee_sv = 1.0f / segments_sv;
      buf_uni_common->links_n_segs = opt_segments;
      buf_uni_common->delta_tee = 1.0f / opt_segments;
    }

  if ( do_rt && first_render )
    {
      const int n_tri_p_segment = 2 * opt_sides;
      const int n_tri_p_link = n_tri_p_segment * opt_segments;
      const int n_tri_p_pass = n_tri_p_link * n_instances;

      const int n_vtx_p_segment = 2 * ( opt_sides + 1 );
      const int n_vtx_p_link = n_vtx_p_segment * opt_segments;
      const int n_vtx_p_pass = n_vtx_p_link * n_instances;

      if ( !rt_geo_links )
        rt_geo_links
          .init( raytrace )
          .color_set( color_dim_gray, color_salmon )
          .bind( VR_BUF_IDX_sampler, sampler, texid_hw )
          .grouping_set( RT_Geometry_Indexed );

      buf_links_as_idx.assure(3*n_tri_p_pass);
      buf_links_tex_coor.assure(n_vtx_p_pass);
      buf_links_vertex_g.assure(n_vtx_p_pass);
      buf_links_normal_g.assure(n_vtx_p_pass);

      rt_geo_links
```

```
        .bind( VR_BUF_IDX_indices, buf_links_as_idx, 3*n_tri_p_pass )
        .bind( VR_BUF_IDX_pos, buf_links_vertex_g, n_vtx_p_pass )
        .bind( VR_BUF_IDX_normal, buf_links_normal_g )
        .bind( VR_BUF_IDX_tcoor, buf_links_tex_coor );
    }

  if ( do_rt )
    {
      QSet& qs = vh.qs;
      buf_uni_common.to_dev();
      qs.cb.begin( { vk::CommandBufferUsageFlagBits::eOneTimeSubmit } );
      comp_links.record_dispatch(qs.cb,opt_segments,n_instances);
      qs.cb.end();
      qs.q.submit( vk::SubmitInfo{ 0, nullptr, nullptr, 1, &qs.cb }, nullptr );
      qs.q.waitIdle();
      return;
    }


  VPipeline& pipe = shadow_volumes ? pipe_set_2_sv : pipe_set_2;

  pipe.record_draw_instanced(cb, segments+1, n_instances);
}


void
World::frame_callback()
{
  // This routine called whenever window needs to be updated, but
  // before commands are recorded.

  // Get any waiting keyboard commands.
  //
  cb_keyboard();

  if ( vh.have_raytrace ) vh.opt_want_raytrace_now = opt_want_raytrace;

  // This is a hack for 2023 HW 2.
  while ( !balls.empty() && balls.back()->world_time_expire <= world_time )
    balls.pop_back();
  //
  // Balls must be last in list and not part of a link.

  const double time_now = time_wall_fp();
  const int time_step_count_start = time_step_count;
  const int nmp = gpu_info.cuda_prop.multiProcessorCount;
  int bl_p_sm_max = 1024;
  for ( int i=0; i<gpu_info.num_kernels; i++ )
    set_min
      (bl_p_sm_max,
       gpu_info.get_max_active_blocks_per_mp(i, opt_block_size, 0) );
  const int grid_size = bl_p_sm_max * nmp;

  const bool opt_cuda = opt_physics_method != GP_cpu;
```

```
  if ( opt_cuda )
    {
      CE( cudaEventRecord(frame_start_ce,0) );
      data_cpu_to_gpu_constants();
      data_cpu_to_gpu_dynamic();
    }

  if ( !opt_pause || opt_single_frame || opt_single_time_step )
    {
      /// Advance simulation state.

      // Amount of time since the user saw the last frame.
      //
      const double wall_delta_t = time_now - last_frame_wall_time;

      // Compute amount by which to advance simulation state for this frame.
      //
      const double duration =
        vh.video_frame_duration() ?:
        opt_single_time_step ? opt_single_time_step * opt_time_step_duration :
        opt_single_frame ? 1/60.0 :
        wall_delta_t;

      const double world_time_target = world_time + duration;
      const double wall_time_limit = time_now + 0.05;

      while ( world_time < world_time_target )
        {
          if ( opt_cuda )
            {
              launch_time_step(opt_time_step_duration,grid_size,opt_block_size);
            }
          else
            {
              time_step_cpu(opt_time_step_duration);
            }
          time_step_count++;
          world_time += opt_time_step_duration;
          const double time_right_now = time_wall_fp();
          if ( time_right_now > wall_time_limit ) break;
        }

      // Reset these, just in case they were set.
      //
      opt_single_frame = false;
      opt_single_time_step = 0;
    }

  if ( opt_cuda )
    {
      data_gpu_to_cpu_dynamic();
      CE( cudaEventRecord(frame_stop_ce,0) );
    }
```

```
  frame_timer.phys_end();
  frame_timer.work_amt_set(time_step_count-time_step_count_start);

  float cuda_time = 0;
  if ( opt_cuda )
    {
      CE(cudaEventSynchronize(frame_stop_ce));
      CE(cudaEventElapsedTime(&cuda_time,frame_start_ce,frame_stop_ce));
    }
  frame_timer.cuda_frame_time_set(cuda_time);

  last_frame_wall_time = time_now;

}

int
main(int argv, char **argc)
{
  pVulkan_Helper pvulkan_helper(argv,argc);
  World world(pvulkan_helper,argv,argc);

  world.run();
}
```

```
/// LSU EE 4702-1 (Fall 2025), GPU Programming
//
 /// Homework 3 -- SOLUTION
 //
 //  Assignment: https://www.ece.lsu.edu/gpup/2025/hw03.pdf
 //
 /// Based on Homework 2 and cpu-only/demo-06-rend-pipe.cc


#include <gp/coord.h>
#include <gp/colors.h>
#include "frame_buffer.h"

 ///
 /// Our_3D: Low-Level 3D Graphics Library.
 ///
enum Topology { Topology_Individual, Topology_Strip, Topology_Fan };

class Our_3D {
public:
  Our_3D(pFrame_Buffer& fbp)
    :frame_buffer(fbp)
  {
    vtx_colors = nullptr;
    vtx_coors = nullptr;
    vtx_normals = nullptr;
    topology = Topology_Individual;
    opt_tri_normals_always = false;
  };

  Our_3D& transform_eye_from_object_set(const pMatrix& m)
  {
    eye_from_object = m;
    return *this;
  }
  Our_3D& transform_clip_from_eye_set(const pMatrix& m)
  {
    clip_from_eye = m;
    return *this;
  }
  Our_3D& light_location_set(const pCoor c)
  {
    light_location = c;
    return *this;
  }
  Our_3D& light_color_set(const pColor c)
  {
    light_color = c;
    return *this;
  }

  Our_3D& vtx_normals_set()
  {
```

```
  vtx_normals = nullptr;
  return *this;
}

Our_3D& vtx_normals_set(vector<pVect4>& c)
{
  vtx_normals = c.data();
  return *this;
}

Our_3D& vtx_coors_set(vector<pCoor>& c)
{
  vtx_coors = &c;
  return *this;
}

Our_3D& vtx_colors_set(vector<pColor>& c)
{
  vtx_colors = &c;
  return *this;
}

// Specify whether vertices are grouped for individual triangles, a
// triangle strip, or a triangle fan in the next call to
// draw_rasterization.
//
Our_3D& topology_set
( Topology top = Topology_Individual, int n_vertices = 0 )
{
  topology = top;
  topology_n_vertices = top == Topology_Individual ? 3 : n_vertices;
  return *this;
}

Our_3D& rendering_stats_reset()
{
  stats_n_triangles = stats_n_vertices = stats_n_fragments = 0;
  return *this;
}

Our_3D& draw_rasterization();

pFrame_Buffer& frame_buffer;
pMatrix eye_from_object, clip_from_eye;
pCoor light_location;
pColor light_color;
vector<pCoor> *vtx_coors;
pVect4 *vtx_normals;
vector<pColor> *vtx_colors;
Topology topology;
int topology_n_vertices;
int64_t stats_n_triangles, stats_n_vertices, stats_n_fragments;

bool opt_tri_normals_always;
```

```
  bool opt_tryout1, opt_tryout2;

};

enum HW3_Variation { HW3_Triangles, HW3_Fans, HW3_ENUM_SIZE };

 ///
 ///   World: An application.
 ///
class World : public World_Base {
public:
  World(pFrame_Buffer& fb):World_Base(fb),gc(fb){}
  Our_3D gc;

  // Called each time a frame needs to be rendered.
  virtual void render(bool first_time)
  {
    if ( first_time ) init_scene();
    render_scene();
  }

  // Called Once
  void init_scene();

  // Called for each frame.
  void render_scene();

  pCoor eye_location, light_location;
  pColor light_color;
  pCoor *adj_location;
  pVect eye_direction;
  bool opt_tryout1,opt_tryout2;

  int opt_hw3_variation;
  bool opt_hw3_p1_strips;
  float dod_angle1, dod_angle2;

};



const char* const hw3_variation_str[] =
  { "P2: TRIANGLES", "P3: FANS" };

void
World::init_scene()
{
  eye_location = pCoor(-2.6,.5,7.9);
  light_location = pCoor(-4.1,.9,5.1);
  eye_direction = pVect(.1,0,-1);
  adj_location = &eye_location;
  opt_tryout1 = opt_tryout2 = false;
  gc.opt_tryout1 = opt_tryout1;
  gc.opt_tryout2 = opt_tryout2;
```

```
  light_color = color_white * 100;

  dod_angle1 = dod_angle2 = 0;
  opt_hw3_variation = 0;
  opt_hw3_p1_strips = false;
}

pVect vec_rand() {return pVect(drand48(),drand48(),drand48());}

void
World::render_scene()
{
  pFrame_Buffer& frame_buffer = gc.frame_buffer;

  switch ( frame_buffer.keyboard_key ) {
  case FB_KEY_LEFT: adj_location->x -= 0.1; break;
  case FB_KEY_RIGHT: adj_location->x += 0.1; break;
  case FB_KEY_PAGE_UP: adj_location->y += 0.1; break;
  case FB_KEY_PAGE_DOWN: adj_location->y -= 0.1; break;
  case FB_KEY_UP: adj_location->z -= 0.1; break;
  case FB_KEY_DOWN: adj_location->z += 0.1; break;
  case 'a': case 'A': opt_hw3_variation++;
    if ( opt_hw3_variation == HW3_ENUM_SIZE ) opt_hw3_variation = 0;
    break;
  case '1': opt_hw3_p1_strips = !opt_hw3_p1_strips; break;
  case 'e': case 'E': adj_location = &eye_location; break;
  case 'l': case 'L': adj_location = &light_location; break;
  case 'v': case 'V': gc.opt_tri_normals_always = !gc.opt_tri_normals_always;
    break;
  case 'y': gc.opt_tryout1 = opt_tryout1 = !opt_tryout1; break;
  case 'Y': gc.opt_tryout2 = opt_tryout2 = !opt_tryout2; break;
  case 'w': dod_angle1 += numbers::pi/30; break;
  case 'W': dod_angle1 -= numbers::pi/30; break;
  case 'q': dod_angle2 += numbers::pi/30; break;
  case 'Q': dod_angle2 -= numbers::pi/30; break;
  case '+': case '=':
    if ( !frame_buffer.keyboard_control ) light_color *= 1.1;
    break;
  case '-': case '_':
    if ( !frame_buffer.keyboard_control ) light_color *= 1/1.1;
    break;

  }

  gc.frame_buffer.fbprintf
    ( "Light Location [ %.2f, %.2f, %.2f ],  Light Intensity %.2f ('+-')  "
      "Eye Location [ %.2f, %.2f, %.2f ]\n",
      light_location.x, light_location.y, light_location.z,
      light_color.r,
      eye_location.x, eye_location.y, eye_location.z );
  frame_buffer.fbprintf
    ("HW3 P1: %s ('1')  HW3 Dodec %s ('a')  Spin ('wWqQ')  "
     "Tryout1 %s ('y')  Tryout2 %s ('Y')\n",
     opt_hw3_p1_strips ? "STR" : "TRI",
```

```
    hw3_variation_str[opt_hw3_variation],
    opt_tryout1 ? "ON " : "OFF", opt_tryout2 ? "ON " : "OFF");

gc.light_location_set( light_location ).light_color_set( light_color );

///
/// Prepare Transformations
///

/// Specifications of User's Monitor
//
//  First, the width of the user's monitor in object space coordinates.
//
const float width_m = 1.6;  // Wow, that's a big monitor! (m is for meter.)
const float qn = 1;         // Distance from eye to monitor along z axis.
//
//  To determine the height of the user's monitor we need to know the
//  aspect ratio of the window we are painting.
//
const uint win_width = frame_buffer.width_get();
const uint win_height = frame_buffer.height_get();
//
//  win_width and win_height are pixel-space coordinates.
//
const float aspect = 1.0 * win_width / win_height;
//
//  Using the aspect ratio and the width we can compute the height.
//
const float height_m = width_m / aspect;
//
//  width_m and height_m are in object-space coordinates.

// Compute transformation from Object Space to Eye Space ..
//
pMatrix_Translate center_eye(-eye_location);
pMatrix_Rotation rotate_eye(eye_direction,pVect(0,0,-1));
pMatrix eye_from_object = rotate_eye * center_eye;
//
// .. and give the transformation to Our3D:
//
gc.transform_eye_from_object_set(eye_from_object);

// Compute transformation from Eye Space to Clip Space ..
//
pMatrix_Frustum clip_from_eye
  ( -width_m/2, width_m/2,  -height_m/2, height_m/2,  qn, 5000 );
//   l          r           b            t            n   f
//   The letters l,r,b,t,n,f are from a frustum transform definition.
//
// .. and give the transformation to Our3D:
//
gc.transform_clip_from_eye_set(clip_from_eye);
```

```
///
/// Prepare Triangles
///

// List of coordinates in object space and list of colors.
//
vector<pCoor> coors_os;
vector<pColor> colors;

/// Triangle to Mark Light Source
//
// Insert a triangle at light location, facing user. Light is top point.
//
pVect light_to_eye( light_location, eye_location );
float l_sz = 0.1;
pVect lax = cross( pVect( 0, 1, 0 ), light_to_eye ).normal() * l_sz;
pVect lay = cross( light_to_eye, lax ).normal() * l_sz;
coors_os << light_location
         << light_location - lax - lay << light_location + lax - lay;
colors << color_yellow << color_yellow << color_yellow;

/// Yellow and Multicolored Triangles
//
coors_os << pCoor(-7,2,-2) << pCoor(-7,4,-2) << pCoor(-5,2,-2);
colors   << color_red      << color_green    << color_blue;

coors_os << pCoor(-7,4,-2) << pCoor(-5,4,-2) << pCoor(-5,2,-2);
colors << color_lsu_spirit_gold << color_lsu_spirit_gold
       << color_lsu_spirit_gold;

///
/// Send Triangles to Our3D for Rendering
///
//
gc.vtx_normals_set();            // Tell Our3D to compute triangle normals.
gc.topology_set(Topology_Individual);  // Use individual triangle grouping.
gc.vtx_coors_set(coors_os);      // Coordinates to render.
gc.vtx_colors_set(colors);       // Colors to render.
gc.draw_rasterization();         // Tell Our3D to render now.


///
/// Render the Problem 1 Shape
///

// First remove the coordinates and colors inserted above.
//
coors_os.clear(); colors.clear();

if ( opt_hw3_p1_strips )
  {
    // Render shape using a triangle strip. (When Problem 1 solved.)

    gc.topology_set( Topology_Strip );
```

```
      /// Homework 3
      //
      //  Problem 1 part a solution goes here.

      /// SOLUTION -- Problem 1a
      //
      // Emit the first two vertices of the strip.
      //
      coors_os << pCoor(-5,1,-2)          << pCoor(-7,1,-2);
      colors   << color_lsu_spirit_purple << color_lsu_spirit_purple;

      // Complete first triangle.
      //
      coors_os << pCoor(-5,-1,-2);
      colors   << color_lsu_spirit_purple;

      // Complete second triangle.
      //
      coors_os << pCoor(-7,-1,-2);
      colors << color_lsu_spirit_gold;

      // Complete third triangle.
      //
      coors_os << pCoor(-6,-2,-2);
      colors << color_green;

      // Complete fourth triangle, ending the strip.
      //
      coors_os << pCoor(-7,-2,-2);
      colors << color_salmon;

#if 0
      // Original code.
      coors_os << pCoor(-6,-2,-2) << pCoor(-7,-2,-2) << pCoor(-7,-1,-2);
      colors   << color_salmon    << color_salmon    << color_salmon;

      coors_os << pCoor(-7,1,-2) << pCoor(-5,1,-2) << pCoor(-5,-1,-2);
      colors << color_lsu_spirit_purple << color_lsu_spirit_purple
             << color_lsu_spirit_purple;

      coors_os << pCoor(-7,-1,-2) << pCoor(-7,1,-2) << pCoor(-5,-1,-2);
      colors << color_lsu_spirit_gold << color_lsu_spirit_gold
             << color_lsu_spirit_gold;

      coors_os << pCoor(-7,-1,-2) << pCoor(-6,-2,-2) << pCoor(-5,-1,-2);
      colors   << color_green     << color_green     << color_green;
#endif

    }
  else
    {
      // Render shape using individual triangles.
```

```
    gc.topology_set( Topology_Individual );

    /// Homework 3
    //
    //  This block does not have to be modified.

    coors_os << pCoor(-6,-2,-2) << pCoor(-7,-2,-2) << pCoor(-7,-1,-2);
    colors   << color_salmon    << color_salmon    << color_salmon;

    coors_os << pCoor(-7,1,-2) << pCoor(-5,1,-2) << pCoor(-5,-1,-2);
    colors << color_lsu_spirit_purple << color_lsu_spirit_purple
           << color_lsu_spirit_purple;

    coors_os << pCoor(-7,-1,-2) << pCoor(-7,1,-2) << pCoor(-5,-1,-2);
    colors << color_lsu_spirit_gold << color_lsu_spirit_gold
           << color_lsu_spirit_gold;

    coors_os << pCoor(-7,-1,-2) << pCoor(-6,-2,-2) << pCoor(-5,-1,-2);
    colors   << color_green     << color_green     << color_green;
  }


gc.vtx_normals_set();            // Tell Our3D to compute triangle normals.
gc.vtx_coors_set(coors_os);      // Coordinates to render.
gc.vtx_colors_set(colors);       // Colors to render.
gc.rendering_stats_reset();
gc.draw_rasterization();         // Tell Our3D to render now.
pStringF
  p1stats("P1 Shape: %ld vertices, %ld triangles, %ld fragments.",
   gc.stats_n_vertices, gc.stats_n_triangles, gc.stats_n_fragments);


// First remove the coordinates and colors inserted above.
//
coors_os.clear(); colors.clear();

///
/// Render a Cylinder
///

// Container for vertex normals.
vector<pVect4> normals_os;

// Specifications of Cylinder
//
pCoor cyl_center(1,-1,-3);
pVect cyl_axis( 0, 0.5, 0.05 );
float cyl_radius = 4;
//
// Compute axes of circle. (Base of cylinder, top of cylinder.)
//
pNorm ax(0,-cyl_axis.z,cyl_axis.y);  // Find an axis orthogonal to cyl_axis.
pNorm ay = cross( cyl_axis, ax );
//
```

```
  int n_segs = 20;  // Number of faces in prism (approximating a cylinder.)
  //
  float delta_theta = 2 * M_PI / n_segs;

  for ( int i=0; i<=n_segs; i++ )
    {
      float theta = i * delta_theta;
      pVect4 n = ax * cosf(theta) + ay * sinf(theta);
      pCoor c = cyl_center + cyl_radius * n;

      coors_os << c + cyl_axis << c;
      normals_os << n << n;
      colors << color_cyan << color_cyan;
    }

  ///
  /// Send Triangles to Our3D for Rendering
  ///
  //
  gc.vtx_normals_set(normals_os);  // This time we're providing normals.
  gc.topology_set(Topology_Strip); // Use a triangle strip grouping.
  gc.vtx_coors_set(coors_os);
  gc.vtx_colors_set(colors);
  gc.draw_rasterization();

  gc.rendering_stats_reset();

  ///
  /// Homework 3
  ///
  {
    // Dodecahedron Center
    //
    pCoor center = cyl_center + pVect(0,1.5,0);
    pMatrix_Rotation dod_spin1( pNorm(0.1,1,0.1), dod_angle1 );
    pMatrix_Rotation dod_spin2( pNorm(1,-0.1,0), dod_angle2 );
    pMatrix dod_spin = dod_spin1 * dod_spin2;
    pVect ax = dod_spin.col_get(0);
    pVect ay = dod_spin.col_get(1);
    pVect az = dod_spin.col_get(2);

    // Edge Length
    //
    const float len_edge = 2;

    // Distance from pentagon center to pentagon vertex.
    //
    const float rc_pentagon = len_edge / ( 2 * sinf(numbers::pi/5) );

    // Inscribed Sphere Radius
    //
    float ri = len_edge * sqrtf( sqrt(5) * pow(numbers::phi,5) / 20 );

    // Use same coordinate axes as the cylinder.
```

```
pCoor center_pentagon_0 = center - ri * az;

// Dihedral Angle
//
float theta_dihedral [[maybe_unused]] = 2 * atanf( numbers::phi );

// Dodecahedron Vertex Coordinates -- Including Duplicates
//
// The number of items in coors will be a multiple of 5.
//
vector< pCoor > coors;
//
// First pentagon:  coors[0] through coors[4].
// Second pentagon: coors[5] through coors[9].
// And so on.

///
/// Compute Coordinates of First Pentagon
///
const float d_theta = 2 * numbers::pi / 5;
for ( int i=0; i<5; i++ )
  {
    const float theta = d_theta * i;
    pVect dir = cosf(theta) * ax + sinf(theta) * ay;
    pCoor pos = center_pentagon_0 + rc_pentagon * dir;
    coors << pos;
  }

///
/// Compute Coordinates of the Next Five Pentagons
///
for ( int i=0; i<5; i++ )
  {
    pCoor p0 = coors[i], p1 = coors[(i+1)%5];
    pMatrix_Translate to_p0( p0 );
    pMatrix_Rotation rot2( pNorm(p0,p1), theta_dihedral );
    pMatrix_Translate to_Origin( pVect( p0, pCoor(0,0,0) ) );
    pMatrix m = to_p0 * rot2 * to_Origin;
    for ( int j=0; j<5; j++ ) coors << m * coors[j];
  }

///
/// Compute Coordinates of Last Six Pentagons
///

pMatrix_Translate from_ctr( pVect(center,pCoor(0,0,0)) );
pMatrix_Translate to_ctr( center );
pMatrix_Rotation rot3(ax,numbers::pi);
pMatrix_Rotation rot4(az,0.2*numbers::pi);
pMatrix ms = to_ctr * rot4 * rot3 * from_ctr;
const int n_vtx_demi = coors.size();
for ( int i=0; i<n_vtx_demi; i++ ) coors << ms * coors[i];

///
```

```
    /// Send Triangles to Our3D for Rendering
    ///

    coors_os.clear();
    colors.clear();
    gc.vtx_normals_set();

    switch ( opt_hw3_variation ){

    case HW3_Triangles:

      /// SOLUTION -- Problem 2
      //
      for ( size_t i=0; i<coors.size(); i+=5 )

        // Render the pentagon by emitting three triangles, all
        // share pentagon vertex 0 (index i).
        //
        for ( int j=1; j<4; j++ )
          coors_os << coors[ i ] << coors[ i + j ] << coors[ i + j + 1 ];

      gc.topology_set(Topology_Individual);

      break;

    case HW3_Fans:

      // Problem 3
      // Don't change this code, modify the code in Our_3D.

      gc.topology_set(Topology_Fan,5); // Use a fan grouping.
      coors_os = coors;

      break;

    default:
      assert( false );
    }

    //  for ( auto _: coors_os ) colors << color_lsu_spirit_purple;
    for ( auto _: coors_os ) colors << color_light_slate_gray;
    gc.vtx_coors_set(coors_os);
    gc.vtx_colors_set(colors);
    gc.draw_rasterization();

  }

  gc.frame_buffer.fbprintf
    ("%s   Dodecahedron: %ld vertices, %ld triangles, %ld fragments.\n",
     p1stats.s,
     gc.stats_n_vertices, gc.stats_n_triangles, gc.stats_n_fragments);

}
```

```
Our_3D&
Our_3D::draw_rasterization()
{
  ///
  /// Prepare Transformations
  ///

  // Specifications of User's Monitor
  //
  const uint win_width = frame_buffer.width_get();
  const uint win_height = frame_buffer.height_get();
  //
  // win_width and win_height are pixel-space coordinates.

  // Transform from Clip to Pixel
  //
  pMatrix_Translate recenter(pVect(1,1,0));
  pMatrix_Scale scale( win_width/2, win_height/2, 1 );
  pMatrix window_from_clip = scale * recenter;
  pMatrix ws_from_os = window_from_clip * clip_from_eye * eye_from_object;

  ///
  /// Rasterize Triangle and Write Frame Buffer
  ///

  // Get pointer to the depth (z) buffer.
  //
  float* const zbuffer = frame_buffer.buffer_depth_get();
  //
  // The depth buffer uses the same index as the frame buffer.

  auto& coors_os = *vtx_coors;
  auto& colors = *vtx_colors;

  const bool topology_strip = topology == Topology_Strip;
  const bool topology_fan [[maybe_unused]] = topology == Topology_Fan;

  const ssize_t stop = coors_os.size() - 2;

  /// SOLUTION -- Problem 3
  //
  // Triangle Fan Plan:
  //
  //  - Set increment to 1 (as in triangle strips).
  //  - Vertex i Notes:
  //      Vertex i is in fan number i / topology_n_vertices
  //         which is i / 5 for the pentagons.
  //      Let j = i % topology_n_vertices
  //         remember that % is the modulus operator.
  //      The first vertex of the fan is
  //         idx_00 = i / topology_n_vertices * topology_n_vertices ..
  //         .. remembering that integer division rounds down.
  //      When j < topology_n_vertices - 2 ..
  //      .. the following vertices form a triangle of the fan:
```

```
//         idx_00,  i + 1,  i + 2.
//      When j == topology_n_vertices - 2 and j == topology_n_vertices - 1
//      .. should not render a triangle.
//
//  Code immediately below sets increment to 1 for fan (and keeps it
//  at one for strip).
//
//  In the vertex loop idx_0 is computed as described above when the
//  topology is a fan, and the triangle is skipped as described
//  above.

//  Increment by 1 for both strips and fans.
//
size_t inc = topology == Topology_Individual ? 3 : 1;

stats_n_vertices += coors_os.size();

for ( ssize_t i=0; i<stop; i += inc )
  {
    /// SOLUTION -- Problem 3
    //
    //  Compute idx_0 appropriately.
    //
    ssize_t idx_0 =
      topology_fan ? i / topology_n_vertices * topology_n_vertices : i;
    ssize_t idx_1 = i + 1;
    ssize_t idx_2 = i + 2;
    //
    // If triangle would be part of two fans, skip triangle.
    //
    if ( topology_fan && idx_2 - idx_0 >= topology_n_vertices ) continue;

    // Get next triangle's object space coordinates ..
    //
    pCoor o0 = coors_os[idx_0],  o1 = coors_os[idx_1],  o2 = coors_os[idx_2];
    pCoor_Homogenized_Keep_w w0( ws_from_os * o0 );
    pCoor_Homogenized_Keep_w w1( ws_from_os * o1 );
    pCoor_Homogenized_Keep_w w2( ws_from_os * o2 );

    pColor c0 = colors[idx_0], c1 = colors[idx_1], c2 = colors[idx_2];

    stats_n_triangles++;

    // Compute triangle normal (for lighting).
    //
    pNorm tn = cross(o0,o1,o2);

    // Non-null if user has provided vertex normals and wants to use them.
    vector<pVect4> npd;
    const bool use_normals = !opt_tri_normals_always && vtx_normals;
    if ( use_normals )
      for ( auto idx: { idx_0, idx_1, idx_2 } ) npd << vtx_normals[idx];
    pVect4* np = use_normals ? &npd[0] : nullptr;
```

```
        pVect w20(w2,w0), w21(w2,w1);
        const float db0 = 0.5/max(fabs(w20.x),fabs(w20.y));
        const float db1 = 0.5/max(fabs(w21.x),fabs(w21.y));

        // Quick and dirty method of skipping huge triangles.
        if ( db0 * db1 < 1e-7 ) continue;

        // Used for computing perspective-correct barycentric coordinates.
        const float w0ow2 = w0.w/w2.w, w0ow1 = w0.w/w1.w;
        const float w1ow2 = w1.w/w2.w, w1ow0 = w1.w/w0.w;

        // Rasterize the Triangle.
        // Iterate over triangle using barycentric coordinates.
        //
        for ( float b0=0; b0<=1; b0 += db0 )
          for ( float b1=0; b1<=1-b0; b1 += db1 )
            {
              const float b2 = 1 - b0 - b1;

              // Each iteration operates on one fragment of triangle.

              // Window-Space Coordinates of Fragment
              //
              pCoor wf = b0 * w0 + b1 * w1 + b2 * w2;

              stats_n_fragments++;

              if ( uint(wf.x) >= win_width || uint(wf.y) >= win_height ) continue;

              // Compute the index into the frame and depth buffers.
              //
              const size_t idx = wf.x + int(wf.y) * win_width;

              // Depth (Z) Test
              //
              float zinv = wf.z;
              if ( zinv < 0 || zinv > 1 ) continue;
              if ( zbuffer[ idx ] < zinv ) continue;

              zbuffer[ idx ] = zinv;

              // Compute perspective-correct barycentric coordinates.
              //
              float bc0 = b0 / ( b2 * w0ow2 + b1 * w0ow1 + b0 );
              float bc1 = b1 / ( b2 * w1ow2 + b0 * w1ow0 + b1 );
              float bc2 = 1 - bc0 - bc1;

              // Find object-space coordinate of fragment.
              //
              pCoor of =     bc0 * o0 + bc1 * o1 + bc2 * o2;

              // Blend color of three vertices together.
              //
              /// SOLUTION -- Problem 1b
```

```
            //
            // When rendering a strip set all colors equal to the
            // color of the last (provoking) vertex of the triangle.
            //
            pColor color = topology_strip ? c2 : bc0 * c0 + bc1 * c1 + bc2 * c2;
            //
            // It would also be correct to use the color of first
            // vertex of the triangle (c0), or even the middle vertex
            // color (c1), so long as the particular vertex position
            // (first, middle, last) is used consistently.

            // If provided, blend vertex normals, otherwise use triangle normal.
            //
            pVect4 n = np ? bc0 * np[0] + bc1 * np[1] + bc2 * np[2] : tn;

            // Compute lighted color.
            //
            pNorm f_to_l(of,light_location);
            float phase = fabs(dot(n,f_to_l));
            pColor lighted_color = phase / f_to_l.mag_sq * light_color * color;

            //
            // Write the frame (color) buffer with the lighted color
            //
            frame_buffer[ idx ] = lighted_color.int_rgb();
          }
      }
  return *this;
}


int
main(int argc, char **argv)
{
  pFrame_Buffer demo_frame_buffer(argc,argv);
  WRender render( demo_frame_buffer );
  World our_world( demo_frame_buffer );
  WRENDER_INSERT(render, our_world);
  render.run();
  return 0;
}
```

```
/// LSU EE 4702-1 (Fall 2025), GPU Programming
//
 /// Homework 4 -- SOLUTION
 //
 //  Assignment: https://www.ece.lsu.edu/gpup/2025/hw04.pdf
 //
 //  Placed solution in routines vs_main, gs_main, and vs_main and
 //  surrounding code.
 //
 //  DID NOT put solution in vs_main_clean, gs_main_clean, and vs_main_clean.
 //  Left them untouched.

// Specify version of OpenGL Shading Language.
//
#version 460

#extension GL_GOOGLE_include_directive : enable
#include <light.h>
#include "shader-common-generic-lighting.h"
#include "links-shdr-common.h"

///
 /// Shader Uniforms
///

layout ( binding = BIND_UNI_COMMON ) uniform Common_Uniform
{
  Shdr_Uni_Common com;
};

// Convenience Variables
//
bool opt_tryout1 = com.tryout.x != 0;
bool opt_tryout2 = com.tryout.y != 0;
bool opt_tryout3 = com.tryout.z != 0;
float opt_tryoutf = com.tryoutf.x;

bool opt_hw04_stel = com.opt_hw04.x != 0;
bool opt_hw04_dot = com.opt_hw04.y != 0;
bool opt_hw04_tex = com.opt_hw04.z != 0;
float opt_hw04_height_stel = com.tryoutf.y;

// Note: this uniform is also defined in transform.h.
layout ( binding = BIND_TRANSFORM ) uniform Uni_Transform
{
  //   Course Name           OpenGL Name
  mat4 eye_from_object;   // gl_ModelViewMatrix
  mat4 clip_from_eye;     // gl_ProjectionMatrix
  mat4 clip_from_object;  // gl_ModelViewProjectionMatrix
  mat4 object_from_eye;   // gl_ModelViewMatrixInverse
  mat4 eye_from_clip;     // gl_ProjectionMatrixInverse
  mat4 object_from_clip;  // gl_ModelViewProjectionMatrixInverse
};
```

```
#ifdef HW4_TRI
#ifdef _VERTEX_SHADER_

 /// Vertex Shader Inputs
///
layout ( location = LOC_IN_POS ) in vec4 in_vertex;
layout ( location = LOC_IN_COLOR ) in vec4 in_color;
layout ( location = LOC_IN_MAT4 ) in mat4 in_mat_object_from_local;

layout (location = 0) out VG
{
  vec4 vertex_e;
  vec4 color;
} Out;

void
vs_main()
{
  // Homework 4 - Put solution here and in surrounding code.

  // Transform to clip space ..
  //
  gl_Position = clip_from_object * in_mat_object_from_local * in_vertex;
  //
  // .. and eye-space.
  //
  Out.vertex_e = eye_from_object * in_mat_object_from_local * in_vertex;

  // Pass color unchanged.
  //
  Out.color = in_color;
}

#endif

#ifdef _GEOMETRY_SHADER_

// Indicate type of input primitive expected by geometry shader.
//
layout ( triangles ) in;

 /// SOLUTION -- Problem 2
//
//   Set max_vertices to six for the two triangles needed in stellation.
//
layout ( triangle_strip, max_vertices = 6 ) out;

layout (location = 0) in VG
{
  vec4 vertex_e;
  vec4 color;
} In[3];
```

```
layout (location = 0) out GF
{
  vec4 vertex_e;
  vec4 color;
  vec3 normal_e;

  /// SOLUTION -- Problem 1
  vec2 tcoor;
} Out;

void
gs_main()
{
  // Homework 4 - Put solution here and in surrounding code.

  /// Geometry Shader
  //
  // Notes:
  //
  //   - In[0] is the same vertex on all three triangles used to
  //     render a dodecahedron face.

  // Compute Triangle Normal
  //
  vec3 v12 = In[2].vertex_e.xyz - In[1].vertex_e.xyz;
  vec3 v10 = In[0].vertex_e.xyz - In[1].vertex_e.xyz;
  vec3 tn = normalize( cross(v12,v10) );
  float edge_len = length(v12);

  // Compute Center of Pentagon
  //
  const float pi = radians(180);  // No, pi is not a built in. Nor tau.
  vec3 v12o = cross(tn,v12) * float( 1.0 / ( 2 * tan( 2 * pi / 10 ) ) );
  vec4 ctr_pentagon_e = vec4( In[1].vertex_e.xyz + 0.5f * v12 + v12o, 1 );
  //
  // The calculation above is correct when In[1] and In[2] are an edge
  // of the pentagon.


  /// SOLUTION -- Problem 2
  //
  //  Compute information needed to transform an eye space coordinate
  //  to pentagon local space, in which the pentagon inscribed circle
  //  has a radius of one.

  // Radius of inscribed circle in eye-space coordinates.
  //
  float ri_pent = length( v12o );

  // Use vertex In[0] to compute ax since it is the same pentagon
  // vertex for all  three triangles.
  //
  vec3 ax = normalize( In[0].vertex_e.xyz - ctr_pentagon_e.xyz );
```

```
  vec3 ay = cross(tn,ax);

  // Size axes so that the axes will scale from eye space to local space.
  //
  vec3 vx = ax / ri_pent;
  vec3 vy = ay / ri_pent;
  //
  // These axes can be used to find the pentagon local coordinates, or
  // the axes can be used to compute a transformation matrix which
  // will compute the pentagon local coordinates.

  // Compute a matrix that will transform from eye space to pentagon
  // local space. Notice that pentagon local space is 2d.
  //
  // Start with a translation matrix.
  //
  mat4 tr = mat4(1.0); tr[3] = -ctr_pentagon_e;
  //
  // Then use axes to scale and rotate.
  //
  mat4 rot = mat4( transpose( mat3( vx, vy, vec3(0) ) ) );
  //
  // Multiplying these two yields the transformation matrix.
  //
  mat4 m = rot * tr;
  mat4x2 m2 = mat4x2(m);

  /// SOLUTION -- Problem 2
  //
  // Compute a vertex of the stellated face if stellation is on.
  //
  vec4 ctr_stel_e =
    opt_hw04_stel
    ? vec4( ctr_pentagon_e.xyz + tn * opt_hw04_height_stel * edge_len, 1 )
    : ctr_pentagon_e;

  for ( int i=0; i<3; i++ )
    {
      // This vertex (point) eye-space coordinates.
      //
      vec4 pt_i_e = In[i].vertex_e;
      //
      // The next vertex (point) (in triangle order) eye-space coordinates.
      //
      vec4 pt_n_e = In[(i+1)%3].vertex_e;

      // Check whether pt_i_e and pt_n_e form an edge of the pentagon.
      //
      bool is_pentagon_edge = distance(pt_i_e,pt_n_e) < edge_len * 1.1f;

      if ( opt_tryout3 )
        {
          /// SOLUTION -- Problem 1a
```

```
    // Move vertices toward pentagon center.
    //
    vec3 to_ctr = vec3( ctr_pentagon_e - In[i].vertex_e );
    vec4 px_e = vec4( In[i].vertex_e.xyz + 0.1 * to_ctr, 1 );

    gl_Position = clip_from_eye * px_e;
    Out.vertex_e = px_e;
    Out.color = In[ i ].color;
    Out.normal_e = tn;
    Out.tcoor = m2 * Out.vertex_e;
    EmitVertex();
    continue;
  }

if ( opt_hw04_stel )
  {
    /// SOLUTION -- Problem 2
    //
    // Emit one triangle if this is a pentagon edge.

    if ( !is_pentagon_edge ) continue;

    // Compute normal for triangle.
    //
    vec3 n = normalize
      ( cross( vec3( pt_i_e-ctr_stel_e ), vec3( pt_n_e-ctr_stel_e ) ) );

    vec4 pts[3] = { pt_i_e, pt_n_e, ctr_stel_e };
    for ( int j=0; j<3; j++ )
      {
        Out.vertex_e = pts[j];
        gl_Position = clip_from_eye * Out.vertex_e;
        Out.color = In[ i ].color;
        Out.normal_e = n;

        // Compute pentagon local coordinates, either using
        // dot products or the transformation matrix.
        //
        // Setting this to true selects the alternate solution.
        bool use_dot_products = false;
        //
        if ( use_dot_products )
          {
            // This is correct, but longer.
            vec3 vctr = Out.vertex_e.xyz - ctr_pentagon_e.xyz;
            Out.tcoor = vec2( dot(vctr,vx), dot(vctr,vy) );
          }
        else
          {
            Out.tcoor = m2 * Out.vertex_e;
          }
        EmitVertex();
      }
```

```
          EndPrimitive();
          continue;
        }

      gl_Position = gl_in[i].gl_Position;
      Out.vertex_e = In[i].vertex_e;
      Out.color = In[ i ].color;
      Out.normal_e = tn;
      Out.tcoor = m2 * pt_i_e;
      EmitVertex();
    }
  EndPrimitive(); // Triangle Strip Re-Start
}

#endif

#ifdef _FRAGMENT_SHADER_

 /// Fragment Shader Inputs
//
layout ( location = 0 ) in GF
{
  vec4 vertex_e;
  vec4 color;
  vec3 normal_e;

  /// SOLUTION
 vec2 tcoor;

} In;

layout ( binding = BIND_TEXUNIT ) uniform sampler2D tex_unit_0;

 /// Fragment Shader Output
//
layout ( location = 0 ) out vec4 out_frag_color;

void
fs_main()
{
  // Homework 4 - Put solution here and in surrounding code.

  vec4 our_color = gl_FrontFacing ? In.color : vec4(1,0,0,1);

  /// SOLUTION -- Problem 1
  //
  //  Use gold if close enough to center.
  //
  if ( opt_hw04_dot && length(In.tcoor) < 0.5 ) our_color = vec4(1,1,0,1);

  vec4 lighted_color = generic_lighting( In.vertex_e, our_color, In.normal_e );

  if ( opt_hw04_tex )
    {
```

```
      // Scale texture coordinates so they fit inside inscribed circle.
      //
      float a = pow(2,-0.5);
      vec2 tc = ( In.tcoor + vec2(a,a) ) * vec2(a,a);
      tc.y = 1 - tc.y;

      // Check whether we are inside the square ..
      //
      bool out_of_range = any( greaterThan(tc,vec2(1)) )
        || any( lessThan(tc,vec2(0)) );
      //
      // .. if so, read the texture and use it to modulate the color.
      //
      if ( !out_of_range )
        lighted_color = lighted_color * texture(tex_unit_0,tc);
    }

  out_frag_color = lighted_color;
}
#endif

#endif // HW4_TRI
```

## /// Homework 4:

```
//
 ///  Code below should not be part of the solution.
 //   Use it for experimentation or keep it unmodified, as a reference.

#ifdef HW4_CLEAN
#ifdef _VERTEX_SHADER_

 /// Vertex Shader Inputs
///
layout ( location = LOC_IN_POS ) in vec4 in_vertex;
layout ( location = LOC_IN_COLOR ) in vec4 in_color;
layout ( location = LOC_IN_MAT4 ) in mat4 in_mat_object_from_local;

layout (location = 0) out VG
{
  vec4 vertex_e;
  vec4 color;
} Out;

void
vs_main_clean()
{
  /// Not part of solution. Modify vs_main instead.

  // Transform to clip space ..
  //
  gl_Position = clip_from_object * in_mat_object_from_local * in_vertex;
  //
  // .. and eye-space.
```

```
  //
  Out.vertex_e = eye_from_object * in_mat_object_from_local * in_vertex;

  // Pass color unchanged.
  //
  Out.color = in_color;

  /// Not part of solution. Modify vs_main instead.
}

#endif

#ifdef _GEOMETRY_SHADER_

// Indicate type of input primitive expected by geometry shader.
//
layout ( triangles ) in;
layout ( triangle_strip, max_vertices = 3 ) out;

layout (location = 0) in VG
{
  vec4 vertex_e;
  vec4 color;
} In[3];

layout (location = 0) out GF
{
  vec4 vertex_e;
  vec4 color;
  vec3 normal_e;

} Out;

void
gs_main_clean()
{
  /// Not part of solution. Modify gs_main instead.

  /// Geometry Shader
  //
  // Notes:
  //
  //   - In[0] is the same vertex on all three triangles used to
  //     render a dodecahedron face.

  // Compute Triangle Normal
  //
  vec3 v12 = In[2].vertex_e.xyz - In[1].vertex_e.xyz;
  vec3 v10 = In[0].vertex_e.xyz - In[1].vertex_e.xyz;
  vec3 tn = normalize( cross(v12,v10) );
  float edge_len = length(v12);

  // Compute Center of Pentagon
  //
```

```
  const float pi = radians(180);  // No, pi is not a built in. Nor tau.
  vec3 v12o = cross(tn,v12) * float( 1.0 / ( 2 * tan( 2 * pi / 10 ) ) );
  vec4 ctr_pentagon_e = vec4( In[1].vertex_e.xyz + 0.5f * v12 + v12o, 1 );
  //
  // The calculation above is correct when In[1] and In[2] are an edge
  // of the pentagon.

  /// Not part of solution. Modify gs_main instead.

  // Emit one triangle.
  //
  for ( int i=0; i<3; i++ )
    {
      // For convenience, write this vertex and next one in to variables.
      //
      vec4 pt_i_e = In[ i ].vertex_e;
      vec4 pt_n_e = In[ (i+1)%3 ].vertex_e;

      // Check whether pt_i_e and pt_n_e form an edge of the pentagon.
      //
      bool is_pentagon_edge = distance(pt_i_e,pt_n_e) < edge_len * 1.1f;

      gl_Position = gl_in[i].gl_Position;
      Out.vertex_e = In[i].vertex_e;
      Out.color = In[i].color;
      Out.normal_e = tn;
      EmitVertex();
    }
  EndPrimitive(); // Triangle Strip Re-Start

  /// Not part of solution. Modify gs_main instead.
}

#endif

#ifdef _FRAGMENT_SHADER_

 /// Fragment Shader Inputs
//
layout ( location = 0 ) in GF
{
  vec4 vertex_e;
  vec4 color;
  vec3 normal_e;
} In;

layout ( binding = BIND_TEXUNIT ) uniform sampler2D tex_unit_0;

 /// Fragment Shader Output
//
layout ( location = 0 ) out vec4 out_frag_color;

void
fs_main_clean()
```

```
{
  /// Not part of solution. Modify fs_main instead.

  vec2 tc = vec2(0.5,0.5);
  vec4 texel = texture(tex_unit_0,tc);

  vec4 our_color = gl_FrontFacing ? In.color : vec4(1,0,0,1);
  vec4 lighted_color = generic_lighting( In.vertex_e, our_color, In.normal_e );
  out_frag_color = lighted_color;
}

#endif // _FRAGMENT_SHADER_

#endif // HW4_CLEAN
```

```
/// LSU EE 4702-1 (Fall 2025), GPU Programming
//
 /// Homework 5 (EE 4702 Fall 2025) -- SOLUTION
 //
 //  Assignment: https://www.ece.lsu.edu/gpup/2025/hw05.pdf
 //
 //  The solution has been be put in this file, hw05-shdr.cc, and hw05.cc

// Specify version of OpenGL Shading Language.
//
#version 460


#extension GL_GOOGLE_include_directive : enable
#include <light.h>
#include "shader-common-generic-lighting.h"
#include "links-shdr-common.h"

#ifdef BIND_UNI_COMMON
layout ( binding = BIND_UNI_COMMON ) uniform Common_Uniform
{
  Shdr_Uni_Common com;
};
#endif

bool opt_tryout1 = com.tryout.x != 0;
bool opt_tryout2 = com.tryout.y != 0;
bool opt_tryout3 = com.tryout.z != 0;
float opt_tryoutf = com.tryoutf.x;

bool opt_hw05_stel = com.opt_hw05.x != 0;
bool opt_hw05_dot = com.opt_hw05.y != 0;
bool opt_hw05_tex = com.opt_hw05.z != 0;
float opt_hw05_height_stel = com.tryoutf.y;


///
 /// Shader Uniforms
///

// Note: this uniform is also defined in transform.h.
layout ( binding = BIND_TRANSFORM ) uniform Uni_Transform
{
  //   Course Name           OpenGL Name
  mat4 eye_from_object;   // gl_ModelViewMatrix
  mat4 clip_from_eye;     // gl_ProjectionMatrix
  mat4 clip_from_object;  // gl_ModelViewProjectionMatrix
  mat4 object_from_eye;   // gl_ModelViewMatrixInverse
  mat4 eye_from_clip;     // gl_ProjectionMatrixInverse
  mat4 object_from_clip;  // gl_ModelViewProjectionMatrixInverse
};

///
```

```
 /// Storage Buffers
///

#ifdef BIND_HW5_COLOR
layout ( binding = BIND_HW5_COLOR ) buffer cb1 { vec4 buf_hw05_color[]; };
#endif
#ifdef BIND_HW5_MATRIX
layout ( binding = BIND_HW5_MATRIX ) buffer cb2 { mat4 buf_hw05_matrix[]; };
#endif
#ifdef BIND_DOD_COORS_L
layout ( binding = BIND_DOD_COORS_L ) buffer cb3 { vec4 buf_dod_coors_l[]; };
#endif


#ifdef HW5_TRI

#ifdef _VERTEX_SHADER_

 /// Vertex Shader Inputs
///
layout ( location = LOC_IN_POS ) in vec4 in_vertex;
layout ( location = LOC_IN_COLOR ) in vec4 in_color;
layout ( location = LOC_IN_MAT4 ) in mat4 in_mat;

// Declare some outputs as a group. These are grouped together
// because they are used by the generic lighting routine.
//
layout (location = 0) out VG
{
  vec4 vertex_e;
  vec4 color;
} Out;

void
vs_main()
{
  // Transform vertex to clip space.
  gl_Position = clip_from_object * in_mat * in_vertex;

  // Compute eye-space vertex coordinate.
  // These are outputs of the vertex shader and inputs to the frag shader.
  //
  Out.vertex_e = eye_from_object * in_mat * in_vertex;
  Out.color = in_color;
}



#endif

#ifdef _GEOMETRY_SHADER_

// Indicate type of input primitive expected by geometry shader.
//
```

```
layout ( triangles ) in;
layout ( triangle_strip, max_vertices = 9 ) out;

layout (location = 0) in VG
{
  vec4 vertex_e;
  vec4 color;

} In[3];

layout (location = 0) out GF
{
  vec4 vertex_e;
  vec4 color;
  vec3 normal_e;

  vec2 tcoor;
} Out;

void
gs_main()
{
  /// Geometry Shader
  //

   /// Geometry Shader

  // Compute Triangle Normal
  //
  vec3 v12 = In[2].vertex_e.xyz - In[1].vertex_e.xyz;
  vec3 v10 = In[0].vertex_e.xyz - In[1].vertex_e.xyz;
  vec3 tn = normalize( cross(v12,v10) );
  float edge_len = length(v12);

  // Compute Center of Pentagon
  //
  const float pi = radians(180);  // No, pi is not a built in. Nor tau.
  vec3 v12o = cross(tn,v12) * float( 1.0 / ( 2 * tan( 2 * pi / 10 ) ) );
  vec4 ctr_pentagon_e = vec4( In[1].vertex_e.xyz + 0.5f * v12 + v12o, 1 );
  //
  // The calculation above is correct when In[1] and In[2] are an edge
  // of the pentagon.

  float rc_pent = length( v12o );
  vec3 ax = normalize( In[0].vertex_e.xyz - ctr_pentagon_e.xyz );
  vec3 ay = cross(tn,ax);
  vec3 vx = ax / rc_pent, vy = ay / rc_pent;
  mat4 tr = mat4(1.0); tr[3] = -ctr_pentagon_e;
  mat4 rot = mat4( transpose( mat3( vx, vy, vec3(0) ) ) );
  mat4x2 m2 = mat4x2( rot * tr );

  vec4 ctr_stel_e =
    opt_hw05_stel
    ? vec4( ctr_pentagon_e.xyz + tn * opt_hw05_height_stel * edge_len, 1 )
```

```
    : ctr_pentagon_e;

  // Emit one triangle.
  //
  for ( int i=0; i<3; i++ )
    {
      // This vertex (point) eye-space coordinates.
      //
      vec4 pt_i_e = In[i].vertex_e;
      //
      // The next vertex (point) (in triangle order) eye-space coordinates.
      //
      vec4 pt_n_e = In[(i+1)%3].vertex_e;

      // Check whether pt_i_e and pt_n_e form an edge of the pentagon.
      //
      bool is_pentagon_edge = distance(pt_i_e,pt_n_e) < edge_len * 1.1f;

      Out.tcoor = m2 * pt_i_e;

      if ( opt_hw05_stel )
        {
          if ( !is_pentagon_edge ) continue;

          vec4 pts[3] = { pt_i_e, pt_n_e, ctr_stel_e };
          vec3 n = normalize
            ( cross( vec3( pt_i_e-ctr_stel_e ), vec3( pt_n_e-ctr_stel_e ) ) );

          for ( int j=0; j<3; j++ )
            {
              Out.vertex_e = pts[j];
              gl_Position = clip_from_eye * pts[j];
              Out.color = In[i].color;
              Out.normal_e = n;
              Out.tcoor = m2 * pts[j];
              EmitVertex();
            }

          EndPrimitive();
          continue;
        }

      gl_Position = gl_in[i].gl_Position;
      Out.vertex_e = In[i].vertex_e;
      Out.color = In[i].color;
      Out.normal_e = tn;
      EmitVertex();
    }
  EndPrimitive(); // Triangle Strip Re-Start
}

#endif

#endif // HW5_TRI
```

```
#ifdef HW5_PTS_PROB_1

#ifdef _VERTEX_SHADER_

 /// Vertex Shader Inputs
///
#ifdef LOC_IN_POS
layout ( location = LOC_IN_POS ) in vec4 in_vertex;
#endif
#ifdef LOC_IN_INT1
layout ( location = LOC_IN_INT1 ) in int in_color_idx;
#endif
#ifdef LOC_IN_MAT4
layout ( location = LOC_IN_MAT4 ) in mat4 in_mat;
#endif
#ifdef LOC_IN_NORMAL
layout ( location = LOC_IN_NORMAL ) in vec3 in_normal;
#endif

void
vs_main_pts_prob_1()
{
}

#endif

#ifdef _GEOMETRY_SHADER_

// Indicate type of input primitive expected by geometry shader.
//
layout ( points ) in;
layout ( triangle_strip, max_vertices = 15 ) out;

layout (location = 0) out GF
{
  vec4 vertex_e;
  flat vec4 color;
  flat vec3 normal_e;

  vec2 tcoor;

} Out;

void
gs_main_pts_prob_1()
{
  const float pi = radians(180);
  const float delta_th = 2 * pi / 5;

  // Consecutive numbering of primitives (vertices in this case
  // because the input topology is points).
  //
```

```
const int prim_id = gl_PrimitiveIDIn;

/// SOLUTION -- Problem 1
//
// Use gl_PrimitiveIDIn to determine which dodecahedron we are to
// render, dod_idx, and which face of that dodecahedron, face_idx.
//
int dod_idx = gl_PrimitiveIDIn / 12;
int face_idx = gl_PrimitiveIDIn % 12;
//
// Note: dod_idx will be used in the same way color_idx was used
// in the original points code, gs_main_pts_clean. It will
// also be used to retrieve the transformation matrix that in
// the original code was a vertex shader input.

vec4 color = buf_hw05_color[dod_idx];
mat4 eye_from_local = eye_from_object * buf_hw05_matrix[dod_idx];

vec4 dod_ctr_e = eye_from_local * vec4(0,0,0,1);

// Retrieve pentagon vertex coordinates from the storage buffer ..
// .. and transform them to eye space.
//
// Start by computing the index of the first vertex of this
// pentagon ..
//
int pt0_idx = face_idx * 5;
//
// .. then retrieve and transform them.
//
vec4 pent_pts_e[5];
for ( int i=0; i<5; i++ )
  pent_pts_e[i] = eye_from_local * buf_dod_coors_l[pt0_idx+i];

// Compute the pentagon tangent and use as a local z axis.
//
vec3 az =
  normalize( cross( vec3( pent_pts_e[2] - pent_pts_e[1] ),
                    vec3( pent_pts_e[0] - pent_pts_e[1] ) ) );

// Compute the pentagon center.
//
float edge_len = distance( pent_pts_e[1], pent_pts_e[2] );
// ( 5^(1/2) GoldenRatio^5 / 20 )^(1/2)
float ri_e = edge_len * 1.11351636441;
vec4 pent_ctr_e = dod_ctr_e + vec4(ri_e * az,0);
//
// The pentagon center could also have been computed by taking
// the average of the pentagon vertices.

vec3 vx = vec3( pent_pts_e[0] - pent_ctr_e );
vec3 vy = cross( az, vx );

// Compute a matrix, m2, that will transform from eye coordinates to
```

```
  // pentagon-local coordinates (2D) where the center is at the origin.
  //
  mat4 tr = mat4(1.0); tr[3] = -pent_ctr_e;
  mat4 scale = mat4( 1 / ( cos( 2 * pi / 10 ) * dot( vx, vx ) ) );
  mat4 rot = mat4( transpose( mat3( vx, vy, vec3(0) ) ) );
  mat4x2 m2 = mat4x2( scale * rot * tr );

  // Compute the coordinates of the center of the stellation, and
  // use it if ctr_stel_e is true.
  //
  vec4 ctr_stel_e =
    opt_hw05_stel
    ? vec4( pent_ctr_e.xyz + az * opt_hw05_height_stel * edge_len, 1 )
    : pent_ctr_e;

  for ( int i=0; i<5; i++ )
    {
      // Put the coordinates in an array so that a loop can be
      // used to emit the vertices.
      //
      vec4 pts[3] = { ctr_stel_e, pent_pts_e[i], pent_pts_e[(i+1)%5] };

      vec3 n = normalize
        ( cross( vec3(pts[0] - pts[1]), vec3(pts[0] - pts[2]) ) );
      for ( int j=0; j<3; j++ )
        {
          Out.vertex_e = pts[j];
          gl_Position = clip_from_eye * Out.vertex_e;
          Out.color = color;
          Out.normal_e = n;
          Out.tcoor = m2 * pts[j];
          EmitVertex();
        }
      EndPrimitive();
      //
      // Note: It would be possible to emit a triangle strip.
      // Possible final exam question.
    }
}

#endif

#endif


#ifdef HW5_INST_PROB_2

#ifdef _VERTEX_SHADER_

 /// Vertex Shader Inputs
///
 /// SOLUTION -- Problem 2
 //
 //  Remove in_mat and in_ball_idx ..
```

```
 //  .. because those don't apply to every dodecahedron.
 //
 //  Add in_tcoor because they do apply to every dodecahedron.
 //
layout ( location = LOC_IN_POS ) in vec4 in_vertex;
layout ( location = LOC_IN_TCOOR ) in vec2 in_tcoor;

// Declare some outputs as a group. These are grouped together
// because they are used by the generic lighting routine.
//
layout (location = 0) out VG
{
  vec4 vertex_e;
  vec4 color;
  vec2 tcoor;
} Out;

void
vs_main_inst_prob_2()
{
  // Transform vertex to clip space.

  const int instance_id = gl_InstanceIndex;
  //
  // Values range for 0 to n_instances-1.

  const int vertex_id = gl_VertexIndex;
  //
  // Values range from 0 to number of vertices - 1.
  // Values range from 0 to bset_inst_prob2.size() - 1.

  /// SOLUTION -- Problem 2
  //
  //  Retrieve transformation matrix using gl_InstanceIndex ..
  //
  mat4 object_from_local = buf_hw05_matrix[gl_InstanceIndex];
  //
  // .. and use it, in part, to compute clip-space coordinates.
  //
  gl_Position = clip_from_object * object_from_local * in_vertex;

  // Compute eye-space vertex coordinate.
  //
  Out.vertex_e = eye_from_object * object_from_local * in_vertex;
  Out.color = buf_hw05_color[gl_InstanceIndex];
  Out.tcoor = in_tcoor;
}



#endif

#ifdef _GEOMETRY_SHADER_
```

```
// Indicate type of input primitive expected by geometry shader.
//
layout ( triangles ) in;

 /// SOLUTION -- Problem 2
//
//   Reduce number of vertices from 9 to 3.
//
layout ( triangle_strip, max_vertices = 3 ) out;

layout (location = 0) in VG
{
  vec4 vertex_e;
  vec4 color;
  vec2 tcoor;
} In[3];

layout (location = 0) out GF
{
  vec4 vertex_e;
  vec4 color;
  vec3 normal_e;

  vec2 tcoor;
} Out;

void
gs_main_inst_prob_2()
{
  /// Geometry Shader
  //

  /// SOLUTION -- Problem 2
  //
  //  Here there is just one of the pentagon with one vertex in the
  //  center of the pentagon and the other two forming an edge.

  // Compute Triangle Normal
  //
  vec3 v12 = In[2].vertex_e.xyz - In[1].vertex_e.xyz;
  vec3 v10 = In[0].vertex_e.xyz - In[1].vertex_e.xyz;
  vec3 tn = normalize( cross(v12,v10) );
  float edge_len = length(v12);

  vec4 ctr_pentagon_e = In[0].vertex_e;

  vec4 ctr_stel_e =
    opt_hw05_stel
    ? vec4( ctr_pentagon_e.xyz + tn * opt_hw05_height_stel * edge_len, 1 )
    : ctr_pentagon_e;

  if ( opt_hw05_stel )
    {
      vec3 v10s = ctr_stel_e.xyz - In[1].vertex_e.xyz;
```

```
      tn = normalize( cross(v12, v10s ) );
    }

  // Emit one triangle.
  //
  for ( int i=0; i<3; i++ )
    {
      if ( i == 0 && opt_hw05_stel )
        {
          Out.vertex_e = ctr_stel_e;
          gl_Position = clip_from_eye * Out.vertex_e;
        }
      else
        {
          Out.vertex_e = In[i].vertex_e;
          gl_Position = gl_in[i].gl_Position;
        }

      Out.color = In[i].color;
      Out.tcoor = In[i].tcoor;
      Out.normal_e = tn;
      EmitVertex();
    }
  EndPrimitive(); // Triangle Strip Re-Start
}

#endif

#endif // HW5_INST_PROB_2




#ifdef HW5_PTS_CLEAN

#ifdef _VERTEX_SHADER_

 /// Vertex Shader Inputs
///
layout ( location = LOC_IN_POS ) in vec4 in_vertex;
layout ( location = LOC_IN_INT1 ) in int in_color_idx;
layout ( location = LOC_IN_MAT4 ) in mat4 in_mat;
layout ( location = LOC_IN_NORMAL ) in vec3 in_normal;

// Declare some outputs as a group. These are grouped together
// because they are used by the generic lighting routine.
//
layout (location = 0) out VG
{
  mat4 obj_from_local;
  vec4 pent_pt0_l;
  vec3 dod_to_pctr_l;
  int color_idx;
```

```
} Out;

void
vs_main_pts_clean()
{
  Out.obj_from_local = in_mat;
  Out.pent_pt0_l = in_vertex;
  Out.dod_to_pctr_l = in_normal;
  Out.color_idx = in_color_idx;
}

#endif

#ifdef _GEOMETRY_SHADER_

// Indicate type of input primitive expected by geometry shader.
//
layout ( points ) in;
layout ( triangle_strip, max_vertices = 15 ) out;

layout (location = 0) in VG
{
  mat4 obj_from_local;
  vec4 pent_pt0_l;
  vec3 dod_to_pctr_l;
  int color_idx;

} In[1];

layout (location = 0) out GF
{
  vec4 vertex_e;
  flat vec4 color;
  flat vec3 normal_e;

  vec2 tcoor;

} Out;

void
gs_main_pts_clean()
{

  const float pi = radians(180);
  const float delta_th = 2 * pi / 5;

  mat4 eye_from_local = eye_from_object * In[0].obj_from_local;

  vec4 dod_ctr_e = eye_from_local * vec4(0,0,0,1);
  vec4 pent_ctr_l = vec4( In[0].dod_to_pctr_l, 1 );
  vec4 pent_ctr_e = eye_from_local * pent_ctr_l;
  vec4 pt0_l = In[0].pent_pt0_l;
  vec4 pt0_e = eye_from_local * pt0_l;
  vec3 az = normalize( pent_ctr_e.xyz - dod_ctr_e.xyz );
```

```
  vec3 vx = pt0_e.xyz - pent_ctr_e.xyz;
  vec3 vy = cross( az, vx );
  mat4 tr = mat4(1.0); tr[3] = -pent_ctr_e;
  mat4 scale = mat4( 1 / ( cos( 2 * pi / 10 ) * dot( vx, vx ) ) );
  mat4 rot = mat4( transpose( mat3( vx, vy, vec3(0) ) ) );
  mat4x2 m2 = mat4x2( scale * rot * tr );

  vec4 pent_pts_e[5];
  pent_pts_e[0] = pt0_e;

  vec4 color = buf_hw05_color[In[0].color_idx];

  for ( int i=1; i<5; i++ )
    pent_pts_e[i] = pent_ctr_e
      + vec4( cos(i*delta_th) * vx + sin(i*delta_th) * vy, 0 );

  float edge_len = distance(pent_pts_e[1],pent_pts_e[2]);
  vec4 ctr_stel_e =
    opt_hw05_stel
    ? vec4( pent_ctr_e.xyz + az * opt_hw05_height_stel * edge_len, 1 )
    : pent_ctr_e;

  for ( int i=0; i<5; i++ )
    {
      vec4 pts[3] = { ctr_stel_e, pent_pts_e[i], pent_pts_e[(i+1)%5] };
      vec3 n = normalize
        ( cross( pts[0].xyz - pts[1].xyz, pts[0].xyz - pts[2].xyz ) );
      for ( int j=0; j<3; j++ )
        {
          Out.vertex_e = pts[j];
          gl_Position = clip_from_eye * Out.vertex_e;
          Out.color = color;
          Out.normal_e = n;
          Out.tcoor = m2 * pts[j];
          EmitVertex();
        }
      EndPrimitive();
    }
}

#endif

#endif

#ifdef _FRAGMENT_SHADER_

 /// Fragment Shader Inputs
//
layout ( location = 0 ) in GF
{
  vec4 vertex_e;
  vec4 color;
  vec3 normal_e;
  vec2 tcoor;
```

```
} In;

#ifdef BIND_TEXUNIT
layout ( binding = BIND_TEXUNIT ) uniform sampler2D tex_unit_0;
#endif

 /// Fragment Shader Output
//
layout ( location = 0 ) out vec4 out_frag_color;

void
fs_main()
{
  vec4 our_color = gl_FrontFacing ? In.color : vec4(1,0,0,1);

  float cd = length(In.tcoor);
  if ( opt_hw05_dot && cd < 0.5 ) our_color = vec4(1,1,0,1);
  float a = pow(2,-0.5);
  vec2 tc = ( In.tcoor + vec2(a,a) ) * vec2(a,a);
  tc.y = 1 - tc.y;

  vec4 texel = texture(tex_unit_0,tc);
  bool out_of_range = any( greaterThan(tc,vec2(1)) )
    || any( lessThan(tc,vec2(0)) );

  vec4 lighted_color = generic_lighting( In.vertex_e, our_color, In.normal_e );

  if ( opt_hw05_tex && !out_of_range )
    lighted_color = lighted_color * texel;

  out_frag_color = lighted_color;
}
#endif
```

```
/// LSU EE EE 7722/ EE 4702-X, GPU Prog/MicroArch
//
//
 /// Homework 5 (EE 4702 Fall 2025) -- SOLUTION
 //
 //  Assignment: https://www.ece.lsu.edu/gpup/2025/hw05.pdf
 //
 //  The solution should be put in this file, hw05.cc, and hw05-shdr.cc


/// Purpose
//
//   Demonstrate simulation of point masses connected by springs.


/// What Code Does

// Simulates balls connected by springs over a platform. Balls and
// springs can be initialized in different arrangements (called
// scenes). Currently scene 1 is a tree-like arrangement. The platform
// consists of tiles, some are purple-tinted mirrors (showing a
// reflection of the ball), the others show the course syllabus.



///  Keyboard Commands
 //
 /// Object (Eye, Light, Ball) Location or Push
 //   Arrows (←,→,↑,↓) Page Up, Page Down
 //        Move object or push ball, depending on mode.
 //        Shift + KEY: motion is 5x faster.
 //        Ctrl + KEY : motion is 5x slower.
 //   'e': Move eye.
 //   'l': Move light.
 //   'b': Move head (first) ball. (Change position but not velocity.)
 //   'B': Push head ball. (Add velocity.)
 //
 /// Eye Direction and GUI Options
 //
 //   'Home', 'End', 'Delete', 'Insert'
 //         Turn the eye direction. Home should rotate eye direction
 //         up, End should rotate eye down, Delete should rotate eye
 //         left, Insert should rotate eye right. The eye direction
 //         vector is displayed in the upper left.
 //  'C-='  (Ctrl =) Increase green text size.
 //  'C--'  (Ctrl -) Decrease green text size.
 //  'F12'  Write screenshot to PNG file.

 /// Simulation Execution Options
 //  (Also see variables below.)
 //
 //  'p'    Pause simulation. (Press again to resume.)
```

```
//  ' '    (Space bar.) Advance simulation by 1/30 second.
//  'S- '  (Shift Space bar.) Advance simulation by one time step.

/// Performance Tuning, Graphics Effects, and Debugging Options
//
//  'y'    Toggle value of opt_tryout1. Intended for experiments and debugging.
//  'Y'    Toggle value of opt_tryout2. Intended for experiments and debugging.
//  'n'    Toggle use of surface normals for spheres.
//  'o'    Toggle shadow effect.
//  'r'    Toggle mirror effect.
//  '!'    Toggle rendering of spheres (balls).
//  '@'    Toggle rendering of links.
//  '#'    Toggle rendering of platform.


/// Scene and Simulated System Options
//  (Also see variables below.)
//
//  'w'    Twirl balls around axis formed by head and tail. (Prob 2 soln).
//  '1'    Set up scene 1.
//  '2'    Set up scene 2.
//  '3'    Set up scene 3.
//  '4'    Set up scene 4.
//  'h'    Freeze position of first (head) ball. (Press again to release.)
//  't'    Freeze position of last (tail) ball. (Press again to release.)
//  's'    Stop balls.
//  'g'    Turn gravity on and off.

/// Variables
//   Selected program variables can be modified using the keyboard.
//   Use "Tab" to cycle through the variable to be modified, the
//   name of the variable is displayed next to "VAR" on the bottom
//   line of green text.

//  'Tab' Cycle to next variable.
//  '`'   Cycle to previous variable.
//  '+'   Increase variable value.
//  '-'   Decrease variable value.
//
//  VAR Spring Constant - Set spring constant.
//  VAR Time Step Duration - Set physics time step.
//  VAR Air Resistance - Set air resistance.
//  VAR Light Intensity - The light intensity.
//  VAR Gravity - Gravitational acceleration. (Turn on/off using 'g'.)

#define MAIN_INCLUDE
#undef VBUFFER_STRICT
//  #define VBUFFER_STRICT_WATCH 0xd10d270000000018
#include <vhelper.h>

#include <vstroke.h>
#include <gp/coord.h>
#include <gp/pstring.h>
#include <gp/misc.h>
```

```
#include <vutil-texture.h>
#include <vutil-pipeline.h>
#include "shapes.h"

#include <cuda_runtime.h>

#include <gp/colors.h>

#include <gp/util-containers.h>
#include <gp/coord-containers.h>
#include <gp/cuda-gpuinfo.h>
#include "links.cuh"

#include "links-shdr-common.h"

#include <numbers>


///
/// Main Data Structures
///
//
// class World: All data about scene.


// Set to false for homework assignments to avoid burying important
// variables.
constexpr bool opt_all_ui_variables = false;

class World;

enum GPU_Physics_Method { GP_cpu, GP_cuda, GP_ENUM_SIZE };
const char* const gpu_physics_method_str[] = { "CPU", "CUDA" };



enum Shader_Option { SO_Plain, SO_HW5_Triangles, SO_HW5_Points_Clean,
  SO_HW5_Points_Prob_1, SO_HW5_Inst_Prob_2, SO_ENUM_SIZE };

const char *sh_names[] =
  { "PLAIN", "HW5-Tri", "HW5-Pts-Cln", "HW5-Pts-P1",
    "HW5-Inst-P2"
  };

// Object Holding Ball State
//
class Ball {
public:
  Ball() { init(); }
  Ball(pCoor pos){ init(); position = pos; constants_update(); }
  Ball(pCoor pos, const Ball &bd)
  { *this = bd; position = pos; constants_update(); }
```

```
void init()
  {
    velocity = pVect(0,0,0);
    omega = pVect(0,0,0);
    radius = 1;
    mass = 0;
    mass_inv = 1;
    density = 1.00746;
    fdt_to_do = 0;
    locked = false;
    color = color_lsu_spirit_gold;
    specularity = 0;
    contact = false;
    orientation_set(pVect(0,1,0),0);
    world_time_expire = 3600.0 * 24 * 365 * 1000;
  }

pCoor position;
pVect velocity;

// Orientation of ball. Both orientation and omatrix updated by time
// step code.
pQuat orientation;
pMatrix3x3p omatrix;

pVect omega;                    // Spin rate and axis.
double world_time_expire;

int idx; // Position in balls_pos_rad;

union {
  float4 m_r_fdt;
  struct {
    float mass;
    float radius;
    float fdt_to_do; // Radius / moment of inertia.
    float _;
  }; };

float mass_min; // Mass below which simulation is unstable.
float radius_sq, radius_inv;  // Pre-computed based on radius.
float density;
float mass_inv;

bool locked;

pVect force;
pVect torque;
pColor color;
float specularity;              // Amount of shininess, in [0,1].
bool contact;                   // When true, ball rendered in gray.
float spring_constant_sum;      // Used to compute minimum mass.

void orientation_set(pNorm dir, float angle)
```

```
    { orientation_set(pQuat(dir,angle)); }
  void orientation_set(pQuat ori)
    {
      orientation = ori;
      omatrix = pMatrix3x3_Rotation(orientation);
    }
  void constants_update();
  pVect point_rot_vel(pNorm tact_dir);
  pVect point_rot_vel(pVect tact_dir);
  void apply_tan_force(pNorm tact_dir, pNorm force_dir, double force_mag);
  void apply_tan_force(pNorm tact_dir, pVect force);
  void push(pVect amt);
  void translate(pVect amt);
  void stop();
  void freeze();
};


void
Ball::constants_update()
{
  assert( radius );
  radius_sq = radius * radius;
  if ( mass >= 0 )
    {
      mass = 4.0 / 3 * numbers::pi * radius_sq * radius * density;
      mass_inv = 1 / mass;
    }
  radius_inv = 1 / radius;
  // FYI, notice simplifications:
  //   const float mo_inertia = 0.4 * mass * radius_sq;
  //   fdt_to_do = radius / mo_inertia;
  fdt_to_do = radius_inv * 2.5 * mass_inv;
}

pVect
Ball::point_rot_vel(pNorm direction)
{
  /// Return velocity of point on surface of ball.
  //
  return radius * cross( omega, direction );
}

pVect
Ball::point_rot_vel(pVect rel_pos)
{
  /// Return velocity of point relative to center.
  //
  return cross( omega, rel_pos );
}

void
Ball::apply_tan_force(pNorm tact_dir, pVect force)
```

```
{
  torque += cross(tact_dir, force);
}

void
Ball::apply_tan_force(pNorm tact_dir, pNorm force_dir, double force_mag)
{
  apply_tan_force(tact_dir, force_mag * force_dir);
}

struct Link_Args {
  pColor natural_color = color_red;
  float distance_relaxed = 0;
  float spring_constant = 0;
  float link_stiffness = 0;
  float radius = 0;
};

class Link {
public:
  Link(Ball *b1, Ball *b2, Link_Args args = {}):ball1(b1),ball2(b2),
     snapped(false)
  { init(args); }
  Link(Link *link, pVect cb1p, pVect cb2p, float drp):
    ball1(link->ball1), ball2(link->ball2), cb1(cb1p), cb2(cb2p),
    is_surface_connection(true), is_renderable(false),
    is_simulatable(true), distance_relaxed(drp), spring_constant(0),
    radius(0),snapped(false){link->is_simulatable = false;}

  Link(Link *link, pVect cb1p, pVect cb2p, Link_Args args = {} ):
    ball1(link->ball1), ball2(link->ball2), cb1(cb1p), cb2(cb2p),
    is_surface_connection(true), is_renderable(false),
    is_simulatable(true),
    distance_relaxed(args.distance_relaxed),
    spring_constant(args.spring_constant),
    radius(0),snapped(false){link->is_simulatable = false;}

  void init(Link_Args& args)
  {
    assert( args.distance_relaxed == 0 );
    natural_color = args.natural_color;
    spring_constant = args.spring_constant;
    radius = args.radius;
    init();
  }
  void init()
    {
      assert( ball1->radius > 0 );
      assert( ball2->radius > 0 );
      is_simulatable = true;
      is_renderable = true;
      pNorm n12(ball1->position,ball2->position);
      const float rad_sum = ball1->radius + ball2->radius;
      is_surface_connection = n12.magnitude >= rad_sum;
```

```
      pMatrix3x3 b1rot = ball1->omatrix;
      pMatrix3x3 b2rot = ball2->omatrix;

      pVect b1_y = b1rot * pVect(0,1,0);
      pVect b1_x = b1rot * pVect(1,0,0);
      bool b1_dir_is_y = fabs(dot(b1_y,n12)) < 0.999;
      pVect b1_dir_loc = b1_dir_is_y ? b1_y : b1_x;

      con_x = pNorm(cross(b1_dir_loc,n12));
      con_y = cross(n12,con_x);
      b1_dir = mult_MTV( b1rot, con_y );
      b2_dir = mult_MTV( b2rot, con_y );

      if ( !is_surface_connection )
        {
          distance_relaxed = n12.magnitude;
          cb1 = cb2 = pVect(0,0,0);
          is_renderable = false;
          return;
        }

      distance_relaxed = n12.magnitude - rad_sum;
      cb1 = ball1->radius * mult_MTV( b1rot,  n12 );
      cb2 = ball2->radius * mult_MTV( b2rot, -n12 );
    }
  int serial;
  int idx;  // Position in lis_pos1, etc.
  Ball* const ball1;
  Ball* const ball2;
  union {
    int2 ball_idx;
    struct { int ball1_idx, ball2_idx; };
  };
  pVect cb1, cb2, con_x, con_y, b1_dir, b2_dir;
  bool is_surface_connection;
  bool is_renderable, is_simulatable;
  union {
    float2 dr_sc;
    struct {
      float distance_relaxed;
      float spring_constant;
    };
  };
  float radius;
  bool snapped;
  pColor natural_color;
  pColor color;

  pVect spring_force_12;
  pVect torque1,torque2;

};
```

```
class Surface {
public:
  Ball *b0, *b1, *b2;
  pColor color = color_blue;
  pTCoor tc0, tc1, tc2;
  union {
    int4 b_idx;
    struct { int b0_idx, b1_idx, b2_idx, _; };
  };
  Surface* operator ->() { return this; }
};

// Declare containers for Balls and Links. The containers are derived
// from std::vector and use overloaded operator += for push_back. (See
// include/util-containers.h.)
//
typedef pVectorI<Link> Links;
typedef pVectorI<Ball> Balls;


class World {
public:
  World(pVulkan_Helper &fb, int argc, char **argv)
    :vh(fb), ff_state(fb.qs), raytrace(fb.qs),
     frame_timer(vh.frame_timer), transform(fb.qs),
     shapes(ff_state), sphere(ff_state),
     opt_tryout1(fb.qs.opt_tryout1),opt_tryout2(fb.qs.opt_tryout2),
     opt_tryout3(fb.qs.opt_tryout3)
  {init(argc,argv);}
  void init(int argc, char **argv);
  void init_graphics();
  void run();
  void frame_callback();
  void render(vk::CommandBuffer& cb);
  void render_objects(vk::CommandBuffer& cb, Render_Option render_option);
  void objects_erase();
  void cb_keyboard();
  void modelview_update();

  pVulkan_Helper& vh;
  VFixed_Function_State_Manager ff_state;
  VRaytrace raytrace;
  pFrame_Timer& frame_timer;
  VTransform transform;
  Shapes shapes;
  Sphere sphere;

  bool opt_want_raytrace;

  pVariable_Control variable_control;
  double world_time;
  double last_frame_wall_time;
  float opt_time_step_duration;
```

```
int time_step_count;
float opt_gravity_accel;      // Value chosen by user.
pVect gravity_accel;          // Set to zero when opt_gravity is false;
bool opt_gravity;
bool opt_head_lock, opt_tail_lock;
bool &opt_tryout1, &opt_tryout2, &opt_tryout3;  // For ad-hoc experiments.
float opt_tryoutf;
bool opt_sphere_true;
VBufferV<Shdr_Uni_Common> buf_uni_common;

// Tiled platform for ball.
//
float platform_xmin, platform_xmax, platform_zmin, platform_zmax;
float platform_pi_xwidth_inv;
VVertex_Buffer_Set bset_platform_tex, bset_platform_mir;
VPipeline pipe_platform_mir, pipe_platform_tex;
RT_Geo_Package rt_geo_platform_tex, rt_geo_platform_mir;

vk::Sampler sampler;
VTexture texid_hw;
VTexture texid_syl;
VTexture texid_emacs;

void platform_update();
bool platform_collision_possible(pCoor pos);

pCoor light_location;
float opt_light_intensity;
enum { MI_Eye, MI_Light, MI_Ball, MI_Ball_V, MI_COUNT } opt_move_item;
bool opt_pause;
bool opt_single_frame;      // Simulate for one frame.
int opt_single_time_step;  // Simulate for one time step.

VBufferV<Uni_Lighting> uni_light, uni_light_ambient, uni_light_all;
VBufferV<Uni_Lighting> uni_light_reflected;
VBufferV<Uni_Lighting> *puni_light_curr;

pCoor eye_location;
pVect eye_direction;
pMatrix transform_projection;
pMatrix modelview;

pVect adj_vector;
double adj_t_prev;
double adj_t_stop;
double adj_duration;

uint opt_shader;

bool opt_mirror;
int opt_mirror_depth;
bool opt_shadows;
bool opt_shadow_volumes;      // Make shadow volumes visible.
bool opt_normal_sphere;
```

```
int opt_hide_stuff;

int last_setup; // Last scene set up.

void ball_setup_1(bool big);  // Silly Tree
void ball_setup_2(); // Hanging thing.
void ball_setup_3(); // Corona virus particle.
void ball_setup_4(); // Wheel.
void ball_setup_5(); // Top.
void ball_setup_6(); // Wheel with webbing between spokes.
void ball_setup_7();
void setup_at_end();
void time_step_cpu(double);
void balls_stop();
void balls_freeze();
void balls_translate(pVect amt, int idx);
void balls_translate(pVect amt);
void balls_push(pVect amt, int idx);
void balls_push(pVect amt);
void balls_twirl();
void lock_update();

Ball *make_marker(pCoor pos, pColor col, float radius = 0.2 );
Balls balls_marker; // Written to balls right after setup.

float opt_spring_constant, opt_spring_damp;
float opt_friction_dynamic;
float opt_air_resistance;
float distance_relaxed;
int chain_length;
Balls balls;
Ball *head_ball, *tail_ball;
Links links;
vector<Surface> surfaces;
int opt_sphere_slices;
float opt_omega;  // For disk and top.

Links link_new
( Ball *ball1, Ball *ball2,
  float stiffness = 0.2, float spring_constant = 0);
Links link_new
( Ball *ball1, Ball *ball2, Link_Args largs );

/// CUDA Stuff

int opt_physics_method;
GPU_Info gpu_info;
cudaEvent_t frame_start_ce, frame_stop_ce;
int opt_block_size;
bool gpu_const_data_stale;
bool cpu_phys_data_stale;

void data_cpu_to_gpu_constants();
```

```
  void data_cpu_to_gpu_dynamic();
  void data_gpu_to_cpu_dynamic();
  CPU_GPU_Common c;  // c is for common.  See links.cuh.

  void render_link_start();
  void render_link_gather(Link *link);
  void render_link_render(vk::CommandBuffer& cb, bool shadow_volumes = false);
  void render_link_render_sv(vk::CommandBuffer& cb)
  { render_link_render(cb,true); };

  VPipeline pipe_set_2, pipe_set_2_sv;
  double world_time_link_update;

  VCompute comp_links;
  VBuffer<int> buf_links_as_idx; // Indices for links' acceleration structures.
  VBuffer<vec2> buf_links_tex_coor;
  VBuffer<vec4> buf_links_vertex_g;
  VBuffer<vec4> buf_links_normal_g;
  RT_Geo_Package rt_geo_links;

  int opt_sides, opt_segments;

  vector< VBufferVV<vec4>* > lis_bufs;
  VBufferVV<vec4> lis_pos1, lis_pos2;
  VBufferVV<vec4> lis_v1, lis_v2;
  VBufferVV<vec4> lis_b1_ydir, lis_b2_ydir, lis_tex_rad;

  VVertex_Buffer_Set bset_surfaces;
  VPipeline pipe_surfaces;
  RT_Geo_Package rt_geo_surfaces;
  void render_surfaces(vk::CommandBuffer& cb);


  /// Homework 5 (2025)
  //
  VVertex_Buffer_Set bset_doda;
  VPipeline pipe_doda;
  VVertex_Buffer_Set bset_doda_pts_clean;
  VPipeline pipe_doda_pts_clean;
  VPipeline pipe_doda_pts_prob_1;
  VVertex_Buffer_Set bset_doda_inst_prob_2;
  VPipeline pipe_doda_inst_prob_2;
  void render_doda(vk::CommandBuffer& cb);

  VBufferVV<pCoor> buf_dod_coors_l;
  VBufferVV<pColor> buf_hw05_color;
  VBufferVV<mat4> buf_hw05_matrix;
  VBufferVV<pVect> buf_hw05_vector;
  VBufferVV<pCoor> buf_hw05_coor;

  float opt_hw05_height_stel;
  bool opt_hw05_dot, opt_hw05_tex, opt_hw05_stel;
  pVariable_Control_Elt *hw05_vc_height_stel;
};
```

```
enum { OH_Links = 1, OH_Platform = 2, OH_Sphere = 4 };

void
World::init_graphics()
{
  vh.want_raytrace().init();
  vh.opt_record_every_time = true;
  vh.display_cb_set([&](){frame_callback();});
  vh.cbs_cmd_record.push_back( [&](vk::CommandBuffer& cb){ render(cb); });
  sphere.ppuni_light = &puni_light_curr;
  sphere.transform = &transform;

  opt_want_raytrace = false;

  if ( vh.have_raytrace )
    {
      raytrace.init();
      for ( auto b: { &bset_platform_mir, &bset_platform_tex, &bset_surfaces,
                      &bset_doda
        } )
        b->usage_set
          ( vk::BufferUsageFlagBits::eVertexBuffer
            | vk::BufferUsageFlagBits::eStorageBuffer );
      buf_links_as_idx.init
        ( vh.qs, vk::BufferUsageFlagBits::eStorageBuffer, 0,
          vk::MemoryPropertyFlagBits::eDeviceLocal );
      buf_links_tex_coor.init
        ( vh.qs, vk::BufferUsageFlagBits::eStorageBuffer, 0,
          vk::MemoryPropertyFlagBits::eDeviceLocal );
      buf_links_vertex_g.init
        ( vh.qs, vk::BufferUsageFlagBits::eStorageBuffer, 0,
          vk::MemoryPropertyFlagBits::eDeviceLocal );
      buf_links_normal_g.init
        ( vh.qs, vk::BufferUsageFlagBits::eStorageBuffer, 0,
          vk::MemoryPropertyFlagBits::eDeviceLocal );
    }

  buf_uni_common.init(vh.qs,vk::BufferUsageFlagBits::eUniformBuffer);
  sphere.buf_uni_common = buf_uni_common;

  lis_bufs =
    { &lis_pos1, &lis_pos2,
      &lis_v1, &lis_v2,
      &lis_b1_ydir, &lis_b2_ydir,
      &lis_tex_rad };

  for ( auto b: lis_bufs )
    b->init(vh.dev_phys, vh.dev, vk::BufferUsageFlagBits::eStorageBuffer );

  ///
  /// Graphical Model Initialization
  ///
```

```
opt_head_lock = false;
opt_tail_lock = false;
opt_tryout1 = opt_tryout2 = opt_tryout3 = false;
opt_normal_sphere = true;
opt_sphere_true = true;

eye_location = pCoor(20,13.1,-25);
eye_direction = pVect(-0.42,-0.09,0.9);

platform_xmin = -40; platform_xmax = 40;
platform_zmin = -40; platform_zmax = 40;

texid_syl.init
  ( vh.qs, P_Image_Read( vh.gp_root_get() / "vulkan" / "gpup.png", 255 ) );
texid_emacs.init( vh.qs, P_Image_Read("mult.png",-1) );
texid_hw.init( vh.qs, P_Image_Read("wiki-dod.png",255) );
sampler = vh.qs.dev.createSampler
  ( { {},
      vk::Filter::eLinear,  // magFilter
      vk::Filter::eLinear,  // minFilter
      vk::SamplerMipmapMode::eLinear,
      // Also: eRepeat, eMirroredRepeat, eClampToEdge, etc.
      vk::SamplerAddressMode::eRepeat,
      vk::SamplerAddressMode::eRepeat,
      vk::SamplerAddressMode::eRepeat,
      0.0f,  // mipLodBias
      true, // anisotropyEnable,
      16.0f, // maxAnisotropy
      false, vk::CompareOp::eNever,  // compareEnable, compareOp
      0.0f, VK_LOD_CLAMP_NONE,  // min and max LOD
      vk::BorderColor::eFloatOpaqueBlack,
      false // unnormalizedCoordinates
      } );

sphere.texture_set(sampler,texid_emacs);

opt_light_intensity = 50;
light_location = pCoor(platform_xmax,platform_xmax,platform_zmin);
uni_light.init(vh.qs,vk::BufferUsageFlagBits::eUniformBuffer);
uni_light_all.init(vh.qs,vk::BufferUsageFlagBits::eUniformBuffer);
uni_light_reflected.init(vh.qs,vk::BufferUsageFlagBits::eUniformBuffer);
uni_light_ambient.init(vh.qs,vk::BufferUsageFlagBits::eUniformBuffer);

variable_control.insert(opt_light_intensity,"Light Intensity");

opt_move_item = MI_Eye;
opt_pause = false;
opt_single_time_step = 0;
opt_single_frame = false;

platform_update();
modelview_update();

adj_t_stop = 0;
```

```
  adj_duration = 0.25;

  opt_sides = 20;
  opt_segments = 16;
  if ( opt_all_ui_variables )
    {
      variable_control.insert(opt_sides,"Link Sides");
      variable_control.insert(opt_segments,"Link Segments",5,1);
    }
  opt_hide_stuff = 0;

  if ( vh.have_raytrace )
    {
      raytrace.buf_light_set(uni_light_all);
    }

  opt_tryoutf = 0;
  variable_control.insert_linear(opt_tryoutf,"Tryout F",0.1);

  opt_hw05_height_stel = 1;
  hw05_vc_height_stel =
    &variable_control.insert_linear
    (opt_hw05_height_stel,"opt_hw05_height_stel",0.1);

  opt_hw05_dot = true;
  opt_hw05_tex = true;
  opt_hw05_stel = true;
  variable_control.make_current(hw05_vc_height_stel);
}

void
World::run()
{
  vh.message_loop_spin();

  uni_light.destroy();
  uni_light_ambient.destroy();
  uni_light_all.destroy();
  uni_light_reflected.destroy();
  transform.destroy();
  buf_uni_common.destroy();

  for ( auto b: lis_bufs ) b->destroy();

  pipe_platform_tex.destroy(); pipe_platform_mir.destroy();
  bset_platform_tex.destroy(); bset_platform_mir.destroy();
  pipe_set_2.destroy();
  pipe_set_2_sv.destroy();

  texid_syl.destroy();
  texid_emacs.destroy();
  texid_hw.destroy();
  vh.dev.destroySampler(sampler);
  shapes.destroy();
```

```
  sphere.destroy();

  bset_surfaces.destroy();
  pipe_surfaces.destroy();
  rt_geo_surfaces.destroy();

  bset_doda.destroy();
  pipe_doda.destroy();
  bset_doda_pts_clean.destroy();
  bset_doda_inst_prob_2.destroy();
  pipe_doda_pts_clean.destroy();
  pipe_doda_pts_prob_1.destroy();
  pipe_doda_inst_prob_2.destroy();
  buf_dod_coors_l.destroy();
  buf_hw05_color.destroy();
  buf_hw05_matrix.destroy();
  buf_hw05_vector.destroy();
  buf_hw05_coor.destroy();


  buf_links_as_idx.destroy();
  buf_links_tex_coor.destroy();
  buf_links_normal_g.destroy();
  buf_links_vertex_g.destroy();
  rt_geo_platform_tex.destroy();
  rt_geo_platform_mir.destroy();
  rt_geo_links.destroy();
  comp_links.destroy();
  raytrace.destroy();

  vh.finish();
}

void
World::render_surfaces(vk::CommandBuffer& cb)
{
  if ( !pipe_surfaces )
    pipe_surfaces
      .init( vh.qs )
      .ds_follow( puni_light_curr )
      .ds_follow( transform )
      .ds_uniform_use( "BIND_UNI_COMMON", buf_uni_common )
      .topology_set( vk::PrimitiveTopology::eTriangleList )
      .create();

  bset_surfaces.reset( pipe_surfaces );

  for ( Surface& s: surfaces )
    {
      Ball &b0 = *s.b0, &b1 = *s.b1, &b2 = *s.b2;
      pNorm snorm = cross(b0.position,b1.position,b2.position);
      bset_surfaces << b0.position << b1.position << b2.position;
      bset_surfaces << s.color << s.color << s.color;
      bset_surfaces << snorm << snorm << snorm;
```

```
    }

  bset_surfaces.to_dev();

  if ( vh.have_raytrace && !rt_geo_surfaces )
    rt_geo_surfaces.init( raytrace );

  if ( vh.raytrace_active() )
    {
      rt_geo_surfaces.bind( bset_surfaces, RT_Geometry_Triangles );
      return;
    }

  pipe_surfaces.record_draw( cb, bset_surfaces );
}

void
World::render_doda(vk::CommandBuffer& cb)
{
  auto shader_stages_all =
    vk::ShaderStageFlagBits::eVertex |
    vk::ShaderStageFlagBits::eGeometry |
    vk::ShaderStageFlagBits::eFragment;

  if ( !buf_hw05_color )
    {
      buf_dod_coors_l.init( vh.qs, vk::BufferUsageFlagBits::eStorageBuffer );
      buf_hw05_color.init( vh.qs, vk::BufferUsageFlagBits::eStorageBuffer );
      buf_hw05_matrix.init( vh.qs, vk::BufferUsageFlagBits::eStorageBuffer );
      buf_hw05_vector.init( vh.qs, vk::BufferUsageFlagBits::eStorageBuffer );
      buf_hw05_coor.init( vh.qs, vk::BufferUsageFlagBits::eStorageBuffer );
    }

  if ( !pipe_doda )
    pipe_doda
      .init( vh.qs )
      .ds_follow( puni_light_curr )
      .ds_follow( transform )
      .ds_uniform_use( "BIND_UNI_COMMON", buf_uni_common )
      .ds_use( sampler, texid_hw )
      .shader_inputs_info_set<pCoor, pColor, pMatrix>()
      .shader_code_set
      ("hw05-shdr.cc",
       "vs_main();",
       "gs_main();",
       "fs_main();", "#define HW5_TRI\n" )
      .topology_set( vk::PrimitiveTopology::eTriangleList )
      .create();

  if ( !pipe_doda_pts_clean )
    {
      pipe_doda_pts_clean
        .init( vh.qs )
        .ds_follow( puni_light_curr )
```

```
      .ds_follow( transform )
      .ds_use( sampler, texid_hw )
      .ds_uniform_use( "BIND_UNI_COMMON", buf_uni_common );

    pipe_doda_pts_clean
      .ds_storage_follow
      ( "BIND_HW5_COLOR", buf_hw05_color, shader_stages_all );

    pipe_doda_pts_clean
      .shader_inputs_info_set<pCoor, pVect, pMatrix, int>()
      .shader_code_set
      ("hw05-shdr.cc",
       "vs_main_pts_clean();",
       "gs_main_pts_clean();",
       "fs_main();", "#define HW5_PTS_CLEAN\n" )
      .topology_set( vk::PrimitiveTopology::ePointList );

    pipe_doda_pts_clean.create();
  }

if ( !pipe_doda_pts_prob_1 )
  {
    // Problem 1: Modify some of the calls below or add new ones.

    pipe_doda_pts_prob_1
      .init( vh.qs )
      .ds_follow( puni_light_curr )
      .ds_follow( transform )
      .ds_use( sampler, texid_hw )
      .ds_uniform_use( "BIND_UNI_COMMON", buf_uni_common );

    /// SOLUTION -- Problem 1
    //
    // In addition to buf_hw05_color, bind buf_hw05_matrix
    // and buf_dod_coors_l to the Problem 1 pipeline.
    //
    pipe_doda_pts_prob_1
      .ds_storage_follow
      ( "BIND_HW5_COLOR", buf_hw05_color, shader_stages_all )
      .ds_storage_follow
      ( "BIND_HW5_MATRIX", buf_hw05_matrix, shader_stages_all )
      .ds_storage_follow
      ( "BIND_DOD_COORS_L", buf_dod_coors_l, shader_stages_all );

    pipe_doda_pts_prob_1
      .shader_inputs_info_set()
      .shader_code_set
      ("hw05-shdr-sol.cc",
       "vs_main_pts_prob_1();",
       "gs_main_pts_prob_1();",
       "fs_main();", "#define HW5_PTS_PROB_1\n" );

    pipe_doda_pts_prob_1
      .topology_set( vk::PrimitiveTopology::ePointList )
```

```
      .create();
  }

if ( !pipe_doda_inst_prob_2 )
  {
    /// SOLUTION -- Problem 2
    //
    //  In addition to buf_hw05_color, bind buf_hw05_matrix
    //  to the pipeline.
    //
    pipe_doda_inst_prob_2
      .init( vh.qs )
      .ds_follow( puni_light_curr )
      .ds_follow( transform )
      .ds_use( sampler, texid_hw )
      .ds_uniform_use( "BIND_UNI_COMMON", buf_uni_common )
      .ds_storage_follow
      ( "BIND_HW5_COLOR", buf_hw05_color, shader_stages_all )
      .ds_storage_follow
      ( "BIND_HW5_MATRIX", buf_hw05_matrix, shader_stages_all );

    /// SOLUTION -- Problem 2
    //
    //  Remove the int and pMatrix from the buffer set (and so
    //  shader_inputs_info_set), but add pTCoor.
    //
    pipe_doda_inst_prob_2
      .shader_inputs_info_set<pCoor,pTCoor>()
      .shader_code_set
      ("hw05-shdr-sol.cc",
       "vs_main_inst_prob_2();",
       "gs_main_inst_prob_2();",
       "fs_main();", "#define HW5_INST_PROB_2\n" );

    pipe_doda_inst_prob_2
      .topology_set( vk::PrimitiveTopology::eTriangleList )
      .create();
  }

// Compute Dodecahedron Vertex Coordinates -- Including Duplicates
//
if ( buf_dod_coors_l.size() == 0 )
  {
    // Dodecahedron Center
    //
    pCoor center(0,0,0);
    pVect ax(1,0,0), ay(0,1,0), az(0,0,1);

    // Inscribed Sphere Radius
    //
    float ri = 1;

    const float len_edge = ri / sqrt( sqrt(5) * pow(numbers::phi,5) / 20 );
```

```
    // Distance from pentagon center to pentagon vertex.
    //
    const float rc_pentagon = len_edge / ( 2 * sinf(numbers::pi/5) );

    // Use same coordinate axes as the cylinder.
    pCoor center_pentagon_0 = center + ri * az;

    // Dihedral Angle
    //
    float theta_dihedral = 2 * atanf( numbers::phi );

    ///
    /// Compute Coordinates of First Pentagon
    ///
    const float d_theta = 2 * numbers::pi / 5;
    for ( int i=0; i<5; i++ )
      {
        const float theta = d_theta * i;
        pVect dir = cosf(theta) * ax + sinf(theta) * ay;
        pCoor pos = center_pentagon_0 + rc_pentagon * dir;
        buf_dod_coors_l << pos;
      }

    ///
    /// Compute Coordinates of the Next Five Pentagons
    ///
    for ( int i=0; i<5; i++ )
      {
        pCoor p0 = buf_dod_coors_l[i], p1 = buf_dod_coors_l[(i+1)%5];
        pMatrix_Translate to_p0( p0 );
        pMatrix_Rotation rot2( pNorm(p0,p1), -theta_dihedral );
        pMatrix_Translate to_Origin( pVect( p0, pCoor(0,0,0) ) );
        pMatrix m = to_p0 * rot2 * to_Origin;
        for ( int j=4; j>=0; j-- ) buf_dod_coors_l << m * buf_dod_coors_l[j];
      }

    ///
    /// Compute Coordinates of Last Six Pentagons
    ///

    pMatrix_Rotation rot3(ay,numbers::pi);
    const int n_vtx_demi = buf_dod_coors_l.size();
    for ( int i=0; i<n_vtx_demi; i++ )
      buf_dod_coors_l << rot3 * buf_dod_coors_l[i];

    buf_dod_coors_l.to_dev();
  }

// First pentagon:  buf_dod_coors_l[0] through buf_dod_coors_l[4].
// Second pentagon: buf_dod_coors_l[5] through buf_dod_coors_l[9].
// And so on.

switch( opt_shader ){
```

```
  case SO_HW5_Triangles:
    /// Shown for reference, don't modify.
    //
    bset_doda.reset( pipe_doda );

    for ( Ball* b: balls )
      {
        pMatrix rot(b->omatrix);
        pMatrix_Translate tra(b->position);
        pMatrix_Scale sca(b->radius);
        pMatrix m = tra * sca * rot;

        /// Triangles.
        //
        for ( size_t i=0; i<buf_dod_coors_l.size(); i+=5 )
          // Render the pentagon by emitting three triangles, all
          // share pentagon vertex 0 (index i).
          //
          for ( int j=1; j<4; j++ )
            bset_doda
              << buf_dod_coors_l[ i ]
              << buf_dod_coors_l[ i + j ]
              << buf_dod_coors_l[ i + j + 1 ];

        for ( int i=0; i<3*3*12; i++ ) bset_doda << b->color << m;

      }

    bset_doda.to_dev();
    pipe_doda.record_draw( cb, bset_doda );

    break;

  case SO_HW5_Points_Clean:
    /// Shown for reference, don't modify.
    //
    bset_doda_pts_clean.reset( pipe_doda_pts_clean );
    buf_hw05_color.clear();

    for ( Ball* b: balls )
      {
        pMatrix rot(b->omatrix);
        pMatrix_Translate tra(b->position);
        pMatrix_Scale sca(b->radius);
        pMatrix m = tra * sca * rot;

        const int color_idx = buf_hw05_color.size();
        buf_hw05_color << b->color;

        /// Points Clean -- Do Not Modify
        //
        for ( size_t i=0; i<buf_dod_coors_l.size(); i+=5 )
          {
            pNorm n( cross( buf_dod_coors_l[i+2],
```

```
                                buf_dod_coors_l[i+1],
                                buf_dod_coors_l[i  ] ) );
                  // pCoor pent_ctr = dod_ctr + n; // Note: inscribed radius is 1.

                  bset_doda_pts_clean << m << buf_dod_coors_l[i] << n << color_idx;
                }
              }

            bset_doda_pts_clean.to_dev();
            buf_hw05_color.to_dev();
            pipe_doda_pts_clean.record_draw( cb, bset_doda_pts_clean );

            break;

          case SO_HW5_Points_Prob_1:
            {
              /// Problem 1 solution goes here and in shader file, hw05-shdr.cc.
              //

              bset_doda_inst_prob_2.reset( pipe_doda_inst_prob_2 );
              buf_hw05_color.clear();

              /// SOLUTION -- Problem 1
              buf_hw05_matrix.clear();

              for ( Ball* b: balls )
                {
                  pMatrix rot(b->omatrix);
                  pMatrix_Translate tra(b->position);
                  pMatrix_Scale sca(b->radius);
                  pMatrix m = tra * sca * rot;

                  buf_hw05_color << b->color;

                  /// SOLUTION -- Problem 1
                  //
                  //  Put transformation matrix in a storage buffer rather than
                  //  in a buffer set.
                  //
                  buf_hw05_matrix << m;

                  /// Points
                  //
                  // Zero inputs, so defined-out code below not needed.
#if 0
                  for ( size_t i=0; i<buf_dod_coors_l.size(); i+=5 )
                    {

                      pNorm n( cross( buf_dod_coors_l[i+2],
                                      buf_dod_coors_l[i+1],
                                      buf_dod_coors_l[i  ] ) );

                      bset_doda_pts_prob_1 << m << dod_idx << buf_dod_coors_l[i] << n;
                    }
```

```
#endif
        }

      /// SOLUTION -- Problem 1
      //
      // Send matrices and colors to the GPU.
      //
      buf_hw05_matrix.to_dev();
      buf_hw05_color.to_dev();
      //
      // Compute the number of "vertices", which is the number of
      // dodecahedra times 12.
      //
      const int n_vertices = buf_hw05_matrix.size() * 12;

      pipe_doda_pts_prob_1.record_draw( cb, n_vertices );
    }
    break;

  case SO_HW5_Inst_Prob_2:
    {
      /// Problem 2 solution goes here and in shader file, hw05-shdr.cc.
      //

      /// SOLUTION -- Problem 2

      // If buffer set is empty then initialize it with a dodecahedron
      // local-space vertices and texture coordinates. This should
      // only be done once because local-space coordinates don't
      // change.
      //
      if ( bset_doda_inst_prob_2.size() == 0 )
        {
          bset_doda_inst_prob_2.reset( pipe_doda_inst_prob_2 );

          for ( size_t i=0; i<buf_dod_coors_l.size(); i += 5 )
            {
              pNorm az =
                cross( pVect(buf_dod_coors_l[i+1], buf_dod_coors_l[ i+2 ] ),
                       pVect(buf_dod_coors_l[i+1], buf_dod_coors_l[ i+0 ] ) );

              pCoor pent_ctr_l = pCoor(0,0,0) + az;
              pNorm ax( pent_ctr_l, buf_dod_coors_l[i] );
              pNorm ay = cross(az,ax);
              float pent_ri = ax.magnitude * cosf( 2 * numbers::pi / 10 );
              pMatrix_Translate to_ctr(-pent_ctr_l);
              pMatrix_Rows rot(ax,ay,pVect(0,0,0));
              pMatrix_Scale scale(1/pent_ri);
              pMatrix m = scale * rot * to_ctr;

              // Write five triangles for each face, so that the
              // geometry shader can more easily tessellate.
              //
              // For each vertex write its dodecahedron local-space
```

```
          // coordinate. Also write a pentagon local space 2D
          // coordinate, to be used as a texture coordinate.

          // Start by computing local space coordinates and
          // temporarily storing them in a std::vector
          //
          vector<pCoor> coors;
          for ( int j=0; j<5; j++ )
            coors << pent_ctr_l
                  << buf_dod_coors_l[ i + j ]
                  << buf_dod_coors_l[ i + ( j + 1 ) % 5 ];
          //
          // Next, insert each coordinate into the buffer set, and
          // also insert a texture coordinate (pentagon local
          // space coordinate) computed using the dodecahedron
          // coordinate.
          //
          for ( pCoor c: coors )
            bset_doda_inst_prob_2 << c << (m * c).tcoor();
        }
      //
      // Don't forget that the code above is executed just
      // once per run. Therefore we only need to send
      // the buffer set to the GPU just once.
      //
      bset_doda_inst_prob_2.to_dev();
    }

  /// SOLUTION -- Problem 2
  //
  //  Update storage buffers with the latest transformation
  //  matrices and colors for each dodecahedron.

  buf_hw05_color.clear();
  buf_hw05_matrix.clear();

  for ( Ball* b: balls )
    {
      pMatrix rot(b->omatrix);
      pMatrix_Translate tra(b->position);
      pMatrix_Scale sca(b->radius);
      pMatrix m = tra * sca * rot;

      buf_hw05_color << b->color;
      buf_hw05_matrix << m;
    }

  buf_hw05_color.to_dev();
  buf_hw05_matrix.to_dev();

  const int n_instances = buf_hw05_matrix.size();

  pipe_doda_inst_prob_2.record_draw_instanced
    ( cb, bset_doda_inst_prob_2, n_instances );
```

```
    }
    break;

  default: assert( false );
  }


}

void
World::platform_update()
{
  const float tile_count = 19;
  const float ep = 1.00001;
  const float delta_x = ( platform_xmax - platform_xmin ) / tile_count * ep;
  const float zdelta = ( platform_zmax - platform_zmin ) / tile_count * ep;

  const float trmin = 0.05;
  const float trmax = 0.7;
  const float tsmin = 0;
  const float tsmax = 0.4;
  pColor color_platform( color_white * 0.9 );

  if ( !pipe_platform_mir )
    pipe_platform_mir
      .init( ff_state )
      .ds_set( color_lsu_spirit_purple )
      .ds_follow( puni_light_curr )
      .ds_follow( transform )
      .topology_set( vk::PrimitiveTopology::eTriangleList )
      .create();
  bset_platform_mir.reset( pipe_platform_mir ).tolerate_tcoords();

  if ( !pipe_platform_tex )
    pipe_platform_tex
      .init( ff_state )
      .ds_set_material(color_platform)
      .ds_follow( puni_light_curr )
      .ds_follow( transform )
      .topology_set( vk::PrimitiveTopology::eTriangleList )
      .ds_use( sampler, texid_syl )
      .create();
  bset_platform_tex.reset( pipe_platform_tex );

  const pVect platform_normal(0,1,0);
  pTCoor t00(trmin,tsmin), t01(trmin,tsmax), t10(trmax,tsmin), t11(trmax,tsmax);
  bool even = true;

  for ( int i = 0; i < tile_count; i++ )
    {
      const float x0 = platform_xmin + i * delta_x;
      const float x1 = x0 + delta_x;
      const float y = 0;
      for ( float z = platform_zmin; z < platform_zmax; z += zdelta )
```

```
        {
          auto& bset = even ? bset_platform_tex : bset_platform_mir;
          const float z1 = z+zdelta;
          pCoor p00(x0,y,z), p01(x0,y,z1), p10(x1,y,z), p11(x1,y,z1);

          bset << t11 << platform_normal << p00;
          bset << t10 << platform_normal << p01;
          bset << t00 << platform_normal << p11;

          bset << t11 << platform_normal << p00;
          bset << t00 << platform_normal << p11;
          bset << t01 << platform_normal << p10;

          even = !even;
        }
    }

  bset_platform_mir.to_dev();
  bset_platform_tex.to_dev();

  if ( vh.have_raytrace && !rt_geo_platform_mir )
    {
      pColor mirror_color(color_lsu_spirit_purple);
      mirror_color.a = -0.5;
      rt_geo_platform_mir
        .init( raytrace )
        .color_front_set( mirror_color )
        .bind( bset_platform_mir, RT_Geometry_Triangles );
      rt_geo_platform_tex
        .init( raytrace )
        .color_front_set( color_platform )
        .bind( VR_BUF_IDX_sampler, sampler, texid_syl )
        .bind( bset_platform_tex, RT_Geometry_Triangles );
    }
}

void
World::modelview_update()
{
  pMatrix_Translate center_eye(-eye_location);
  pMatrix_Rotation rotate_eye(eye_direction,pVect(0,0,-1));
  modelview = rotate_eye * center_eye;
}

void
World::render_objects(vk::CommandBuffer& cb, Render_Option option)
{
  //  const float shininess_ball = 5;
  pColor spec_color(1,1,1);
  //  pColor spec_light = color_white * opt_light_intensity;

  const bool hide_links = opt_hide_stuff & OH_Links;
  const bool hide_platform = opt_hide_stuff & OH_Platform;
  const bool hide_sphere = opt_hide_stuff & OH_Sphere;
```

```
  sphere.init(opt_sphere_slices);

  if ( option == RO_Shadow_Volumes )
    {
      if ( !hide_sphere )
        {
          sphere.light_pos = light_location;
          sphere.render_bunch_gather_sv(world_time);
          for ( Ball *ball: balls )
            sphere.render_shadow_volume(cb,ball->radius,ball->position);

          sphere.render_bunch_render_sv(cb);
        }

      if ( !hide_links )
        {
          render_link_start();
          for ( Link *link: links )
            if ( link->is_renderable ) render_link_gather(link);
          render_link_render_sv(cb);
        }
    }
  else
    {
      sphere.rt_geo_sphere.rt_wanted_set(!hide_sphere);
      if ( !hide_sphere && opt_shader == SO_Plain )
        {
          sphere.render_bunch_gather(world_time);
          for ( Ball *ball: balls )
            {
              if ( ball->contact )
                sphere.color = color_gray;
              else if ( ball->mass > 0 && ball->mass < ball->mass_min )
                sphere.color = color_red;
              else if ( ball->mass > 0 && ball->locked )
                sphere.color = color_pale_green;
              else
                sphere.color = ball->color;

              sphere.color.a = ball->specularity ? -ball->specularity : 1;
              sphere.render
                (cb, ball->radius,ball->position,
                 pMatrix_Rotation(ball->orientation));
            }

          sphere.render_bunch_render(cb, opt_sphere_true);
        }

      rt_geo_links.rt_wanted_set(!hide_links);
      if ( !hide_links )
        {
          render_link_start();
          for ( Link *link: links )
```

```
            if ( link->is_renderable ) render_link_gather(link);
          render_link_render(cb);
        }
    }

  rt_geo_platform_tex.rt_wanted_set(!hide_platform);
  rt_geo_platform_mir.rt_wanted_set(!hide_platform);

  if ( option != RO_Shadow_Volumes && opt_shader != SO_Plain )
    render_doda(cb);

  if ( option != RO_Shadow_Volumes )
    render_surfaces(cb);

  if ( vh.raytrace_active() ) return;
  if ( hide_platform ) return;

  if ( option == RO_Shadow_Volumes ) return;

  //
  // Render Platform
  //
  pipe_platform_tex.record_draw(cb,bset_platform_tex);
}

void
World::render(vk::CommandBuffer& cb)
{
  const bool do_raytrace = vh.raytrace_active();

  /// Emit a Graphical Representation of Simulation State
  //

  // Understanding of the code below not required for introductory
  // lectures.

  // That said, much of the complexity of the code is to show
  // the ball shadow and reflection.


  const int win_width = vh.get_width();
  const int win_height = vh.get_height();
  const float aspect = float(win_width) / win_height;
  const float n_dist = 0.01;
  const float xr = .8 * n_dist;

  transform.eye_from_global_set( modelview );
  transform.clip_from_eye_set
    ( pMatrix_Frustum(-xr,xr,-xr/aspect,xr/aspect,n_dist,5000) );
  // pMatrix_Frustum: left, right, bottom, top, near, far

  float fb_dark = 0.3;

  vh.clearValues[0].color =
```

```
  vk::ClearColorValue( array<float, 4>( { {fb_dark,fb_dark,fb_dark,0.5} } ) );

/// Lighting
//
auto& light_all = uni_light_all->cgl_LightSource[0];

light_all.position = transform.eye_from_global * light_location;

light_all.constantAttenuation = 0.5;
light_all.linearAttenuation = 1.0;
light_all.quadraticAttenuation = 0;

pColor ambient_color(0x555555);

uni_light_all->cgl_LightModel.ambient = ambient_color;

light_all.diffuse = opt_light_intensity * color_white;
light_all.ambient = 12 * color_white;
light_all.specular = opt_light_intensity * color_white;

uni_light_reflected = uni_light_all.get_val();
uni_light_reflected->cgl_LightSource[0].position =
  transform.eye_from_global * pMatrix_Scale(1,-1,1) * light_location;

uni_light_ambient = uni_light_all.get_val();
auto& light_amb = uni_light_ambient->cgl_LightSource[0];
light_amb.diffuse = 12 * color_white;
light_amb.specular = color_black;

uni_light = uni_light_all.get_val();
uni_light->cgl_LightModel.ambient = color_black;

uni_light.to_dev();
uni_light_ambient.to_dev();
uni_light_all.to_dev();
uni_light_reflected.to_dev();

puni_light_curr = &uni_light_all;

buf_uni_common->tryout =
  ivec4(opt_tryout1,opt_tryout2,opt_tryout3,opt_normal_sphere);
buf_uni_common->opt_hw05 =
  ivec4(opt_hw05_stel,opt_hw05_dot,opt_hw05_tex,0);
buf_uni_common->tryoutf = vec4(opt_tryoutf, opt_hw05_height_stel, 0, 0);
buf_uni_common->light_pos_g = light_location;
if ( do_raytrace )
  {
    raytrace.buf_uni_common->opt_tryout = buf_uni_common->tryout;
    raytrace.buf_uni_common->opt_tryoutf = buf_uni_common->tryoutf;
    raytrace.buf_uni_common->opt_shadows = opt_shadows;
    raytrace.buf_uni_common->opt_mirror = opt_mirror;
    raytrace.buf_light_set(uni_light_all);
  }
```

```
  const double time_now = time_wall_fp();
  const bool blink_visible = int64_t(time_now*3) & 1;
# define BLINK(txt,pad) ( blink_visible ? txt : pad )

  vh.fbprintf("%s\n",frame_timer.frame_rate_text_get());

#if 0
  vh.fbprintf
    ("Raytrace: %s ('R')   Compiled: %s\n",
     !vh.have_raytrace ? "NA" :
     opt_want_raytrace ? ( raytrace.rt_ready ? "ON " : "???" ) : "OFF",
#ifdef __OPTIMIZE__
     "WITH OPTIMIZATION"
#else
     BLINK("WITHOUT OPTIMIZATION","")
#endif
     );
#endif

  vh.fbprintf
    ("Time Step: %8d  World Time: %11.6f  %s\n",
     time_step_count, world_time,
     opt_pause ? BLINK("PAUSED, 'p' to unpause, SPC or S-SPC to step.","") :
     "Press 'p' to pause."
     );

  vh.fbprintf
    ("Eye location: [%5.1f, %5.1f, %5.1f]  "
     "Eye direction: [%+.2f, %+.2f, %+.2f]\n",
     eye_location.x, eye_location.y, eye_location.z,
     eye_direction.x, eye_direction.y, eye_direction.z);

  Ball& ball = *balls[0];

  if ( 0 )
  vh.fbprintf
    ("Head Ball Pos  [%5.1f,%5.1f,%5.1f] Vel [%+5.1f,%+5.1f,%+5.1f]\n",
     ball.position.x,ball.position.y,ball.position.z,
     ball.velocity.x,ball.velocity.y,ball.velocity.z );

  if ( 0 )
  vh.fbprintf
    ("%4zd Balls  %4zd Links  Physics: %s ('a' to change)  \n",
     balls.size(), links.size(), gpu_physics_method_str[opt_physics_method]);


  vh.fbprintf
    ("Hide: %c%c%c ('!@#')  Effect: %c%c ('or')  Sphere: %s  ('z')  "
     "Tryout 1:%s ('y')  2:%s ('Y')  3:%s ('x')  "
     "Normals: %s ('n')\n",
     opt_hide_stuff & OH_Sphere ? 'S' : '_',
     opt_hide_stuff & OH_Links ? 'L' : '_',
     opt_hide_stuff & OH_Platform ? 'P' : '_',
     opt_shadows ? 'S' : '_',
```

```
    opt_mirror ? 'M' : '_',
    opt_sphere_true ? "TRUE" : "TRI" ,
    opt_tryout1 ? BLINK("ON ","   ") : "OFF",
    opt_tryout2 ? BLINK("ON ","   ") : "OFF",
    opt_tryout3 ? BLINK("ON ","   ") : "OFF",
    opt_normal_sphere ? "SPHERE" : "TRI");

  vh.fbprintf("Shader: %s ('v')  "
              "HW05 DOT: %s ('d')  TEX: %s ('D')  STELL: %s ('c')\n",
              sh_names[opt_shader],
              opt_hw05_dot ? "ON " : BLINK("OFF","   "),
              opt_hw05_tex ? "ON " : BLINK("OFF","   "),
              opt_hw05_stel ? "ON " : BLINK("OFF","   ")
             );

  pVariable_Control_Elt* const cvar = variable_control.current;
  vh.fbprintf("VAR %s = %.5f  (TAB or SH-TAB to change, +/- to adjust)\n",
                      cvar->name,cvar->get_val());

  // Render Marker for Light Source
  //
  shapes.record_tetrahedron(cb,transform,light_location,0.15);

  //
  // Render ball reflection.  (Will be blended with dark tiles.)
  //

  const bool hide_platform = opt_hide_stuff & OH_Platform;

  if ( do_raytrace )
    {
      raytrace.tform_update(transform);
      render_objects(cb,RO_Normally);
      raytrace.buf_uni_common.to_dev();

      return;
    }

  buf_uni_common.to_dev();

  vk::ColorComponentFlags clr_all
    ( vk::ColorComponentFlagBits::eR | vk::ColorComponentFlagBits::eG |
      vk::ColorComponentFlagBits::eB | vk::ColorComponentFlagBits::eA );

  if ( !hide_platform )
    {
      // Save current state of fixed-functionality state.
      //
      ff_state.push();

      if ( opt_mirror )
        {
          /// Mirror Effect
          //
```

```
//  - Step 1:  Write location of mirrors into stencil buffer.
//
//  - Step 2a: Prepare a matrix that transforms to mirrored location.
//  - Step 2b: Set up blending to blend mirrored objects with mirrors.
//
//  - Step 3:  Render all objects except mirrors.

///  Step 1: Write location of mirrors into stencil buffer.
//
//   Write stencil value 2 at location of dark (mirrored) tiles.
//

vk::StencilOpState s_op_apply
  ( vk::StencilOp::eReplace, vk::StencilOp::eKeep,
    vk::StencilOp::eKeep, vk::CompareOp::eNever, 2, 2, 2 );
// sfail, spass, dfail, compare;  cmask, wmask, ref

ff_state.pipelineDepthStencilStateCreateInfo_get()
  .setStencilTestEnable(true)
  .setFront(s_op_apply).setBack(s_op_apply);

// Render mirrored tiles (but only to write stencil buffer).
//
pipe_platform_mir.record_draw(cb,bset_platform_mir);

cb.nextSubpass(vk::SubpassContents::eInline); vh.qs.subpass++;

// Prepare to write only stenciled locations.
//
vk::StencilOpState s_op_use
  ( vk::StencilOp::eKeep, vk::StencilOp::eKeep,
    vk::StencilOp::eKeep, vk::CompareOp::eEqual, 2, 2, 2 );
// sfail, spass, dfail, compare;  cmask, wmask, ref
//
// Stencil test passes if stencil value = 2.
// Don't change stencil buffer.

ff_state.pipelineDepthStencilStateCreateInfo_get()
  .setFront(s_op_use).setBack(s_op_use);

///  Step 2a: Prepare a matrix that transforms to mirrored location.
//   Use a transform that reflects objects to other side of platform.
//
const pMatrix prev = transform.eye_from_global;
const pMatrix_Scale reflect(1,-1,1);
transform.eye_from_global = prev * reflect;

// Reflected front face should still be treated as the front face.
//
ff_state.pipelineRasterizationStateCreateInfo_get()
  .setFrontFace(vk::FrontFace::eClockwise);

puni_light_curr = &uni_light_reflected;
```

```
        /// Step 3: Render all objects.

        render_objects(cb,RO_Mirrored);

        transform.eye_from_global = prev;

        ff_state.pipelineRasterizationStateCreateInfo_get()
          .setFrontFace(vk::FrontFace::eCounterClockwise);
      }

    // blend operation, blend factors, blend constants
    ff_state.pipelineColorBlendStateCreateInfo_get()
      .setBlendConstants( array<float,4>{.5,.5,.5,.5} );

    ff_state.pipelineColorBlendAttachmentState_get() =
      vk::PipelineColorBlendAttachmentState
      ( true,                    // blendEnable
        vk::BlendFactor::eConstantAlpha,  // srcColorBlendFactor
        vk::BlendFactor::eConstantAlpha,  // dstColorBlendFactor
        vk::BlendOp::eAdd,       // colorBlendOp
        vk::BlendFactor::eOne,   // srcAlphaBlendFactor
        vk::BlendFactor::eZero,  // dstAlphaBlendFactor
        vk::BlendOp::eAdd,       // alphaBlendOp
        clr_all      // colorWriteMask
        );

    puni_light_curr = &uni_light_all;;
    pipe_platform_mir.record_draw(cb,bset_platform_mir);

    ff_state.pop();
  }

if ( !opt_shadows )
  {
    // Render.
    //
    render_objects(cb,RO_Normally);
  }
else
  {
    //
    // First pass, render using only ambient light.
    //
    puni_light_curr = &uni_light_ambient;

    // Send balls, tiles, and platform to opengl.
    //
    render_objects(cb,RO_Normally);

    //
    // Second pass, find un-shadowed areas and add on light0.
    //
```

```
/// Write Shadow Volumes
//
vk::ClearAttachment clr_att
  ( vk::ImageAspectFlagBits::eStencil, 0,
    vk::ClearDepthStencilValue(0,0) );
vk::ClearRect clr_rect( vk::Rect2D({}, vh.s_extent), 0, 1 );
cb.clearAttachments( clr_att, clr_rect );

ff_state.push();

vk::StencilOpState s_op_apply_f
  ( vk::StencilOp::eKeep, vk::StencilOp::eIncrementAndWrap,
    vk::StencilOp::eKeep, vk::CompareOp::eAlways, -1, -1, 0 );
vk::StencilOpState s_op_apply_b
  ( vk::StencilOp::eKeep, vk::StencilOp::eDecrementAndWrap,
    vk::StencilOp::eKeep, vk::CompareOp::eAlways, -1, -1, 0 );
// sfail, spass, dfail, compare; c-mask, w-mask, ref
// ref is -1 (all 1's) because we want to operate on all bits in
// stencil value.

ff_state.pipelineDepthStencilStateCreateInfo_get()
  .setStencilTestEnable(true)
  .setDepthWriteEnable(false)
  .setFront(s_op_apply_f).setBack(s_op_apply_b);

if ( !opt_shadow_volumes )
  {
    ff_state.pipelineColorBlendAttachmentState_get()
      .setColorWriteMask( {} );
  }
else
  {
    ff_state.pipelineColorBlendAttachmentState_get() =
      vk::PipelineColorBlendAttachmentState
      ( true,                   // blendEnable
        vk::BlendFactor::eConstantAlpha,  // srcColorBlendFactor
        vk::BlendFactor::eConstantAlpha,  // dstColorBlendFactor
        vk::BlendOp::eAdd,       // colorBlendOp
        vk::BlendFactor::eOne,   // srcAlphaBlendFactor
        vk::BlendFactor::eZero,  // dstAlphaBlendFactor
        vk::BlendOp::eAdd,       // alphaBlendOp
        clr_all      // colorWriteMask
        );

    ff_state.pipelineColorBlendStateCreateInfo_get()
      .setBlendConstants( array<float,4>{.5,.5,.5,.5} );

    ff_state.pipelineDepthStencilStateCreateInfo_get()
      .setDepthWriteEnable(true);
  }

// Write stencil with shadow locations based on shadow volumes
// cast by light0 (light_location).  Shadowed locations will
// have a positive stencil value.
```

```
      //
      render_objects(cb,RO_Shadow_Volumes);

      cb.nextSubpass(vk::SubpassContents::eInline); vh.qs.subpass++;

      // Use stencil test to prevent writes to shadowed areas.
      //
      vk::StencilOpState s_op_apply_fb
        ( vk::StencilOp::eKeep, vk::StencilOp::eKeep,
          vk::StencilOp::eKeep, vk::CompareOp::eEqual, -1, -1, 0 );
      // sfail, spass, dfail, compare;  c-mask, w-mask, ref
      //
      // Stencil test passes if stencil value == 0.

      ff_state.pipelineDepthStencilStateCreateInfo_get()
        .setFront(s_op_apply_fb).setBack(s_op_apply_fb)
        .setDepthCompareOp(vk::CompareOp::eEqual);

      // Allow pixels to be re-written.
      //
      vk::ColorComponentFlags cmask_all
        ( vk::ColorComponentFlagBits::eR | vk::ColorComponentFlagBits::eG |
          vk::ColorComponentFlagBits::eB | vk::ColorComponentFlagBits::eA );

      ff_state.pipelineColorBlendAttachmentState_get() =
        vk::PipelineColorBlendAttachmentState
        ( true,                    // blendEnable
          vk::BlendFactor::eOne,  // srcColorBlendFactor
          vk::BlendFactor::eOne,  // dstColorBlendFactor
          vk::BlendOp::eAdd,       // colorBlendOp
          vk::BlendFactor::eOne,   // srcAlphaBlendFactor
          vk::BlendFactor::eOne,  // dstAlphaBlendFactor
          vk::BlendOp::eAdd,       // alphaBlendOp
          cmask_all      // colorWriteMask
          );

      puni_light_curr = &uni_light;

      // Render.
      //
      render_objects(cb,RO_Normally);

      ff_state.pop();
    }

  puni_light_curr = &uni_light_all;
}


void
World::cb_keyboard()
{
  const int key = vh.keyboard_key_get();
  if ( !key ) return;
```

```
  pVect adjustment(0,0,0);
  pVect user_rot_axis(0,0,0);
  const bool kb_mod_s = vh.keyboard_shift;
  const bool kb_mod_c = vh.keyboard_control;
  const float move_amt = kb_mod_s ? 2.0 : kb_mod_c ? 0.08 : 0.4;

  switch ( key ) {
  case FB_KEY_LEFT: adjustment.x = -move_amt; break;
  case FB_KEY_RIGHT: adjustment.x = move_amt; break;
  case FB_KEY_PAGE_UP: adjustment.y = move_amt; break;
  case FB_KEY_PAGE_DOWN: adjustment.y = -move_amt; break;
  case FB_KEY_DOWN: adjustment.z = move_amt; break;
  case FB_KEY_UP: adjustment.z = -move_amt; break;
  case FB_KEY_DELETE: user_rot_axis.y = 1; break;
  case FB_KEY_INSERT: user_rot_axis.y =  -1; break;
  case FB_KEY_HOME:
    if ( opt_move_item == MI_Light ) adjustment.x = 1; else user_rot_axis.x = 1;
    break;
  case FB_KEY_END: user_rot_axis.x = -1; break;
  case '1': ball_setup_1(kb_mod_c); break;
  case '2': ball_setup_2(); break;
  case '3': ball_setup_3(); break;
  case '4': ball_setup_4(); break;
  case '5': ball_setup_5(); break;
  case '6': ball_setup_6(); break;
  case '7': ball_setup_7(); break;
  case 'a':
    opt_physics_method++;
    if ( opt_physics_method == GP_ENUM_SIZE ) opt_physics_method = GP_cpu;
    break;
  case 'b': opt_move_item = MI_Ball; break;
  case 'B': opt_move_item = MI_Ball_V; break;

  case 'c': case 'C': opt_hw05_stel = !opt_hw05_stel;
    if ( opt_hw05_stel ) variable_control.make_current(hw05_vc_height_stel);
    break;
  case 'd': opt_hw05_dot = !opt_hw05_dot; break;
  case 'D': opt_hw05_tex = !opt_hw05_tex; break;

  case 'e': case 'E': opt_move_item = MI_Eye; break;
  case 'g': case 'G': opt_gravity = !opt_gravity; break;
  case 'h': case 'H': opt_head_lock = !opt_head_lock; break;
  case '!': opt_hide_stuff ^= OH_Sphere; break;
  case '@': opt_hide_stuff ^= OH_Links; break;
  case '#': opt_hide_stuff ^= OH_Platform; break;
  case 't': case 'T': opt_tail_lock = !opt_tail_lock; break;
  case 'l': case 'L': opt_move_item = MI_Light; break;
  case 'r': opt_mirror = !opt_mirror; break;
#if 0
  case 'R':
    opt_want_raytrace = !opt_want_raytrace;
    world_time_link_update = 0;
    break;
#endif
```

```
  case 'o': opt_shadows = !opt_shadows; break;
  case 'O':
    opt_shadow_volumes = !opt_shadow_volumes;
    if ( opt_shadow_volumes ) opt_shadows = true;
    break;
  case 'n': case 'N': opt_normal_sphere = !opt_normal_sphere; break;
  case 'p': case 'P': opt_pause = !opt_pause; break;
  case 's': case 'S': balls_stop(); break;
  case 'v': opt_shader++;
    if ( opt_shader >= sizeof(sh_names)/sizeof(sh_names[0]) ) opt_shader = 0;
    break;
  case 'V':
    if ( !opt_shader ) opt_shader = sizeof(sh_names)/sizeof(sh_names[0]);
    opt_shader--;
    break;
  case 'w': case 'W': balls_twirl(); break;
  case 'x': opt_tryout3 = !opt_tryout3; break;
  case 'y': opt_tryout1 = !opt_tryout1; break;
  case 'Y': opt_tryout2 = !opt_tryout2; break;
  case 'z': opt_sphere_true = !opt_sphere_true; break;
  case ' ':
    if ( kb_mod_s && kb_mod_c ) opt_single_time_step = 10;
    else if ( kb_mod_s ) opt_single_time_step = 100;
    else if ( kb_mod_c ) opt_single_time_step = 1; else opt_single_frame = true;
    opt_pause = true;
    break;
  case FB_KEY_TAB:
    if ( !kb_mod_s ) { variable_control.switch_var_right(); break; }
  case 96: variable_control.switch_var_left(); break;
  case '-':case '_': variable_control.adjust_lower(); break;
  case '+':case '=': variable_control.adjust_higher(); break;
  default: printf("Unknown key, %d\n",key); break;
  }

  gravity_accel.y = opt_gravity ? -opt_gravity_accel : 0;

  opt_mirror_depth = raytrace.recursion_max_set( opt_mirror_depth + 1 ) - 1;
  lock_update();
  sphere.bunch_invalidate();

  // Update eye_direction based on keyboard command.
  //
  if ( user_rot_axis.x || user_rot_axis.y )
    {
      pMatrix_Rotation rotall(eye_direction,pVect(0,0,-1));
      user_rot_axis *= invert(rotall);
      eye_direction *= pMatrix_Rotation(user_rot_axis, numbers::pi * 0.03);
      modelview_update();
    }

  // Update eye_location based on keyboard command.
  //
  if ( adjustment.x || adjustment.y || adjustment.z )
    {
```

```
      const double angle =
        fabs(eye_direction.y) > 0.99
        ? 0 : atan2(eye_direction.x,-eye_direction.z);
      pMatrix_Rotation rotall(pVect(0,1,0),-angle);
      adjustment *= rotall;

      switch ( opt_move_item ){
      case MI_Ball:
        adj_vector = adjustment;
        if ( adj_t_stop == 0 )
          adj_t_prev = adj_t_stop = world_time;
        adj_t_stop += adj_duration;
        break;
      case MI_Ball_V: balls_push(adjustment,0); break;
      case MI_Light:
        if ( key == FB_KEY_HOME )
          {
            pVect l_to_b(light_location,balls[0]->position);
            adjustment = 0.25 * l_to_b;
          }
        light_location += adjustment;
        break;
      case MI_Eye: eye_location += adjustment; break;
      default: break;
      }
      modelview_update();
    }

}




void
World::init(int argc, char **argv)
{
  chain_length = 14;

  opt_time_step_duration = 0.00003;
  if ( opt_all_ui_variables )
    variable_control.insert(opt_time_step_duration,"Time Step Duration");

  distance_relaxed = 15.0 / chain_length;
  opt_spring_constant = 500;
  if ( opt_all_ui_variables )
    variable_control.insert(opt_spring_constant,"Spring Constant");

  opt_spring_damp = 0.7;
  if ( opt_all_ui_variables )
    variable_control.insert(opt_spring_damp,"Spring Damping").set_max(1);

  opt_friction_dynamic = 0.2;
  if ( opt_all_ui_variables )
    variable_control.insert(opt_friction_dynamic,"Dynamic Friction");
```

```
opt_omega = 10;
if ( opt_all_ui_variables )
  variable_control.insert(opt_omega,"Initial Rotation Rate");

opt_gravity_accel = 9.8;
opt_gravity = true;
gravity_accel = pVect(0,-opt_gravity_accel,0);
if ( opt_all_ui_variables )
  variable_control.insert(opt_gravity_accel,"Gravity");

opt_air_resistance = 0.01;
if ( opt_all_ui_variables )
  variable_control.insert(opt_air_resistance,"Air Resistance");

world_time = 0;
time_step_count = 0;
last_frame_wall_time = time_wall_fp();
frame_timer.work_unit_set("Steps / s");

world_time_link_update = 0;

opt_mirror = true;
opt_mirror_depth = 2;
if ( opt_all_ui_variables )
  variable_control.insert(opt_mirror_depth,"Mirror Depth",1,0);
opt_shadows = true;
opt_shadow_volumes = false;
opt_shader = SO_HW5_Triangles;

/// Init CUDA

// Get information about GPU and its ability to run CUDA.
//
int device_count;
cudaGetDeviceCount(&device_count); // Get number of GPUs.
if ( device_count == 0 )
  {
    fprintf(stderr,"No GPU found, exiting.\n");
    exit(1);
  }

// Print information about the available GPUs.
//
gpu_info_print();

// Use course library routine to choose a good GPU for CUDA.
//
int dev = gpu_choose_index();
CE(cudaSetDevice(dev));
printf("Using GPU %d\n",dev);

gpu_info.get_gpu_info(dev);
cuda_setup(&gpu_info);
```

```
// Print information about time_step routine.
//
printf("\nCUDA Routine Resource Usage:\n");

int block_size_max = 4096;

for ( int i=0; i<gpu_info.num_kernels; i++ )
  {
    printf("For %s:\n", gpu_info.ki[i].name);
    printf("  %6zd shared, %zd const, %zd loc, %d regs; "
           "%d max threads per block.\n",
           gpu_info.ki[i].cfa.sharedSizeBytes,
           gpu_info.ki[i].cfa.constSizeBytes,
           gpu_info.ki[i].cfa.localSizeBytes,
           gpu_info.ki[i].cfa.numRegs,
           gpu_info.ki[i].cfa.maxThreadsPerBlock);
    set_min(block_size_max,gpu_info.ki[i].cfa.maxThreadsPerBlock);
  }

opt_physics_method = GP_cuda;
gpu_const_data_stale = true;
cpu_phys_data_stale = false;

CE( cudaEventCreate(&frame_start_ce) );
CE( cudaEventCreate(&frame_stop_ce) );

opt_block_size = 32;

if ( opt_all_ui_variables )
  variable_control.insert_power_of_2
    (opt_block_size, "Block Size", 32, block_size_max);

c.alloc_n_balls = 0;
c.alloc_n_links = 0;

init_graphics();

opt_sphere_slices = 35;
sphere.opt_texture = true;
sphere.init(opt_sphere_slices);
if ( opt_all_ui_variables )
  variable_control.insert(opt_sphere_slices,"Sphere Slices");

const char* const links_shader_code_path = "links-shdr-links.cc";

const auto bvec_vs = vk::ShaderStageFlagBits::eVertex;
const auto bvec_vgs =
  vk::ShaderStageFlagBits::eVertex | vk::ShaderStageFlagBits::eGeometry;

pipe_set_2
  .init( ff_state )
  .ds_follow( transform )
  .ds_follow( puni_light_curr )
```

```
    .ds_uniform_use( "BIND_UNI_COMMON", buf_uni_common )
    .ds_set_material( color_dim_gray, color_salmon )
    .ds_use( sampler, texid_hw )
    .ds_storage_follow( "BIND_LINKS_POS1", lis_pos1, bvec_vs )
    .ds_storage_follow( "BIND_LINKS_POS2", lis_pos2, bvec_vs )
    .ds_storage_follow( "BIND_LINKS_V1", lis_v1, bvec_vs )
    .ds_storage_follow( "BIND_LINKS_V2", lis_v2, bvec_vs )
    .ds_storage_follow( "BIND_LINKS_B1_YDIR", lis_b1_ydir, bvec_vs )
    .ds_storage_follow( "BIND_LINKS_B2_YDIR", lis_b2_ydir, bvec_vs )
    .ds_storage_follow( "BIND_LINKS_TEX_RAD", lis_tex_rad, bvec_vgs )
    .shader_inputs_info_set( )
    .shader_code_set
    (links_shader_code_path,
     "vs_main_2(); ",     // Name of vertex shader main routine.
     "gs_main_2();",      // Name of geometry shader main routine.
     "fs_main();"         // Name of fragment shader main routine.
     )
    .topology_set( vk::PrimitiveTopology::eLineStrip )
    .create();

  if ( vh.have_raytrace )
    comp_links
      .init( vh.qs )
      .ds_uniform_use( "BIND_UNI_COMMON", buf_uni_common )
      .ds_uniform_use( "BIND_TRANSFORM", raytrace.buf_uni_transform )
      .ds_storage_follow( "BIND_LINKS_POS1", lis_pos1 )
      .ds_storage_follow( "BIND_LINKS_POS2", lis_pos2 )
      .ds_storage_follow( "BIND_LINKS_V1", lis_v1 )
      .ds_storage_follow( "BIND_LINKS_V2", lis_v2 )
      .ds_storage_follow( "BIND_LINKS_B1_YDIR", lis_b1_ydir )
      .ds_storage_follow( "BIND_LINKS_B2_YDIR", lis_b2_ydir )
      .ds_storage_follow( "BIND_LINKS_TEX_RAD", lis_tex_rad )
      .ds_storage_follow( "BIND_LINKS_AS_IDX", buf_links_as_idx )
      .ds_storage_follow( "BIND_LINKS_TEX_COOR", buf_links_tex_coor )
      .ds_storage_follow( "BIND_LINKS_VERTEX_G", buf_links_vertex_g )
      .ds_storage_follow( "BIND_LINKS_NORMAL_G", buf_links_normal_g )
      .shader_code_set(links_shader_code_path, "comp_main(); ")
      .create();

  pipe_set_2_sv
    .init( ff_state ).ds_use( uni_light )
    .ds_follow( transform )
    .ds_uniform_use( "BIND_UNI_COMMON", buf_uni_common )
    .ds_storage_follow( "BIND_LINKS_POS1", lis_pos1 )
    .ds_storage_follow( "BIND_LINKS_POS2", lis_pos2 )
    .ds_storage_follow( "BIND_LINKS_V1", lis_v1 )
    .ds_storage_follow( "BIND_LINKS_V2", lis_v2 )
    .ds_storage_follow( "BIND_LINKS_B1_YDIR", lis_b1_ydir )
    .ds_storage_follow( "BIND_LINKS_B2_YDIR", lis_b2_ydir )
    .ds_storage_follow( "BIND_LINKS_TEX_RAD", lis_tex_rad )
    .shader_inputs_info_set( )
    .shader_code_set
    (links_shader_code_path,
     "vs_main_2_sv(); ",     // Name of vertex shader main routine.
```

```
      "gs_main_2_sv();",      // Name of geometry shader main routine.
      "fs_main_sv();"         // Name of fragment shader main routine.
      )
     .topology_set( vk::PrimitiveTopology::eLineStrip )
     .create();

  ball_setup_2();
  return;
  // For benchmarking.
  opt_pause = true;
  opt_hide_stuff = OH_Links | OH_Platform;
  opt_shadows = false;
  opt_mirror = false;
}

Ball*
World::make_marker(pCoor position, pColor color, float radius )
{
  Ball* const ball = new Ball;
  ball->position = position;
  ball->locked = true;
  ball->velocity = pVect(0,0,0);
  ball->radius = radius;
  ball->mass = -1;
  ball->contact = false;
  ball->color = color;
  balls_marker.push_back(ball);
  return ball;
}

void
World::lock_update()
{
  // This routine called when options like opt_head_lock might have
  // changed.

  // Update locked status.
  //
  if ( head_ball ) head_ball->locked = opt_head_lock;
  if ( tail_ball ) tail_ball->locked = opt_tail_lock;

  // Re-compute minimum mass needed for stability.
  //
  for ( Ball *ball: balls ) ball->spring_constant_sum = 0;
  const double dtis = pow( opt_time_step_duration, 2 );
  for ( Link *link: links )
    {
      Ball* const b1 = link->ball1;
      Ball* const b2 = link->ball2;
      b1->spring_constant_sum += opt_spring_constant;
      b2->spring_constant_sum += opt_spring_constant;
    }
  for ( Ball *ball: balls )
    {
```

```
      ball->mass_min = ball->spring_constant_sum * dtis;
      ball->constants_update();
    }
  gpu_const_data_stale = true;
}

void
World::balls_twirl()
{
  if ( !head_ball || !tail_ball ) return;

  pNorm axis(head_ball->position, tail_ball->position);
  const float omega = 0.1 * opt_omega;

  for ( Ball *ball: balls )
    {
      pVect b_to_top(ball->position,head_ball->position);
      const float dist_along_axis = dot(b_to_top,axis);
      const float lsq = b_to_top.mag_sq() - dist_along_axis * dist_along_axis;
      ball->omega += omega * axis;
      if ( lsq <= 1e-5 ) continue;
      const float dist_to_axis = sqrt(lsq);
      pNorm rot_dir = cross(b_to_top,axis);
      ball->velocity += omega * dist_to_axis * rot_dir;
    }
}

void
World::objects_erase()
{
  balls.erase();
  links.erase();
  surfaces.clear();
}

Links
World::link_new
(Ball *ball1, Ball *ball2, float stiffness, float spring_constant_p)
{
  Link_Args largs
    { .spring_constant = spring_constant_p, .link_stiffness = stiffness };
  return link_new(ball1,ball2,largs);
}

Links
World::link_new
(Ball *ball1, Ball *ball2, Link_Args largs )
{
  const float spring_constant_p = largs.spring_constant;
  const float stiffness = largs.link_stiffness;
  Links links_rv;
  assert( ball1->radius > 0 );
  assert( ball2->radius > 0 );
  const float spring_constant = spring_constant_p ?: opt_spring_constant;
```

```
  Link* const rlink = new Link(ball1,ball2);
  rlink->radius = largs.radius;
  links_rv += rlink;

  pNorm n12(ball1->position,ball2->position);
  const float rad_sum = ball1->radius + ball2->radius;
  pMatrix3x3 b1rot = ball1->omatrix;
  pMatrix3x3 b2rot = ball2->omatrix;
  pCoor ctr = ball1->position
    + ( ball1->radius + 0.5 * ( n12.magnitude - rad_sum ) ) * n12;

  pVect con_x = rlink->con_x;
  pVect con_y = rlink->con_y;

  const float lrad = stiffness * ball1->radius;

  const int n_springs = 4;
  const double delta_theta = 2 * numbers::pi / n_springs;

  for ( int i=0; i<n_springs; i++ )
    {
      const double theta = i * delta_theta;
      pVect convec = lrad * ( cos(theta) * con_x + sin(theta) * con_y );
      pCoor con = ctr + convec;
      pVect cb1 = mult_MTV( b1rot, con - ball1->position );
      pVect cb2 = mult_MTV( b2rot, con - ball2->position );
      links_rv +=
        new Link(rlink, cb1, cb2,
                 { .distance_relaxed=0, .spring_constant=spring_constant } );
    }
  return links_rv;
}

///
/// CUDA Code
///

float4 tof4(pCoor c)
{
  float4 f4;
  f4.x = c.x;
  f4.y = c.y;
  f4.z = c.z;
  f4.w = c.w;
  return f4;
}

float4 tof4(pVect c)
{
  float4 f4;
  f4.x = c.x;
  f4.y = c.y;
  f4.z = c.z;
```

```
  f4.w = 0;
  return f4;
}

pCoor topc(float4 c)
{
  pCoor f4;
  f4.x = c.x;
  f4.y = c.y;
  f4.z = c.z;
  f4.w = c.w;
  return f4;
}

void
World::data_cpu_to_gpu_constants()
{
  c.platform_xmin = platform_xmin;
  c.platform_xmax = platform_xmax;
  c.platform_zmax = platform_zmax;
  c.platform_zmin = platform_zmin;

  c.n_balls = balls.size();
  c.n_links = links.size();
  c.n_surfaces = surfaces.size();
  c.opt_tryout1 = opt_tryout1;
  c.opt_tryout2 = opt_tryout2;
  c.opt_tryoutf = opt_tryoutf;
  c.opt_spring_constant = opt_spring_constant;
  c.opt_spring_damp = opt_spring_damp;
  c.opt_air_resistance = opt_air_resistance;
  c.opt_friction_dynamic = opt_friction_dynamic;
  c.opt_head_lock = opt_head_lock;
  c.opt_tail_lock = opt_tail_lock;
  c.gravity_accel = gravity_accel;

  data_cpu_to_gpu_common(&c);
}



void
World::data_cpu_to_gpu_dynamic()
{
  const int n_balls = balls.size();
  const int n_links = links.size();
  const int n_surfaces = surfaces.size();
  const bool need_alloc =
    n_balls > c.alloc_n_balls || n_links > c.alloc_n_links
    || n_surfaces > c.alloc_n_surfaces;
  const bool need_init = c.alloc_n_balls == 0;

#define ALLOC_MOVE_ZERO(n_balls,balls,memb,do_move,do_zero)                  \
  {                                                                          \
    const int size_elt_bytes = sizeof(c.balls.memb[0]);                      \
```

```
    const int size_bytes = n_balls * size_elt_bytes;                          \
    if ( need_alloc )                                                         \
      {                                                                       \
        if ( !need_init )                                                     \
          {                                                                   \
            CE( cudaFree( c.balls.memb ) );                                   \
            free(c.h_##balls.memb);                                           \
          }                                                                   \
        CE( cudaMalloc( &c.balls.memb, size_bytes ) );                        \
        c.h_##balls.memb = (decltype(c.balls.memb)) malloc( size_bytes );     \
      }                                                                       \
    if ( do_move )                                                            \
      {                                                                       \
        for ( int i=0; i<n_balls; i++ )                                       \
          c.h_##balls.memb[i] = balls[i]->memb;                               \
        CE( cudaMemcpy                                                        \
            ( c.balls.memb, c.h_##balls.memb,                                 \
              size_bytes, cudaMemcpyDefault ) );                              \
      }                                                                       \
    else if ( do_zero )                                                       \
      {                                                                       \
        CE( cudaMemset( c.balls.memb, 0, size_bytes ) );                      \
      }                                                                       \
  }

#define MOVEc(num,ptr,memb) \
  ALLOC_MOVE_ZERO(num,ptr,memb,gpu_const_data_stale,0)
#define MOVE(num,ptr,memb) ALLOC_MOVE_ZERO(num,ptr,memb,1,0)
#define ALLOC(num,ptr,memb) ALLOC_MOVE_ZERO(num,ptr,memb,0,0)
#define ZERO(num,ptr,memb) ALLOC_MOVE_ZERO(num,ptr,memb,0,1)
  MOVEc(n_balls,balls,position);
  MOVEc(n_balls,balls,velocity);
  ZERO(n_balls,balls,force);

  MOVEc(n_balls,balls,orientation);
  MOVEc(n_balls,balls,omatrix);
  MOVEc(n_balls,balls,omega);
  ZERO(n_balls,balls,torque);
  MOVEc(n_balls,balls,m_r_fdt);
  MOVEc(n_balls,balls,locked);

  MOVEc(n_links,links,ball_idx);
  MOVEc(n_links,links,cb1);
  MOVEc(n_links,links,cb2);
  MOVEc(n_links,links,dr_sc);
  MOVEc(n_links,links,is_simulatable);
  MOVEc(n_links,links,is_surface_connection);

  MOVEc(n_surfaces,surfaces,b_idx);

#undef MOVE
#undef MOVEc
#undef ALLOC
#undef ZERO
```

```
#undef ALLOC_MOVE
  if ( need_alloc )
    {
      c.alloc_n_balls = n_balls;
      c.alloc_n_links = n_links;
      c.alloc_n_surfaces = n_surfaces;
      data_cpu_to_gpu_common(&c);
    }
  gpu_const_data_stale = false;
}

void
World::data_gpu_to_cpu_dynamic()
{
  const int n_balls = balls.size();

#define MOVE(n_balls,balls,memb)                                          \
  {                                                                       \
    const int size_elt_fr_bytes = sizeof(c.balls.memb[0]);                \
    const int size_bytes = n_balls * size_elt_fr_bytes;                   \
    CE( cudaMemcpy                                                        \
        ( c.h_##balls.memb, c.balls.memb, size_bytes,                     \
          cudaMemcpyDeviceToHost ) );                                     \
    for ( int i=0; i<n_balls; i++ )                                       \
      balls[i]->memb = c.h_##balls.memb[i];                               \
  }

  MOVE(n_balls,balls,position);
  MOVE(n_balls,balls,velocity);
  MOVE(n_balls,balls,orientation);
  MOVE(n_balls,balls,omatrix);
  MOVE(n_balls,balls,omega);

#undef MOVE

}


///
/// Physical Simulation Code
///

 /// Initialize Simulation
//

void
World::ball_setup_1(bool big)
{
  // Arrange balls to form sort of tree.

  last_setup = 1;

  double ball_rad = 1;  // Warning: Re-assigned, changes behavior of nb.
```

```
pCoor first_pos(17.2,ball_rad,-20.2);

pVect dir_up(0.1,3,0.1);
pVect dir_z(3*distance_relaxed,0,0);
pVect dir_x(0,0,3*distance_relaxed);

// Remove objects from the simulated objects lists, balls and links.
// The delete operator is used on objects in the lists.
//
objects_erase();

auto nb =
  [&] (pCoor pos) {
    Ball* const ball = new Ball;
    ball->position = pos;
    ball->locked = false;
    ball->velocity = pVect(0,0,0);
    ball->radius = ball_rad;
    ball->contact = false;
    balls += ball;
    return ball;
    };

tail_ball = nb(first_pos);
tail_ball->specularity = 0.5;

pCoor last_pos = tail_ball->position;

const int n_tiers = big ? 32 : 8;
const double shrink_final = 0.17;
const double shrink = pow(shrink_final,1.0/n_tiers);

Balls tail_balls;
for ( int i=1; i<n_tiers; i++ )
  {
    ball_rad *= shrink;
    Ball* const ball = nb(last_pos + (i==1?1.0:ball_rad) * dir_up);
    ball->specularity = 0.5;
    tail_balls += ball;
    links += link_new(balls[balls-2],ball,0.8);
    last_pos = ball->position;
  }

head_ball = balls[balls-1];

pNorm dir_zn(dir_z);
pNorm dir_xn(dir_x);
for ( Ball *ball: tail_balls )
  {
    if ( ball == head_ball ) break;
    pNorm to_top(ball->position,head_ball->position);
    const float r = to_top.magnitude;
    int num_hairs = max(4,int(7*r));
    ball_rad = ball->radius / 4;
```

```
      const float delta_theta = 2 * numbers::pi / num_hairs;
      for ( int i=0; i<num_hairs; i++ )
        {
          const float theta = i * delta_theta;
          pCoor pos = ball->position +
            r * cos(theta) * dir_xn + r * sin(theta) * dir_zn;
          Ball* const ballr = nb(pos);
          links += link_new(ballr,ball);
        }
    }

  opt_head_lock = false;
  opt_tail_lock = true;
  setup_at_end();
}

void
World::ball_setup_2()
{
  // Arrange balls to form a suspended thing.

  last_setup = 2;

  pCoor first_pos(17.2,14.8f,-20.2);

  pVect dir_dn(0.5,first_pos.y/8,0.5);
  pVect dir_z(3*distance_relaxed,0,0);
  pVect dir_x(0,0,3*distance_relaxed);

  // Remove objects from the simulated objects lists, balls and links.
  // The delete operator is used on objects in the lists.
  //
  objects_erase();

  float ball_rad = 0.3;  // Warning: Re-assigned, changes behavior of nb.

  auto nb =
    [&] (pCoor pos) {
      Ball* const ball = new Ball;
      ball->position = pos;
      ball->locked = false;
      ball->velocity = pVect(0,0,0);
      ball->radius = ball_rad;
      ball->contact = false;
      ball->specularity = 0.5;
      balls += ball;
      return ball;
      };

  head_ball = nb(first_pos);
  for ( int j: {-1,1} )
    for ( int i: {-1,0,1} )
      {
        Ball* const ball = nb(first_pos - dir_dn + i * dir_x + j * dir_z );
```

```
      if ( i != 0 ) links += link_new(head_ball,ball);
      if ( i != -1 ) links += link_new(balls[balls-2],ball);
      if ( j == 1 && i != 0 )  links += link_new(balls[balls-4],ball);
    }

  Ball* const tail_start = nb(first_pos - dir_dn);
  links += link_new(balls[balls-6],tail_start);
  links += link_new(balls[balls-3],tail_start);
  pCoor last_pos = tail_start->position;
  Balls tail_balls;
  for ( int i=1; i<5; i++ )
    {
      Ball* const ball = nb(last_pos - 0.5 * dir_dn);
      tail_balls += ball;
      links += link_new(balls[balls-2],ball);
      last_pos = ball->position;
    }

  tail_ball = balls[balls-1];

  pNorm dir_zn(dir_z);
  pNorm dir_xn(dir_x);
  int num_hairs = 20;
  ball_rad = 0.1;
  for ( Ball *ball: tail_balls )
    {
      const float delta_theta = 2 * numbers::pi / num_hairs;
      for ( int i=0; i<num_hairs; i++ )
        {
          const float theta = i * delta_theta;
          const float r = ball->radius * 10;
          pCoor pos = ball->position +
            r * cos(theta) * dir_xn + r * sin(theta) * dir_zn;
          Ball* const ballr = nb(pos);
          links += link_new(ballr,ball);
        }
    }

  opt_head_lock = true;    // Head ball will be frozen in space.
  opt_tail_lock = false;   // Tail ball can move freely.
  setup_at_end();
}

void
World::ball_setup_3()
{
  // Arrange links to form what I would eventually learn to be a
  // corona virus.

  const bool sarcov = true;

  last_setup = 3;
  objects_erase();
```

```
  pCoor first_pos(17.2,14.8f,-10.2);

  float ball_radius = 1;

  auto nb =
    [&] (pCoor pos) {
      Ball* const ball = new Ball;
      ball->position = pos;
      ball->locked = false;
      ball->velocity = pVect(0,0,0);
      ball->radius = ball_radius;
      ball->contact = false;
      balls += ball;
      return ball;
      };

  head_ball = nb(first_pos);
  head_ball->specularity = 0.5;
  ball_radius = ball_radius * ( sarcov ? 0.08 : .1 );
  const int n_spikes = sarcov ? 100 : 500;

  for ( int i=0; i<n_spikes; i++ )
    {
      pNorm dir(0.5-drand48(),0.5-drand48(),0.5-drand48());
      const float len = head_ball->radius * ( sarcov ? 1.5 : 3.0 );
      Ball* const ball = nb(first_pos + len * dir);
      links += link_new(ball,head_ball,2);
    }

  tail_ball = balls[1];

  opt_head_lock = false;    // Head ball will be frozen in space.
  opt_tail_lock = false;   // Tail ball can move freely.

  setup_at_end();
}

pVect vec_rand() {return pVect(drand48(),drand48(),drand48());}

void
World::ball_setup_4()
{
  // Render a wheel (disk of balls with one in the center).

  last_setup = 4;
  objects_erase();

  pCoor center_pos = pCoor(17.2,14.8f,-10.2) + 2*vec_rand();

  const int n_balls = 4 * ( 4 + random()%4 );
  float ring_outer_radius = 5 + 2 * drand48();
  float ball_radius = 0.2 * 2 * numbers::pi * ring_outer_radius / n_balls;

  pNorm c_to_eye(center_pos,eye_location);
```

```
  pNorm az = c_to_eye + 0.5 * vec_rand() + pVect(0,0.5,0);

  pNorm ax = fabs(az.x) < fabs(az.y)
    ? pVect(0,az.z,-az.y) : pVect(az.z,0,-az.x);
  pVect ay(az,ax);

  // Default settings for balls.
  //
  Ball ball_default;
  ball_default.specularity = 0.5;
  ball_default.velocity = pVect(0,0,0);
  ball_default.radius = ball_radius;
  ball_default.color = color_light_slate_gray;

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

  const float d_theta = 2 * numbers::pi / n_balls;

  // Place balls in a circle.
  //
  for ( int i=0; i<n_balls; i++ )
    {
      const float theta = d_theta * i;
      pCoor pos = center_pos + vx * cosf(theta) + vy * sinf(theta);

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

      // Set color of first ball to blue, others to white.
      ball->color = i == 0 ? color_blue : color_white;
    }

  const float link_stiffness = 10;

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

  // Place a ball in the center and connect it to all the other balls.
  if ( true )
    {
      Ball* const center_ball =
        balls.emplace_back( new Ball(center_pos,ball_default) );

      for ( int i=0; i<n_balls; i++ )
        links += link_new( center_ball, balls[i], 2 );
    }

  pNorm spin_axis = az;
```

```
  pCoor c_of_mass = center_pos;

  for ( Ball* ball: balls )
    {
      pVect cm_to_ball( c_of_mass, ball->position );

      ball->velocity = opt_omega * cross( spin_axis, cm_to_ball );
      ball->omega = opt_omega * spin_axis;
    }

  tail_ball = balls.back();
  head_ball = balls.front();

  opt_head_lock = false; // If true, head ball locked in place.
  opt_tail_lock = false;

  setup_at_end();
}

void
World::ball_setup_5()
{
  // Construct a top. (Top, as in a child's toy.)

  last_setup = 5;
  objects_erase();

  pCoor center_pos(17.2,14.8f,-10.2);

  const int n_balls = 20;
  const float crad = 5;

  const float ball_radius = 0.3 * 2 * numbers::pi * crad / n_balls;

  // Default settings for balls.
  //
  Ball ball_default;
  ball_default.specularity = 0.5;
  ball_default.velocity = pVect(0,0,0);
  ball_default.radius = ball_radius;
  ball_default.color = pColor(0xffbbff); // Plum 1. That's the color's name.

  const float d_theta = 2 * numbers::pi / n_balls;

  pVect vz = pVect(1,6,1) - 2*vec_rand();
  pNorm ax( vz.y, -vz.x, 0 );
  pNorm ay = cross(vz,ax);
  pVect vx = crad * ax;
  pVect vy = crad * ay;

  pNorm spin_axis = vz;
  float omega = opt_omega;
  opt_spring_damp = .4;
```

```
  // Construct Top's Main Ring
  //
  for ( int i=0; i<n_balls; i++ )
    {
      const float theta = d_theta * i;
      pCoor pos = center_pos + vx * cosf(theta) + vy * sinf(theta);
      Ball* ball = balls.emplace_back( new Ball(pos,ball_default) );

      // Find the coordinate on the spin axis closest to the new ball.
      //
      pCoor axis_cp =
        center_pos + spin_axis * dot( spin_axis, pVect(center_pos,pos) );
      //
      // Use this coordinate to compute a vector giving the ball's
      // motion when the top is spinning around axis spin_axis.
      //
      pVect cp_to_p(axis_cp,pos);
      ball->velocity = omega * cross(spin_axis,cp_to_p);
    }

  // Make one of the balls on the top's main ring a different color.
  balls[0]->color = color_blue;

  // Connect the balls on the main ring with links.
  //
  for ( int i=0; i<n_balls; i++ )
    links += link_new( balls[i], balls[(i+1)%n_balls], 2 );

  // Complete the top with balls above and below the ring, and one in
  // the center.
  //
  pNorm az(vz);
  pCoor top_pos = center_pos + az * crad * 1.5;
  pCoor bot_pos = center_pos - az * crad * 1.5;

  head_ball = balls.emplace_back( new Ball(top_pos,ball_default) );
  tail_ball = balls.emplace_back( new Ball(bot_pos,ball_default) );
  Ball* ctr_ball = balls.emplace_back( new Ball(center_pos,ball_default) );

  for ( int i=0; i<n_balls; i += 2 )
    {
      links += link_new( head_ball, balls[i], 2 );
      links += link_new( tail_ball, balls[i], 2 );
    }

  for ( int i=0; i<n_balls; i++ )
    links += link_new( ctr_ball, balls[i], 2 );

  for ( auto b: balls ) b->omega = omega * spin_axis;

  opt_head_lock = false;    // Head ball will be frozen in space.
  opt_tail_lock = false;   // Tail ball can move freely.

  setup_at_end();
```

```
}

void
World::ball_setup_6()
{
  // Render what will become a parachute, probably.

  last_setup = 6;
  objects_erase();

  pCoor center_pos = pCoor(17.2,14.8f,-10.2) + 2*vec_rand();

  const int n_balls = 4 * ( 4 + random()%4 );
  float ring_outer_radius = 5 + 2 * drand48();
  float ring_inner_radius = ring_outer_radius * ( 0.1 + drand48() * 0.4 );
  float ball_radius = 0.2 * 2 * numbers::pi * ring_outer_radius / n_balls;

  pNorm c_to_eye(center_pos,eye_location);
  pNorm az = c_to_eye + 0.5 * vec_rand() + pVect(0,0.5,0);

  pNorm ax = fabs(az.x) < fabs(az.y)
    ? pVect(0,az.z,-az.y) : pVect(az.z,0,-az.x);
  pVect ay(az,ax);

  // Default settings for balls.
  //
  Ball ballo_default;
  ballo_default.specularity = 0.5;
  ballo_default.velocity = pVect(0,0,0);
  ballo_default.radius = ball_radius;
  ballo_default.color = color_light_slate_gray;
  Ball balli_default = ballo_default;
  balli_default.radius = ballo_default.radius * 0.6;

  const float d_theta = 2 * numbers::pi / n_balls;

  const float link_stiffness = 10;

  vector<Ball*> balls_outer, balls_inner;

  // Place balls in a circle.
  //
  for ( int i=0; i<n_balls; i++ )
    {
      const float theta = d_theta * i;
      pVect pos_n = cosf(theta) * ax + sinf(theta) * ay;
      pCoor pos_o = center_pos + ring_outer_radius * pos_n;
      pCoor pos_i = center_pos + ring_inner_radius * pos_n;

      // Construct new ball and add it to the ball list.
      //
      Ball* ball_o = balls_outer.emplace_back( new Ball(pos_o,ballo_default) );
      Ball* ball_i = balls_inner.emplace_back( new Ball(pos_i,balli_default) );
      Link *link =
```

```
        new Link( ball_o, ball_i,
                  { .spring_constant = opt_spring_constant/200 } );
      links.push_back(link);

      // Set color of first ball to blue, others to white.
      ball_i->color = ball_o->color =
        i == 0 ? color_blue :
        theta <= numbers::pi/2 ? color_lsu_spirit_gold :
        color_white;
    }

  // Connect each ball to its neighbor with a link.
  //
  for ( auto bc: { &balls_outer, &balls_inner } )
    for ( int i=0; i<n_balls; i++ )
      {
        balls.push_back( bc[0][i] );
        links += link_new( bc[0][i], bc[0][(i+1)%n_balls], link_stiffness );
      }

  for ( int i=0; i<n_balls; i++ )
    {
      const int j = ( i + 1 ) % n_balls;
      Ball* b00 = balls_inner[i];
      Ball* b01 = balls_outer[i];
      Ball* b10 = balls_inner[j];
      Ball* b11 = balls_outer[j];
      surfaces.emplace_back(b00,b01,b11,color_light_gray);
      surfaces.emplace_back(b10,b00,b11,color_light_gray);
    }

  pNorm spin_axis = az;
  pCoor c_of_mass = center_pos;
  float omega_init = opt_omega * 0.1;

  if ( 0 )
  for ( Ball* ball: balls )
    {
      pVect cm_to_ball( c_of_mass, ball->position );

      ball->velocity = omega_init * cross( spin_axis, cm_to_ball );
      ball->omega = omega_init * spin_axis;
    }

  tail_ball = balls.back();
  head_ball = balls.front();

  opt_head_lock = false; // If true, head ball locked in place.
  opt_tail_lock = false;

  setup_at_end();
}

void
```

```
World::ball_setup_7()
{
  // Render what will become a parachute, probably.

  last_setup = 7;
  objects_erase();
  setup_at_end();
}

void
World::setup_at_end()
{
  balls.take(balls_marker);
  lock_update();
  time_step_count = 0;
  world_time = 0;
  world_time_link_update = -1;
  for ( uint i=0; i<balls.size(); i++ ) balls[i]->idx = i;
  for ( Link *l : links )
    { l->ball1_idx = l->ball1->idx; l->ball2_idx = l->ball2->idx; }
  for ( auto& s: surfaces )
    { s.b0_idx = s.b0->idx;  s.b1_idx = s.b1->idx;  s.b2_idx = s.b2->idx; }
}


 /// Advance Simulation State by delta_t Seconds
//
void
World::time_step_cpu(double delta_t)
{
  const int n_balls = balls;
  const int n_links = links;

  // Smoothly move ball in response to user input.
  //
  if ( adj_t_stop )
    {
      const double dt = min(world_time,adj_t_stop) - adj_t_prev;
      pVect adj = dt/adj_duration * adj_vector;
      balls_translate(adj,0);
      adj_t_prev = world_time;
      if ( world_time >= adj_t_stop ) adj_t_stop = 0;
    }

#pragma omp parallel for
  for ( int bi=0; bi<n_balls; bi++ )
    {
      Ball* const ball = balls[bi];
      assert( ball->fdt_to_do );
      ball->force = ball->mass * gravity_accel;
      ball->torque = pVect(0,0,0);
    }

  const int n_surfaces = surfaces.size();
```

```
  for ( int i=0; i<n_surfaces; i++ )
    {
      Surface& s = surfaces[i];
      Ball& b0 = *s.b0;
      Ball& b1 = *s.b1;
      Ball& b2 = *s.b2;
      pNorm snorm = cross(b0.position,b1.position,b2.position);
      pVect vel = (1.0/3) * ( b0.velocity + b1.velocity + b2.velocity );
      float speed = dot(snorm,vel);
      float fmag =
        -speed * fabs(speed) * snorm.magnitude * 0.5f * opt_air_resistance;
      pVect f = fmag * snorm;
      b0.force += f;
      b1.force += f;
      b2.force += f;
    }

#pragma omp parallel for
  for ( int i=0; i<n_links; i++ )
    {
      Link* const link = links[i];
      if ( !link->is_simulatable ) continue;

      //
      // Spring Force from Neighbor Balls
      //

      Ball* const ball1 = link->ball1;
      Ball* const ball2 = link->ball2;

      /// Notes:
      //
      // link->cb1
      //   Coordinate where link touches ball1, in ball1's local space.
      //
      // ball1->omatrix
      //   Ball's orientation transformation ..
      //   .. which transforms vectors in ball1's local space to object space.
      //

      // Find position and velocity of the point where the link touches
      // the surface of ball 1 ...
      //
      pVect dir1 = ball1->omatrix * link->cb1;
      pCoor pos1 = ball1->position + dir1;
      pVect vel1 = ball1->velocity + ball1->point_rot_vel(dir1);

      // ... and ball 2.
      //
      pVect dir2 = ball2->omatrix * link->cb2;
      pCoor pos2 = ball2->position + dir2;
      pVect vel2 = ball2->velocity + ball2->point_rot_vel(dir2);

      // Construct a normalized (Unit) Vector from ball to neighbor
```

```
    // based on link connection points and ball centers.
    //
    pNorm link_dir(pos1,pos2);
    pNorm c_to_c(ball1->position,ball2->position);

    const float link_length = link_dir.magnitude;

    // Compute the speed of ball's end of link towards neighbor's end of link.
    //
    pVect delta_v = vel2 - vel1;
    float delta_s = dot( delta_v, link_dir );

    // Compute by how much the spring is stretched (positive value)
    // or compressed (negative value).
    //
    const float spring_stretch = link_length - link->distance_relaxed;

    // Determine whether spring is gaining energy (whether its length
    // is getting further from its relaxed length).
    //
    const bool gaining_e = ( delta_s > 0.0 ) == ( spring_stretch > 0 );

    const float blend_length_factor = 0.1; // Just a guess.
    const float blend_s = blend_length_factor * c_to_c.magnitude;
    const float blend = min( 1.0f, fabs(delta_s) / blend_s );

    // Use a smaller spring constant when spring is loosing energy,
    // a quick and dirty way of simulating energy loss due to spring
    // friction.
    //
    const float sc_base = link->spring_constant ?: opt_spring_constant;
    const float spring_constant =
      gaining_e ? sc_base :
      sc_base * ( 1.0f - blend + blend * opt_spring_damp );

    const float force_mag = spring_constant * spring_stretch;
    pVect spring_force_12 = force_mag * link_dir;

    // Apply forces affecting linear momentum.
    //
    link->spring_force_12 = spring_force_12;

    if ( ! link->is_surface_connection ) continue;

    // Apply torque.  Why doesn't the ball radius matter?
    //
    link->torque1 = cross(pNorm(dir1), spring_force_12);
    link->torque2 = cross(pNorm(dir2), spring_force_12);
  }

// Note: Because two links can reference the same ball this should
// not be done in parallel.
for ( Link *link: links )
  {
```

```
      if ( !link->is_simulatable ) continue;
      Ball* const ball1 = link->ball1;
      Ball* const ball2 = link->ball2;
      ball1->force += link->spring_force_12;
      ball2->force -= link->spring_force_12;
      ball1->torque += link->torque1;
      ball2->torque -= link->torque2;
    }

  ///
  /// Update Position of Each Ball
  ///

#pragma omp parallel for
  for ( int bi=0; bi<n_balls; bi++ )
    {
      Ball* const ball = balls[bi];
      if ( ball->locked )
        {
          ball->velocity = pVect(0,0,0);
          ball->omega = pVect(0,0,0);
          continue;
        }

      // Update Velocity
      //
      // This code assumes that force on ball is constant over time
      // step. This is clearly wrong when balls are moving with
      // respect to each other because the springs are changing
      // length. This inaccuracy will make the simulation unstable
      // when spring constant is large for the time step.
      //
      const float mass = max( ball->mass, ball->mass_min );

      pVect delta_v = ( ball->force / mass ) * delta_t;
      const float r = ball->radius;

      if ( platform_collision_possible(ball->position)
           && fabs(ball->position.y) < r )
        {
          const bool above = ball->position.y >= 0;
          const float spring_constant_plat =
            ball->velocity.y < 0 == above ? 100000 : 50000;
          const float fric_coefficient = opt_friction_dynamic;
          const float force_away = ( above ? 1 : -1 )
            * ( r - fabs(ball->position.y) ) * spring_constant_plat;
          const float delta_v_away = force_away / mass * delta_t;
          const float fric_force_mag = fric_coefficient * force_away;
          pNorm surface_v(ball->velocity.x,0,ball->velocity.z);
          const float delta_v_surf = fric_force_mag / mass * delta_t;

          if ( delta_v_surf > surface_v.magnitude )
            {
              // Ignoring other forces?
```

```
              delta_v =
                pVect(-ball->velocity.x,delta_v.y,-ball->velocity.z);
            }
          else
            {
              delta_v -= delta_v_surf * surface_v;
            }
          delta_v.y += delta_v_away;
        }

      ball->velocity += delta_v;

      // Air Resistance
      //
      const pNorm ball_dir(ball->velocity);
      const float delta_s_air_raw =
        opt_air_resistance * ball_dir.mag_sq * delta_t;
      const float delta_s_air = min( ball_dir.magnitude, delta_s_air_raw );
      ball->velocity -= delta_s_air * ball_dir;

      // Update Position
      //
      // Assume that velocity is constant.
      //
      ball->position += ball->velocity * delta_t;

      ball->omega += delta_t * ball->fdt_to_do * ball->torque;

      pNorm axis(ball->omega);

      // Update Orientation
      //
      // If ball isn't spinning fast skip expensive rotation.
      //
      if ( axis.mag_sq < 0.000001 ) continue;

      // Update ball orientation.
      //
      ball->orientation_set
        ( pQuat(axis,delta_t * axis.magnitude) * ball->orientation );
    }
}

bool
World::platform_collision_possible(pCoor pos)
{
  // Assuming no motion in x or z axes.
  //
  return pos.x >= platform_xmin && pos.x <= platform_xmax
    && pos.z >= platform_zmin && pos.z <= platform_zmax;
}

 /// External Modifications to State
//
```

```
//   These allow the user to play with state while simulation
//   running.

// Move the ball.
//
void Ball::translate(pVect amt) {position += amt;}

// Add velocity to the ball.
//
void Ball::push(pVect amt) {velocity += amt;}

// Set the velocity to zero.
//
void Ball::stop() {velocity = pVect(0,0,0); }

// Set the velocity and rotation (not yet supported) to zero.
//
void Ball::freeze() {velocity = pVect(0,0,0); }



void World::balls_translate(pVect amt,int b){head_ball->translate(amt);}
void World::balls_push(pVect amt,int b){head_ball->push(amt);}
void World::balls_translate(pVect amt)
{ for ( Ball *ball: balls ) ball->translate(amt); }
void World::balls_push(pVect amt)
{ for ( Ball *ball: balls ) ball->push(amt); }
void World::balls_stop()
{ for ( Ball *ball: balls ) ball->stop(); }
void World::balls_freeze(){balls_stop();}


void
World::render_link_start()
{
  if ( world_time_link_update == world_time ) return;

  // Empty the containers in preparation for a new pass.
  for ( auto b: lis_bufs ) b->clear();
}

void
World::render_link_gather(Link *link)
{
  // Place data needed to render link (the argument to this function)
  // in arrays. Just before rendering those arrays will be placed in
  // buffer objects.


  // If we've already collected links, just return.  This happens
  // when doing multiple passes to render shadows and reflections.
  //
  if ( world_time_link_update == world_time ) return;
```

```
    Ball *const ball1 = link->ball1;
    Ball *const ball2 = link->ball2;

    // Direction of link relative to center off ball.
    //
    pVect dir1 = ball1->omatrix * link->cb1;

    // Place on surface of ball where link connects.
    //
    pCoor pos1 = ball1->position + dir1;

    // Direction and connection location for ball 2.
    //
    pVect dir2 = ball2->omatrix * link->cb2;
    pCoor pos2 = ball2->position + dir2;

    pVect p1p2(pos1,pos2);
    pNorm p1p2n(p1p2);

    // Radius of link.
    //
    const float rad = link->radius ?: ball1->radius * 0.3;

    // Compute scale factors for texture.
    const float tex_aspect_ratio = 8.5 / 11;  // Yes, inches. Can't remember mm.
    const float page_width_o = 2.5;  // Width of texture in object-space coords.
    const float tex_t_scale = link->distance_relaxed / page_width_o;
    const float tex_angle_scale = rad * tex_aspect_ratio / page_width_o;

    pNorm dirn1(dir1);
    pNorm dirn2(dir2);

    // Vectors used to describe the cubic Hermite curve.
    //
    pVect4 v1 = p1p2n.magnitude * dirn1;
    pVect4 v2 = p1p2n.magnitude * dirn2;

    // Convert link's local x and y axes to global coordinates.
    //
    pVect4 b1_ydir = ball1->omatrix * link->b1_dir;
    pVect4 b2_ydir = ball2->omatrix * link->b2_dir;


    // Place data needed to render links in lists. Note that
    // the += operator appends the right-hand side to the container
    // on the left-hand side.
    //
    link->idx = lis_pos1.size();
    lis_pos1 << pos1;
    lis_pos2 << pos2;
    lis_v1 << v1;
    lis_v2 << v2;
    lis_b1_ydir << b1_ydir;
```

```
  lis_b2_ydir << b2_ydir;
  lis_tex_rad << vec4( tex_t_scale, tex_angle_scale, rad, 0);
}



void
World::render_link_render(vk::CommandBuffer& cb, bool shadow_volumes)
{
  // Perform a rendering pass to render all the links collected
  // by the routine render_link_2.


  // Check whether this is the first time we are rendering this data.
  //
  const bool first_render =
    world_time_link_update != world_time
    || opt_segments != buf_uni_common->links_n_segs
    || opt_sides != buf_uni_common->sides;
  world_time_link_update = world_time;
  const size_t n_instances = lis_pos1.size();
  const bool do_rt = vh.raytrace_active();
  const int segments = shadow_volumes ? max(2,opt_segments/4) : opt_segments;
  const int segments_sv = max(2,opt_segments/4);

  if ( !n_instances ) return;

  if ( first_render )
    {
      for ( auto b: lis_bufs ) b->to_dev();
      buf_uni_common->sides = opt_sides;
      buf_uni_common->links_n_segs_sv = segments_sv;
      buf_uni_common->delta_tee_sv = 1.0f / segments_sv;
      buf_uni_common->links_n_segs = opt_segments;
      buf_uni_common->delta_tee = 1.0f / opt_segments;
    }

  if ( do_rt && first_render )
    {
      const int n_tri_p_segment = 2 * opt_sides;
      const int n_tri_p_link = n_tri_p_segment * opt_segments;
      const int n_tri_p_pass = n_tri_p_link * n_instances;

      const int n_vtx_p_segment = 2 * ( opt_sides + 1 );
      const int n_vtx_p_link = n_vtx_p_segment * opt_segments;
      const int n_vtx_p_pass = n_vtx_p_link * n_instances;

      if ( !rt_geo_links )
        rt_geo_links
          .init( raytrace )
          .color_set( color_dim_gray, color_salmon )
          .bind( VR_BUF_IDX_sampler, sampler, texid_hw )
          .grouping_set( RT_Geometry_Indexed );
```

```
      buf_links_as_idx.assure(3*n_tri_p_pass);
      buf_links_tex_coor.assure(n_vtx_p_pass);
      buf_links_vertex_g.assure(n_vtx_p_pass);
      buf_links_normal_g.assure(n_vtx_p_pass);

      rt_geo_links
        .bind( VR_BUF_IDX_indices, buf_links_as_idx, 3*n_tri_p_pass )
        .bind( VR_BUF_IDX_pos, buf_links_vertex_g, n_vtx_p_pass )
        .bind( VR_BUF_IDX_normal, buf_links_normal_g )
        .bind( VR_BUF_IDX_tcoor, buf_links_tex_coor );
    }

  if ( do_rt )
    {
      QSet& qs = vh.qs;
      buf_uni_common.to_dev();
      qs.cb.begin( { vk::CommandBufferUsageFlagBits::eOneTimeSubmit } );
      comp_links.record_dispatch(qs.cb,opt_segments,n_instances);
      qs.cb.end();
      qs.q.submit( vk::SubmitInfo{ 0, nullptr, nullptr, 1, &qs.cb }, nullptr );
      qs.q.waitIdle();
      return;
    }


  VPipeline& pipe = shadow_volumes ? pipe_set_2_sv : pipe_set_2;

  pipe.record_draw_instanced(cb, segments+1, n_instances);
}


void
World::frame_callback()
{
  // This routine called whenever window needs to be updated, but
  // before commands are recorded.

  // Get any waiting keyboard commands.
  //
  cb_keyboard();

  if ( vh.have_raytrace ) vh.opt_want_raytrace_now = opt_want_raytrace;

  // This is a hack for 2023 HW 2.
  while ( !balls.empty() && balls.back()->world_time_expire <= world_time )
    balls.pop_back();
  //
  // Balls must be last in list and not part of a link.

  const double time_now = time_wall_fp();
  const int time_step_count_start = time_step_count;
  const int nmp = gpu_info.cuda_prop.multiProcessorCount;
  int bl_p_sm_max = 1024;
  for ( int i=0; i<gpu_info.num_kernels; i++ )
```

```
    set_min
      (bl_p_sm_max,
       gpu_info.get_max_active_blocks_per_mp(i, opt_block_size, 0) );
  const int grid_size = bl_p_sm_max * nmp;

  const bool opt_cuda = opt_physics_method != GP_cpu;
  if ( opt_cuda )
    {
      CE( cudaEventRecord(frame_start_ce,0) );
      data_cpu_to_gpu_constants();
      data_cpu_to_gpu_dynamic();
    }

  if ( !opt_pause || opt_single_frame || opt_single_time_step )
    {
      /// Advance simulation state.

      // Amount of time since the user saw the last frame.
      //
      const double wall_delta_t = time_now - last_frame_wall_time;

      // Compute amount by which to advance simulation state for this frame.
      //
      const double duration =
        vh.video_frame_duration() ?:
        opt_single_time_step ? opt_single_time_step * opt_time_step_duration :
        opt_single_frame ? 1/60.0 :
        wall_delta_t;

      const double world_time_target = world_time + duration;
      const double wall_time_limit = time_now + 0.05;

      while ( world_time < world_time_target )
        {
          if ( opt_cuda )
            {
              launch_time_step(opt_time_step_duration,grid_size,opt_block_size);
            }
          else
            {
              time_step_cpu(opt_time_step_duration);
            }
          time_step_count++;
          world_time += opt_time_step_duration;
          const double time_right_now = time_wall_fp();
          if ( time_right_now > wall_time_limit ) break;
        }

      // Reset these, just in case they were set.
      //
      opt_single_frame = false;
      opt_single_time_step = 0;
    }
```

```
  if ( opt_cuda )
    {
      data_gpu_to_cpu_dynamic();
      CE( cudaEventRecord(frame_stop_ce,0) );
    }

  frame_timer.phys_end();
  frame_timer.work_amt_set(time_step_count-time_step_count_start);

  float cuda_time = 0;
  if ( opt_cuda )
    {
      CE(cudaEventSynchronize(frame_stop_ce));
      CE(cudaEventElapsedTime(&cuda_time,frame_start_ce,frame_stop_ce));
    }
  frame_timer.cuda_frame_time_set(cuda_time);

  last_frame_wall_time = time_now;

}

int
main(int argv, char **argc)
{
  pVulkan_Helper pvulkan_helper(argv,argc);
  World world(pvulkan_helper,argv,argc);

  world.run();
}
```

# 19 Fall 2024 Solutions

```
/// LSU EE 4702-1 (Fall 2024), GPU Programming
//
//
 /// Homework 1 -- SOLUTION
 //
 //  Assignment: https://www.ece.lsu.edu/gpup/2024/hw01.pdf
 //
 //  Solution is in routine render_hw01, search for SOLUTION.

#include <span>
#include <gp/coord.h>
#include <gp/texture-util.h>
#include "frame_buffer.h"

// Definitions for iVect, iCoor
#include "hw01-misc.h"


enum Wrap_Mode { WM_Repeat, WM_Mirror, WM_Transparent, WM_ENUM_SIZE };
const char* wrap_mode_str[] = { "REPEAT", "MIRROR", "TRANSPARENT" };


struct HW01_Data {
  HW01_Data():inited(false){};
  bool inited;
  bool tryout1, tryout2;
  pCoor img_box[4];  // Or, just use floats.
  uint img_box_dragging;
  vector<uint32_t> img_pixels;
  int img_px_width, img_px_height;
  float zoom;
  int wrap_mode;
};

void kbd_input_handle(pFrame_Buffer& fb, HW01_Data& hw01_data);
void image_box_drag(pFrame_Buffer& fb, HW01_Data& hw01_data);
void line_draw( pFrame_Buffer& fb, iCoor p0, iCoor p1, uint32_t co);
void aa_rectangle_draw
( pFrame_Buffer& fb, iCoor p00, int wd, int ht, uint32_t color );
void aa_rectangle_draw
( pFrame_Buffer& fb, iCoor p00, iCoor p11, uint32_t color );
void parallelogram_draw
( pFrame_Buffer& fb, iCoor p00, iVect vx,  iVect vy, uint32_t color );


void
render_hw01(pFrame_Buffer& fb, HW01_Data& hw01_data)
{
  // Window Size, in Pixels
  //
  const int win_width = fb.width_get();
  const int win_height = fb.height_get();
  const auto& tryout1 [[maybe_unused]] = hw01_data.tryout1;
```

```
const auto& tryout2 [[maybe_unused]] = hw01_data.tryout2;

kbd_input_handle(fb,hw01_data);
image_box_drag(fb,hw01_data); // Update image box corners.


/// Coordinates Used for Placement of Other Objects
//
const int sq_slen = min(win_width,win_height) * .45;
const int sq_x0 = sq_slen * 0.1,   sq_y0 = win_height - 150;
const int sq_x1 = sq_x0 + sq_slen, sq_y1 = sq_y0 - sq_slen;
//
// One vertex of the square is at (sq_x0,sq_y0) and the opposite
// vertex is at (sq_x1,sq_y1).

//                            RRGGBB
const uint32_t color_red = 0xff0000;
const uint32_t color_blue =    0xff;
const uint32_t color_green = 0xff00;

/// Demo Code: Draw Sine Waves  -- No need to modify this.
//
int num_waves = 4;
float plot_ht = sq_slen/float(num_waves);
for ( float
        yb = sq_y0 - plot_ht/2,  freq = 4 * M_PI / ( win_width - 40 - sq_x1 );
      yb > sq_y1;
      yb -= plot_ht,  freq *= 2 )
  for ( int x = 20;  x < win_width - 20;  x++ )
    {
      int y = yb + sin( x * freq ) * plot_ht * .45;
      fb[ y * win_width + x ] = 0xff00;
    }

/// Demo Code: Draw Lines Radiating From a Point -- No need to modify.
//
//  This code calls the line_draw routine.
//
int n_rects = 24;
int n_pts = n_rects * 3;
float c_r1 = sq_slen/6;  // Inner radius
float c_r2 = sq_slen/2;  // Outer radius.
pCoor c_ctr( c_r2 + 20, c_r2 + 20 ); // Center of "circle"
float delta_theta = 2 * M_PI / n_pts;
vector<pVect> vecs;
for ( int i=0; i<n_pts; i++ )
  vecs.push_back( pVect( cosf(i*delta_theta), sinf(i*delta_theta), 0 ) );
for ( int i=0; i<n_pts; i+=3 )
  {
    pVect v1 = vecs[i+1] * ( c_r2 - c_r1 );
    pVect v2 = c_r1 * ( vecs[i+2] - vecs[i] );
    pCoor p1 = c_ctr + c_r1 * vecs[i];
    parallelogram_draw(fb, p1, v1, v2, 0xffffff );
  }
```

```
/// Homework 1: No need to modify code above.


/// Image Box Variables
///
const auto& img_box = hw01_data.img_box;
//
// Image Box Vertices (Corners)
//
//   p3 --- p2
//   |       |
//   p0 --- p1
//
const pCoor p0 [[maybe_unused]] = img_box[0];
const pCoor p1 [[maybe_unused]] = img_box[1];
const pCoor p2 [[maybe_unused]] = img_box[2];
const pCoor p3 [[maybe_unused]] = img_box[3];

/// SOLUTION
//
//  Compute width of image box, correct even if image box rotated.
//
pNorm v01(p0,p1);
pNorm v03(p0,p3);
//
const int box_wd = v01.magnitude;
const int box_ht = v03.magnitude;
//
// Comment out previous code for pedagogical reasons.
//
// float box_wd = p1.x - p0.x;
// float box_ht = p3.y - p0.y;

// Image pixel array, and width and height, in pixels.
//
const auto& img_pixels = hw01_data.img_pixels;
const int img_wd = hw01_data.img_px_width;
const int img_ht = hw01_data.img_px_height;

// Display Options
//
const float zoom [[maybe_unused]] = hw01_data.zoom;
const int wrap_mode = hw01_data.wrap_mode;

//  float img_scale_y = img_ht / box_ht;

/// SOLUTION -- Problems 1 and 2
//
// Problem 2: Adjust img_scale_x based on zoom.
//
float img_scale_x = float(img_wd) / ( box_wd * zoom );
//
// Problem 1: Make y scale same as x scale to preserve aspect ratio.
```

```
//
const float img_scale_y = img_scale_x;


// Write img_pixels to Frame Buffer at Image Box (p0,p1,p2,p3)
//
for ( int y=0; y<box_ht; y++ )
  for ( int x=0; x<box_wd; x++ )
    {
      // Compute x and y frame buffer coordinates.
      //
      // Original Code.
      // int fb_x = p0.x + x;
      // int fb_y = p0.y + y;
      //
      /// SOLUTION -- Problem 4
      //
      // Vector v01 points from p0 to p1 and its length is 1.
      // Vector v03 points from p0 to p3 and its length is 1.
      //
      // Variables x and y give the location of the frame buffer pixel
      // relative to p0. With no rotation we can compute fb_x by
      // just adding: fb_x = p0.x + x.
      //
      // But with rotation we need to use v01 *and* v03 to compute
      // fb_x:
      //
      int fb_x = p0.x + x * v01.x + y * v03.x;
      //
      // Similarly for fb_y:
      //
      int fb_y = p0.y + x * v01.y + y * v03.y;
      //
      // Some tried solving the problem by computing the angle by
      // which the image box is rotated, then using cosine and sine
      // of that angle to compute the rotated frame buffer
      // coordinates.  That's not necessary because cosine and sine
      // are just ratios of lengths of right triangle sides that we
      // already know:
      // Note: v01.x = cos( theta ) = ( p1.x - p0.x ) / box_wd
      //  where theta = arctan2( p1.y-p0.y, p1.x-p0.x ).
      // Notice that there is no need to compute theta since v01.x is
      // exactly what we need.

      // Compute preliminary image pixel coordinates.
      //
      int img_x_raw = x * img_scale_x;
      int img_y_raw = y * img_scale_y;
      //
      // These will be used to compute the pixel coordinates that
      // take the wrap mode into account.

      int img_x, img_y;
```

```
    // Adjust img_x and img_y based on wrap mode.
    //
    switch ( wrap_mode ) {
      /// SOLUTION -- Problem 3
    case WM_Repeat:
      // For repeat, just use the modulo operator.
      img_x = img_x_raw % img_wd;
      img_y = img_y_raw % img_ht;
      break;
    case WM_Mirror:
      {
        // For the mirror mode we need to know whether to apply mirroring
        // to that axis. So first we compute how many tiles are to
        // the left of img_x_raw, nx:
        //
        int nx = img_x_raw / img_wd;
        //
        // If there are an even number of tiles (including 0) we don't
        // mirror, otherwise we do.
        //
        int xm = img_x_raw % img_wd;
        img_x =  nx % 2  ?  img_wd - xm - 1  :  xm;
        //                  Mirror---------     Don't Mirror

        // Do the same for the y axis.
        int ny = img_y_raw / img_ht;
        int ym = img_y_raw % img_ht;
        img_y = ny % 2 ? img_ht - ym - 1 : ym;
        break;
      }
    case WM_Transparent:
      if ( img_x_raw >= img_wd || img_y_raw >= img_ht ) continue;
      img_x = img_x_raw;  img_y = img_y_raw;
      break;
    default:
      assert( false ); // Check for an unexpected wrap mode.
    }

    // Check whether image pixel coordinates are valid ..
    //
    const size_t img_idx = img_y * img_wd + img_x;
    const bool out_of_range = img_idx >= img_pixels.size();
    //
    // .. and read pixel if they are valid, otherwise use color red.
    //
    uint32_t img_pixel = out_of_range ? color_red : img_pixels[ img_idx ];
    //
    // Finally, write image pixel into frame buffer.
    //
    fb[ fb_y * win_width + fb_x ] = img_pixel;
  }


/// Homework 1: No need to modify code below.
```

```
  //

  // Draw Lines Around Image Box
  //
  line_draw(fb, img_box[0], img_box[1], color_blue );
  line_draw(fb, img_box[1], img_box[2], color_green );
  line_draw(fb, img_box[2], img_box[3], color_red );
  line_draw(fb, img_box[0], img_box[3], 0xffffff );

  // Draw Image Box Corners
  //
  for ( auto& c: hw01_data.img_box )
    {
      int hwid = 10;
      pVect diag(hwid,hwid,0);
      pCoor ll = c - diag, ur = c + diag;
      bool inside = fb.mouse_x >= ll.x && fb.mouse_x <= ur.x
        && fb.mouse_y >= ll.y && fb.mouse_y <= ur.y;
      if ( inside && fb.mouse_button_left_state == GLUT_DOWN )
        hw01_data.img_box_dragging = distance(hw01_data.img_box,&c);
      uint32_t color = inside ? 0xff0000 : 0xffffff;
      aa_rectangle_draw(fb, ll, ur, color );
    }

}


void
line_draw
( pFrame_Buffer& fb, iCoor p0, iCoor p1, uint32_t co)
{
  int x0 = p0.x, y0 = p0.y;
  int x1 = p1.x, y1 = p1.y;

  const iVect delta = p1 - p0;
  const int width = fb.width_get();

  if ( abs( delta.x ) < abs( delta.y ) )
    {
      if ( y0 > y1 ) { swap(x0,x1); swap(y0,y1); }
      float dxdy = float(delta.x) / delta.y;
      float x = x0;
      for ( int yi = y0;  yi < y1;  yi++ )
        {
          fb[ yi * width + x ] = co;
          x += dxdy;
        }
    }
  else
    {
      if ( x0 > x1 ) { swap(x0,x1); swap(y0,y1); }
      float dydx = float(delta.y) / delta.x;
      float y = y0;
      for ( int xi = x0;  xi < x1;  xi++ )
```

```
        {
          fb[ int(y) * width + xi ] = co;
          y += dydx;
        }
    }
}


void
parallel_lines_draw
( pFrame_Buffer& fb, iCoor p0, iCoor pb, iVect v, uint32_t co)
{
  line_draw( fb, p0, p0 + v, co );
  line_draw( fb, pb, pb + v, co );
}

void
parallelogram_draw
( pFrame_Buffer& fb, iCoor p00, iVect vx,  iVect vy, uint32_t color )
{
  parallel_lines_draw(fb, p00, p00+vy, vx, color);
  parallel_lines_draw(fb, p00, p00+vx, vy, color);
}

void
aa_rectangle_draw
( pFrame_Buffer& fb, iCoor p00, int wd, int ht, uint32_t color )
{
  const int win_width = fb.width_get(), win_height = fb.height_get();

  // Compute coordinate of upper-right corner.
  //
  iCoor p11 = p00 + iVect(wd,ht);
  //
  // Coordinates p00 and p11 may be outside of the window.

  assert( wd >= 0 && ht >= 0 );  // Raise a fatal error if they are not.

  // Check whether rectangle will be completely outside the window.
  //
  if ( p00.x >= win_width || p00.y >= win_height ) return;
  if ( p11.x <= 0 || p11.y <= 0 ) return;

  // Compute clipped rectangle lower-left coordinate.
  //
  iCoor p00c( max( 0, p00.x ), max( 0, p00.y ) );
  //
  // Note: The /clipped/ rectangle is the portion of the rectangle
  // that is within the window (frame buffer).

  // Compute clipped rectangle upper-right coordinate.
  //
  iCoor p11c( min( win_width-1, p11.x ), min( win_height-1, p11.y ) );
```

```
  // Compute the other two clipped rectangle coordinates.
  //
  iCoor p01c( p00c.x, p11c.y ), p10c( p11c.x, p00c.y );

  // Draw visible vertical lines.
  //
  if ( p00.x == p00c.x ) line_draw( fb, p00c, p01c, color );
  if ( p11.x == p11c.x ) line_draw( fb, p10c, p11c, color );

  // Draw visible horizontal lines.
  //
  if ( p00.y == p00c.y ) line_draw( fb, p00c, p10c, color );
  if ( p11.y == p11c.y ) line_draw( fb, p01c, p11c, color );

}

void
aa_rectangle_draw
( pFrame_Buffer& fb, iCoor p00, iCoor p11, uint32_t color )
{
  iVect diag = p11 - p00;
  aa_rectangle_draw(fb, p00, diag.x, diag.y, color);
}


void
image_box_drag(pFrame_Buffer& fb, HW01_Data& hw01_data)
{
  // Coordinates of mouse pointer.
  //
  const pCoor mouse(fb.mouse_x,fb.mouse_y,0);

  auto& img_box = hw01_data.img_box;
  auto& drag_idx = hw01_data.img_box_dragging;

  // Turn off mouse dragging if button up, don't bother checking if it's on.
  if ( fb.mouse_button_left_state == GLUT_UP && drag_idx != 4 ) drag_idx = 4;

  const bool rot = drag_idx & 1;

  if ( drag_idx < 4 )
    {
      const int nbr_x_idx = (drag_idx+1) & 3, nbr_y_idx = (drag_idx+3) & 3;
      const int opp_idx = (drag_idx+2) & 3;
      pNorm pox_oldn(img_box[opp_idx],img_box[nbr_x_idx]);
      pNorm poy_oldn(img_box[opp_idx],img_box[nbr_y_idx]);
      pNorm doldn(img_box[opp_idx],img_box[drag_idx]);
      pVect dnew(img_box[opp_idx],mouse);
      if ( rot )
        {
          if ( pox_oldn.magnitude > 40 && poy_oldn.magnitude > 40 )
            {
              float costh = dot(dnew,doldn), sinth = cross(dnew,doldn).z;
              pVect pox_new = pox_oldn.magnitude / doldn.magnitude
```

```
                * (  costh * pox_oldn + sinth * poy_oldn );
              pVect poy_new = poy_oldn.magnitude / doldn.magnitude
                * ( -sinth * pox_oldn + costh * poy_oldn );
              img_box[drag_idx] = mouse;
              img_box[nbr_x_idx] = img_box[opp_idx] + pox_new;
              img_box[nbr_y_idx] = img_box[opp_idx] + poy_new;
            }
        }
      else
        {
          float lx = dot( pox_oldn, dnew ), ly = dot( poy_oldn, dnew );
          if ( fabs(lx) > 40 && fabs(ly) > 40 )
            {
              img_box[drag_idx] = mouse;
              img_box[nbr_x_idx] = img_box[opp_idx] + lx * pox_oldn;
              img_box[nbr_y_idx] = img_box[opp_idx] + ly * poy_oldn;
            }
        }
    }
}

void
image_box_reset(pFrame_Buffer& fb, HW01_Data& hw01_data)
{
  float img_aspect = float(hw01_data.img_px_width)/hw01_data.img_px_height;
  const int win_width = fb.width_get();
  const int win_height = fb.height_get();

  int box_width =
    img_aspect > 1 ? win_width / 2 : img_aspect * win_width / 2;
  int box_height =
    img_aspect < 1 ? win_height / 2 : box_width / img_aspect;

  pCoor img_center(win_width/2,win_height/2,0);
  pVect box_hw(box_width/2,0,0);
  pVect box_hh(0,box_height/2,0);

  hw01_data.img_box[0] = img_center - box_hw - box_hh;
  hw01_data.img_box[1] = img_center + box_hw - box_hh;
  hw01_data.img_box[2] = img_center + box_hw + box_hh;
  hw01_data.img_box[3] = img_center - box_hw + box_hh;
}

void
kbd_input_handle(pFrame_Buffer& fb, HW01_Data& hw01_data)
{
  if ( !hw01_data.inited )
    {
      hw01_data.inited = true;
      hw01_data.tryout1 = false;
      hw01_data.tryout2 = false;

      P_Image_Read image("pipelines_2x-crop-lab.png");
      hw01_data.img_pixels.reserve(image.size);
```

```
      hw01_data.img_px_height = image.height;
      hw01_data.img_px_width = image.width;
      auto sc = [&](uint32_t v){ return MaxRGB == 65535 ? v >> 8 : v; };
      // Copy image so that first pixel is lower-left.
      for ( int idx = image.size;  (idx-=image.width) >= 0; )
        for ( auto& pp: span(&image.pp[idx],image.width) )
          hw01_data.img_pixels.push_back
            ( sc(pp.red)<<16 | sc(pp.green)<<8 | sc(pp.blue) );

      image_box_reset(fb,hw01_data);
      hw01_data.img_box_dragging = 4;

      hw01_data.zoom = 1;
      hw01_data.wrap_mode = WM_Repeat;
     }

  switch ( fb.keyboard_key ) {
  case FB_KEY_RIGHT: case FB_KEY_LEFT:
    {
      break;
    }
  case FB_KEY_UP: case FB_KEY_DOWN:
    {
      break;
    }

  case 'R': image_box_reset(fb,hw01_data); break;

  case 'y':
    hw01_data.tryout1 = !hw01_data.tryout1; break;
  case 'Y':
    hw01_data.tryout2 = !hw01_data.tryout2; break;

  case 'w': case 'W': hw01_data.wrap_mode++;
    if ( hw01_data.wrap_mode == WM_ENUM_SIZE ) hw01_data.wrap_mode = 0;
    break;

  case 'u': hw01_data.zoom /= 1.1; break;
  case 'U': hw01_data.zoom /= 1.5; break;
  case 'z': hw01_data.zoom *= 1.1; break;
  case 'Z': hw01_data.zoom *= 1.5; break;

  default:
    if ( false && fb.keyboard_key ) printf("Unknown key, %d\n",fb.keyboard_key);
    break;
  };

  fb.fbprintf("Zoom: %.2f ('z','u')  Wrap Mode: %s ('w')  "
              "Tryout 1,2:  %s,%s ('y','Y')\n",
              hw01_data.zoom, wrap_mode_str[hw01_data.wrap_mode],
              hw01_data.tryout1 ? "TRUE " : "FALSE",
              hw01_data.tryout2 ? "TRUE " : "FALSE");

}
```

```
int
main(int argc, char **argv)
{
  pFrame_Buffer demo_frame_buffer(argc,argv);

  HW01_Data hw01_data;
  demo_frame_buffer.show
    ( [&](pFrame_Buffer& fb){ render_hw01(fb,hw01_data); } );
  return 0;
}
```

```
/// # LSU EE 4702-X (Fall 2024), GPU Programming
//
 /// ## Homework 2 -- SOLUTION
 //
 //  Assignment: https://www.ece.lsu.edu/gpup/2024/hw02.pdf
 //
 /// ## Modify this file only.
 //
 /// ## Put Solution and other Work In Routines:
 //
 //    World::ball_setup_prob_0(), World::ball_setup_prob_1(), and
 //    World::ball_setup_prob_2().
 //
 // Based on the Links Project Code
 //

/// # Purpose
//
//   Demonstrate simulation of point masses connected by springs.


/// # What Code Does

// Simulates balls connected by springs over a platform. Balls and
// springs can be initialized in different arrangements (called
// scenes). consists of tiles, some are purple-tinted mirrors (showing
// a reflection of the ball), the others show the course syllabus.



/// # Keyboard Commands
 //
 /// ## Object (Eye, Light, Ball) Location or Push
 //   Arrows (←,→,↑,↓) Page Up, Page Down
 //        Move object or push ball, depending on mode.
 //        Shift + KEY: motion is 5x faster.
 //        Ctrl + KEY : motion is 5x slower.
 //   'e': Move eye.
 //   'l': Move light.
 //   'b': Move head (first) ball. (Change position but not velocity.)
 //   'B': Push head ball. (Add velocity.)
 //
 /// ## Eye Direction and GUI Options
 //
 //   'Home', 'End', 'Delete', 'Insert'
 //         Turn the eye direction. Home should rotate eye direction
 //         up, End should rotate eye down, Delete should rotate eye
 //         left, Insert should rotate eye right. The eye direction
 //         vector is displayed in the upper left.
 //  'C-='  (Ctrl =) Increase green text size.
 //  'C--'  (Ctrl -) Decrease green text size.
 //  'F12'  Write screenshot to PNG file.
```

```
/// Simulation Execution Options
//  (Also see variables below.)
//
//  'p'    Pause simulation. (Press again to resume.)
//  ' '    (Space bar.) Advance simulation by 1/30 second.
//  'S- '  (Shift Space bar.) Advance simulation by one time step.

/// Performance Tuning, Graphics Effects, and Debugging Options
//
//  'y'    Toggle value of opt_tryout1. Intended for experiments and debugging.
//  'Y'    Toggle value of opt_tryout2. Intended for experiments and debugging.
//  'n'    Toggle use of surface normals for spheres.
//  'o'    Toggle shadow effect.
//  'r'    Toggle mirror effect.
//  '!'    Toggle rendering of spheres (balls).
//  '@'    Toggle rendering of links.
//  '#'    Toggle rendering of platform.


/// Scene and Simulated System Options
//  (Also see variables below.)
//
//  'w'    Twirl balls around axis formed by head and tail. (Prob 2 soln).
//  '1'    Set up scene 1.
//  '2'    Set up scene 2.
//  '3'    Set up scene 3.
//  '4'    Set up scene 4.
//  'h'    Freeze position of first (head) ball. (Press again to release.)
//  't'    Freeze position of last (tail) ball. (Press again to release.)
//  's'    Stop balls.
//  'g'    Turn gravity on and off.

/// Variables
//   Selected program variables can be modified using the keyboard.
//   Use "Tab" to cycle through the variable to be modified, the
//   name of the variable is displayed next to "VAR" on the bottom
//   line of green text.

//  'Tab' Cycle to next variable.
//  '`'   Cycle to previous variable.
//  '+'   Increase variable value.
//  '-'   Decrease variable value.
//
//  VAR Spring Constant - Set spring constant.
//  VAR Time Step Duration - Set physics time step.
//  VAR Air Resistance - Set air resistance.
//  VAR Light Intensity - The light intensity.
//  VAR Gravity - Gravitational acceleration. (Turn on/off using 'g'.)

#define MAIN_INCLUDE
#undef VBUFFER_STRICT
//  #define VBUFFER_STRICT_WATCH 0xd10d270000000018
#include <vhelper.h>
```

```
#include <vstroke.h>
#include <gp/coord.h>
#include <gp/pstring.h>
#include <gp/misc.h>

#include <vutil-texture.h>
#include <vutil-pipeline.h>
#include "shapes.h"

#include <cuda_runtime.h>

#include <gp/colors.h>

#include <gp/util-containers.h>
#include <gp/coord-containers.h>
#include <gp/cuda-gpuinfo.h>
#include "links.cuh"

#include "links-shdr-common.h"


///
/// Main Data Structures
///
//
// class World: All data about scene.


class World;

enum GPU_Physics_Method { GP_cpu, GP_cuda, GP_ENUM_SIZE };
const char* const gpu_physics_method_str[] = { "CPU", "CUDA" };
const char *sh_names[] = { "PLAIN", "SET 1", "SET 2" };


// Object Holding Ball State
//
class Ball {
public:
  Ball() { init(); }
  Ball(pCoor pos){ init(); position = pos; constants_update(); }
  Ball(pCoor pos, const Ball &bd)
  { *this = bd; position = pos; constants_update(); }

  void init()
    {
      velocity = pVect(0,0,0);
      omega = pVect(0,0,0);
      radius = 1;
      mass = 0;
      mass_inv = 1;
      density = 1.00746;
      fdt_to_do = 0;
```

```
    locked = false;
    color = color_lsu_spirit_gold;
    specularity = 0;
    contact = false;
    orientation_set(pVect(0,1,0),0);
    world_time_expire = 3600.0 * 24 * 365 * 1000;
  }

pCoor position;
pVect velocity;
pQuat orientation;
pMatrix3x3p omatrix;
pVect omega;                    // Spin rate and axis.
double world_time_expire;

int idx; // Position in balls_pos_rad;

union {
  float4 m_r_fdt;
  struct {
    float mass;
    float radius;
    float fdt_to_do; // Radius / moment of inertia.
    float _;
  }; };

float mass_min; // Mass below which simulation is unstable.
float radius_sq, radius_inv;  // Pre-computed based on radius.
float density;
float mass_inv;

bool locked;

pVect force;
pVect torque;
pColor color;
float specularity;              // Amount of shininess, in [0,1].
bool contact;                   // When true, ball rendered in gray.
float spring_constant_sum;      // Used to compute minimum mass.

void orientation_set(pNorm dir, float angle)
  { orientation_set(pQuat(dir,angle)); }
void orientation_set(pQuat ori)
  {
    orientation = ori;
    omatrix = pMatrix3x3_Rotation(orientation);
  }
void constants_update();
pVect point_rot_vel(pNorm tact_dir);
pVect point_rot_vel(pVect tact_dir);
void apply_tan_force(pNorm tact_dir, pNorm force_dir, double force_mag);
void apply_tan_force(pNorm tact_dir, pVect force);
void push(pVect amt);
void translate(pVect amt);
```

```
  void stop();
  void freeze();
};


void
Ball::constants_update()
{
  assert( radius );
  radius_sq = radius * radius;
  if ( mass >= 0 )
    {
      mass = 4.0 / 3 * M_PI * radius_sq * radius * density;
      mass_inv = 1 / mass;
    }
  radius_inv = 1 / radius;
  // FYI, notice simplifications:
  //   const float mo_inertia = 0.4 * mass * radius_sq;
  //   fdt_to_do = radius / mo_inertia;
  fdt_to_do = radius_inv * 2.5 * mass_inv;
}

pVect
Ball::point_rot_vel(pNorm direction)
{
  /// Return velocity of point on surface of ball.
  //
  return radius * cross( omega, direction );
}

pVect
Ball::point_rot_vel(pVect rel_pos)
{
  /// Return velocity of point relative to center.
  //
  return cross( omega, rel_pos );
}

void
Ball::apply_tan_force(pNorm tact_dir, pVect force)
{
  torque += cross(tact_dir, force);
}

void
Ball::apply_tan_force(pNorm tact_dir, pNorm force_dir, double force_mag)
{
  apply_tan_force(tact_dir, force_mag * force_dir);
}

struct Link_Args {
  pColor natural_color = color_red;
  float distance_relaxed = 0;
  float spring_constant = 0;
```

```
  float link_stiffness = 0;
  float radius = 0;
};

class Link {
public:
  Link(Ball *b1, Ball *b2, Link_Args args = {}):ball1(b1),ball2(b2),
     snapped(false)
  { init(args); }
  Link(Link *link, pVect cb1p, pVect cb2p, float drp):
    ball1(link->ball1), ball2(link->ball2), cb1(cb1p), cb2(cb2p),
    is_surface_connection(true), is_renderable(false),
    is_simulatable(true), distance_relaxed(drp), spring_constant(0),
    radius(0),snapped(false){link->is_simulatable = false;}

  Link(Link *link, pVect cb1p, pVect cb2p, Link_Args args = {} ):
    ball1(link->ball1), ball2(link->ball2), cb1(cb1p), cb2(cb2p),
    is_surface_connection(true), is_renderable(false),
    is_simulatable(true),
    distance_relaxed(args.distance_relaxed),
    spring_constant(args.spring_constant),
    radius(0),snapped(false){link->is_simulatable = false;}

  void init(Link_Args& args)
  {
    assert( args.distance_relaxed == 0 );
    natural_color = args.natural_color;
    spring_constant = args.spring_constant;
    radius = args.radius;
    init();
  }
  void init()
    {
      assert( ball1->radius > 0 );
      assert( ball2->radius > 0 );
      is_simulatable = true;
      is_renderable = true;
      pNorm n12(ball1->position,ball2->position);
      const float rad_sum = ball1->radius + ball2->radius;
      is_surface_connection = n12.magnitude >= rad_sum;

      pMatrix3x3 b1rot = ball1->omatrix;
      pMatrix3x3 b2rot = ball2->omatrix;

      pVect b1_y = b1rot * pVect(0,1,0);
      pVect b1_x = b1rot * pVect(1,0,0);
      bool b1_dir_is_y = fabs(dot(b1_y,n12)) < 0.999;
      pVect b1_dir_loc = b1_dir_is_y ? b1_y : b1_x;

      con_x = pNorm(cross(b1_dir_loc,n12));
      con_y = cross(n12,con_x);
      b1_dir = mult_MTV( b1rot, con_y );
      b2_dir = mult_MTV( b2rot, con_y );
```

```
      if ( !is_surface_connection )
        {
          distance_relaxed = n12.magnitude;
          cb1 = cb2 = pVect(0,0,0);
          is_renderable = false;
          return;
        }

      distance_relaxed = n12.magnitude - rad_sum;
      cb1 = ball1->radius * mult_MTV( b1rot,  n12 );
      cb2 = ball2->radius * mult_MTV( b2rot, -n12 );
    }
  int serial;
  int idx;  // Position in lis_pos1, etc.
  Ball* const ball1;
  Ball* const ball2;
  union {
    int2 ball_idx;
    struct { int ball1_idx, ball2_idx; };
  };
  pVect cb1, cb2, con_x, con_y, b1_dir, b2_dir;
  bool is_surface_connection;
  bool is_renderable, is_simulatable;
  union {
    float2 dr_sc;
    struct {
      float distance_relaxed;
      float spring_constant;
    };
  };
  float radius;
  bool snapped;
  pColor natural_color;
  pColor color;

  pVect spring_force_12;
  pVect torque1,torque2;

};

class Surface {
public:
  Ball *b0, *b1, *b2;
  pColor color = color_blue;
  pTCoor tc0, tc1, tc2;
  union {
    int4 b_idx;
    struct { int b0_idx, b1_idx, b2_idx, _; };
  };
  Surface* operator ->() { return this; }
};

// Declare containers for Balls and Links. The containers are derived
// from std::vector and use overloaded operator += for push_back. (See
```

```
// include/util-containers.h.)
//
typedef pVectorI<Link> Links;
typedef pVectorI<Ball> Balls;

pVect vec_rand() {return pVect(drand48(),drand48(),drand48());}

class HW_Info
{
public:
  HW_Info(){set();};
  int n_balls, n_lines;
  float ball_radius;
  pNorm axis;
  pCoor center_pos,  line_0_bot_pos, line_0_top_pos;

  void set()
  {
    center_pos = pCoor(12.2,4.8f,0) + 2*vec_rand();
    n_balls = 4 + random()%8;
    n_lines = 32 + random()%20;

    pNorm az = pVect( drand48(), 10, drand48() );
    axis = az;
    pNorm ax =
      fabs(az.x) < fabs(az.y) ? pVect(0,az.z,-az.y) : pVect(az.z,0,-az.x);
    pNorm ay = cross(az,ax);

    float r_bot = 10 + 10 * drand48();
    ball_radius = 0.2;
    line_0_bot_pos = center_pos + r_bot * ax;
    line_0_top_pos =
      center_pos + ( 10 + 10 * drand48() ) * az
      + drand48() * r_bot * ax + 3*(drand48()-0.5) * r_bot * ay;
  }
};

enum Shader_Option { SO_Plain, SO_Instances, SO_True, SO_ENUM_SIZE };

class World {
public:
  World(pVulkan_Helper &fb, int argc, char **argv)
    :vh(fb), ff_state(fb.qs), raytrace(fb.qs),
     frame_timer(vh.frame_timer), transform(fb.qs),
     shapes(ff_state), sphere(ff_state),
     opt_tryout1(fb.qs.opt_tryout1),opt_tryout2(fb.qs.opt_tryout2)
  {init(argc,argv);}
  void init(int argc, char **argv);
  void init_graphics();
  void run();
  void frame_callback();
  void render(vk::CommandBuffer& cb);
  void render_objects(vk::CommandBuffer& cb, Render_Option render_option);
```

```
void objects_erase();
void cb_keyboard();
void modelview_update();

pVulkan_Helper& vh;
VFixed_Function_State_Manager ff_state;
VRaytrace raytrace;
pFrame_Timer& frame_timer;
VTransform transform;
Shapes shapes;
Sphere sphere;

bool opt_want_raytrace;

pVariable_Control variable_control;
double world_time;
double last_frame_wall_time;
float opt_time_step_duration;
int time_step_count;
float opt_gravity_accel;      // Value chosen by user.
pVect gravity_accel;          // Set to zero when opt_gravity is false;
bool opt_gravity;
bool opt_head_lock, opt_tail_lock;
bool &opt_tryout1, &opt_tryout2;  // For ad-hoc experiments.
float opt_tryoutf;
bool opt_sphere_true;
VBufferV<Shdr_Uni_Common> buf_uni_common;

// Tiled platform for ball.
//
float platform_xmin, platform_xmax, platform_zmin, platform_zmax;
float platform_pi_xwidth_inv;
VVertex_Buffer_Set bset_platform_tex, bset_platform_mir;
VPipeline pipe_platform_mir, pipe_platform_tex;
RT_Geo_Package rt_geo_platform_tex, rt_geo_platform_mir;

vk::Sampler sampler;
VTexture texid_hw;
VTexture texid_syl;
VTexture texid_emacs;

void platform_update();
bool platform_collision_possible(pCoor pos);

pCoor light_location;
float opt_light_intensity;
enum { MI_Eye, MI_Light, MI_Ball, MI_Ball_V, MI_COUNT } opt_move_item;
bool opt_pause;
bool opt_single_frame;      // Simulate for one frame.
int opt_single_time_step;  // Simulate for one time step.

VBufferV<Uni_Lighting> uni_light, uni_light_ambient, uni_light_all;
VBufferV<Uni_Lighting> uni_light_reflected;
VBufferV<Uni_Lighting> *puni_light_curr;
```

```
pCoor eye_location;
pVect eye_direction;
pMatrix transform_projection;
pMatrix modelview;

pVect adj_vector;
double adj_t_prev;
double adj_t_stop;
double adj_duration;

uint opt_shader;

bool opt_mirror;
int opt_mirror_depth;
bool opt_shadows;
bool opt_shadow_volumes;      // Make shadow volumes visible.
bool opt_normal_sphere;
int opt_hide_stuff;

int last_setup; // Last scene set up.

void ball_setup_1(bool big);  // Silly Tree
void ball_setup_2(); // Hanging thing.
void ball_setup_3(); // Corona virus particle.
void ball_setup_4(); // Wheel.
void ball_setup_5(); // Top.
void ball_setup_6(); // Wheel with webbing between spokes.
void ball_setup_7();
void setup_at_end();
void time_step_cpu(double);
void balls_stop();
void balls_freeze();
void balls_translate(pVect amt, int idx);
void balls_translate(pVect amt);
void balls_push(pVect amt, int idx);
void balls_push(pVect amt);
void balls_twirl();
void lock_update();

Ball *make_marker(pCoor pos, pColor col, float radius = 0.2 );
Balls balls_marker; // Written to balls right after setup.

float opt_spring_constant, opt_spring_damp;
float opt_friction_dynamic;
float opt_air_resistance;
float distance_relaxed;
int chain_length;
Balls balls;
Ball *head_ball, *tail_ball;
Links links;
vector<Surface> surfaces;
int opt_sphere_slices;
```

```
float opt_omega;  // For disk and top.

Links link_new
( Ball *ball1, Ball *ball2,
  float stiffness = 0.2, float spring_constant = 0);
Links link_new
( Ball *ball1, Ball *ball2, Link_Args largs );

/// CUDA Stuff

int opt_physics_method;
GPU_Info gpu_info;
cudaEvent_t frame_start_ce, frame_stop_ce;
int opt_block_size;
bool gpu_const_data_stale;
bool cpu_phys_data_stale;

void data_cpu_to_gpu_constants();
void data_cpu_to_gpu_dynamic();
void data_gpu_to_cpu_dynamic();
CPU_GPU_Common c;  // c is for common.  See links.cuh.

void render_link_start();
void render_link_gather(Link *link);
void render_link_render(vk::CommandBuffer& cb, bool shadow_volumes = false);
void render_link_render_sv(vk::CommandBuffer& cb)
{ render_link_render(cb,true); };

VPipeline pipe_set_2, pipe_set_2_sv;
double world_time_link_update;

VCompute comp_links;
VBuffer<int> buf_links_as_idx; // Indices for links' acceleration structures.
VBuffer<vec2> buf_links_tex_coor;
VBuffer<vec4> buf_links_vertex_g;
VBuffer<vec4> buf_links_normal_g;
RT_Geo_Package rt_geo_links;

int opt_sides, opt_segments;

vector< VBufferVV<vec4>* > lis_bufs;
VBufferVV<vec4> lis_pos1, lis_pos2;
VBufferVV<vec4> lis_v1, lis_v2;
VBufferVV<vec4> lis_b1_ydir, lis_b2_ydir, lis_tex_rad;

VVertex_Buffer_Set bset_surfaces;
VPipeline pipe_surfaces;
RT_Geo_Package rt_geo_surfaces;
void render_surfaces(vk::CommandBuffer& cb);

/// Stuff for 2024 Homework 2

void ball_setup_hw2( char version );
```

```
  void ball_setup_prob_0();
  void ball_setup_prob_1();
  void ball_setup_prob_2();

  HW_Info h;
  bool opt_freeze_config;

};

void
World::ball_setup_hw2( char version )
{
  last_setup = version - '0';
  objects_erase();

  /// Homework 2: Do not modify this routine.
  // Your solution and other work goes in World::ball_setup_prob_0(),
  // World::ball_setup_prob_1(), and World::ball_setup_prob_2().

  if ( !opt_freeze_config ) h.set();

  switch ( version ){
  case '0': ball_setup_prob_0(); break;
  case '1': ball_setup_prob_1(); break;
  case '2': ball_setup_prob_2(); break;
  default: assert( false );
  }
  setup_at_end();
}

enum { OH_Links = 1, OH_Platform = 2, OH_Sphere = 4 };

void
World::init_graphics()
{
  vh.want_raytrace().init();
  vh.opt_record_every_time = true;
  vh.display_cb_set([&](){frame_callback();});
  vh.cbs_cmd_record.push_back( [&](vk::CommandBuffer& cb){ render(cb); });
  sphere.ppuni_light = &puni_light_curr;
  sphere.transform = &transform;

  opt_want_raytrace = true;

  if ( vh.have_raytrace )
    {
      raytrace.init();
      for ( auto b: { &bset_platform_mir, &bset_platform_tex, &bset_surfaces } )
        b->usage_set
          ( vk::BufferUsageFlagBits::eVertexBuffer
            | vk::BufferUsageFlagBits::eStorageBuffer );
      buf_links_as_idx.init
        ( vh.qs, vk::BufferUsageFlagBits::eStorageBuffer, 0,
          vk::MemoryPropertyFlagBits::eDeviceLocal );
```

```
    buf_links_tex_coor.init
      ( vh.qs, vk::BufferUsageFlagBits::eStorageBuffer, 0,
        vk::MemoryPropertyFlagBits::eDeviceLocal );
    buf_links_vertex_g.init
      ( vh.qs, vk::BufferUsageFlagBits::eStorageBuffer, 0,
        vk::MemoryPropertyFlagBits::eDeviceLocal );
    buf_links_normal_g.init
      ( vh.qs, vk::BufferUsageFlagBits::eStorageBuffer, 0,
        vk::MemoryPropertyFlagBits::eDeviceLocal );
  }

buf_uni_common.init(vh.qs,vk::BufferUsageFlagBits::eUniformBuffer);
sphere.buf_uni_common = buf_uni_common;

lis_bufs =
  { &lis_pos1, &lis_pos2,
    &lis_v1, &lis_v2,
    &lis_b1_ydir, &lis_b2_ydir,
    &lis_tex_rad };

for ( auto b: lis_bufs )
  b->init(vh.dev_phys, vh.dev, vk::BufferUsageFlagBits::eStorageBuffer );

///
/// Graphical Model Initialization
///

opt_head_lock = false;
opt_tail_lock = false;
opt_tryout1 = opt_tryout2 = false;
opt_normal_sphere = true;
opt_sphere_true = true;
opt_tryoutf = 0;
variable_control.insert_linear(opt_tryoutf,"Tryout F",0.1);

eye_location = pCoor(24.2,11.6,-38.7);
eye_direction = pVect(-0.42,-0.09,0.9);

platform_xmin = -40; platform_xmax = 40;
platform_zmin = -40; platform_zmax = 40;

texid_syl.init
  ( vh.qs, P_Image_Read( vh.gp_root_get() / "vulkan" / "gpup.png", 255 ) );
texid_emacs.init( vh.qs, P_Image_Read("mult.png",-1) );
texid_hw.init( vh.qs, P_Image_Read("hw05-assign.png",255) );
sampler = vh.qs.dev.createSampler
  ( { {},
      vk::Filter::eLinear,  // magFilter
      vk::Filter::eLinear,  // minFilter
      vk::SamplerMipmapMode::eLinear,
      // Also: eRepeat, eMirroredRepeat, eClampToEdge, etc.
      vk::SamplerAddressMode::eRepeat,
      vk::SamplerAddressMode::eRepeat,
      vk::SamplerAddressMode::eRepeat,
```

```
        0.0f,  // mipLodBias
        true, // anisotropyEnable,
        16.0f, // maxAnisotropy
        false, vk::CompareOp::eNever,  // compareEnable, compareOp
        0.0f, VK_LOD_CLAMP_NONE,  // min and max LOD
        vk::BorderColor::eFloatOpaqueBlack,
        false // unnormalizedCoordinates
        } );

  sphere.texture_set(sampler,texid_emacs);

  opt_light_intensity = 50;
  light_location = pCoor(platform_xmax,platform_xmax,platform_zmin);
  uni_light.init(vh.qs,vk::BufferUsageFlagBits::eUniformBuffer);
  uni_light_all.init(vh.qs,vk::BufferUsageFlagBits::eUniformBuffer);
  uni_light_reflected.init(vh.qs,vk::BufferUsageFlagBits::eUniformBuffer);
  uni_light_ambient.init(vh.qs,vk::BufferUsageFlagBits::eUniformBuffer);

  variable_control.insert(opt_light_intensity,"Light Intensity");

  opt_move_item = MI_Eye;
  opt_pause = false;
  opt_single_time_step = 0;
  opt_single_frame = false;

  platform_update();
  modelview_update();

  adj_t_stop = 0;
  adj_duration = 0.25;

  opt_sides = 20;
  opt_segments = 16;
  //  variable_control.insert(opt_sides,"Link Sides");
  //  variable_control.insert(opt_segments,"Link Segments",5,1);
  opt_hide_stuff = 0;

  if ( vh.have_raytrace )
    {
      raytrace.buf_light_set(uni_light_all);
    }

  opt_freeze_config = false;
}

void
World::run()
{
  vh.message_loop_spin();

  uni_light.destroy();
  uni_light_ambient.destroy();
  uni_light_all.destroy();
  uni_light_reflected.destroy();
```

```
  transform.destroy();
  buf_uni_common.destroy();

  for ( auto b: lis_bufs ) b->destroy();

  pipe_platform_tex.destroy(); pipe_platform_mir.destroy();
  bset_platform_tex.destroy(); bset_platform_mir.destroy();
  pipe_set_2.destroy();
  pipe_set_2_sv.destroy();

  texid_syl.destroy();
  texid_emacs.destroy();
  texid_hw.destroy();
  vh.dev.destroySampler(sampler);
  shapes.destroy();
  sphere.destroy();

  bset_surfaces.destroy();
  pipe_surfaces.destroy();
  rt_geo_surfaces.destroy();

  buf_links_as_idx.destroy();
  buf_links_tex_coor.destroy();
  buf_links_normal_g.destroy();
  buf_links_vertex_g.destroy();
  rt_geo_platform_tex.destroy();
  rt_geo_platform_mir.destroy();
  rt_geo_links.destroy();
  comp_links.destroy();
  raytrace.destroy();

  vh.finish();
}

void
World::render_surfaces(vk::CommandBuffer& cb)
{
  if ( !pipe_surfaces )
    pipe_surfaces
      .init( vh.qs )
      .ds_follow( puni_light_curr )
      .ds_follow( transform )
      .ds_uniform_use( "BIND_UNI_COMMON", buf_uni_common )
      .topology_set( vk::PrimitiveTopology::eTriangleList )
      .create();

  bset_surfaces.reset( pipe_surfaces );

  for ( Surface& s: surfaces )
    {
      Ball &b0 = *s.b0, &b1 = *s.b1, &b2 = *s.b2;
      pNorm snorm = cross(b0.position,b1.position,b2.position);
      bset_surfaces << b0.position << b1.position << b2.position;
      bset_surfaces << s.color << s.color << s.color;
```

```
      bset_surfaces << snorm << snorm << snorm;
    }

  bset_surfaces.to_dev();

  if ( vh.have_raytrace && !rt_geo_surfaces )
    rt_geo_surfaces.init( raytrace );

  if ( vh.raytrace_active() )
    {
      rt_geo_surfaces.bind( bset_surfaces, RT_Geometry_Triangles );
      return;
    }

  pipe_surfaces.record_draw( cb, bset_surfaces );
}

void
World::platform_update()
{
  const float tile_count = 19;
  const float ep = 1.00001;
  const float delta_x = ( platform_xmax - platform_xmin ) / tile_count * ep;
  const float zdelta = ( platform_zmax - platform_zmin ) / tile_count * ep;

  const float trmin = 0.05;
  const float trmax = 0.7;
  const float tsmin = 0;
  const float tsmax = 0.4;
  pColor color_platform( color_white * 0.9 );

  if ( !pipe_platform_mir )
    pipe_platform_mir
      .init( ff_state )
      .ds_set( color_lsu_spirit_purple )
      .ds_follow( puni_light_curr )
      .ds_follow( transform )
      .topology_set( vk::PrimitiveTopology::eTriangleList )
      .create();
  bset_platform_mir.reset( pipe_platform_mir ).tolerate_tcoords();

  if ( !pipe_platform_tex )
    pipe_platform_tex
      .init( ff_state )
      .ds_set_material(color_platform)
      .ds_follow( puni_light_curr )
      .ds_follow( transform )
      .topology_set( vk::PrimitiveTopology::eTriangleList )
      .ds_use( sampler, texid_syl )
      .create();
  bset_platform_tex.reset( pipe_platform_tex );

  const pVect platform_normal(0,1,0);
  pTCoor t00(trmin,tsmin), t01(trmin,tsmax), t10(trmax,tsmin), t11(trmax,tsmax);
```

```
  bool even = true;

  for ( int i = 0; i < tile_count; i++ )
    {
      const float x0 = platform_xmin + i * delta_x;
      const float x1 = x0 + delta_x;
      const float y = 0;
      for ( float z = platform_zmin; z < platform_zmax; z += zdelta )
        {
          auto& bset = even ? bset_platform_tex : bset_platform_mir;
          const float z1 = z+zdelta;
          pCoor p00(x0,y,z), p01(x0,y,z1), p10(x1,y,z), p11(x1,y,z1);

          bset << t11 << platform_normal << p00;
          bset << t10 << platform_normal << p01;
          bset << t00 << platform_normal << p11;

          bset << t11 << platform_normal << p00;
          bset << t00 << platform_normal << p11;
          bset << t01 << platform_normal << p10;

          even = !even;
        }
    }

  bset_platform_mir.to_dev();
  bset_platform_tex.to_dev();

  if ( vh.have_raytrace && !rt_geo_platform_mir )
    {
      pColor mirror_color(color_lsu_spirit_purple);
      mirror_color.a = -0.5;
      rt_geo_platform_mir
        .init( raytrace )
        .color_front_set( mirror_color )
        .bind( bset_platform_mir, RT_Geometry_Triangles );
      rt_geo_platform_tex
        .init( raytrace )
        .color_front_set( color_platform )
        .bind( VR_BUF_IDX_sampler, sampler, texid_syl )
        .bind( bset_platform_tex, RT_Geometry_Triangles );
    }
}

void
World::modelview_update()
{
  pMatrix_Translate center_eye(-eye_location);
  pMatrix_Rotation rotate_eye(eye_direction,pVect(0,0,-1));
  modelview = rotate_eye * center_eye;
}

void
World::render_objects(vk::CommandBuffer& cb, Render_Option option)
```

```
{
  //  const float shininess_ball = 5;
  pColor spec_color(1,1,1);
  //  pColor spec_light = color_white * opt_light_intensity;

  const bool hide_links = opt_hide_stuff & OH_Links;
  const bool hide_platform = opt_hide_stuff & OH_Platform;
  const bool hide_sphere = opt_hide_stuff & OH_Sphere;

  sphere.init(opt_sphere_slices);

  if ( option == RO_Shadow_Volumes )
    {
      if ( !hide_sphere )
        {
          sphere.light_pos = light_location;
          sphere.render_bunch_gather_sv(world_time);
          for ( Ball *ball: balls )
            sphere.render_shadow_volume(cb,ball->radius,ball->position);

          sphere.render_bunch_render_sv(cb);
        }

      if ( !hide_links )
        {
          render_link_start();
          for ( Link *link: links )
            if ( link->is_renderable ) render_link_gather(link);
          render_link_render_sv(cb);
        }
    }
  else
    {
      sphere.rt_geo_sphere.rt_wanted_set(!hide_sphere);
      if ( !hide_sphere )
        {
          sphere.render_bunch_gather(world_time);
          for ( Ball *ball: balls )
            {
              if ( ball->contact )
                sphere.color = color_gray;
              else if ( ball->mass > 0 && ball->mass < ball->mass_min )
                sphere.color = color_red;
              else if ( ball->mass > 0 && ball->locked )
                sphere.color = color_pale_green;
              else
                sphere.color = ball->color;

              sphere.color.a = ball->specularity ? -ball->specularity : 1;
              sphere.render
                (cb, ball->radius,ball->position,
                 pMatrix_Rotation(ball->orientation));
            }
```

```
          sphere.render_bunch_render(cb, opt_sphere_true);
        }

      rt_geo_links.rt_wanted_set(!hide_links);
      if ( !hide_links )
        {
          render_link_start();
          for ( Link *link: links )
            if ( link->is_renderable ) render_link_gather(link);
          render_link_render(cb);
        }
    }

  rt_geo_platform_tex.rt_wanted_set(!hide_platform);
  rt_geo_platform_mir.rt_wanted_set(!hide_platform);

  if ( option != RO_Shadow_Volumes )
    render_surfaces(cb);

  if ( vh.raytrace_active() ) return;
  if ( hide_platform ) return;

  if ( option == RO_Shadow_Volumes ) return;

  //
  // Render Platform
  //
  pipe_platform_tex.record_draw(cb,bset_platform_tex);
}

void
World::render(vk::CommandBuffer& cb)
{
  const bool do_raytrace = vh.raytrace_active();

  /// Emit a Graphical Representation of Simulation State
  //

  // Understanding of the code below not required for introductory
  // lectures.

  // That said, much of the complexity of the code is to show
  // the ball shadow and reflection.


  const int win_width = vh.get_width();
  const int win_height = vh.get_height();
  const float aspect = float(win_width) / win_height;
  const float n_dist = 0.01;
  const float xr = .8 * n_dist;

  transform.eye_from_global_set( modelview );
  transform.clip_from_eye_set
    ( pMatrix_Frustum(-xr,xr,-xr/aspect,xr/aspect,n_dist,5000) );
```

```
// pMatrix_Frustum: left, right, bottom, top, near, far

vh.clearValues[0].color =
  vk::ClearColorValue( array<float, 4>( { { 0, 0, 0, 0.5 } } ) );

/// Lighting
//
auto& light_all = uni_light_all->cgl_LightSource[0];

light_all.position = transform.eye_from_global * light_location;

light_all.constantAttenuation = 0.5;
light_all.linearAttenuation = 1.0;
light_all.quadraticAttenuation = 0;

pColor ambient_color(0x555555);

uni_light_all->cgl_LightModel.ambient = ambient_color;

light_all.diffuse = opt_light_intensity * color_white;
light_all.ambient = color_black;
light_all.specular = opt_light_intensity * color_white;

uni_light_reflected = uni_light_all.get_val();
uni_light_reflected->cgl_LightSource[0].position =
  transform.eye_from_global * pMatrix_Scale(1,-1,1) * light_location;

uni_light_ambient = uni_light_all.get_val();
auto& light_amb = uni_light_ambient->cgl_LightSource[0];
light_amb.diffuse = color_black;
light_amb.specular = color_black;

uni_light = uni_light_all.get_val();
uni_light->cgl_LightModel.ambient = color_black;

uni_light.to_dev();
uni_light_ambient.to_dev();
uni_light_all.to_dev();
uni_light_reflected.to_dev();

puni_light_curr = &uni_light_all;

buf_uni_common->tryout =
  ivec4(opt_tryout1,opt_tryout2,opt_normal_sphere,false);
buf_uni_common->tryoutf = vec4(opt_tryoutf, 0, 0, 0);
buf_uni_common->light_pos_g = light_location;
if ( do_raytrace )
  {
    raytrace.buf_uni_common->opt_tryout = buf_uni_common->tryout;
    raytrace.buf_uni_common->opt_tryoutf = buf_uni_common->tryoutf;
    raytrace.buf_uni_common->opt_shadows = opt_shadows;
    raytrace.buf_uni_common->opt_mirror = opt_mirror;
    raytrace.buf_light_set(uni_light_all);
  }
```

```
  const double time_now = time_wall_fp();
  const bool blink_visible = int64_t(time_now*3) & 1;
# define BLINK(txt,pad) ( blink_visible ? txt : pad )

  vh.fbprintf("%s\n",frame_timer.frame_rate_text_get());

  vh.fbprintf
    ("Raytrace: %s ('R')   Compiled: %s\n",
     !vh.have_raytrace ? "NA" :
     opt_want_raytrace ? ( raytrace.rt_ready ? "ON " : "???" ) : "OFF",
#ifdef __OPTIMIZE__
     "WITH OPTIMIZATION"
#else
     BLINK("WITHOUT OPTIMIZATION","")
#endif
     );

  vh.fbprintf
    ("Time Step: %8d  World Time: %11.6f  %s\n",
     time_step_count, world_time,
     opt_pause ? BLINK("PAUSED, 'p' to unpause, SPC or S-SPC to step.","") :
     "Press 'p' to pause."
     );

  vh.fbprintf
    ("Eye location: [%5.1f, %5.1f, %5.1f]  "
     "Eye direction: [%+.2f, %+.2f, %+.2f] ",
     eye_location.x, eye_location.y, eye_location.z,
     eye_direction.x, eye_direction.y, eye_direction.z);

  if ( balls )
    {
      Ball& ball = *balls[0];

      vh.fbprintf
        ("Head Ball Pos  [%5.1f,%5.1f,%5.1f] Vel [%+5.1f,%+5.1f,%+5.1f]\n",
         ball.position.x,ball.position.y,ball.position.z,
         ball.velocity.x,ball.velocity.y,ball.velocity.z );
    }

  vh.fbprintf
    ("Scene: %d ('0-7')  Config: %s ('c') "
     "Tryout 1: %s  ('y')  Tryout 2: %s  ('Y')\n",
     last_setup, opt_freeze_config ? BLINK("FROZEN","      ") : "RANDOM",
     opt_tryout1 ? BLINK("ON ","   ") : "OFF",
     opt_tryout2 ? BLINK("ON ","   ") : "OFF" );

#if 0
  vh.fbprintf
    ("Hide: %c%c%c ('!@#')  Effect: %c%c ('or')  Sphere: %s  ('z')  "
     "Tryout 1: %s  ('y')  Tryout 2: %s  ('Y')  Normals: %s ('n')\n",
     opt_hide_stuff & OH_Sphere ? 'S' : '_',
     opt_hide_stuff & OH_Links ? 'L' : '_',
```

```
      opt_hide_stuff & OH_Platform ? 'P' : '_',
      opt_shadows ? 'S' : '_',
      opt_mirror ? 'M' : '_',
      opt_sphere_true ? "TRUE" : "TRI" ,
      opt_tryout1 ? BLINK("ON ","   ") : "OFF",
      opt_tryout2 ? BLINK("ON ","   ") : "OFF",
      opt_normal_sphere ? "SPHERE" : "TRI");
#endif

  pVariable_Control_Elt* const cvar = variable_control.current;
  vh.fbprintf("VAR %s = %.5f  (TAB or SH-TAB to change, +/- to adjust)\n",
                      cvar->name,cvar->get_val());

  // Render Marker for Light Source
  //
  shapes.record_tetrahedron(cb,transform,light_location,0.15);

  //
  // Render ball reflection.  (Will be blended with dark tiles.)
  //

  const bool hide_platform = opt_hide_stuff & OH_Platform;

  if ( do_raytrace )
    {
      raytrace.tform_update(transform);
      render_objects(cb,RO_Normally);
      raytrace.buf_uni_common.to_dev();

      return;
    }

  buf_uni_common.to_dev();

  vk::ColorComponentFlags clr_all
    ( vk::ColorComponentFlagBits::eR | vk::ColorComponentFlagBits::eG |
      vk::ColorComponentFlagBits::eB | vk::ColorComponentFlagBits::eA );

  if ( !hide_platform )
    {
      // Save current state of fixed-functionality state.
      //
      ff_state.push();

      if ( opt_mirror )
        {
          /// Mirror Effect
          //
          //  - Step 1:  Write location of mirrors into stencil buffer.
          //
          //  - Step 2a: Prepare a matrix that transforms to mirrored location.
          //  - Step 2b: Set up blending to blend mirrored objects with mirrors.
          //
          //  - Step 3:  Render all objects except mirrors.
```

```
///   Step 1: Write location of mirrors into stencil buffer.
//
//    Write stencil value 2 at location of dark (mirrored) tiles.
//

vk::StencilOpState s_op_apply
  ( vk::StencilOp::eReplace, vk::StencilOp::eKeep,
    vk::StencilOp::eKeep, vk::CompareOp::eNever, 2, 2, 2 );
// sfail, spass, dfail, compare;  cmask, wmask, ref

ff_state.pipelineDepthStencilStateCreateInfo_get()
  .setStencilTestEnable(true)
  .setFront(s_op_apply).setBack(s_op_apply);

// Render mirrored tiles (but only to write stencil buffer).
//
pipe_platform_mir.record_draw(cb,bset_platform_mir);

cb.nextSubpass(vk::SubpassContents::eInline); vh.qs.subpass++;

// Prepare to write only stenciled locations.
//
vk::StencilOpState s_op_use
  ( vk::StencilOp::eKeep, vk::StencilOp::eKeep,
    vk::StencilOp::eKeep, vk::CompareOp::eEqual, 2, 2, 2 );
// sfail, spass, dfail, compare;  cmask, wmask, ref
//
// Stencil test passes if stencil value = 2.
// Don't change stencil buffer.

ff_state.pipelineDepthStencilStateCreateInfo_get()
  .setFront(s_op_use).setBack(s_op_use);

///   Step 2a: Prepare a matrix that transforms to mirrored location.
//    Use a transform that reflects objects to other side of platform.
//
const pMatrix prev = transform.eye_from_global;
const pMatrix_Scale reflect(1,-1,1);
transform.eye_from_global = prev * reflect;

// Reflected front face should still be treated as the front face.
//
ff_state.pipelineRasterizationStateCreateInfo_get()
  .setFrontFace(vk::FrontFace::eClockwise);

puni_light_curr = &uni_light_reflected;

///  Step 3:  Render all objects.

render_objects(cb,RO_Mirrored);

transform.eye_from_global = prev;
```

```
        ff_state.pipelineRasterizationStateCreateInfo_get()
          .setFrontFace(vk::FrontFace::eCounterClockwise);
      }

    // blend operation, blend factors, blend constants
    ff_state.pipelineColorBlendStateCreateInfo_get()
      .setBlendConstants( array<float,4>{.5,.5,.5,.5} );

    ff_state.pipelineColorBlendAttachmentState_get() =
      vk::PipelineColorBlendAttachmentState
      ( true,                    // blendEnable
        vk::BlendFactor::eConstantAlpha,  // srcColorBlendFactor
        vk::BlendFactor::eConstantAlpha,  // dstColorBlendFactor
        vk::BlendOp::eAdd,       // colorBlendOp
        vk::BlendFactor::eOne,   // srcAlphaBlendFactor
        vk::BlendFactor::eZero,  // dstAlphaBlendFactor
        vk::BlendOp::eAdd,       // alphaBlendOp
        clr_all      // colorWriteMask
        );

    puni_light_curr = &uni_light_all;;
    pipe_platform_mir.record_draw(cb,bset_platform_mir);

    ff_state.pop();
  }

if ( !opt_shadows )
  {
    // Render.
    //
    render_objects(cb,RO_Normally);
  }
else
  {
    //
    // First pass, render using only ambient light.
    //
    puni_light_curr = &uni_light_ambient;

    // Send balls, tiles, and platform to opengl.
    //
    render_objects(cb,RO_Normally);

    //
    // Second pass, find un-shadowed areas and add on light0.
    //


    /// Write Shadow Volumes
    //
    vk::ClearAttachment clr_att
      ( vk::ImageAspectFlagBits::eStencil, 0,
        vk::ClearDepthStencilValue(0,0) );
    vk::ClearRect clr_rect( vk::Rect2D({}, vh.s_extent), 0, 1 );
```

```
cb.clearAttachments( clr_att, clr_rect );

ff_state.push();

vk::StencilOpState s_op_apply_f
  ( vk::StencilOp::eKeep, vk::StencilOp::eIncrementAndWrap,
    vk::StencilOp::eKeep, vk::CompareOp::eAlways, -1, -1, 0 );
vk::StencilOpState s_op_apply_b
  ( vk::StencilOp::eKeep, vk::StencilOp::eDecrementAndWrap,
    vk::StencilOp::eKeep, vk::CompareOp::eAlways, -1, -1, 0 );
// sfail, spass, dfail, compare; c-mask, w-mask, ref
// ref is -1 (all 1's) because we want to operate on all bits in
// stencil value.

ff_state.pipelineDepthStencilStateCreateInfo_get()
  .setStencilTestEnable(true)
  .setDepthWriteEnable(false)
  .setFront(s_op_apply_f).setBack(s_op_apply_b);

if ( !opt_shadow_volumes )
  {
    ff_state.pipelineColorBlendAttachmentState_get()
      .setColorWriteMask( {} );
  }
else
  {
    ff_state.pipelineColorBlendAttachmentState_get() =
      vk::PipelineColorBlendAttachmentState
      ( true,                   // blendEnable
        vk::BlendFactor::eConstantAlpha,  // srcColorBlendFactor
        vk::BlendFactor::eConstantAlpha,  // dstColorBlendFactor
        vk::BlendOp::eAdd,       // colorBlendOp
        vk::BlendFactor::eOne,   // srcAlphaBlendFactor
        vk::BlendFactor::eZero,  // dstAlphaBlendFactor
        vk::BlendOp::eAdd,       // alphaBlendOp
        clr_all      // colorWriteMask
        );

    ff_state.pipelineColorBlendStateCreateInfo_get()
      .setBlendConstants( array<float,4>{.5,.5,.5,.5} );

    ff_state.pipelineDepthStencilStateCreateInfo_get()
      .setDepthWriteEnable(true);
  }

// Write stencil with shadow locations based on shadow volumes
// cast by light0 (light_location).  Shadowed locations will
// have a positive stencil value.
//
render_objects(cb,RO_Shadow_Volumes);

cb.nextSubpass(vk::SubpassContents::eInline); vh.qs.subpass++;

// Use stencil test to prevent writes to shadowed areas.
```

```
      //
      vk::StencilOpState s_op_apply_fb
        ( vk::StencilOp::eKeep, vk::StencilOp::eKeep,
          vk::StencilOp::eKeep, vk::CompareOp::eEqual, -1, -1, 0 );
      // sfail, spass, dfail, compare;  c-mask, w-mask, ref
      //
      // Stencil test passes if stencil value == 0.

      ff_state.pipelineDepthStencilStateCreateInfo_get()
        .setFront(s_op_apply_fb).setBack(s_op_apply_fb)
        .setDepthCompareOp(vk::CompareOp::eEqual);

      // Allow pixels to be re-written.
      //
      vk::ColorComponentFlags cmask_all
        ( vk::ColorComponentFlagBits::eR | vk::ColorComponentFlagBits::eG |
          vk::ColorComponentFlagBits::eB | vk::ColorComponentFlagBits::eA );

      ff_state.pipelineColorBlendAttachmentState_get() =
        vk::PipelineColorBlendAttachmentState
        ( true,                   // blendEnable
          vk::BlendFactor::eOne,  // srcColorBlendFactor
          vk::BlendFactor::eOne,  // dstColorBlendFactor
          vk::BlendOp::eAdd,       // colorBlendOp
          vk::BlendFactor::eOne,   // srcAlphaBlendFactor
          vk::BlendFactor::eOne,  // dstAlphaBlendFactor
          vk::BlendOp::eAdd,       // alphaBlendOp
          cmask_all      // colorWriteMask
          );

      puni_light_curr = &uni_light;

      // Render.
      //
      render_objects(cb,RO_Normally);

      ff_state.pop();
    }

  puni_light_curr = &uni_light_all;
}


void
World::cb_keyboard()
{
  const int key = vh.keyboard_key_get();
  if ( !key ) return;
  pVect adjustment(0,0,0);
  pVect user_rot_axis(0,0,0);
  const bool kb_mod_s = vh.keyboard_shift;
  const bool kb_mod_c = vh.keyboard_control;
  const float move_amt = kb_mod_s ? 2.0 : kb_mod_c ? 0.08 : 0.4;
```

```
  switch ( key ) {
  case FB_KEY_LEFT: adjustment.x = -move_amt; break;
  case FB_KEY_RIGHT: adjustment.x = move_amt; break;
  case FB_KEY_PAGE_UP: adjustment.y = move_amt; break;
  case FB_KEY_PAGE_DOWN: adjustment.y = -move_amt; break;
  case FB_KEY_DOWN: adjustment.z = move_amt; break;
  case FB_KEY_UP: adjustment.z = -move_amt; break;
  case FB_KEY_DELETE: user_rot_axis.y = 1; break;
  case FB_KEY_INSERT: user_rot_axis.y =  -1; break;
  case FB_KEY_HOME:
    if ( opt_move_item == MI_Light ) adjustment.x = 1; else user_rot_axis.x = 1;
    break;
  case FB_KEY_END: user_rot_axis.x = -1; break;
  case '0': case '1': case '2': ball_setup_hw2( key ); break;
  case '3': ball_setup_3(); break;
  case '4': ball_setup_4(); break;
  case '5': ball_setup_5(); break;
  case '6': ball_setup_6(); break;
  case '7': ball_setup_7(); break;
  case 'a':
    opt_physics_method++;
    if ( opt_physics_method == GP_ENUM_SIZE ) opt_physics_method = GP_cpu;
    break;
  case 'b': opt_move_item = MI_Ball; break;
  case 'B': opt_move_item = MI_Ball_V; break;

  case 'c': opt_freeze_config = !opt_freeze_config; break;

  case 'e': case 'E': opt_move_item = MI_Eye; break;
  case 'g': case 'G': opt_gravity = !opt_gravity; break;
  case 'h': case 'H': opt_head_lock = !opt_head_lock; break;
#if 0
  case '!': opt_hide_stuff ^= OH_Sphere; break;
  case '@': opt_hide_stuff ^= OH_Links; break;
  case '#': opt_hide_stuff ^= OH_Platform; break;
#endif
  case 't': case 'T': opt_tail_lock = !opt_tail_lock; break;
  case 'l': case 'L': opt_move_item = MI_Light; break;
  //  case 'r': opt_mirror = !opt_mirror; break;
  case 'R':
    opt_want_raytrace = !opt_want_raytrace;
    world_time_link_update = 0;
    break;
#if 0
  case 'o': opt_shadows = !opt_shadows; break;
  case 'O':
    opt_shadow_volumes = !opt_shadow_volumes;
    if ( opt_shadow_volumes ) opt_shadows = true;
    break;
  case 'n': case 'N': opt_normal_sphere = !opt_normal_sphere; break;
#endif
  case 'p': case 'P': opt_pause = !opt_pause; break;
  case 's': case 'S': balls_stop(); break;
#if 0
```

```
  case 'v': case 'V': opt_shader++;
    if ( opt_shader >= sizeof(sh_names)/sizeof(sh_names[0]) ) opt_shader = 0;
    break;
  case 'w': case 'W': balls_twirl(); break;
#endif
  case 'y': opt_tryout1 = !opt_tryout1; break;
  case 'Y': opt_tryout2 = !opt_tryout2; break;
  //  case 'z': opt_sphere_true = !opt_sphere_true; break;
  case ' ':
    if ( kb_mod_s && kb_mod_c ) opt_single_time_step = 10;
    else if ( kb_mod_s ) opt_single_time_step = 100;
    else if ( kb_mod_c ) opt_single_time_step = 1; else opt_single_frame = true;
    opt_pause = true;
    break;
  case FB_KEY_TAB:
    if ( !kb_mod_s ) { variable_control.switch_var_right(); break; }
  case 96: variable_control.switch_var_left(); break;
  case '-':case '_': variable_control.adjust_lower(); break;
  case '+':case '=': variable_control.adjust_higher(); break;
  default: printf("Unknown key, %d\n",key); break;
  }

  gravity_accel.y = opt_gravity ? -opt_gravity_accel : 0;

  opt_mirror_depth = raytrace.recursion_max_set( opt_mirror_depth + 1 ) - 1;
  lock_update();
  sphere.bunch_invalidate();

  // Update eye_direction based on keyboard command.
  //
  if ( user_rot_axis.x || user_rot_axis.y )
    {
      pMatrix_Rotation rotall(eye_direction,pVect(0,0,-1));
      user_rot_axis *= invert(rotall);
      eye_direction *= pMatrix_Rotation(user_rot_axis, M_PI * 0.03);
      modelview_update();
    }

  // Update eye_location based on keyboard command.
  //
  if ( adjustment.x || adjustment.y || adjustment.z )
    {
      const double angle =
        fabs(eye_direction.y) > 0.99
        ? 0 : atan2(eye_direction.x,-eye_direction.z);
      pMatrix_Rotation rotall(pVect(0,1,0),-angle);
      adjustment *= rotall;

      switch ( opt_move_item ){
      case MI_Ball:
        adj_vector = adjustment;
        if ( adj_t_stop == 0 )
          adj_t_prev = adj_t_stop = world_time;
        adj_t_stop += adj_duration;
```

```
        break;
      case MI_Ball_V: balls_push(adjustment,0); break;
      case MI_Light:
        if ( key == FB_KEY_HOME )
          {
            pVect l_to_b(light_location,balls[0]->position);
            adjustment = 0.25 * l_to_b;
          }
        light_location += adjustment;
        break;
      case MI_Eye: eye_location += adjustment; break;
      default: break;
      }
      modelview_update();
    }

}




void
World::init(int argc, char **argv)
{
  chain_length = 14;

  const bool all_vars = false;

  opt_time_step_duration = 0.00003;
  if ( all_vars )
    variable_control.insert(opt_time_step_duration,"Time Step Duration");

  distance_relaxed = 15.0 / chain_length;
  opt_spring_constant = 500;
  variable_control.insert(opt_spring_constant,"Spring Constant");

  opt_spring_damp = 0.7;
  if ( all_vars )
    variable_control.insert(opt_spring_damp,"Spring Damping").set_max(1);

  opt_friction_dynamic = 0.2;
  if ( all_vars )
    variable_control.insert(opt_friction_dynamic,"Dynamic Friction");

  opt_omega = 10;
  if ( all_vars )
    variable_control.insert(opt_omega,"Initial Rotation Rate");

  opt_gravity_accel = 9.8;
  opt_gravity = true;
  gravity_accel = pVect(0,-opt_gravity_accel,0);
  variable_control.insert(opt_gravity_accel,"Gravity");

  opt_air_resistance = 0.01;
```

```
variable_control.insert(opt_air_resistance,"Air Resistance");

world_time = 0;
time_step_count = 0;
last_frame_wall_time = time_wall_fp();
frame_timer.work_unit_set("Steps / s");

world_time_link_update = 0;

opt_mirror = true;
opt_mirror_depth = 3;
if ( all_vars )
  variable_control.insert(opt_mirror_depth,"Mirror Depth",1,0);
opt_shadows = true;
opt_shadow_volumes = false;
opt_shader = SO_Plain;

/// Init CUDA

// Get information about GPU and its ability to run CUDA.
//
int device_count;
cudaGetDeviceCount(&device_count); // Get number of GPUs.
if ( device_count == 0 )
  {
    fprintf(stderr,"No GPU found, exiting.\n");
    exit(1);
  }

// Print information about the available GPUs.
//
gpu_info_print();

// Use course library routine to choose a good GPU for CUDA.
//
int dev = gpu_choose_index();
CE(cudaSetDevice(dev));
printf("Using GPU %d\n",dev);

gpu_info.get_gpu_info(dev);
cuda_setup(&gpu_info);

// Print information about time_step routine.
//
printf("\nCUDA Routine Resource Usage:\n");

int block_size_max = 4096;

for ( int i=0; i<gpu_info.num_kernels; i++ )
  {
    printf("For %s:\n", gpu_info.ki[i].name);
    printf("  %6zd shared, %zd const, %zd loc, %d regs; "
           "%d max threads per block.\n",
           gpu_info.ki[i].cfa.sharedSizeBytes,
```

```
            gpu_info.ki[i].cfa.constSizeBytes,
            gpu_info.ki[i].cfa.localSizeBytes,
            gpu_info.ki[i].cfa.numRegs,
            gpu_info.ki[i].cfa.maxThreadsPerBlock);
      set_min(block_size_max,gpu_info.ki[i].cfa.maxThreadsPerBlock);
    }

  opt_physics_method = GP_cuda;
  gpu_const_data_stale = true;
  cpu_phys_data_stale = false;

  CE( cudaEventCreate(&frame_start_ce) );
  CE( cudaEventCreate(&frame_stop_ce) );

  opt_block_size = 32;

  if ( all_vars )
    variable_control.insert_power_of_2
      (opt_block_size, "Block Size", 32, block_size_max);

  c.alloc_n_balls = 0;
  c.alloc_n_links = 0;

  init_graphics();

  opt_sphere_slices = 35;
  sphere.opt_texture = true;
  sphere.init(opt_sphere_slices);
  if ( all_vars )
    variable_control.insert(opt_sphere_slices,"Sphere Slices");

  const char* const links_shader_code_path = "links-shdr-links.cc";

  const auto bvec_vs = vk::ShaderStageFlagBits::eVertex;
  const auto bvec_vgs =
    vk::ShaderStageFlagBits::eVertex | vk::ShaderStageFlagBits::eGeometry;

  pipe_set_2
    .init( ff_state )
    .ds_follow( transform )
    .ds_follow( puni_light_curr )
    .ds_uniform_use( "BIND_UNI_COMMON", buf_uni_common )
    .ds_set_material( color_dim_gray, color_salmon )
    .ds_use( sampler, texid_hw )
    .ds_storage_follow( "BIND_LINKS_POS1", lis_pos1, bvec_vs )
    .ds_storage_follow( "BIND_LINKS_POS2", lis_pos2, bvec_vs )
    .ds_storage_follow( "BIND_LINKS_V1", lis_v1, bvec_vs )
    .ds_storage_follow( "BIND_LINKS_V2", lis_v2, bvec_vs )
    .ds_storage_follow( "BIND_LINKS_B1_YDIR", lis_b1_ydir, bvec_vs )
    .ds_storage_follow( "BIND_LINKS_B2_YDIR", lis_b2_ydir, bvec_vs )
    .ds_storage_follow( "BIND_LINKS_TEX_RAD", lis_tex_rad, bvec_vgs )
    .shader_inputs_info_set( )
    .shader_code_set
    (links_shader_code_path,
```

```
    "vs_main_2(); ",     // Name of vertex shader main routine.
    "gs_main_2();",      // Name of geometry shader main routine.
    "fs_main();"         // Name of fragment shader main routine.
    )
  .topology_set( vk::PrimitiveTopology::eLineStrip )
  .create();

if ( vh.have_raytrace )
  comp_links
    .init( vh.qs )
    .ds_uniform_use( "BIND_UNI_COMMON", buf_uni_common )
    .ds_uniform_use( "BIND_TRANSFORM", raytrace.buf_uni_transform )
    .ds_storage_follow( "BIND_LINKS_POS1", lis_pos1 )
    .ds_storage_follow( "BIND_LINKS_POS2", lis_pos2 )
    .ds_storage_follow( "BIND_LINKS_V1", lis_v1 )
    .ds_storage_follow( "BIND_LINKS_V2", lis_v2 )
    .ds_storage_follow( "BIND_LINKS_B1_YDIR", lis_b1_ydir )
    .ds_storage_follow( "BIND_LINKS_B2_YDIR", lis_b2_ydir )
    .ds_storage_follow( "BIND_LINKS_TEX_RAD", lis_tex_rad )
    .ds_storage_follow( "BIND_LINKS_AS_IDX", buf_links_as_idx )
    .ds_storage_follow( "BIND_LINKS_TEX_COOR", buf_links_tex_coor )
    .ds_storage_follow( "BIND_LINKS_VERTEX_G", buf_links_vertex_g )
    .ds_storage_follow( "BIND_LINKS_NORMAL_G", buf_links_normal_g )
    .shader_code_set(links_shader_code_path, "comp_main(); ")
    .create();

pipe_set_2_sv
  .init( ff_state ).ds_use( uni_light )
  .ds_follow( transform )
  .ds_uniform_use( "BIND_UNI_COMMON", buf_uni_common )
  .ds_storage_follow( "BIND_LINKS_POS1", lis_pos1 )
  .ds_storage_follow( "BIND_LINKS_POS2", lis_pos2 )
  .ds_storage_follow( "BIND_LINKS_V1", lis_v1 )
  .ds_storage_follow( "BIND_LINKS_V2", lis_v2 )
  .ds_storage_follow( "BIND_LINKS_B1_YDIR", lis_b1_ydir )
  .ds_storage_follow( "BIND_LINKS_B2_YDIR", lis_b2_ydir )
  .ds_storage_follow( "BIND_LINKS_TEX_RAD", lis_tex_rad )
  .shader_inputs_info_set( )
  .shader_code_set
  (links_shader_code_path,
    "vs_main_2_sv(); ",     // Name of vertex shader main routine.
    "gs_main_2_sv();",      // Name of geometry shader main routine.
    "fs_main_sv();"         // Name of fragment shader main routine.
    )
  .topology_set( vk::PrimitiveTopology::eLineStrip )
  .create();

ball_setup_hw2('1');
return;
// For benchmarking.
opt_pause = true;
opt_hide_stuff = OH_Links | OH_Platform;
opt_shadows = false;
opt_mirror = false;
```

```
}

Ball*
World::make_marker(pCoor position, pColor color, float radius )
{
  Ball* const ball = new Ball;
  ball->position = position;
  ball->locked = true;
  ball->velocity = pVect(0,0,0);
  ball->radius = radius;
  ball->mass = -1;
  ball->contact = false;
  ball->color = color;
  balls_marker.push_back(ball);
  return ball;
}

void
World::lock_update()
{
  // This routine called when options like opt_head_lock might have
  // changed.

  // Update locked status.
  //
  if ( head_ball ) head_ball->locked = opt_head_lock;
  if ( tail_ball ) tail_ball->locked = opt_tail_lock;

  // Re-compute minimum mass needed for stability.
  //
  for ( Ball *ball: balls ) ball->spring_constant_sum = 0;
  const double dtis = pow( opt_time_step_duration, 2 );
  for ( Link *link: links )
    {
      Ball* const b1 = link->ball1;
      Ball* const b2 = link->ball2;
      b1->spring_constant_sum += opt_spring_constant;
      b2->spring_constant_sum += opt_spring_constant;
    }
  for ( Ball *ball: balls )
    {
      ball->mass_min = ball->spring_constant_sum * dtis;
      ball->constants_update();
    }
  gpu_const_data_stale = true;
}

void
World::balls_twirl()
{
  if ( !head_ball || !tail_ball ) return;

  pNorm axis(head_ball->position, tail_ball->position);
  const float omega = 0.1 * opt_omega;
```

```
  for ( Ball *ball: balls )
    {
      pVect b_to_top(ball->position,head_ball->position);
      const float dist_along_axis = dot(b_to_top,axis);
      const float lsq = b_to_top.mag_sq() - dist_along_axis * dist_along_axis;
      ball->omega += omega * axis;
      if ( lsq <= 1e-5 ) continue;
      const float dist_to_axis = sqrt(lsq);
      pNorm rot_dir = cross(b_to_top,axis);
      ball->velocity += omega * dist_to_axis * rot_dir;
    }
}

void
World::objects_erase()
{
  balls.erase();
  links.erase();
  surfaces.clear();
}

Links
World::link_new
(Ball *ball1, Ball *ball2, float stiffness, float spring_constant_p)
{
  Link_Args largs
    { .spring_constant = spring_constant_p, .link_stiffness = stiffness };
  return link_new(ball1,ball2,largs);
}

Links
World::link_new
(Ball *ball1, Ball *ball2, Link_Args largs )
{
  const float spring_constant_p = largs.spring_constant;
  const float stiffness = largs.link_stiffness;
  Links links_rv;
  assert( ball1->radius > 0 );
  assert( ball2->radius > 0 );
  const float spring_constant = spring_constant_p ?: opt_spring_constant;

  Link* const rlink = new Link(ball1,ball2);
  rlink->radius = largs.radius;
  links_rv += rlink;

  pNorm n12(ball1->position,ball2->position);
  const float rad_sum = ball1->radius + ball2->radius;
  pMatrix3x3 b1rot = ball1->omatrix;
  pMatrix3x3 b2rot = ball2->omatrix;
  pCoor ctr = ball1->position
    + ( ball1->radius + 0.5 * ( n12.magnitude - rad_sum ) ) * n12;

  pVect con_x = rlink->con_x;
```

```
  pVect con_y = rlink->con_y;

  const float lrad = stiffness * ball1->radius;

  const int n_springs = 4;
  const double delta_theta = 2 * M_PI / n_springs;

  for ( int i=0; i<n_springs; i++ )
    {
      const double theta = i * delta_theta;
      pVect convec = lrad * ( cos(theta) * con_x + sin(theta) * con_y );
      pCoor con = ctr + convec;
      pVect cb1 = mult_MTV( b1rot, con - ball1->position );
      pVect cb2 = mult_MTV( b2rot, con - ball2->position );
      links_rv +=
        new Link(rlink, cb1, cb2,
                 { .distance_relaxed=0, .spring_constant=spring_constant } );
    }
  return links_rv;
}

///
/// CUDA Code
///

float4 tof4(pCoor c)
{
  float4 f4;
  f4.x = c.x;
  f4.y = c.y;
  f4.z = c.z;
  f4.w = c.w;
  return f4;
}

float4 tof4(pVect c)
{
  float4 f4;
  f4.x = c.x;
  f4.y = c.y;
  f4.z = c.z;
  f4.w = 0;
  return f4;
}

pCoor topc(float4 c)
{
  pCoor f4;
  f4.x = c.x;
  f4.y = c.y;
  f4.z = c.z;
  f4.w = c.w;
  return f4;
}
```

```
void
World::data_cpu_to_gpu_constants()
{
  c.platform_xmin = platform_xmin;
  c.platform_xmax = platform_xmax;
  c.platform_zmax = platform_zmax;
  c.platform_zmin = platform_zmin;

  c.n_balls = balls.size();
  c.n_links = links.size();
  c.n_surfaces = surfaces.size();
  c.opt_tryout1 = opt_tryout1;
  c.opt_tryout2 = opt_tryout2;
  c.opt_tryoutf = opt_tryoutf;
  c.opt_spring_constant = opt_spring_constant;
  c.opt_spring_damp = opt_spring_damp;
  c.opt_air_resistance = opt_air_resistance;
  c.opt_friction_dynamic = opt_friction_dynamic;
  c.opt_head_lock = opt_head_lock;
  c.opt_tail_lock = opt_tail_lock;
  c.gravity_accel = gravity_accel;

  data_cpu_to_gpu_common(&c);
}


void
World::data_cpu_to_gpu_dynamic()
{
  const int n_balls = balls.size();
  const int n_links = links.size();
  const int n_surfaces = surfaces.size();
  const bool need_alloc =
    n_balls > c.alloc_n_balls || n_links > c.alloc_n_links
    || n_surfaces > c.alloc_n_surfaces;
  const bool need_init = c.alloc_n_balls == 0;

#define ALLOC_MOVE_ZERO(n_balls,balls,memb,do_move,do_zero)                \
  {                                                                        \
    const int size_elt_bytes = sizeof(c.balls.memb[0]);                    \
    const int size_bytes = n_balls * size_elt_bytes;                       \
    if ( need_alloc )                                                      \
      {                                                                    \
        if ( !need_init )                                                  \
          {                                                                \
            CE( cudaFree( c.balls.memb ) );                                \
            free(c.h_##balls.memb);                                        \
          }                                                                \
        CE( cudaMalloc( &c.balls.memb, size_bytes ) );                     \
        c.h_##balls.memb = (decltype(c.balls.memb)) malloc( size_bytes );  \
      }                                                                    \
    if ( do_move )                                                         \
      {                                                                    \
```

```
        for ( int i=0; i<n_balls; i++ )                                    \
          c.h_##balls.memb[i] = balls[i]->memb;                            \
        CE( cudaMemcpy                                                     \
            ( c.balls.memb, c.h_##balls.memb,                              \
              size_bytes, cudaMemcpyDefault ) );                           \
      }                                                                    \
    else if ( do_zero )                                                    \
      {                                                                    \
        CE( cudaMemset( c.balls.memb, 0, size_bytes ) );                   \
      }                                                                    \
  }

#define MOVEc(num,ptr,memb) \
  ALLOC_MOVE_ZERO(num,ptr,memb,gpu_const_data_stale,0)
#define MOVE(num,ptr,memb) ALLOC_MOVE_ZERO(num,ptr,memb,1,0)
#define ALLOC(num,ptr,memb) ALLOC_MOVE_ZERO(num,ptr,memb,0,0)
#define ZERO(num,ptr,memb) ALLOC_MOVE_ZERO(num,ptr,memb,0,1)
  MOVEc(n_balls,balls,position);
  MOVEc(n_balls,balls,velocity);
  ZERO(n_balls,balls,force);

  MOVEc(n_balls,balls,orientation);
  MOVEc(n_balls,balls,omatrix);
  MOVEc(n_balls,balls,omega);
  ZERO(n_balls,balls,torque);
  MOVEc(n_balls,balls,m_r_fdt);
  MOVEc(n_balls,balls,locked);

  MOVEc(n_links,links,ball_idx);
  MOVEc(n_links,links,cb1);
  MOVEc(n_links,links,cb2);
  MOVEc(n_links,links,dr_sc);
  MOVEc(n_links,links,is_simulatable);
  MOVEc(n_links,links,is_surface_connection);

  MOVEc(n_surfaces,surfaces,b_idx);

#undef MOVE
#undef MOVEc
#undef ALLOC
#undef ZERO
#undef ALLOC_MOVE
  if ( need_alloc )
    {
      c.alloc_n_balls = n_balls;
      c.alloc_n_links = n_links;
      c.alloc_n_surfaces = n_surfaces;
      data_cpu_to_gpu_common(&c);
    }
  gpu_const_data_stale = false;
}

void
World::data_gpu_to_cpu_dynamic()
```

```
{
  const int n_balls = balls.size();

#define MOVE(n_balls,balls,memb)                                          \
  {                                                                       \
    const int size_elt_fr_bytes = sizeof(c.balls.memb[0]);                \
    const int size_bytes = n_balls * size_elt_fr_bytes;                   \
    CE( cudaMemcpy                                                        \
        ( c.h_##balls.memb, c.balls.memb, size_bytes,                     \
          cudaMemcpyDeviceToHost ) );                                     \
    for ( int i=0; i<n_balls; i++ )                                       \
      balls[i]->memb = c.h_##balls.memb[i];                               \
  }

  MOVE(n_balls,balls,position);
  MOVE(n_balls,balls,velocity);
  MOVE(n_balls,balls,orientation);
  MOVE(n_balls,balls,omatrix);
  MOVE(n_balls,balls,omega);

#undef MOVE

}


///
/// Physical Simulation Code
///

 /// Initialize Simulation
//

void
World::ball_setup_prob_0()
{
  /// Homework 2
  //
  //  Use this routine for experimentation.
  //  Put the solution to Problem 1 in World::ball_setup_prob_1() and
  //  put the solution to Problem 2 in World::ball_setup_prob_2().

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

  // A point on the axis used to construct the hyperboloid.
  //
  const pCoor center_pos{h.center_pos};
  //
  // This will also be the center of the circle that forms the base
  // of the hyperboloid.

  // 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};
//
// The balls along this line are yellow (in the unmodified code).

// The number of balls in one "line".
//
const int n_balls = h.n_balls;
//
// The yellow balls form one line.

// The number of lines.
//
const int n_lines = h.n_lines;

const float ball_radius{h.ball_radius};


// Drop a marker ball at the center position.
//
make_marker( center_pos, color_green, ball_radius*5 );
//
// Feel free to insert markers at additional points to help debug.

// Settings for the link.  No need to modify this.
Link_Args largs{
  .spring_constant = opt_spring_constant / 10,
  .link_stiffness = 50,
  .radius = ball_radius * 0.7f
};

// Container to hold pointers to balls. Organized to facilitate putting
// links between balls.
//
vector<vector<Ball*>> balls_lines;

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

for ( int j=0; j<n_lines; j++ )
  {
    // Construct a container to hold the current line.
    //
    vector<Ball*>& balls_line = balls_lines.emplace_back();

    // Compute the endpoints of the current line to draw.
    // Needs to be modified for Problem 1 and maybe Problem 2.
    //
    pCoor line_bot_pos = line_0_bot_pos + j * grid_pitch;
    pCoor line_top_pos = line_0_top_pos + j * grid_pitch;
    //
    // In the first iteration these should match line_0_top_pos,
    // line_0_bot_pos.
```

```
    // Note: For Problem 1, there is no need to modify the code below,
    // in particular in the "i" loop and the code that follows.

    // Compute a vector that advances along a line, from bottom to top.
    //
    pVect line_pitch = (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++ )
      {
        // Compute the position along the line between line_bot_pos
        // and line_top_pos.
        //
        pCoor pos = line_bot_pos + i * line_pitch;

        // Create a new ball, and add it to "our" list, balls_line,
        // and the simulators list, balls.
        //
        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.
        //
        // Since ball is a pointer it is still possible to set
        // members of ball, such as ball->color, after ball was
        // added to the lists.

        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;
      }
  }

// Put links between balls in adjacent lines.
//
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++ )
      {
        // Add a link along line j. (From bottom to top.)
        //
        if ( i ) links += link_new( bl0[i-1], bl0[i], largs );

        // Add a link from line j to line j+1.
        //
        links += link_new( bl0[i], bl1[i], largs );
      }
  }
```

```
  // Choose a head and tail balls. These can be locked in place.
  //
  tail_ball = balls_lines[0].front();
  head_ball = balls_lines[0].back();
  opt_head_lock = false;  opt_tail_lock = false;
}

void
World::ball_setup_prob_1()
{
  /// Homework 2
  //
  //  Put solution to Problem 1 in this routine.

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

  // A point on the axis used to construct the hyperboloid.
  //
  const pCoor center_pos{h.center_pos};
  //
  // This will also be the center of the circle that forms the base
  // of the hyperboloid.

  // 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};
  //
  // The balls along this line are yellow (in the unmodified code).

  // The number of balls in one "line".
  //
  const int n_balls = h.n_balls;
  //
  // The yellow balls form one line.

  // The number of lines.
  //
  const int n_lines = h.n_lines;

  const float ball_radius{h.ball_radius};


  // Drop a marker ball at the center position.
  //
  make_marker( center_pos, color_green, ball_radius*5 );
  //
  // Feel free to insert markers at additional points to help debug.

  // Settings for the link.  No need to modify this.
  Link_Args largs{
```

```
    .spring_constant = opt_spring_constant / 10,
    .link_stiffness = 50,
    .radius = ball_radius * 0.7f
  };

  // Container to hold pointers to balls. Organized to facilitate putting
  // links between balls.
  //
  vector<vector<Ball*>> balls_lines;

  /// SOLUTION -- Problem 1
  //
  //  Solution Plan:
  //
  //    - Find axes of bottom circle.
  //    - Find center of top circle.
  //    - Find axes of top circle. (Instead could find twist angle.)
  //
  /// Find Axes for the Bottom Circle
  //
  //  The x axis is directly constructed using center_pos and line_0_top_pos.
  pVect vx_bot( center_pos, line_0_bot_pos );
  //
  //  The y axis is found using the just-computed vx_bot and the given axis.
  pVect vy_bot = cross(axis,vx_bot);
  //
  /// Find Center of Top Circle
  //
  //  Compute the coordinates of the center of the top circle.
  //  The top circle is in the plane defined by point line_0_top_pos
  //  and the normal in variable axis.
  //
  //  Line from bottom plane (and circle) to top plane (and circle).
  pVect v0tb( line_0_bot_pos, line_0_top_pos );
  //
  //  The dot product gives the distance from one plane to the other.
  pVect vht = dot(v0tb,axis) * axis;
  //
  //  Use the distance (vht) to find the center of the top circle.
  pCoor ctr_top_pos = center_pos + vht;
  //
  /// Find Axes of Top Circle
  //
  //  The bottom circle and the top circle are in parallel planes and
  //  so vx_bot and vy_bot could be used for both. If vx_bot,vy_bot
  //  were used for the upper circle we'd need to compute a "twist"
  //  angle. The problem with using a twist angle is that we'd need to
  //  compute a sine and cosine twice per iteration, once for theta
  //  and once for theta+twist_angle. Rather than do that a second set
  //  of axes will be computed so that the same angle could be used
  //  for both.
  //
  pVect vx_top( ctr_top_pos, line_0_top_pos );
  pVect vy_top = cross(axis,vx_top);
```

```
  //
  //  Place a marker at the center of the top circle.
  make_marker( ctr_top_pos, color_blue, ball_radius * 5 );
  //
  const float delta_theta = 2 * M_PI / n_lines;

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

  for ( int j=0; j<n_lines; j++ )
    {
      // Construct a container to hold the current line.
      //
      vector<Ball*>& balls_line = balls_lines.emplace_back();

      // Compute the endpoints of the current line to draw.
      // Needs to be modified for Problem 1 and maybe Problem 2.
      //
#if 0
      // Original code (before solution)
      pCoor line_bot_pos = line_0_bot_pos + j * grid_pitch;
      pCoor line_top_pos = line_0_top_pos + j * grid_pitch;
#endif
      //
      // In the first iteration these should match line_0_top_pos,
      // line_0_bot_pos.

      /// SOLUTION -- Problem 1
      //
      //  Use the upper and lower circle axes to find the top and bottom
      //  endpoints of the line rotated by theta.
      //
      const float theta = j * delta_theta;
      pCoor line_bot_pos =
        center_pos + vx_bot * cosf(theta) + vy_bot * sinf(theta);
      pCoor line_top_pos =
        ctr_top_pos + vx_top * cosf(theta) + vy_top * sinf(theta);

      // Note: For Problem 1, there is no need to modify the code below,
      // in particular in the "i" loop and the code that follows.

      // Compute a vector that advances along a line, from bottom to top.
      //
      pVect line_pitch = (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++ )
        {
          // Compute the position along the line between line_bot_pos
          // and line_top_pos.
          //
          pCoor pos = line_bot_pos + i * line_pitch;
```

```
          // Create a new ball, and add it to "our" list, balls_line,
          // and the simulators list, balls.
          //
          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.
          //
          // Since ball is a pointer it is still possible to set
          // members of ball, such as ball->color, after ball was
          // added to the lists.

          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;
        }
    }

  // Put links between balls in adjacent lines.
  //
  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++ )
        {
          // Add a link along line j. (From bottom to top.)
          //
          if ( i ) links += link_new( bl0[i-1], bl0[i], largs );

          // Add a link from line j to line j+1.
          //
          links += link_new( bl0[i], bl1[i], largs );
        }
    }

  // Choose a head and tail balls. These can be locked in place.
  //
  tail_ball = balls_lines[0].front();
  head_ball = balls_lines[0].back();
  opt_head_lock = false;  opt_tail_lock = false;
}

void
World::ball_setup_prob_2()
{
  /// Homework 2
  //
  //  Put solution to Problem 2 in this routine.

  // The axis used to construct the hyperbolid.
```

```
//
const pNorm axis{h.axis};

// A point on the axis used to construct the hyperboloid.
//
const pCoor center_pos{h.center_pos};
//
// This will also be the center of the circle that forms the base
// of the hyperboloid.

// 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};
//
// The balls along this line are yellow (in the unmodified code).

// The number of balls in one "line".
//
const int n_balls = h.n_balls;
//
// The yellow balls form one line.

// The number of lines.
//
const int n_lines = h.n_lines;

const float ball_radius{h.ball_radius};


// Drop a marker ball at the center position.
//
make_marker( center_pos, color_green, ball_radius*5 );
//
// Feel free to insert markers at additional points to help debug.

// Settings for the link.  No need to modify this.
Link_Args largs{
  .spring_constant = opt_spring_constant / 10,
  .link_stiffness = 50,
  .radius = ball_radius * 0.7f
};

// Container to hold pointers to balls. Organized to facilitate putting
// links between balls.
//
vector<vector<Ball*>> balls_lines;

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

// Compute a vector that advances along a line, from bottom to top.
//
```

```
pVect line_pitch = (1.0/(n_balls-1)) * ( line_0_top_pos - line_0_bot_pos );

/// SOLUTION -- Problem 2
//
//  Define out the sample transformations.
#if 0
// Translate so that coordinate P is moved to P + grid_pitch.
pMatrix_Translate transl( grid_pitch );

// The following are examples of transformation matrices that may or
// may not be useful.
//
// Rotate pi/2 radians around the x axis.
pMatrix_Rotation rot( pVect(1,0,0), M_PI / 2 );
//
pNorm ax(1,2,3), ay(4,5,6), az(7,8,9);
// Rotate so that direction ax becomes pVect(1,0,0) (x-axis), etc.
pMatrix_Rows rot_to_local( ax, ay, az );
//
// Rotate so that direction pVect(1,0,0) becomes ax.
pMatrix_Cols rot_from_local( ax, ay, az );
//
// Examples of pMatrix constructors shown above.
#endif

/// SOLUTION -- Problem 2
//
// Translate coordinate space so center_pos is origin.
//
pMatrix_Translate to_ctr( -center_pos );
//
// Rotate around axis.
//
const float delta_theta = 2 * M_PI / n_lines;
pMatrix_Rotation rot( axis, delta_theta );
//
// Translate back so that center_pos is where it was.
//
pMatrix_Translate from_ctr( center_pos );
//
// Combine the three to get one transformation matrix.
//
pMatrix transform = from_ctr * rot * to_ctr;

for ( int j=0; j<n_lines; j++ )
  {
    // Construct a container to hold the current line.
    //
    vector<Ball*>& balls_line = balls_lines.emplace_back();

    // Construct a line consisting of n_balls balls.
    //
    for ( int i=0; i<n_balls; i++ )
      {
```

```
          pCoor pos;

          // Compute the position along the line between line_bot_pos
          // and line_top_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
            {
              // Use find the position on ball i on this line using a
              // transformation of the position of ball i one the
              // previous line.
              //
              pCoor pos_prev = balls_lines[j-1][i]->position;
              pos = transform * pos_prev;
            }

          // Create a new ball, and add it to "our" list, balls_line,
          // and the simulators list, balls.
          //
          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.
          //
          // Since ball is a pointer it is still possible to set
          // members of ball, such as ball->color, after ball was
          // added to the lists.

          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;
        }
    }

  // Put links between balls in adjacent lines.
  //
  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++ )
        {
          // Add a link along line j. (From bottom to top.)
          //
          if ( i ) links += link_new( bl0[i-1], bl0[i], largs );
```

```
          // Add a link from line j to line j+1.
          //
          links += link_new( bl0[i], bl1[i], largs );
        }
    }

  // Choose a head and tail balls. These can be locked in place.
  //
  tail_ball = balls_lines[0].front();
  head_ball = balls_lines[0].back();
  opt_head_lock = false;  opt_tail_lock = false;
}



void
World::ball_setup_1(bool big)
{
  // Arrange balls to form sort of tree.

  last_setup = 1;

  double ball_rad = 1;  // Warning: Re-assigned, changes behavior of nb.

  pCoor first_pos(17.2,ball_rad,-20.2);

  pVect dir_up(0.1,3,0.1);
  pVect dir_z(3*distance_relaxed,0,0);
  pVect dir_x(0,0,3*distance_relaxed);

  // Remove objects from the simulated objects lists, balls and links.
  // The delete operator is used on objects in the lists.
  //
  objects_erase();

  auto nb =
    [&] (pCoor pos) {
      Ball* const ball = new Ball;
      ball->position = pos;
      ball->locked = false;
      ball->velocity = pVect(0,0,0);
      ball->radius = ball_rad;
      ball->contact = false;
      balls += ball;
      return ball;
      };

  tail_ball = nb(first_pos);
  tail_ball->specularity = 0.5;

  pCoor last_pos = tail_ball->position;

  const int n_tiers = big ? 32 : 8;
  const double shrink_final = 0.17;
```

```
  const double shrink = pow(shrink_final,1.0/n_tiers);

  Balls tail_balls;
  for ( int i=1; i<n_tiers; i++ )
    {
      ball_rad *= shrink;
      Ball* const ball = nb(last_pos + (i==1?1.0:ball_rad) * dir_up);
      ball->specularity = 0.5;
      tail_balls += ball;
      links += link_new(balls[balls-2],ball,0.8);
      last_pos = ball->position;
    }

  head_ball = balls[balls-1];

  pNorm dir_zn(dir_z);
  pNorm dir_xn(dir_x);
  for ( Ball *ball: tail_balls )
    {
      if ( ball == head_ball ) break;
      pNorm to_top(ball->position,head_ball->position);
      const float r = to_top.magnitude;
      int num_hairs = max(4,int(7*r));
      ball_rad = ball->radius / 4;
      const float delta_theta = 2 * M_PI / num_hairs;
      for ( int i=0; i<num_hairs; i++ )
        {
          const float theta = i * delta_theta;
          pCoor pos = ball->position +
            r * cos(theta) * dir_xn + r * sin(theta) * dir_zn;
          Ball* const ballr = nb(pos);
          links += link_new(ballr,ball);
        }
    }

  opt_head_lock = false;
  opt_tail_lock = true;
  setup_at_end();
}

void
World::ball_setup_2()
{
  // Arrange balls to form a suspended thing.

  last_setup = 2;

  pCoor first_pos(17.2,14.8f,-20.2);

  pVect dir_dn(0.5,first_pos.y/8,0.5);
  pVect dir_z(3*distance_relaxed,0,0);
  pVect dir_x(0,0,3*distance_relaxed);

  // Remove objects from the simulated objects lists, balls and links.
```

```
// The delete operator is used on objects in the lists.
//
objects_erase();

float ball_rad = 0.3;  // Warning: Re-assigned, changes behavior of nb.

auto nb =
  [&] (pCoor pos) {
    Ball* const ball = new Ball;
    ball->position = pos;
    ball->locked = false;
    ball->velocity = pVect(0,0,0);
    ball->radius = ball_rad;
    ball->contact = false;
    ball->specularity = 0.5;
    balls += ball;
    return ball;
    };

head_ball = nb(first_pos);
for ( int j: {-1,1} )
  for ( int i: {-1,0,1} )
    {
      Ball* const ball = nb(first_pos - dir_dn + i * dir_x + j * dir_z );
      if ( i != 0 ) links += link_new(head_ball,ball);
      if ( i != -1 ) links += link_new(balls[balls-2],ball);
      if ( j == 1 && i != 0 )  links += link_new(balls[balls-4],ball);
    }

Ball* const tail_start = nb(first_pos - dir_dn);
links += link_new(balls[balls-6],tail_start);
links += link_new(balls[balls-3],tail_start);
pCoor last_pos = tail_start->position;
Balls tail_balls;
for ( int i=1; i<5; i++ )
  {
    Ball* const ball = nb(last_pos - 0.5 * dir_dn);
    tail_balls += ball;
    links += link_new(balls[balls-2],ball);
    last_pos = ball->position;
  }

tail_ball = balls[balls-1];

pNorm dir_zn(dir_z);
pNorm dir_xn(dir_x);
int num_hairs = 20;
ball_rad = 0.1;
for ( Ball *ball: tail_balls )
  {
    const float delta_theta = 2 * M_PI / num_hairs;
    for ( int i=0; i<num_hairs; i++ )
      {
        const float theta = i * delta_theta;
```

```
          const float r = ball->radius * 10;
          pCoor pos = ball->position +
            r * cos(theta) * dir_xn + r * sin(theta) * dir_zn;
          Ball* const ballr = nb(pos);
          links += link_new(ballr,ball);
        }
    }

  opt_head_lock = true;    // Head ball will be frozen in space.
  opt_tail_lock = false;   // Tail ball can move freely.
  setup_at_end();
}

void
World::ball_setup_3()
{
  // Arrange links to form what I would eventually learn to be a
  // corona virus.

  const bool sarcov = true;

  last_setup = 3;
  objects_erase();

  pCoor first_pos(17.2,14.8f,-10.2);

  float ball_radius = 1;

  auto nb =
    [&] (pCoor pos) {
      Ball* const ball = new Ball;
      ball->position = pos;
      ball->locked = false;
      ball->velocity = pVect(0,0,0);
      ball->radius = ball_radius;
      ball->contact = false;
      balls += ball;
      return ball;
      };

  head_ball = nb(first_pos);
  head_ball->specularity = 0.5;
  ball_radius = ball_radius * ( sarcov ? 0.08 : .1 );
  const int n_spikes = sarcov ? 100 : 500;

  for ( int i=0; i<n_spikes; i++ )
    {
      pNorm dir(0.5-drand48(),0.5-drand48(),0.5-drand48());
      const float len = head_ball->radius * ( sarcov ? 1.5 : 3.0 );
      Ball* const ball = nb(first_pos + len * dir);
      links += link_new(ball,head_ball,2);
    }

  tail_ball = balls[1];
```

```
  opt_head_lock = false;    // Head ball will be frozen in space.
  opt_tail_lock = false;   // Tail ball can move freely.

  setup_at_end();
}

void
World::ball_setup_4()
{
  // Render a wheel (disk of balls with one in the center).

  last_setup = 4;
  objects_erase();

  pCoor center_pos = pCoor(17.2,14.8f,-10.2) + 2*vec_rand();

  const int n_balls = 4 * ( 4 + random()%4 );
  float ring_outer_radius = 5 + 2 * drand48();
  float ball_radius = 0.2 * 2 * M_PI * ring_outer_radius / n_balls;

  pNorm c_to_eye(center_pos,eye_location);
  pNorm az = c_to_eye + 0.5 * vec_rand() + pVect(0,0.5,0);

  pNorm ax = fabs(az.x) < fabs(az.y)
    ? pVect(0,az.z,-az.y) : pVect(az.z,0,-az.x);
  pVect ay(az,ax);

  // Default settings for balls.
  //
  Ball ball_default;
  ball_default.specularity = 0.5;
  ball_default.velocity = pVect(0,0,0);
  ball_default.radius = ball_radius;
  ball_default.color = color_light_slate_gray;

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

  const float d_theta = 2 * M_PI / n_balls;

  // Place balls in a circle.
  //
  for ( int i=0; i<n_balls; i++ )
    {
      const float theta = d_theta * i;
      pCoor pos = center_pos + vx * cosf(theta) + vy * sinf(theta);

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

```
      // Set color of first ball to blue, others to white.
      ball->color = i == 0 ? color_blue : color_white;
    }

  const float link_stiffness = 10;

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

  // Place a ball in the center and connect it to all the other balls.
  if ( true )
    {
      Ball* const center_ball =
        balls.emplace_back( new Ball(center_pos,ball_default) );

      for ( int i=0; i<n_balls; i++ )
        links += link_new( center_ball, balls[i], 2 );
    }

  pNorm spin_axis = az;
  pCoor c_of_mass = center_pos;

  for ( Ball* ball: balls )
    {
      pVect cm_to_ball( c_of_mass, ball->position );

      ball->velocity = opt_omega * cross( spin_axis, cm_to_ball );
      ball->omega = opt_omega * spin_axis;
    }

  tail_ball = balls.back();
  head_ball = balls.front();

  opt_head_lock = false; // If true, head ball locked in place.
  opt_tail_lock = false;

  setup_at_end();
}

void
World::ball_setup_5()
{
  // Construct a top. (Top, as in a child's toy.)

  last_setup = 5;
  objects_erase();

  pCoor center_pos(17.2,14.8f,-10.2);

  const int n_balls = 20;
  const float crad = 5;
```

```
const float ball_radius = 0.3 * 2 * M_PI * crad / n_balls;

// Default settings for balls.
//
Ball ball_default;
ball_default.specularity = 0.5;
ball_default.velocity = pVect(0,0,0);
ball_default.radius = ball_radius;
ball_default.color = pColor(0xffbbff); // Plum 1. That's the color's name.

const float d_theta = 2 * M_PI / n_balls;

pVect vz = pVect(1,6,1) - 2*vec_rand();
pNorm ax( vz.y, -vz.x, 0 );
pNorm ay = cross(vz,ax);
pVect vx = crad * ax;
pVect vy = crad * ay;

pNorm spin_axis = vz;
float omega = opt_omega;
opt_spring_damp = .4;

// Construct Top's Main Ring
//
for ( int i=0; i<n_balls; i++ )
  {
    const float theta = d_theta * i;
    pCoor pos = center_pos + vx * cosf(theta) + vy * sinf(theta);
    Ball* ball = balls.emplace_back( new Ball(pos,ball_default) );

    // Find the coordinate on the spin axis closest to the new ball.
    //
    pCoor axis_cp =
      center_pos + spin_axis * dot( spin_axis, pVect(center_pos,pos) );
    //
    // Use this coordinate to compute a vector giving the ball's
    // motion when the top is spinning around axis spin_axis.
    //
    pVect cp_to_p(axis_cp,pos);
    ball->velocity = omega * cross(spin_axis,cp_to_p);
  }

// Make one of the balls on the top's main ring a different color.
balls[0]->color = color_blue;

// Connect the balls on the main ring with links.
//
for ( int i=0; i<n_balls; i++ )
  links += link_new( balls[i], balls[(i+1)%n_balls], 2 );

// Complete the top with balls above and below the ring, and one in
// the center.
//
pNorm az(vz);
```

```
  pCoor top_pos = center_pos + az * crad * 1.5;
  pCoor bot_pos = center_pos - az * crad * 1.5;

  head_ball = balls.emplace_back( new Ball(top_pos,ball_default) );
  tail_ball = balls.emplace_back( new Ball(bot_pos,ball_default) );
  Ball* ctr_ball = balls.emplace_back( new Ball(center_pos,ball_default) );

  for ( int i=0; i<n_balls; i += 2 )
    {
      links += link_new( head_ball, balls[i], 2 );
      links += link_new( tail_ball, balls[i], 2 );
    }

  for ( int i=0; i<n_balls; i++ )
    links += link_new( ctr_ball, balls[i], 2 );

  for ( auto b: balls ) b->omega = omega * spin_axis;

  opt_head_lock = false;    // Head ball will be frozen in space.
  opt_tail_lock = false;   // Tail ball can move freely.

  setup_at_end();
}

void
World::ball_setup_6()
{
  // Render what will become a parachute, probably.

  last_setup = 6;
  objects_erase();

  pCoor center_pos = pCoor(17.2,14.8f,-10.2) + 2*vec_rand();

  const int n_balls = 4 * ( 4 + random()%4 );
  float ring_outer_radius = 5 + 2 * drand48();
  float ring_inner_radius = ring_outer_radius * ( 0.1 + drand48() * 0.4 );
  float ball_radius = 0.2 * 2 * M_PI * ring_outer_radius / n_balls;

  pNorm c_to_eye(center_pos,eye_location);
  pNorm az = c_to_eye + 0.5 * vec_rand() + pVect(0,0.5,0);

  pNorm ax = fabs(az.x) < fabs(az.y)
    ? pVect(0,az.z,-az.y) : pVect(az.z,0,-az.x);
  pVect ay(az,ax);

  // Default settings for balls.
  //
  Ball ballo_default;
  ballo_default.specularity = 0.5;
  ballo_default.velocity = pVect(0,0,0);
  ballo_default.radius = ball_radius;
  ballo_default.color = color_light_slate_gray;
  Ball balli_default = ballo_default;
```

```
  balli_default.radius = ballo_default.radius * 0.6;

  const float d_theta = 2 * M_PI / n_balls;

  const float link_stiffness = 10;

  vector<Ball*> balls_outer, balls_inner;

  // Place balls in a circle.
  //
  for ( int i=0; i<n_balls; i++ )
    {
      const float theta = d_theta * i;
      pVect pos_n = cosf(theta) * ax + sinf(theta) * ay;
      pCoor pos_o = center_pos + ring_outer_radius * pos_n;
      pCoor pos_i = center_pos + ring_inner_radius * pos_n;

      // Construct new ball and add it to the ball list.
      //
      Ball* ball_o = balls_outer.emplace_back( new Ball(pos_o,ballo_default) );
      Ball* ball_i = balls_inner.emplace_back( new Ball(pos_i,balli_default) );
      Link *link =
        new Link( ball_o, ball_i,
                  { .spring_constant = opt_spring_constant/200 } );
      links.push_back(link);

      // Set color of first ball to blue, others to white.
      ball_i->color = ball_o->color =
        i == 0 ? color_blue :
        theta <= M_PI/2 ? color_lsu_spirit_gold :
        color_white;
    }

  // Connect each ball to its neighbor with a link.
  //
  for ( auto bc: { &balls_outer, &balls_inner } )
    for ( int i=0; i<n_balls; i++ )
      {
        balls.push_back( bc[0][i] );
        links += link_new( bc[0][i], bc[0][(i+1)%n_balls], link_stiffness );
      }

  for ( int i=0; i<n_balls; i++ )
    {
      const int j = ( i + 1 ) % n_balls;
      Ball* b00 = balls_inner[i];
      Ball* b01 = balls_outer[i];
      Ball* b10 = balls_inner[j];
      Ball* b11 = balls_outer[j];
      surfaces.emplace_back(b00,b01,b11,color_light_gray);
      surfaces.emplace_back(b10,b00,b11,color_light_gray);
    }

  pNorm spin_axis = az;
```

```
  pCoor c_of_mass = center_pos;
  float omega_init = opt_omega * 0.1;

  if ( 0 )
  for ( Ball* ball: balls )
    {
      pVect cm_to_ball( c_of_mass, ball->position );

      ball->velocity = omega_init * cross( spin_axis, cm_to_ball );
      ball->omega = omega_init * spin_axis;
    }

  tail_ball = balls.back();
  head_ball = balls.front();

  opt_head_lock = false; // If true, head ball locked in place.
  opt_tail_lock = false;

  setup_at_end();
}

void
World::ball_setup_7()
{
  // Render what will become a parachute, probably.

  last_setup = 7;
  objects_erase();
  setup_at_end();
}

void
World::setup_at_end()
{
  balls.take(balls_marker);
  lock_update();
  time_step_count = 0;
  world_time = 0;
  world_time_link_update = -1;
  for ( uint i=0; i<balls.size(); i++ ) balls[i]->idx = i;
  for ( Link *l : links )
    { l->ball1_idx = l->ball1->idx; l->ball2_idx = l->ball2->idx; }
  for ( auto& s: surfaces )
    { s.b0_idx = s.b0->idx;  s.b1_idx = s.b1->idx;  s.b2_idx = s.b2->idx; }
}


 /// Advance Simulation State by delta_t Seconds
//
void
World::time_step_cpu(double delta_t)
{
  const int n_balls = balls;
  const int n_links = links;
```

```
  // Smoothly move ball in response to user input.
  //
  if ( adj_t_stop )
    {
      const double dt = min(world_time,adj_t_stop) - adj_t_prev;
      pVect adj = dt/adj_duration * adj_vector;
      balls_translate(adj,0);
      adj_t_prev = world_time;
      if ( world_time >= adj_t_stop ) adj_t_stop = 0;
    }

#pragma omp parallel for
  for ( int bi=0; bi<n_balls; bi++ )
    {
      Ball* const ball = balls[bi];
      assert( ball->fdt_to_do );
      ball->force = ball->mass * gravity_accel;
      ball->torque = pVect(0,0,0);
    }

  const int n_surfaces = surfaces.size();
  for ( int i=0; i<n_surfaces; i++ )
    {
      Surface& s = surfaces[i];
      Ball& b0 = *s.b0;
      Ball& b1 = *s.b1;
      Ball& b2 = *s.b2;
      pNorm snorm = cross(b0.position,b1.position,b2.position);
      pVect vel = (1.0/3) * ( b0.velocity + b1.velocity + b2.velocity );
      float speed = dot(snorm,vel);
      float fmag =
        -speed * fabs(speed) * snorm.magnitude * 0.5f * opt_air_resistance;
      pVect f = fmag * snorm;
      b0.force += f;
      b1.force += f;
      b2.force += f;
    }

#pragma omp parallel for
  for ( int i=0; i<n_links; i++ )
    {
      Link* const link = links[i];
      if ( !link->is_simulatable ) continue;

      //
      // Spring Force from Neighbor Balls
      //

      Ball* const ball1 = link->ball1;
      Ball* const ball2 = link->ball2;

      /// Notes:
      //
```

```
// link->cb1
//   Coordinate where link touches ball1, in ball1's local space.
//
// ball1->omatrix
//   Ball's orientation transformation ..
//   .. which transforms vectors in ball1's local space to object space.
//

// Find position and velocity of the point where the link touches
// the surface of ball 1 ...
//
pVect dir1 = ball1->omatrix * link->cb1;
pCoor pos1 = ball1->position + dir1;
pVect vel1 = ball1->velocity + ball1->point_rot_vel(dir1);

// ... and ball 2.
//
pVect dir2 = ball2->omatrix * link->cb2;
pCoor pos2 = ball2->position + dir2;
pVect vel2 = ball2->velocity + ball2->point_rot_vel(dir2);

// Construct a normalized (Unit) Vector from ball to neighbor
// based on link connection points and ball centers.
//
pNorm link_dir(pos1,pos2);
pNorm c_to_c(ball1->position,ball2->position);

const float link_length = link_dir.magnitude;

// Compute the speed of ball's end of link towards neighbor's end of link.
//
pVect delta_v = vel2 - vel1;
float delta_s = dot( delta_v, link_dir );

// Compute by how much the spring is stretched (positive value)
// or compressed (negative value).
//
const float spring_stretch = link_length - link->distance_relaxed;

// Determine whether spring is gaining energy (whether its length
// is getting further from its relaxed length).
//
const bool gaining_e = ( delta_s > 0.0 ) == ( spring_stretch > 0 );

const float blend_length_factor = 0.1; // Just a guess.
const float blend_s = blend_length_factor * c_to_c.magnitude;
const float blend = min( 1.0f, fabs(delta_s) / blend_s );

// Use a smaller spring constant when spring is loosing energy,
// a quick and dirty way of simulating energy loss due to spring
// friction.
//
const float sc_base = link->spring_constant ?: opt_spring_constant;
const float spring_constant =
```

```
        gaining_e ? sc_base :
        sc_base * ( 1.0f - blend + blend * opt_spring_damp );

      const float force_mag = spring_constant * spring_stretch;
      pVect spring_force_12 = force_mag * link_dir;

      // Apply forces affecting linear momentum.
      //
      link->spring_force_12 = spring_force_12;

      if ( ! link->is_surface_connection ) continue;

      // Apply torque.
      //
      link->torque1 = cross(pNorm(dir1), spring_force_12);
      link->torque2 = cross(pNorm(dir2), spring_force_12);
    }

  // Note: Because two links can reference the same ball this should
  // not be done in parallel.
  for ( Link *link: links )
    {
      if ( !link->is_simulatable ) continue;
      Ball* const ball1 = link->ball1;
      Ball* const ball2 = link->ball2;
      ball1->force += link->spring_force_12;
      ball2->force -= link->spring_force_12;
      ball1->torque += link->torque1;
      ball2->torque -= link->torque2;
    }

  ///
  /// Update Position of Each Ball
  ///

#pragma omp parallel for
  for ( int bi=0; bi<n_balls; bi++ )
    {
      Ball* const ball = balls[bi];
      if ( ball->locked )
        {
          ball->velocity = pVect(0,0,0);
          ball->omega = pVect(0,0,0);
          continue;
        }

      // Update Velocity
      //
      // This code assumes that force on ball is constant over time
      // step. This is clearly wrong when balls are moving with
      // respect to each other because the springs are changing
      // length. This inaccuracy will make the simulation unstable
      // when spring constant is large for the time step.
      //
```

```
const float mass = max( ball->mass, ball->mass_min );

pVect delta_v = ( ball->force / mass ) * delta_t;
const float r = ball->radius;

if ( platform_collision_possible(ball->position)
     && fabs(ball->position.y) < r )
  {
    const bool above = ball->position.y >= 0;
    const float spring_constant_plat =
      ball->velocity.y < 0 == above ? 100000 : 50000;
    const float fric_coefficient = opt_friction_dynamic;
    const float force_away = ( above ? 1 : -1 )
      * ( r - fabs(ball->position.y) ) * spring_constant_plat;
    const float delta_v_away = force_away / mass * delta_t;
    const float fric_force_mag = fric_coefficient * force_away;
    pNorm surface_v(ball->velocity.x,0,ball->velocity.z);
    const float delta_v_surf = fric_force_mag / mass * delta_t;

    if ( delta_v_surf > surface_v.magnitude )
      {
        // Ignoring other forces?
        delta_v =
          pVect(-ball->velocity.x,delta_v.y,-ball->velocity.z);
      }
    else
      {
        delta_v -= delta_v_surf * surface_v;
      }
    delta_v.y += delta_v_away;
  }

ball->velocity += delta_v;

// Air Resistance
//
const pNorm ball_dir(ball->velocity);
const float delta_s_air_raw =
  opt_air_resistance * ball_dir.mag_sq * delta_t;
const float delta_s_air = min( ball_dir.magnitude, delta_s_air_raw );
ball->velocity -= delta_s_air * ball_dir;

// Update Position
//
// Assume that velocity is constant.
//
ball->position += ball->velocity * delta_t;

ball->omega += delta_t * ball->fdt_to_do * ball->torque;

pNorm axis(ball->omega);

// Update Orientation
//
```

```
      // If ball isn't spinning fast skip expensive rotation.
      //
      if ( axis.mag_sq < 0.000001 ) continue;

      // Update ball orientation.
      //
      ball->orientation_set
        ( pQuat(axis,delta_t * axis.magnitude) * ball->orientation );
    }
}

bool
World::platform_collision_possible(pCoor pos)
{
  // Assuming no motion in x or z axes.
  //
  return pos.x >= platform_xmin && pos.x <= platform_xmax
    && pos.z >= platform_zmin && pos.z <= platform_zmax;
}

 /// External Modifications to State
//
//   These allow the user to play with state while simulation
//   running.

// Move the ball.
//
void Ball::translate(pVect amt) {position += amt;}

// Add velocity to the ball.
//
void Ball::push(pVect amt) {velocity += amt;}

// Set the velocity to zero.
//
void Ball::stop() {velocity = pVect(0,0,0); }

// Set the velocity and rotation (not yet supported) to zero.
//
void Ball::freeze() {velocity = pVect(0,0,0); }



void World::balls_translate(pVect amt,int b){head_ball->translate(amt);}
void World::balls_push(pVect amt,int b){head_ball->push(amt);}
void World::balls_translate(pVect amt)
{ for ( Ball *ball: balls ) ball->translate(amt); }
void World::balls_push(pVect amt)
{ for ( Ball *ball: balls ) ball->push(amt); }
void World::balls_stop()
{ for ( Ball *ball: balls ) ball->stop(); }
void World::balls_freeze(){balls_stop();}
```

```
void
World::render_link_start()
{
  if ( world_time_link_update == world_time ) return;

  // Empty the containers in preparation for a new pass.
  for ( auto b: lis_bufs ) b->clear();
}

void
World::render_link_gather(Link *link)
{
  // Place data needed to render link (the argument to this function)
  // in arrays. Just before rendering those arrays will be placed in
  // buffer objects.


  // If we've already collected links, just return.  This happens
  // when doing multiple passes to render shadows and reflections.
  //
  if ( world_time_link_update == world_time ) return;

  Ball *const ball1 = link->ball1;
  Ball *const ball2 = link->ball2;

  // Direction of link relative to center off ball.
  //
  pVect dir1 = ball1->omatrix * link->cb1;

  // Place on surface of ball where link connects.
  //
  pCoor pos1 = ball1->position + dir1;

  // Direction and connection location for ball 2.
  //
  pVect dir2 = ball2->omatrix * link->cb2;
  pCoor pos2 = ball2->position + dir2;

  pVect p1p2(pos1,pos2);
  pNorm p1p2n(p1p2);

  // Radius of link.
  //
  const float rad = link->radius ?: ball1->radius * 0.3;

  // Compute scale factors for texture.
  const float tex_aspect_ratio = 8.5 / 11;  // Yes, inches. Can't remember mm.
  const float page_width_o = 2.5;  // Width of texture in object-space coords.
  const float tex_t_scale = link->distance_relaxed / page_width_o;
  const float tex_angle_scale = rad * tex_aspect_ratio / page_width_o;

  pNorm dirn1(dir1);
  pNorm dirn2(dir2);
```

```
  // Vectors used to describe the cubic Hermite curve.
  //
  pVect4 v1 = p1p2n.magnitude * dirn1;
  pVect4 v2 = p1p2n.magnitude * dirn2;

  // Convert link's local x and y axes to global coordinates.
  //
  pVect4 b1_ydir = ball1->omatrix * link->b1_dir;
  pVect4 b2_ydir = ball2->omatrix * link->b2_dir;


  // Place data needed to render links in lists. Note that
  // the += operator appends the right-hand side to the container
  // on the left-hand side.
  //
  link->idx = lis_pos1.size();
  lis_pos1 << pos1;
  lis_pos2 << pos2;
  lis_v1 << v1;
  lis_v2 << v2;
  lis_b1_ydir << b1_ydir;
  lis_b2_ydir << b2_ydir;
  lis_tex_rad << vec4( tex_t_scale, tex_angle_scale, rad, 0);
}



void
World::render_link_render(vk::CommandBuffer& cb, bool shadow_volumes)
{
  // Perform a rendering pass to render all the links collected
  // by the routine render_link_2.


  // Check whether this is the first time we are rendering this data.
  //
  const bool first_render =
    world_time_link_update != world_time
    || opt_segments != buf_uni_common->links_n_segs
    || opt_sides != buf_uni_common->sides;
  world_time_link_update = world_time;
  const size_t n_instances = lis_pos1.size();
  const bool do_rt = vh.raytrace_active();
  const int segments = shadow_volumes ? max(2,opt_segments/4) : opt_segments;
  const int segments_sv = max(2,opt_segments/4);

  if ( !n_instances ) return;

  if ( first_render )
    {
      for ( auto b: lis_bufs ) b->to_dev();
      buf_uni_common->sides = opt_sides;
      buf_uni_common->links_n_segs_sv = segments_sv;
```

```
      buf_uni_common->delta_tee_sv = 1.0f / segments_sv;
      buf_uni_common->links_n_segs = opt_segments;
      buf_uni_common->delta_tee = 1.0f / opt_segments;
    }

  if ( do_rt && first_render )
    {
      const int n_tri_p_segment = 2 * opt_sides;
      const int n_tri_p_link = n_tri_p_segment * opt_segments;
      const int n_tri_p_pass = n_tri_p_link * n_instances;

      const int n_vtx_p_segment = 2 * ( opt_sides + 1 );
      const int n_vtx_p_link = n_vtx_p_segment * opt_segments;
      const int n_vtx_p_pass = n_vtx_p_link * n_instances;

      if ( !rt_geo_links )
        rt_geo_links
          .init( raytrace )
          .color_set( color_dim_gray, color_salmon )
          .bind( VR_BUF_IDX_sampler, sampler, texid_hw )
          .grouping_set( RT_Geometry_Indexed );

      buf_links_as_idx.assure(3*n_tri_p_pass);
      buf_links_tex_coor.assure(n_vtx_p_pass);
      buf_links_vertex_g.assure(n_vtx_p_pass);
      buf_links_normal_g.assure(n_vtx_p_pass);

      rt_geo_links
        .bind( VR_BUF_IDX_indices, buf_links_as_idx, 3*n_tri_p_pass )
        .bind( VR_BUF_IDX_pos, buf_links_vertex_g, n_vtx_p_pass )
        .bind( VR_BUF_IDX_normal, buf_links_normal_g )
        .bind( VR_BUF_IDX_tcoor, buf_links_tex_coor );
    }

  if ( do_rt )
    {
      QSet& qs = vh.qs;
      buf_uni_common.to_dev();
      qs.cb.begin( { vk::CommandBufferUsageFlagBits::eOneTimeSubmit } );
      comp_links.record_dispatch(qs.cb,opt_segments,n_instances);
      qs.cb.end();
      qs.q.submit( vk::SubmitInfo{ 0, nullptr, nullptr, 1, &qs.cb }, nullptr );
      qs.q.waitIdle();
      return;
    }


  VPipeline& pipe = shadow_volumes ? pipe_set_2_sv : pipe_set_2;

  pipe.record_draw_instanced(cb, segments+1, n_instances);
}


void
```

```
World::frame_callback()
{
  // This routine called whenever window needs to be updated, but
  // before commands are recorded.

  // Get any waiting keyboard commands.
  //
  cb_keyboard();

  if ( vh.have_raytrace ) vh.opt_want_raytrace_now = opt_want_raytrace;

  // This is a hack for 2023 HW 2.
  while ( !balls.empty() && balls.back()->world_time_expire <= world_time )
    balls.pop_back();
  //
  // Balls must be last in list and not part of a link.

  const double time_now = time_wall_fp();
  const int time_step_count_start = time_step_count;
  const int nmp = gpu_info.cuda_prop.multiProcessorCount;
  int bl_p_sm_max = 1024;
  for ( int i=0; i<gpu_info.num_kernels; i++ )
    set_min
      (bl_p_sm_max,
       gpu_info.get_max_active_blocks_per_mp(i, opt_block_size, 0) );
  const int grid_size = bl_p_sm_max * nmp;

  const bool opt_cuda = opt_physics_method != GP_cpu;
  if ( opt_cuda )
    {
      CE( cudaEventRecord(frame_start_ce,0) );
      data_cpu_to_gpu_constants();
      data_cpu_to_gpu_dynamic();
    }

  if ( !opt_pause || opt_single_frame || opt_single_time_step )
    {
      /// Advance simulation state.

      // Amount of time since the user saw the last frame.
      //
      const double wall_delta_t = time_now - last_frame_wall_time;

      // Compute amount by which to advance simulation state for this frame.
      //
      const double duration =
        vh.video_frame_duration() ?:
        opt_single_time_step ? opt_single_time_step * opt_time_step_duration :
        opt_single_frame ? 1/60.0 :
        wall_delta_t;

      const double world_time_target = world_time + duration;
      const double wall_time_limit = time_now + 0.05;
```

```
      while ( world_time < world_time_target )
        {
          if ( opt_cuda )
            {
              launch_time_step(opt_time_step_duration,grid_size,opt_block_size);
            }
          else
            {
              time_step_cpu(opt_time_step_duration);
            }
          time_step_count++;
          world_time += opt_time_step_duration;
          const double time_right_now = time_wall_fp();
          if ( time_right_now > wall_time_limit ) break;
        }

      // Reset these, just in case they were set.
      //
      opt_single_frame = false;
      opt_single_time_step = 0;
    }

  if ( opt_cuda )
    {
      data_gpu_to_cpu_dynamic();
      CE( cudaEventRecord(frame_stop_ce,0) );
    }

  frame_timer.phys_end();
  frame_timer.work_amt_set(time_step_count-time_step_count_start);

  float cuda_time = 0;
  if ( opt_cuda )
    {
      CE(cudaEventSynchronize(frame_stop_ce));
      CE(cudaEventElapsedTime(&cuda_time,frame_start_ce,frame_stop_ce));
    }
  frame_timer.cuda_frame_time_set(cuda_time);

  last_frame_wall_time = time_now;

}

int
main(int argv, char **argc)
{
  pVulkan_Helper pvulkan_helper(argv,argc);
  World world(pvulkan_helper,argv,argc);

  world.run();
}
```

```
/// LSU EE 4702-1 (Fall 2024), GPU Programming
//
 /// Homework 3 -- SOLUTION
 //
 //  Assignment: https://www.ece.lsu.edu/gpup/2024/hw03.pdf
 //
 /// Based on cpu-only/demo-06-rend-pipe.cc

#include <gp/coord.h>
#include <gp/colors.h>
#include "frame_buffer.h"

enum HW3_Lines {
  Lines_None, Lines_Pixel, Lines_World, Lines_ENUM_SIZE };
const char* const hw3_lines_str[] =
  { "NONE", "PIXEL", "WORLD" };

  ///
 /// Our_3D: Low-Level 3D Graphics Library.
 ///
class Our_3D {
public:
  Our_3D(pFrame_Buffer& fbp)
    :frame_buffer(fbp)
  {
    vtx_colors = nullptr;
    vtx_coors = nullptr;
    vtx_normals = nullptr;

    topology_strip = false;
    opt_light_frag = true;
    opt_lines = Lines_None;
  };

  Our_3D& transform_eye_from_object_set(const pMatrix& m)
  {
    eye_from_object = m;
    return *this;
  }
  Our_3D& transform_clip_from_eye_set(const pMatrix& m)
  {
    clip_from_eye = m;
    return *this;
  }
  Our_3D& light_location_set(const pCoor c)
  {
    light_location = c;
    return *this;
  }
  Our_3D& light_color_set(const pColor c)
  {
    light_color = c;
    return *this;
```

```
  }

  Our_3D& vtx_coors_set(vector<pCoor>& c)
  {
    vtx_coors = &c;
    return *this;
  }

  Our_3D& vtx_colors_set(vector<pColor>& c)
  {
    vtx_colors = &c;
    return *this;
  }

  // Specify whether vertices are grouped for individual triangles or
  // a triangle strip in the next call to draw_rasterization.
  //
  Our_3D& topology_strip_set( bool strip = true )
  {
    topology_strip = strip;
    return *this;
  }

  //  Specify vertex normals to by used by the next draw_rasterization.
  //
  Our_3D& vtx_normals_set(vector<pVect4>& c)
  {
    vtx_normals = c.data();
    return *this;
  }

  Our_3D& draw_rasterization();

  pFrame_Buffer& frame_buffer;
  pMatrix eye_from_object, clip_from_eye;
  pCoor light_location;
  pColor light_color;
  vector<pCoor> *vtx_coors;
  pVect4 *vtx_normals;
  vector<pColor> *vtx_colors;
  bool topology_strip;
  bool opt_tryout1, opt_tryout2;
  bool opt_light_frag;
  int opt_lines;
};

 ///
 ///   World: An application.
 ///
class World : public World_Base {
public:
  World(pFrame_Buffer& fb):World_Base(fb),gc(fb){}
  Our_3D gc;
```

```
  // Called each time a frame needs to be rendered.
  virtual void render(bool first_time)
  {
    if ( first_time ) init_scene();
    render_scene();
  }

  // Called Once
  void init_scene();

  // Called for each frame.
  void render_scene();

  pCoor eye_location, light_location;
  pColor light_color;
  pCoor *adj_location;
  pVect eye_direction;
  bool opt_tryout1,opt_tryout2;
};

void
World::init_scene()
{
  eye_location = pCoor(-1.2,.5,10.2);
  light_location = pCoor(0.7, 1.9, -0.6);
  eye_direction = pVect(.1,0,-1);
  adj_location = &eye_location;
  opt_tryout1 = opt_tryout2 = false;
  gc.opt_tryout1 = opt_tryout1;
  gc.opt_tryout2 = opt_tryout2;
  light_color = color_white * 6.3;
}

void
World::render_scene()
{
  pFrame_Buffer& frame_buffer = gc.frame_buffer;

  //
  /// Process User Key Presses
  //
  switch ( frame_buffer.keyboard_key ) {
  case FB_KEY_LEFT: adj_location->x -= 0.1; break;
  case FB_KEY_RIGHT: adj_location->x += 0.1; break;
  case FB_KEY_PAGE_UP: adj_location->y += 0.1; break;
  case FB_KEY_PAGE_DOWN: adj_location->y -= 0.1; break;
  case FB_KEY_UP: adj_location->z -= 0.1; break;
  case FB_KEY_DOWN: adj_location->z += 0.1; break;
  case 'e': case 'E': adj_location = &eye_location; break;
  case 'l': case 'L': adj_location = &light_location; break;
  case 'y': gc.opt_tryout1 = opt_tryout1 = !opt_tryout1; break;
  case 'Y': gc.opt_tryout2 = opt_tryout2 = !opt_tryout2; break;
  case 'n': case 'N':
    gc.opt_lines++;
```

```
  if ( gc.opt_lines >= Lines_ENUM_SIZE ) gc.opt_lines = Lines_None;
  break;
case 'p': case 'P': gc.opt_light_frag = !gc.opt_light_frag; break;
case '+': case '=':
  if ( !frame_buffer.keyboard_control ) light_color *= 1.1;
  break;
case '-':
  if ( !frame_buffer.keyboard_control ) light_color *= 1/1.1;
  break;
}

///
/// Print "Green Text" Information
///
gc.frame_buffer.fbprintf
  ("Lines %s ('n')  Lighting per %s ('p')  "
   "Tryout1 %s ('y')  Tryout2 %s ('Y') \n",
   hw3_lines_str[gc.opt_lines],
   gc.opt_light_frag ? "FRAGMENT" : "VERTEX  ",
   opt_tryout1 ? "ON " : "OFF", opt_tryout2 ? "ON " : "OFF");
gc.frame_buffer.fbprintf
  ( "Light Location [ %.2f, %.2f, %.2f ],  Light Intensity %.2f ('+-')  "
    "Eye Location [ %.2f, %.2f, %.2f ]\n",
    light_location.x, light_location.y, light_location.z,
    light_color.r,
    eye_location.x, eye_location.y, eye_location.z );

gc.light_location_set( light_location ).light_color_set( light_color );


/// Specifications for Cylinder (Ring)
//
pCoor cyl_center(-1,-1,-3);
int n_segs = 50;
pVect cyl_axis( 0, 0.75, 0.05 );
float cyl_radius = 4;
//
// The cylinder is constructed much further below.

///
/// Prepare Transformations
///

// Specifications of User's Monitor
//
const uint win_width = frame_buffer.width_get();
const uint win_height = frame_buffer.height_get();
//
// win_width and win_height are pixel-space coordinates.
//
const float aspect = 1.0 * win_width / win_height;
const float width_m = 0.8;  // Width of user's monitor.
const float height_m = width_m / aspect;
//
```

```
// width_m and height_m are in eye-space coordinates.

// Specify Transformation from Object to Eye Space: eye_from_object.
//
pMatrix_Translate center_eye(-eye_location);
pMatrix_Rotation rotate_eye(eye_direction,pVect(0,0,-1));
pMatrix eye_from_object = rotate_eye * center_eye;

gc.transform_eye_from_object_set(eye_from_object);

// Transform from Eye to Clip
//
pMatrix_Frustum clip_from_eye
  ( -width_m/2, width_m/2,  -height_m/2, height_m/2,  1, 5000 );

gc.transform_clip_from_eye_set(clip_from_eye);


///
/// Prepare Triangles
///

// List of coordinates and normals in object space and list of colors.
//
vector<pCoor> coors_os;
vector<pVect4> norms_os;
vector<pColor> colors;

// Insert First Triangle
//
// Insert coordinates of a triangle into coordinate list,
// and insert its color into a color list.
//
coors_os << pCoor( 0, 0, 0 ) << pCoor( 9, 6, -5 ) << pCoor( 0, 7, -3 );
colors << color_dark_olive_green << color_dark_olive_green
       << color_dark_olive_green;

// Insert Second Triangle
//
// Add coordinates and color of another triangle
//
coors_os << pCoor(0.3,1.5, -.1) << pCoor(4,2,-5) << pCoor(5,0,0);
pColor gray = 0.2 * color_white;
colors << gray << gray << gray;

// Insert Two More Triangles
//
// Add a square consisting of a gold and multicolored triangle.
//
coors_os << pCoor(-4,0,-3) << pCoor(-4,2,-3) << pCoor(-2,0,-3);
colors << color_gold << color_gold << color_gold;
//
coors_os << pCoor(-4,2,-3) << pCoor(-2,2,-3) << pCoor(-2,0,-3);
colors << color_red << color_green << color_blue;
```

```
/// SOLUTION -- Problem 1 a
//
//  Insert purple triangle.
//
coors_os << pCoor( -2, 2, -3) << pCoor( -2, 1, -3 ) << pCoor( 0, 2, -3 );
colors << color_lsu_spirit_purple << color_lsu_spirit_purple
       << color_lsu_spirit_purple;

// Insert a triangle at light location, facing user. Light is top point.
pVect light_to_eye( light_location, eye_location );
float l_sz = 0.1;
pVect lax = cross( pVect( 0, 1, 0 ), light_to_eye ).normal() * l_sz;
pVect lay = cross( light_to_eye, lax ).normal() * l_sz;
coors_os << light_location
         << light_location - lax - lay << light_location + lax - lay;
colors << color_white << color_white << color_white;

/// SOLUTION -- Problem 1 b
//
//  Insert a triangle with vertices at the top and bottom of the
//  cylinder axis and the third vertex towards the light.
//
pCoor cyl_center_top = cyl_center + cyl_axis;
pNorm cct_to_l( cyl_center_top, light_location);
coors_os << cyl_center << cyl_center_top << cyl_center_top + 2 * cct_to_l;
colors << color_salmon << color_salmon << color_salmon;

// Now compute the triangle normals.
for ( size_t i = 0;  i < coors_os.size();  i += 3 )
  {
    pCoor p1 = coors_os[i], p2 = coors_os[i+1], p3 = coors_os[i+2];
    pVect4 n = cross( p1, p2, p3 ).normal();
    norms_os << n << n << n;
  }

///
/// Send Triangles to Our3D for Rendering
///
// Tell Our_3D that in the next draws vertices will not be grouped
// as triangle strips.
//
gc.topology_strip_set(false);

gc.vtx_coors_set(coors_os);
gc.vtx_colors_set(colors);
gc.vtx_normals_set(norms_os);
gc.draw_rasterization();


///
/// Prepare Ring (Cylinder)
///
```

```
  // Empty the containers so they can be used for the next draw.
  //
  coors_os.clear(); colors.clear(); norms_os.clear();

  // Enter primitives for a cylinder (a short one, looks like a ring).
  //
  pNorm ax(0,-cyl_axis.z,cyl_axis.y);  // Find an axis orthogonal to cyl_axis.
  pNorm ay = cross( cyl_axis, ax );
  float delta_theta = 2 * M_PI / n_segs;

  for ( int i=0; i<=n_segs; i++ )
    {
      float theta = i * delta_theta;
      pVect4 n = ax * cosf(theta) + ay * sinf(theta);
      pCoor c = cyl_center + cyl_radius * n;

      coors_os << c + cyl_axis << c;
      norms_os << n << n;
      colors << color_cyan << color_cyan;
    }

  gc.topology_strip_set(true);
  gc.vtx_normals_set(norms_os);
  gc.vtx_coors_set(coors_os).vtx_colors_set(colors);
  gc.draw_rasterization();
}



 ///
 /// Code Below (in Our3D) is Part of (Fictional) 3D Graphics Library
 ///

Our_3D&
Our_3D::draw_rasterization()
{
  ///
  /// Prepare Transformations
  ///

  // Specifications of User's Monitor
  //
  const uint win_width = frame_buffer.width_get();
  const uint win_height = frame_buffer.height_get();
  //
  // win_width and win_height are pixel-space coordinates.

  // Transform from Clip to Pixel
  //
  pMatrix_Translate recenter(pVect(1,1,0));
  pMatrix_Scale scale( win_width/2, win_height/2, 1 );
  pMatrix window_from_clip = scale * recenter;
  pMatrix ws_from_os = window_from_clip * clip_from_eye * eye_from_object;
  pCoor eye_location = -eye_from_object.col_get(3);
```

```
pColor light_ambient = 0.4 * color_white;

frame_buffer.n_vtx_frame += vtx_coors->size();

///
/// Rasterize Triangle and Write Frame Buffer
///

// Get pointer to the depth (z) buffer.
//
float* const zbuffer = frame_buffer.buffer_depth_get();
//
// The depth buffer uses the same index as the frame buffer.

auto& coors_os = *vtx_coors;
auto& colors = *vtx_colors;

//  Set loop stride based on indicated grouping.
//
const size_t inc = topology_strip ? 1 : 3;

/// SOLUTION -- Problem 2
//
//  Compute lighted colors *before* the main loop if opt_light_frag is false.
//
vector<pColor> lighted_colors;
if ( !opt_light_frag )
  {
    // For performance reasons allocate space in advance, since we know
    // how much space we need.
    //
    lighted_colors.resize(coors_os.size());

    for ( size_t i=0; i < coors_os.size(); i++ )
      {
        pVect4 n = vtx_normals[i];
        pColor color = colors[i];
        pCoor of = coors_os[i];

        // Compute lighted color of vertex i.
        //
        pNorm frag_to_light(of,light_location);
        pVect frag_to_eye(of,eye_location);
        float cos_angle_eye = dot( n, frag_to_eye );
        float cos_angle_light = dot( n, frag_to_light );
        bool illumination_visible =
          ( cos_angle_light > 0 )  ==  ( cos_angle_eye > 0 );
        float clamped_phase =
          illumination_visible ? fabs(cos_angle_light) : 0;
        pColor lighted_color =
          ( light_ambient
            + clamped_phase / frag_to_light.magnitude * light_color )
          * color;
```

```
          // Write lighted color into new lighted_colors container.
          //
          lighted_colors[i] = lighted_color;
        }
    }

  ///
  /// Main Rasterization Loop
  ///
  //
  ///  Outer Loop: Iterate Over Triangles
  //
  for ( size_t i=0; i+2 < coors_os.size(); i += inc )
    {
      frame_buffer.n_tri_frame++; // Count of number of triangles.

      // Get next triangle's object space coordinates ..
      //
      pCoor o0 = coors_os[i+0],  o1 = coors_os[i+1],  o2 = coors_os[i+2];
      //
      // .. convert them into unhomogenized window-space coordinates ..
      //
      pCoor u0( ws_from_os * o0 ), u1( ws_from_os * o1 ), u2( ws_from_os * o2 );
      //
      // .. and homogenize them.
      //
      pCoor_Homogenized w0( u0 ), w1( u1 ), w2( u2 );

      //  Extract colors from list.
      //
      pColor c0 = colors[i], c1 = colors[i+1], c2 = colors[i+2];

      //  Extract normals from list.
      //
      pVect4 n0 = vtx_normals[i], n1 = vtx_normals[i+1], n2 = vtx_normals[i+2];

      // Compute increment that will advance by approximately one pixel along
      //  b0 and b1 dimensions.
      //
      pVect v20(w2,w0), v12(w1,w2);
      float db0 = 0.8/max(fabs(v20.x),fabs(v20.y));
      float db1 = 0.8/max(fabs(v12.x),fabs(v12.y));

      if ( db0 < 0.00025 || db1 < 0.00025 ) continue;

      // Used for computing perspective-correct barycentric coordinates.
      const float u0ou2 = u0.w/u2.w, u0ou1 = u0.w/u1.w;
      const float u1ou2 = u1.w/u2.w, u1ou0 = u1.w/u0.w;

      /// SOLUTION -- Problem 3
      //
      //  Prepare to draw 0.1-object-space-units-wide triangle edges.
      //
      //  Compute value of each object-space barycentric coordinate
```

```
//  (bc0, bc1, pc2) that corresponds to a distance of 0.1 from
//  the opposite edges.
//
float line_wid_o = 0.1;
//
// Compute vectors along each edge.
//
pVect o01(o0,o1), o12(o1,o2), o20(o2,o0);
//
// Compute triangle normal.
//
pVect nt = cross(o20,o01);
//
// Compute an orthogonal vector for each triangle edge.
//
pNorm Oo01(cross(nt,o01)), Oo12(cross(nt,o12)), Oo20(cross(nt,o20));
//
// Compute object-space distance from each vertex to opposite
// edge. For example, dist_0_12 is distance from vertex 0 to
// edge 12.
//
float dist_0_12 = dot( o20, Oo12 );
float dist_1_20 = dot( o01, Oo20 );
float dist_2_01 = dot( o12, Oo01 );
//
// Find object-space barycentric coordinate value at which
// interpolated fragment is line_wid_o units from opposite edge.
// For example, b0o is the value of object-space barycentric
// coordinate bc0 at which the interpolated fragment is at
// distance exactly line_wid_o object-space units from edge 12.
//
float b0o = min(1.0f, line_wid_o / dist_0_12 );
float b1o = min(1.0f, line_wid_o / dist_1_20 );
float b2o = min(1.0f, line_wid_o / dist_2_01 );
//
//
//  Prepare to draw 2-pixel-wide triangle edges.
//
//  The method for finding the barycentric coordinate limits,
//  b0w, b1w, b2w, is similar to the procedure for finding
//  limits on object-space barycentric coordinates, except that
//  these limits are computed using the window-space coordinates
//  of the vertices.
//
float line_wid_p = 2;  // It doesn't have to be two. Adjust to taste.
pVect v01(w0,w1);
pVect wn = cross(v20,v01);
pNorm Ow01(cross(wn,v01)), Ow12(cross(wn,v12)), Ow20(cross(wn,v20));
float b0w = min(1.0f, line_wid_p / dot( v20, Ow12 ) );
float b1w = min(1.0f, line_wid_p / dot( v01, Ow20 ) );
float b2w = min(1.0f, line_wid_p / dot( v12, Ow01 ) );

//
/// Inner Loop: Iterate over Triangle Fragments
```

```
//
for ( float b0=0; b0<=1; b0 += db0 )
  for ( float b1=0; b1<=1-b0; b1 += db1 )
    {
      frame_buffer.n_frag_frame++; // Count of number of fragments.
      const float b2 = 1 - b0 - b1;

      // Compute window-space coordinate of fragment.
      //
      pCoor wf = b0*w0 + b1*w1 + b2*w2;

      // Check whether fragment is within window.
      //
      if ( uint(wf.x) >= win_width || uint(wf.y) >= win_height ) continue;

      // Compute the index into the frame and depth buffers.
      //
      const size_t idx = wf.x + int(wf.y) * win_width;

      // Compute perspective-correct barycentric coordinates.
      //
      float bc0 = b0 / ( b2 * u0ou2 + b1 * u0ou1 + b0 );
      float bc1 = b1 / ( b2 * u1ou2 + b0 * u1ou0 + b1 );
      float bc2 = 1 - bc0 - bc1;
      //
      // These should be used for interpolating everything except
      // pixel-space coordinates.

      switch ( opt_lines ){
      case Lines_None:
        break;

      case Lines_Pixel:

        /// SOLUTION -- Problem 3
        //
        //  If all of the window-space barycentric coordinates
        //  are too far away from an edge skip this fragment.
        //
        if (  b0 > b0w  &&   b1 > b1w  &&   b2 > b2w  ) continue;
        break;

      case Lines_World:

        /// SOLUTION -- Problem 3
        //
        //  If all of the object-space barycentric coordinates
        //  are too far away from an edge skip this fragment.
        //
        if ( bc0 > b0o  &&  bc1 > b1o  &&  bc2 > b2o  ) continue;
        break;

      default: assert( false ); break;
      }
```

```
// Depth (Z) Test
//
float zinv = wf.z;
if ( zinv < 0 || zinv > 1 ) continue;
if ( zbuffer[ idx ] < zinv ) continue;

zbuffer[ idx ] = zinv;


/// SOLUTION -- Problem 2
//
//  Compute the lighted color if opt_light_frag is true,
//  otherwise interpolate the values from lighted_colors.
//
pColor lighted_color;
//
if ( opt_light_frag )
  {
    // Find object-space coordinate of fragment.
    //
    pCoor of =     bc0*o0 + bc1*o1 + bc2*o2;

    // "Blend" normal of each vertex.
    //
    pNorm n =      bc0*n0 + bc1*n1 + bc2*n2;

    // Blend color of each vertex together.
    //
    pColor color = bc0*c0 + bc1*c1 + bc2*c2;

    // Compute Lighted Color
    //
    pNorm frag_to_light(of,light_location);
    pVect frag_to_eye(of,eye_location);
    float cos_angle_eye = dot( n, frag_to_eye );
    float cos_angle_light = dot( n, frag_to_light );
    bool illumination_visible =
      ( cos_angle_light > 0 )  ==  ( cos_angle_eye > 0 );
    float clamped_phase =
      illumination_visible ? fabs(cos_angle_light) : 0;
    lighted_color =
      ( light_ambient
        + clamped_phase / frag_to_light.magnitude * light_color )
      * color;
  }
else
  {
    /// SOLUTION -- Problem 2
    //
    //  Instead of computing lighted color for this fragment
    //  interpolate the vertices' lighted colors.
    //
    lighted_color =
```

```
                    bc0 * lighted_colors[i]
                    + bc1 * lighted_colors[i+1]
                    + bc2 * lighted_colors[i+2];
                }

              //
              // Write the frame (color) buffer with the lighted color
              //
              frame_buffer[ idx ] = lighted_color.int_rgb();
              frame_buffer.n_px_frame++; // Count of number of written pixels.
            }
        }
    return *this;
}


int
main(int argc, char **argv)
{
  pFrame_Buffer demo_frame_buffer(argc,argv);
  WRender render( demo_frame_buffer );
  World our_world( demo_frame_buffer );
  WRENDER_INSERT(render, our_world);
  render.run();
  return 0;
}
```

```
/// LSU EE 4702-1 (Fall 2024), GPU Programming
//
 /// Homework 4
//
//   Modify this file AND hw04.cc.
//   Assignment: https://www.ece.lsu.edu/koppel/gpup/2024/hw04.pdf

// Specify version of OpenGL Shading Language.
//
#version 460

#extension GL_GOOGLE_include_directive : enable

layout ( binding = BIND_MISC ) uniform Uni_Misc
{
  ivec4 uni_misc;
};

bool opt_tryout1 = bool(uni_misc.x);
bool opt_tryout2 = bool(uni_misc.y);


const int ncolors = 10;
layout ( binding = BIND_HW03 ) uniform I_can_forget_this_name_no_problem
{
  vec4 front[ncolors], back[ncolors];
} uc;


 /// Shader Uniforms
//
layout ( binding = BIND_LIGHT_SIMPLE ) uniform Uni_Light
{
  vec4 position;
  vec4 color;
} uni_light;


layout ( binding = BIND_TRANSFORM ) uniform Uni_Transform
{
  // Transformation Matrices
  mat4 eye_from_object, clip_from_eye, clip_from_object;
  mat4 object_from_eye, eye_from_clip;
} ut;


// Declare variables for communication between vertex shader
// and fragment shader.
//
#ifdef _VERTEX_SHADER_

 /// Vertex Shader Inputs
//
```

```
layout (location = LOC_IN_POS) in vec4 in_vertex_o;
layout (location = LOC_IN_NORMAL) in vec3 in_normal_o;
#ifdef LOC_IN_INT1
layout (location = LOC_IN_INT1) in int in_color_idx;
#endif
//
// The CPU must provide these inputs.


 /// Vertex Shader Output
//
layout (location = 0) out Data_VG
{
  vec4 vertex_c, vertex_e;
  vec3 normal_e;
  int color_idx;
};

 /// The Vertex Shader
void
vs_main()
{
  // Transform the vertex object-space coordinate to clip space.
  //
  vertex_c = ut.clip_from_object * in_vertex_o;

  // Compute eye-space coordinates for vertex and normal.
  //
  vertex_e = ut.eye_from_object * in_vertex_o;
  normal_e = normalize( mat3(ut.eye_from_object) * in_normal_o );

  color_idx = in_color_idx;
}
#endif

#ifdef _GEOMETRY_SHADER_

layout ( triangles ) in;
layout ( triangle_strip, max_vertices = 3 ) out;

layout (location = 0) in Data_VG
{
  vec4 vertex_c, vertex_e;
  vec3 normal_e;
  int color_idx;
} In[];

layout (location = 0) out Data_GF
{
  vec4 vertex_e;
  vec3 normal_e;
  int color_idx;
} Out;
```

```
void
gs_main()
{
  int i_last = 2 - gl_PrimitiveIDIn % 2;
  if ( !opt_tryout2 && In[i_last].color_idx == 0 ) return;

  for ( int i=0; i<3; i++ )
    {
      gl_Position = In[i].vertex_c;
      Out.color_idx = In[i_last].color_idx;
      Out.vertex_e = In[i].vertex_e;
      Out.normal_e = In[i].normal_e;
      EmitVertex();
    }
  EndPrimitive();
}

#endif

#ifdef _FRAGMENT_SHADER_

layout (location = 0) in Data_VF
{
  vec4 vertex_e;
  vec3 normal_e;
  flat int color_idx;
};

// Fragment Shader Output
//
layout (location = 0) out vec4 frag_color;
//
// This will get written to frame buffer if tests pass, such as the
// depth (z-buffer) test.

void
fs_main()
{
  // Vector from fragment to light.
  vec3 vec_vl = uni_light.position.xyz - vertex_e.xyz;

  // Distance squared.
  float dist_sq = dot( vec_vl, vec_vl );

  // False if the light illuminates the side we can't see.
  bool lit_side = dot( normal_e, vec_vl )>0 == dot( normal_e, -vertex_e.xyz )>0;

  // Amount of light reaching the fragment.

  /// SOLUTION -- Problem 2
  //
  //  Don't set the attenuation to zero if primitive is lit from behind.
  //
  float attenuation = abs( dot( normal_e, vec_vl ) / dist_sq );
```

```
  // Material color.
  vec4 mat_color = gl_FrontFacing ? uc.front[color_idx] : uc.back[color_idx];

  /// SOLUTION -- Problem 2
  if ( !lit_side )
    {
      // Blend front and back colors, and darken them to account for
      // light going through material.
      //
      mat_color = ( uc.front[color_idx] + uc.back[color_idx] ) * 0.25;
    }

  // Compute lighted color.
  frag_color = mat_color * uni_light.color * attenuation;
  //
  // One important thing fragment shaders do--but not this one--is
  // apply textures.
}
#endif
```

/// # LSU EE 4702-1 (Fall 2024), GPU Programming
//
/// **Homework 4 -- SOLUTION**
//
//  Modify this file AND hw04-shdr.cc
//  Assignment: https://www.ece.lsu.edu/koppel/gpup/2024/hw04.pdf

```
// Include files provided for this course.
//
#define MAIN_INCLUDE
#include <vhelper.h>

#include <gp/coord.h>
#include <gp/pstring.h>
#include <gp/misc.h>
#include <gp/colors.h>

#include <vutil-texture.h>
#include <vutil-pipeline.h>
#include "shapes.h"

struct Uni_Light_Simple {
  vec4 position;
  vec4 color;
};

struct Uni_Misc {
  ivec4 opt_tryout;
};


constexpr int ncolors = 10;
struct HW03_Colors {
  pColor front[ncolors], back[ncolors];
};


struct Hyperboloid_Info {

  pCoor bot_ctr_pos, top_ctr_pos;
  pCoor line_0_bot_pos;
  pVect line_0_dir;

  int n_sides_constructed;
  int serial;
};


class World {
public:
  World(pVulkan_Helper &vh)
```

```
  :vh(vh),ff_state(vh.qs),frame_timer(vh.frame_timer),shapes(ff_state),
   transform(vh.qs){}
void setup_and_run();
void render(vk::CommandBuffer& cb);
void keyboard_handle();

// Class providing utilities, such as showing text.
//
pVulkan_Helper& vh;
VFixed_Function_State_Manager ff_state;

// Class for showing frame timing.
//
pFrame_Timer& frame_timer;

Shapes shapes;

// Class for easy keyboard control of variables.
//
pVariable_Control variable_control;

pCoor light_location;
float opt_light_intensity;
enum { MI_Eye, MI_Light, MI_Ball, MI_Ball_V, MI_COUNT } opt_move_item;

pCoor eye_location, eye_initial_location;
pVect eye_direction;
pNorm eye_initial_direction;

VTransform transform;
VBufferV<Uni_Light_Simple> uni_light_simple;
VBufferV<Uni_Misc> uni_misc;
VPipeline pipe_plain;
VVertex_Buffer_Set bset_plain;

bool opt_tryout1, opt_tryout2;  // For ad-hoc experiments.

void scene_setup_1();
void scene_setup_2();
double world_time, time_last_update;
bool opt_pause;

VBufferV<HW03_Colors> uni_hw03_colors;

/// 2024 Homework 4

enum Pipeline_Variant { PV_Individ, PV_Strip, PV_SIZE };
int pipeline_variant;

Hyperboloid_Info hyperb_info;
void hyperb_setup(bool r);
VPipeline pipe_hyperb;
VPipeline pipe_hyperb_strip;
VVertex_Buffer_Set bset_hyperb;
```

```
  int hyperb_serial;
  int hyperb_setup_n;

  int opt_n_sides;

};

const char* const pipeline_variant_str[] =
  { "INDIVID", "STRIP" };



void
World::scene_setup_1()
{
  hyperb_setup(false);
}

void
World::scene_setup_2()
{
  hyperb_setup(true);
}

float
prand(float max)
{
  double r = random() * (1.0 / RAND_MAX );
  return r * max;
}

float
prand(float min, float max)
{
  assert( min <= max );
  return min + prand( max - min );
}

pVect
norm_rand()
{
  double theta = prand(2*M_PI);
  double eta = prand(2*M_PI);
  pVect az(0,0,1);
  pVect a = pMatrix_Rotation(az,theta) * pVect(1,0,0);
  pVect a2 = cross(az,a);
  pVect b = pMatrix_Rotation(a2,eta) * a;
  float l = b.mag();
  assert( fabs(l-1) < 0.0001 );
  return b;
}

pVect vec_rand(float r1) { return prand(r1) * norm_rand(); }
pVect vec_rand(float r1, float r2) { return prand(r1,r2) * norm_rand(); }
```

```
pVect vec_rand(float rmin, float rmax, pVect axis)
{
  pNorm v = pMatrix_Rotation(axis.normal(),prand(2*M_PI)) * axis.orthogonal();
  return prand(rmin,rmax) * v;
}


void
World::hyperb_setup(bool reset)
{
  world_time = 0;
  time_last_update = time_wall_fp();
  if ( reset ) hyperb_setup_n = 0;
  hyperb_setup_n++;

  Hyperboloid_Info& h = hyperb_info;
  float s = 0.6;

  h.n_sides_constructed = 0;

  switch ( hyperb_setup_n ) {
  case 1:
    // Largish
    h.top_ctr_pos    = pCoor(2.78,0.83,0.89);
    h.bot_ctr_pos    = pCoor(2.68,-1.03,0.52);
    h.line_0_bot_pos = pCoor(2.82,-0.35,-2.95);
    h.line_0_dir     = pNorm(0.67,0.27,0.69);
    break;
  case 2:
    // Food display shape.
    h.top_ctr_pos    = pCoor(2.95,0.42,-0.31);
    h.bot_ctr_pos    = pCoor(2.96,-0.85,-0.42);
    h.line_0_bot_pos = pCoor(2.20,-0.68,-2.47);
    h.line_0_dir     = pNorm(0.69,0.18,0.70);
    break;

  default:
    hyperb_info.bot_ctr_pos = pCoor(3,-1,0) + vec_rand(1);
    h.top_ctr_pos =
      h.bot_ctr_pos + pVect(0,prand(1*s,5*s),0) + vec_rand(0,s,pVect(0,1,0));
    pVect vz(h.bot_ctr_pos,h.top_ctr_pos);
    h.line_0_bot_pos = h.bot_ctr_pos + vec_rand(2*s,6*s,vz);
    pCoor line_0_top_pos = hyperb_info.top_ctr_pos + vec_rand(2*s,6*s,vz);
    hyperb_info.line_0_dir = pNorm(h.line_0_bot_pos,line_0_top_pos);
    break;
  }

  printf("h.top_ctr_pos = pCoor(%.2f,%.2f,%.2f);\n",
         h.top_ctr_pos.x, h.top_ctr_pos.y,h.top_ctr_pos.z);
  printf("h.bot_ctr_pos = pCoor(%.2f,%.2f,%.2f);\n",
         h.bot_ctr_pos.x, h.bot_ctr_pos.y,h.bot_ctr_pos.z);
  printf("h.line_0_bot_pos = pCoor(%.2f,%.2f,%.2f);\n",
         h.line_0_bot_pos.x, h.line_0_bot_pos.y,h.line_0_bot_pos.z);
```

```
  printf("h.line_0_dir = pNorm(%.2f,%.2f,%.2f);\n",
         h.line_0_dir.x, h.line_0_dir.y,h.line_0_dir.z);




}


void
World::setup_and_run()
{
  // Setup Vulkan context, etc.
  vh.init();
  vh.display_cb_set([&](){});
  vh.cbs_cmd_record.push_back( [&](vk::CommandBuffer& cb){ render(cb); });

  pipeline_variant = 0;
  opt_tryout1 = opt_tryout2 = false;
  eye_initial_location = pCoor( 3, .5, 8.6) ;
  eye_location = eye_initial_location;
  eye_initial_direction = pVect(0,0,-1);
  eye_direction = eye_initial_direction;

  opt_light_intensity = 22;
  light_location = pCoor( -6.6, 8.0, 2.5 );

  opt_move_item = MI_Eye;

  uni_misc.init(vh.qs,vk::BufferUsageFlagBits::eUniformBuffer);
  uni_misc->opt_tryout = ivec4(opt_tryout1,opt_tryout2,0,0);

  uni_light_simple.init(vh.qs,vk::BufferUsageFlagBits::eUniformBuffer);

  uni_hw03_colors.init(vh.qs,vk::BufferUsageFlagBits::eUniformBuffer);

  hyperb_info.serial = 0;
  hyperb_serial = -1;
  hyperb_setup_n = 0;
  opt_n_sides = 49;
  variable_control.insert(opt_n_sides,"n_sides",1,3);

  variable_control.insert(opt_light_intensity,"Light Intensity");

  opt_pause = false;
  scene_setup_1();

  // Start the graphics. The function below does not return until the
  // user exits by closing the window.
  //
  vh.message_loop_spin();
  //
  // At this point the user exited and so it's time to
  // clean up.
```

```
  uni_light_simple.destroy();
  uni_misc.destroy();
  pipe_plain.destroy();
  bset_plain.destroy();
  shapes.destroy();
  transform.destroy();

  uni_hw03_colors.destroy();

  // 2024 Homework 4
  pipe_hyperb.destroy();
  pipe_hyperb_strip.destroy();
  bset_hyperb.destroy();

  vh.finish();
}

void
World::render(vk::CommandBuffer& cb)
{
  // This routine called whenever window needs to be updated.

  // Get any waiting keyboard commands.
  //
  keyboard_handle();

  /// Frame Buffer Informational Messages
  //
  //  Print messages using utility functions provided for this course.
  //
  vh.fbprintf("%s\n",frame_timer.frame_rate_text_get());
  const double time_now = time_wall_fp();
  const bool blink_visible = int64_t(time_now*2) & 1;
# define BLINK(txt,pad) ( blink_visible ? txt : pad )
  vh.fbprintf
    ("Compiled: %s\n",
#ifdef __OPTIMIZE__
     "WITH OPTIMIZATION"
#else
     BLINK("WITHOUT OPTIMIZATION","")
#endif
     );

  vh.fbprintf
    ("Eye location: [%5.1f, %5.1f, %5.1f]  "
     "Eye direction: [%+.2f, %+.2f, %+.2f]\n",
     eye_location.x, eye_location.y, eye_location.z,
     eye_direction.x, eye_direction.y, eye_direction.z);

  vh.fbprintf
    ("Light location: [%5.1f, %5.1f, %5.1f]\n",
     light_location.x, light_location.y, light_location.z);
```

```
vh.fbprintf
  ("Pipeline Variant: %s ('v')  Tryout 1: %s  ('y')  Tryout 2: %s  ('Y')  \n",
   pipeline_variant_str[pipeline_variant],
   opt_tryout1 ? BLINK("ON ","   ") : "OFF",
   opt_tryout2 ? BLINK("ON ","   ") : "OFF");


pVariable_Control_Elt* const cvar = variable_control.current;
vh.fbprintf("VAR %s = %.5f  (TAB or ``' to change, +/- to adjust)\n",
            cvar->name,cvar->get_val());


// ---------------------------------------------------------------------
///
/// Specification of Transformation Matrices
///

pMatrix eye_from_global =
  pMatrix_Rotation(eye_direction,pVect(0,0,-1))
  * pMatrix_Translate(-eye_location);

transform.eye_from_global_set( eye_from_global );

/// Setup Projection Transformation: Eye Space -> Clip Space
//
const int win_width = vh.s_extent.width;
const int win_height = vh.s_extent.height;
const float aspect = float(win_width) / win_height;
const float n_dist = 0.01;
const float xr = .8 * n_dist;

// Frustum: left, right, bottom, top, near, far
transform.clip_from_eye_set
  ( pMatrix_Frustum( -xr, xr,               // left, right
                     -xr/aspect, xr/aspect,  // bottom, top
                     n_dist, 5000            // near, far
                     ) );

/// Lighting
//
uni_light_simple->color = color_white * opt_light_intensity;
uni_light_simple->position = transform.eye_from_global * light_location;
uni_light_simple.to_dev();

uni_misc.to_dev();

//
// ---------------------------------------------------------------------

if ( !pipe_plain )
  pipe_plain
    .init( vh.qs )
    .ds_uniform_use( "BIND_LIGHT_SIMPLE", uni_light_simple )
    .ds_uniform_use( "BIND_MISC", uni_misc )
```

```
        .shader_inputs_info_set<pCoor,pNorm,pColor>()
        .shader_code_set
        ("demo-03-shdr.cc", "vs_main();", nullptr, "fs_main();")
        .topology_set( vk::PrimitiveTopology::eTriangleList )
        .create();

    // Reset buffers in this buffer set.
    //
    bset_plain.reset(pipe_plain);

    // Object-space coordinates of a triangle.
    //
    pCoor p0 = { 0, 0,  0 };
    pCoor p1 = { 9, 6, -9 };
    pCoor p2 = { 0, 5, -5 };

    // Find triangle normal using cross product function (in coord.h).
    //
    pNorm tri_norm = cross( p0, p1, p2 );
    pColor color_tri( .2, .900, .1 ); // Red, Green, Blue; Values in [0,1]

    // Insert vertex coordinates into buffer set (bset_plain).
    //
    bset_plain << p0 << p1 << p2;

    // Insert vertex colors and normals.
    //
    bset_plain << color_tri << color_red << color_blue;
    bset_plain << tri_norm << tri_norm << tri_norm;
    //
    // All three vertices here have the same normal.

    // Add a square consisting of a red and green triangle.
    //
    bset_plain << pCoor(-2,0,-2) << pCoor(-4,2,-2) << pCoor(-4,0,-2);
    bset_plain << color_red << color_red << color_red;
    bset_plain << pCoor(-2,0,-2) << pCoor(-2,2,-2) << pCoor(-4,2,-2);
    bset_plain << color_green << color_green << color_green;
    pNorm snorm = cross( pCoor(-4,0,-2), pCoor(-4,2,-2), pCoor(-2,0,-2) );
    bset_plain << snorm << snorm << snorm << snorm << snorm << snorm;

    bset_plain.to_dev();
    pipe_plain.ds_set( transform );
    pipe_plain.record_draw(cb, bset_plain);

    // Update theta_0, the angle by which the turbines should be at.
    //
    if ( !opt_pause ) world_time += time_now - time_last_update;
    time_last_update = time_now;
    const float omega = 1;
    const float theta_0 = world_time * omega;

    Hyperboloid_Info& h = hyperb_info;
```

```
    // Values used in constructing the turbines.
    //

    if ( true )
      {
        uni_hw03_colors->front[0] = color_red;
        uni_hw03_colors->back[0] = color_red;

        uni_hw03_colors->front[1] = color_lsu_spirit_purple;
        uni_hw03_colors->back[1] = color_lsu_spirit_gold;
        uni_hw03_colors->front[2] = color_lsu_business_purple;
        uni_hw03_colors->back[2] = color_gold;
        uni_hw03_colors->front[3] = color_salmon;
        uni_hw03_colors->back[3] = color_rosy_brown;
        uni_hw03_colors->front[4] = color_aquamarine;
        uni_hw03_colors->back[4] = color_peru;;

        uni_hw03_colors->front[5] = color_navajo_white;
        uni_hw03_colors->back[5] = color_white;

        uni_hw03_colors.to_dev();

      }

    if ( !pipe_hyperb )
      pipe_hyperb
        .init( vh.qs )
        .ds_uniform_use( "BIND_LIGHT_SIMPLE", uni_light_simple )
        .ds_uniform_use( "BIND_MISC", uni_misc )
        .ds_uniform_use( "BIND_HW03", uni_hw03_colors )
        .shader_inputs_info_set<pCoor,pNorm,int>()
        .shader_code_set
        ("hw04-shdr-sol.cc", "vs_main();", "gs_main();", "fs_main();")
        .topology_set( vk::PrimitiveTopology::eTriangleList )
        .create();

    if ( !pipe_hyperb_strip )
      pipe_hyperb_strip
        .init( vh.qs )
        .ds_uniform_use( "BIND_LIGHT_SIMPLE", uni_light_simple )
        .ds_uniform_use( "BIND_MISC", uni_misc )
        .ds_uniform_use( "BIND_HW03", uni_hw03_colors )
        .shader_inputs_info_set<pCoor,pNorm,int>()
        .shader_code_set
        ("hw04-shdr-sol.cc", "vs_main();", "gs_main();", "fs_main();")
        .topology_set( vk::PrimitiveTopology::eTriangleStrip )
        .create();

    if ( h.n_sides_constructed != opt_n_sides ) h.serial++;

    pCoor bot_ctr = h.bot_ctr_pos;
    pCoor top_ctr = h.top_ctr_pos;
    pCoor line_0_bot_pos = h.line_0_bot_pos;
    pVect line_0_dir = hyperb_info.line_0_dir;
```

```
pVect hyperb_vz(bot_ctr,top_ctr);
pNorm hyperb_az(hyperb_vz);
pVect bot_vx(bot_ctr,line_0_bot_pos);

pMatrix_Rotation rot(hyperb_az,theta_0);
pMatrix trot =
  pMatrix_Translate( bot_ctr ) * rot * pMatrix_Translate( -bot_ctr );

if ( !bset_hyperb || hyperb_serial != h.serial )
  {
    hyperb_serial = h.serial;
    int n_levels = opt_n_sides;
    int n_sides = opt_n_sides;
    h.n_sides_constructed = n_sides;

    pCoor line_0_top_pos =
      line_0_bot_pos
      + hyperb_az.magnitude / dot( hyperb_az, line_0_dir ) * line_0_dir;

    float delta_theta = 2 * M_PI / n_sides;
    pVect delta_z = (1.0/n_levels ) * hyperb_vz;

    pVect line_v(line_0_bot_pos,line_0_top_pos);
    pVect line_dv = (1.0/n_levels) * line_v;

    vector< vector<pCoor> > coords(n_levels+1);
    vector< vector<pNorm> > norms(n_levels+1);

    pVect top_vx(top_ctr,line_0_top_pos);

    for ( int lev = 0; lev <= n_levels; lev++ )
      {
        coords[lev].resize(n_sides+1);
        norms[lev].resize(n_sides+1);
      }

    // Compute vertex coordinates and normals and write them
    // to CPU containers coords and norms.
    //
    for ( int side = 0; side <= n_sides; side++ )
      {
        float theta = side * delta_theta;
        pMatrix_Rotation rot(hyperb_az,theta);
        pCoor line_bot_pos = bot_ctr + rot * bot_vx;
        pVect l_dv = rot * line_dv;

        for ( int lev = 0; lev <= n_levels; lev++ )
          {
            pCoor p_line = line_bot_pos + lev * l_dv;
            coords[lev][side] = p_line;
            pCoor ctr = bot_ctr + lev * delta_z;
            pVect tc = cross(hyperb_az,pVect(ctr,p_line));
            pNorm n = cross(tc,l_dv);
```

```
          norms[lev][side] = n;
        }
    }

  switch ( pipeline_variant ) {
  case PV_Individ:

    bset_hyperb.reset(pipe_hyperb);
    for ( int side = 0; side < n_sides; side++ )
      for ( int lev = 0; lev < n_levels; lev++ )
        {
          pCoor p00 = coords[lev][side];
          pCoor p01 = coords[lev][side+1];
          pCoor p10 = coords[lev+1][side];
          pCoor p11 = coords[lev+1][side+1];

          bset_hyperb << p00 << p01 << p10;
          bset_hyperb << p01 << p11 << p10;

          pNorm n00 = norms[lev][side];
          pNorm n01 = norms[lev][side+1];
          pNorm n10 = norms[lev+1][side];
          pNorm n11 = norms[lev+1][side+1];

          bset_hyperb << n00 << n01 << n10;
          bset_hyperb << n01 << n11 << n10;

          int c1 = side == 0 ? ( lev%2 ? 5 : 2 ) : 1;
          int c2 = side == 0 ? ( lev%2 ? 3 : 4 ) : 1;

          bset_hyperb << c1 << c1 << c1 << c2 << c2 << c2;
        }
    break;

  case PV_Strip:

    bset_hyperb.reset(pipe_hyperb_strip);

    /// Homework 4
    //
    //  Modify code in this switch and elsewhere if necessary to
    //  properly use triangle strip. Write vertex attributes to
    //  bset_hyperb.

    /// SOLUTION -- Problem 1
    //
    //  Each side loop iteration emits one triangle strip iteration.
    //  The lev loop ends at n_levels rather than n_levels.

    for ( int side = 0; side < n_sides; side++ )
      for ( int lev = 0; lev <= n_levels; lev++ )
        {
          // For triangle strip just emit two vertices per iteration ..
          // .. rather than six vertices.
```

```
                // Same two triangles, but using four fewer vertices!

                pCoor p00 = coords[lev][side];
                pCoor p01 = coords[lev][side+1];

                bset_hyperb << p00 << p01;

                pNorm n00 = norms[lev][side];
                pNorm n01 = norms[lev][side+1];

                bset_hyperb << n00 << n01;

                // Choose color index.
                //
                int c1 = lev == 0 ? 0 : side == 0 ? ( (lev+1)%2 ? 5 : 2 ) : 1;
                int c2 = lev == 0 ? 0 : side == 0 ? ( (lev+1)%2 ? 3 : 4 ) : 1;
                //
                // If lev == 0, set to zero to start triangle strip;
                // if side == 0, choose from among four colors to show structure;
                // otherwise, use color index 1.

                bset_hyperb << c1 << c2;
              }

          break;

        default: assert( false );
        }

        bset_hyperb.to_dev();
      }

    switch ( pipeline_variant ) {
    case PV_Individ:
      pipe_hyperb.ds_set( transform * trot );
      pipe_hyperb.record_draw(cb, bset_hyperb);
      break;
    case PV_Strip:
      pipe_hyperb_strip.ds_set( transform * trot );
      pipe_hyperb_strip.record_draw(cb, bset_hyperb);
      break;
    default: assert( false );
    }


    // Render Marker for Light Source
    //
    shapes.record_tetrahedron(cb,transform,light_location,0.2);

}



void
```

```
World::keyboard_handle()
{
  const int key = vh.keyboard_key_get();
  if ( !key ) return;
  pVect adjustment(0,0,0);
  pVect user_rot_axis(0,0,0);
  const bool kb_mod_s = vh.keyboard_shift;
  const bool kb_mod_c = vh.keyboard_control;
  const float move_amt = kb_mod_s ? 2.0 : kb_mod_c ? 0.08 : 0.4;

  switch ( key ) {
  case FB_KEY_LEFT: adjustment.x = -move_amt; break;
  case FB_KEY_RIGHT: adjustment.x = move_amt; break;
  case FB_KEY_PAGE_UP: adjustment.y = move_amt; break;
  case FB_KEY_PAGE_DOWN: adjustment.y = -move_amt; break;
  case FB_KEY_DOWN: adjustment.z = move_amt; break;
  case FB_KEY_UP: adjustment.z = -move_amt; break;
  case FB_KEY_DELETE: user_rot_axis.y = 1; break;
  case FB_KEY_INSERT: user_rot_axis.y =  -1; break;
  case FB_KEY_HOME: user_rot_axis.x = 1; break;
  case FB_KEY_END: user_rot_axis.x = -1; break;
  case 'n': bset_hyperb.reset(); break;
  case '1': scene_setup_1(); break;
  case '2': scene_setup_2(); break;
  case 'b': case 'B': opt_move_item = MI_Ball; break;
  case 'e': case 'E': opt_move_item = MI_Eye; break;
  case 'l': case 'L': opt_move_item = MI_Light; break;
  case 'p': case 'P': opt_pause = !opt_pause; break;
  case 'v': case 'V':
    hyperb_info.serial++;
    if ( ++pipeline_variant >= PV_SIZE ) pipeline_variant = 0;
    break;
  case 'y': opt_tryout1 = !opt_tryout1;
    uni_misc->opt_tryout.x = int(opt_tryout1);
    break;
  case 'Y': opt_tryout2 = !opt_tryout2;
    uni_misc->opt_tryout.y = int(opt_tryout2);
    break;
  case FB_KEY_TAB:
    if ( !kb_mod_s ) { variable_control.switch_var_right(); break; }
  case 96: variable_control.switch_var_left(); break;
  case '-':case '_': variable_control.adjust_lower(); break;
  case '+':case '=': variable_control.adjust_higher(); break;
  default: printf("Unknown key, %d\n",key); break;
  }

  // Update eye_direction based on keyboard command.
  //
  if ( user_rot_axis.x || user_rot_axis.y )
    {
      pMatrix_Rotation rotall(eye_direction,pVect(0,0,-1));
      user_rot_axis *= invert(rotall);
      eye_direction *= pMatrix_Rotation(user_rot_axis, M_PI * 0.03);
    }
```

```
  // Update eye_location based on keyboard command.
  //
  if ( adjustment.x || adjustment.y || adjustment.z )
    {
      const double angle =
        fabs(eye_direction.y) > 0.99
        ? 0 : atan2(eye_direction.x,-eye_direction.z);
      pMatrix_Rotation rotall(pVect(0,1,0),-angle);
      adjustment *= rotall;

      switch ( opt_move_item ){
      case MI_Light: light_location += adjustment; break;
      case MI_Eye: eye_location += adjustment; break;
      //  case MI_Ball: object_location += adjustment; break;
      default: break;
      }
    }
}


int
main(int argv, char **argc)
{
  pVulkan_Helper pvulkan_helper(argv,argc);
  World world(pvulkan_helper);

  world.setup_and_run();

  return 0;
}
```

```
/// LSU EE 4702-1 (Fall 2024), GPU Programming
//
 /// Homework 5 -- SOLUTION
//
//   Modify this file AND hw05.cc.
//   Assignment: https://www.ece.lsu.edu/koppel/gpup/2024/hw05.pdf

// Specify version of OpenGL Shading Language.
//
#version 460

#extension GL_GOOGLE_include_directive : enable

layout ( binding = BIND_MISC ) uniform Uni_Misc
{
  ivec4 opt_tryout;
  ivec4 opt_assign;
  float opt_tryoutf;
};

bool opt_tryout1 = bool(opt_tryout.x);
bool opt_tryout2 = bool(opt_tryout.y);
bool opt_tryout3 = bool(opt_tryout.z);
int  opt_tryouti = opt_tryout.w;
bool opt_texture = bool(opt_assign.x);


const int ncolors = 10;
layout ( binding = BIND_HW05 ) uniform I_can_forget_this_name_no_problem
{
  vec4 front[ncolors], back[ncolors];

  /// SOLUTION -- Problem 3
  //
  mat4 local_from_object;
  int s_sides, s_levels, n_sides, n_levels;
  float n_sides_inv, n_levels_inv;

} uc;


 /// Shader Uniforms
//
layout ( binding = BIND_LIGHT_SIMPLE ) uniform Uni_Light
{
  vec4 position;
  vec4 color;
} uni_light;


layout ( binding = BIND_TRANSFORM ) uniform Uni_Transform
{
  // Transformation Matrices
```

```
  mat4 eye_from_object, clip_from_eye, clip_from_object;
  mat4 object_from_eye, eye_from_clip;
} ut;

layout ( binding = BIND_HYPERB_COORDS ) buffer sc { vec4 hyperb_coords[]; };
layout ( binding = BIND_HYPERB_NORMS ) buffer sn { vec4 hyperb_norms[]; };


// Declare variables for communication between vertex shader
// and fragment shader.
//
#ifdef _VERTEX_SHADER_

 /// Vertex Shader Inputs
//
// None.


 /// Vertex Shader Output
//
layout (location = 0) out Data_VG
{
  vec4 padding;
};

 /// The Vertex Shader
void
vs_points()
{
  // Pre-Defined Input Variable
  //
  // Reference: OGSL 4.6 Section 7.1.1
  //
  // in int gl_VertexIndex;
  // Integer index of this vertex. Ranges from 0 to n_vertices - 1,
  // where n_vertices is the number of vertices sent to the rendering
  // pipeline.

  // Homework 5: If code is added here, don't forget to
  // declare vertex shader outputs and geometry shader inputs.

}
#endif

#ifdef _GEOMETRY_SHADER_

layout ( points ) in;

 /// SOLUTION -- Problem 3
//
//   Increase max_vertices from 3 to 4.
//
layout ( triangle_strip, max_vertices = 4 ) out;
```

```
layout (location = 0) in Data_VG
{
  vec4 padding;
} In[];

layout (location = 0) out Data_GF
{
  vec4 vertex_e;
  vec3 normal_e;
  int color_idx;
  vec2 tcoor;
} Out;

void
gs_points()
{
  // Pre-Defined Input Variable
  //
  // Reference: OGSL 4.6 Section 7.1.4
  //
  // in int gl_PrimitiveIDIn;
  //
  // Integer index of this primitive. Ranges from 0 to n_primitives-1,
  // where n_primitives depends on the grouping (topology). For
  // vk::PrimitiveTopology::ePointList n_primitives is the same as the
  // number of vertices sent to the rendering pipeline in the draw
  // command.

  /// SOLUTION -- Problem 3
  //
  //  This invocation of the geometry shader will emit two triangles,
  //  something like the loop body in the PV_Individ case in
  //  World::render.
  //
  //  First, compute the level and side based on gl_PrimitiveIDIn.
  //
  int level0 = gl_PrimitiveIDIn % uc.n_levels;
  int side0 = gl_PrimitiveIDIn / uc.n_levels;
  //
  //  Because the draw command specified n_levels * n_sides "vertices"
  //  the value of gl_PrimitiveIDIn will range from 0 to n_levels *
  //  n_sides -1, and so the value of side0 will range from 0 to
  //  n_sides-1, and clearly level0 will range from 0 to n_levels-1.

  //  The two triangles will be drawn using a triangle strip. This
  //  can conveniently by done using a loop nest:
  //
  for ( int dl = 0; dl<2; dl++ )
    for ( int ds = 0; ds < 2; ds ++ )
      {
        int level = level0 + dl;
        int side = side0 + ds;

        // Compute the array index in the same way idxf would.
```

```
        //
        int idx = level + side * uc.s_levels;

        // Retrieve the coordinate and normal.
        //
        vec4 p = hyperb_coords[ idx ];
        vec4 n = hyperb_norms[ idx ];

        // Convert coordinate to clip space and eye space, and normal
        // to eye space.
        //
        gl_Position = ut.clip_from_object * p;
        Out.vertex_e = ut.eye_from_object * p;
        Out.normal_e = normalize( mat3(ut.eye_from_object) * n.xyz );

        // Compute the color index.
        //
        Out.color_idx = side0 != 0 ? 1 :
          ds == 0 ? ( level0 % 2 == 0 ? 2 : 5 ) : ( level0 % 2 == 0 ? 4 : 3 );

        if ( opt_tryout1 )
          {
            // PARTIAL CREDIT Texture coordinates.
            Out.tcoor = vec2( 10 * side * uc.n_sides_inv,
                              1 - level * uc.n_levels_inv );
          }
        else
          {
            // FULL CREDIT Texture coordinates.
            vec4 ploc = uc.local_from_object * p;
            float theta = atan( ploc.y, ploc.x );
            Out.tcoor = vec2( 1.5 * theta, 1 - level * uc.n_levels_inv );
          }

        EmitVertex();
      }
  EndPrimitive();
}

#endif

#ifdef _FRAGMENT_SHADER_

layout ( binding = BIND_TEXUNIT ) uniform sampler2D tex_unit_0;

layout (location = 0) in Data_GF
{
  vec4 vertex_e;
  vec3 normal_e;
  flat int color_idx;
  vec2 tcoor;
};

// Fragment Shader Output
```

```
//
layout (location = 0) out vec4 frag_color;
//
// This will get written to frame buffer if tests pass, such as the
// depth (z-buffer) test.

void
fs_points()
{
  // Homework 5:  NO NEED to modify this routine.

  vec4 texel = opt_texture ? texture(tex_unit_0,tcoor) : vec4(1,1,1,1);

  // Vector from fragment to light.
  vec3 vec_vl = uni_light.position.xyz - vertex_e.xyz;

  // Distance squared.
  float dist_sq = dot( vec_vl, vec_vl );

  // False if the light illuminates the side we can't see.
  bool lit_side = dot( normal_e, vec_vl )>0 == dot( normal_e, -vertex_e.xyz )>0;

  float ambient_attn = 0.01;

  // Amount of light reaching the fragment.
  float attenuation = lit_side ? abs( dot( normal_e, vec_vl ) / dist_sq ) : 0;

  // Material color.
  vec4 mat_color = gl_FrontFacing ? uc.front[color_idx] : uc.back[color_idx];

  // Compute lighted color.
  frag_color =
    texel * mat_color * uni_light.color * ( attenuation + ambient_attn );
  //
  // One important thing fragment shaders do--but not this one--is
  // apply textures.
}
#endif
```

```
/// LSU EE 4702-1 (Fall 2024), GPU Programming
//
 /// Homework 5 -- SOLUTION
//
//   Modify this file AND hw05-shdr.cc
//   Assignment: https://www.ece.lsu.edu/koppel/gpup/2024/hw05.pdf



// Include files provided for this course.
//
#define MAIN_INCLUDE
#include <vhelper.h>

#include <gp/coord.h>
#include <gp/pstring.h>
#include <gp/misc.h>
#include <gp/colors.h>

#include <vutil-texture.h>
#include <vutil-pipeline.h>
#include "shapes.h"

struct Uni_Light_Simple {
  vec4 position;
  vec4 color;
};

struct Uni_Misc {
  ivec4 opt_tryout;
  ivec4 opt_assign;
  float opt_tryoutf;
};


constexpr int ncolors = 10;
struct HW05_Uniform {
  pColor front[ncolors], back[ncolors];

  /// SOLUTION -- Problem 3
  //
  //  Declare members for additional data needed by geometry shader.
  //
  mat4 local_from_object;
  int s_sides, s_levels, n_sides, n_levels;
  float n_sides_inv, n_levels_inv;

};


struct Hyperboloid_Info {

  pCoor bot_ctr_pos, top_ctr_pos;
```

```
  pCoor line_0_bot_pos;
  pVect line_0_dir;

  int n_sides_constructed;
  int serial;
};


class World {
public:
  World(pVulkan_Helper &vh)
    :vh(vh),ff_state(vh.qs),frame_timer(vh.frame_timer),shapes(ff_state),
     transform(vh.qs){}
  void setup_and_run();
  void render(vk::CommandBuffer& cb);
  void keyboard_handle();

  // Class providing utilities, such as showing text.
  //
  pVulkan_Helper& vh;
  VFixed_Function_State_Manager ff_state;

  // Class for showing frame timing.
  //
  pFrame_Timer& frame_timer;

  Shapes shapes;

  // Class for easy keyboard control of variables.
  //
  pVariable_Control variable_control;

  pCoor light_location;
  float opt_light_intensity;
  enum { MI_Eye, MI_Light, MI_Ball, MI_Ball_V, MI_COUNT } opt_move_item;

  pCoor eye_location, eye_initial_location;
  pVect eye_direction;
  pNorm eye_initial_direction;

  VTransform transform;
  VBufferV<Uni_Light_Simple> uni_light_simple;
  VBufferV<Uni_Misc> uni_misc;
  VPipeline pipe_plain;
  VVertex_Buffer_Set bset_plain;

  bool opt_tryout1, opt_tryout2, opt_tryout3;  // For ad-hoc experiments.

  void scene_setup_1();
  void scene_setup_2();
  double world_time, time_last_update;
  bool opt_pause;

  VBufferV<HW05_Uniform> uni_hw05;
```

```
  /// 2024 Homework 4

  enum Pipeline_Variant { PV_Individ, PV_Strip, PV_Points, PV_SIZE };
  int pipeline_variant;

  Hyperboloid_Info hyperb_info;
  void hyperb_setup(bool r);
  VPipeline pipe_hyperb;
  VPipeline pipe_hyperb_strip;
  VVertex_Buffer_Set bset_hyperb;
  int hyperb_serial;
  int hyperb_setup_n;

  int opt_n_sides;


  /// 2024 Homework 5
  VBufferVV<vec4> sto_hyperb_coords;
  VBufferVV<vec4> sto_hyperb_norms;
  VPipeline pipe_hyperb_points;
  vk::Sampler sampler;
  VTexture texture_id_syllabus;
  bool opt_texture;

};

const char* const pipeline_variant_str[] =
  { "INDIVID", "STRIP", "POINTS" };



void
World::scene_setup_1()
{
  hyperb_setup(false);
}

void
World::scene_setup_2()
{
  hyperb_setup(true);
}

float
prand(float max)
{
  double r = random() * (1.0 / RAND_MAX );
  return r * max;
}

float
prand(float min, float max)
{
```

```
  assert( min <= max );
  return min + prand( max - min );
}

pVect
norm_rand()
{
  double theta = prand(2*M_PI);
  double eta = prand(2*M_PI);
  pVect az(0,0,1);
  pVect a = pMatrix_Rotation(az,theta) * pVect(1,0,0);
  pVect a2 = cross(az,a);
  pVect b = pMatrix_Rotation(a2,eta) * a;
  float l = b.mag();
  assert( fabs(l-1) < 0.0001 );
  return b;
}

pVect vec_rand(float r1) { return prand(r1) * norm_rand(); }
pVect vec_rand(float r1, float r2) { return prand(r1,r2) * norm_rand(); }

pVect vec_rand(float rmin, float rmax, pVect axis)
{
  pNorm v = pMatrix_Rotation(axis.normal(),prand(2*M_PI)) * axis.orthogonal();
  return prand(rmin,rmax) * v;
}


void
World::hyperb_setup(bool reset)
{
  world_time = 0;
  time_last_update = time_wall_fp();
  if ( reset ) hyperb_setup_n = 0;
  hyperb_setup_n++;

  Hyperboloid_Info& h = hyperb_info;
  float s = 0.6;

  h.n_sides_constructed = 0;

  switch ( hyperb_setup_n ) {
  case 1:
    // Largish
    h.top_ctr_pos    = pCoor(2.78,0.83,0.89);
    h.bot_ctr_pos    = pCoor(2.68,-1.03,0.52);
    h.line_0_bot_pos = pCoor(2.82,-0.35,-2.95);
    h.line_0_dir     = pNorm(0.67,0.27,0.69);
    break;
  case 2:
    // Food display shape.
    h.top_ctr_pos    = pCoor(2.95,0.42,-0.31);
    h.bot_ctr_pos    = pCoor(2.96,-0.85,-0.42);
    h.line_0_bot_pos = pCoor(2.20,-0.68,-2.47);
```

```
    h.line_0_dir     = pNorm(0.69,0.18,0.70);
    break;

  default:
    hyperb_info.bot_ctr_pos = pCoor(3,-1,0) + vec_rand(1);
    h.top_ctr_pos =
      h.bot_ctr_pos + pVect(0,prand(1*s,5*s),0) + vec_rand(0,s,pVect(0,1,0));
    pVect vz(h.bot_ctr_pos,h.top_ctr_pos);
    h.line_0_bot_pos = h.bot_ctr_pos + vec_rand(2*s,6*s,vz);
    pCoor line_0_top_pos = hyperb_info.top_ctr_pos + vec_rand(2*s,6*s,vz);
    hyperb_info.line_0_dir = pNorm(h.line_0_bot_pos,line_0_top_pos);
    break;
  }

  printf("h.top_ctr_pos = pCoor(%.2f,%.2f,%.2f);\n",
         h.top_ctr_pos.x, h.top_ctr_pos.y,h.top_ctr_pos.z);
  printf("h.bot_ctr_pos = pCoor(%.2f,%.2f,%.2f);\n",
         h.bot_ctr_pos.x, h.bot_ctr_pos.y,h.bot_ctr_pos.z);
  printf("h.line_0_bot_pos = pCoor(%.2f,%.2f,%.2f);\n",
         h.line_0_bot_pos.x, h.line_0_bot_pos.y,h.line_0_bot_pos.z);
  printf("h.line_0_dir = pNorm(%.2f,%.2f,%.2f);\n",
         h.line_0_dir.x, h.line_0_dir.y,h.line_0_dir.z);




}


void
World::setup_and_run()
{
  // Setup Vulkan context, etc.
  vh.init();
  vh.display_cb_set([&](){});
  vh.cbs_cmd_record.push_back( [&](vk::CommandBuffer& cb){ render(cb); });

  pipeline_variant = 0;
  opt_tryout1 = opt_tryout2 = opt_tryout3 = false;
  uni_misc->opt_tryout.w = 1;
  uni_misc->opt_tryoutf = 1;
  eye_initial_location = pCoor( 3, .5, 8.6) ;
  eye_location = eye_initial_location;
  eye_initial_direction = pVect(0,0,-1);
  eye_direction = eye_initial_direction;

  opt_light_intensity = 22;
  light_location = pCoor( -6.6, 8.0, 2.5 );

  opt_move_item = MI_Eye;

  sto_hyperb_coords.init(vh.qs,vk::BufferUsageFlagBits::eStorageBuffer );
  sto_hyperb_norms.init(vh.qs,vk::BufferUsageFlagBits::eStorageBuffer );
```

```
    opt_texture = true;
    sampler = vh.qs.dev.createSampler
      ( { {},
          vk::Filter::eLinear, vk::Filter::eLinear,
          vk::SamplerMipmapMode::eLinear,
          // Also: eRepeat, eMirroredRepeat, eClampToEdge, etc.
          vk::SamplerAddressMode::eRepeat,
          vk::SamplerAddressMode::eRepeat,
          vk::SamplerAddressMode::eRepeat,
          0.0f,
          true, // anisotropyEnable,
          16.0f,
          false,
          vk::CompareOp::eNever,
          0.0f, VK_LOD_CLAMP_NONE,  // min and max LOD
          vk::BorderColor::eFloatOpaqueBlack } );

    texture_id_syllabus.init
      (vh.qs, P_Image_Read("../../../../vulkan/gpup.png",255) );

    uni_misc.init(vh.qs,vk::BufferUsageFlagBits::eUniformBuffer);
    uni_misc->opt_tryout =
      ivec4(opt_tryout1,opt_tryout2,opt_tryout3, uni_misc->opt_tryout.w);
    uni_misc->opt_assign = ivec4(opt_texture,0,0,0);

    uni_light_simple.init(vh.qs,vk::BufferUsageFlagBits::eUniformBuffer);

    uni_hw05.init(vh.qs,vk::BufferUsageFlagBits::eUniformBuffer);

    hyperb_info.serial = 0;
    hyperb_serial = -1;
    hyperb_setup_n = 0;
    opt_n_sides = 81;
    variable_control.insert(opt_n_sides,"n_sides",1,3);

    variable_control.insert(opt_light_intensity,"Light Intensity");

    variable_control.insert(uni_misc->opt_tryout.w,"opt_tryouti");
    variable_control.insert(uni_misc->opt_tryoutf,"opt_tryoutf");

    opt_pause = false;
    scene_setup_1();

    // Start the graphics. The function below does not return until the
    // user exits by closing the window.
    //
    vh.message_loop_spin();
    //
    // At this point the user exited and so it's time to
    // clean up.

    uni_light_simple.destroy();
    uni_misc.destroy();
    pipe_plain.destroy();
```

```
  bset_plain.destroy();
  shapes.destroy();
  transform.destroy();

  uni_hw05.destroy();

  // 2024 Homework 4
  pipe_hyperb.destroy();
  pipe_hyperb_strip.destroy();
  bset_hyperb.destroy();

  pipe_hyperb_points.destroy();
  texture_id_syllabus.destroy();
  vh.dev.destroySampler(sampler);
  sto_hyperb_coords.destroy();
  sto_hyperb_norms.destroy();

  vh.finish();
}

void
World::render(vk::CommandBuffer& cb)
{
  // This routine called whenever window needs to be updated.

  // Get any waiting keyboard commands.
  //
  keyboard_handle();

  /// Frame Buffer Informational Messages
  //
  //  Print messages using utility functions provided for this course.
  //
  vh.fbprintf("%s\n",frame_timer.frame_rate_text_get());
  const double time_now = time_wall_fp();
  const bool blink_visible = int64_t(time_now*2) & 1;
# define BLINK(txt,pad) ( blink_visible ? txt : pad )
  vh.fbprintf
    ("Compiled: %s\n",
#ifdef __OPTIMIZE__
     "WITH OPTIMIZATION"
#else
     BLINK("WITHOUT OPTIMIZATION","")
#endif
     );

  vh.fbprintf
    ("Eye location: [%5.1f, %5.1f, %5.1f]  "
     "Eye direction: [%+.2f, %+.2f, %+.2f]\n",
     eye_location.x, eye_location.y, eye_location.z,
     eye_direction.x, eye_direction.y, eye_direction.z);

  vh.fbprintf
    ("Texture: %s ('t')  Light location: [%5.1f, %5.1f, %5.1f]\n",
```

```
    opt_texture ? "ON " : BLINK("OFF","   "),
    light_location.x, light_location.y, light_location.z);

  vh.fbprintf
    ("Pipeline Variant: %s ('v')  Tryout 1: %s ('y')  "
     "Tryout 2: %s  ('Y')  Tryout 3: %s ('z')\n",
     pipeline_variant_str[pipeline_variant],
     opt_tryout1 ? BLINK("ON ","   ") : "OFF",
     opt_tryout2 ? BLINK("ON ","   ") : "OFF",
     opt_tryout3 ? BLINK("ON ","   ") : "OFF");


  pVariable_Control_Elt* const cvar = variable_control.current;
  vh.fbprintf("VAR %s = %.5f  (TAB or ``' to change, +/- to adjust)\n",
              cvar->name,cvar->get_val());


  // ------------------------------------------------------------------------
  ///
  /// Specification of Transformation Matrices
  ///

  pMatrix eye_from_global =
    pMatrix_Rotation(eye_direction,pVect(0,0,-1))
    * pMatrix_Translate(-eye_location);

  transform.eye_from_global_set( eye_from_global );

  /// Setup Projection Transformation: Eye Space -> Clip Space
  //
  const int win_width = vh.s_extent.width;
  const int win_height = vh.s_extent.height;
  const float aspect = float(win_width) / win_height;
  const float n_dist = 0.01;
  const float xr = .8 * n_dist;

  // Frustum: left, right, bottom, top, near, far
  transform.clip_from_eye_set
    ( pMatrix_Frustum( -xr, xr,                // left, right
                       -xr/aspect, xr/aspect,  // bottom, top
                       n_dist, 5000            // near, far
                       ) );

  /// Lighting
  //
  uni_light_simple->color = color_white * opt_light_intensity;
  uni_light_simple->position = transform.eye_from_global * light_location;
  uni_light_simple.to_dev();

  uni_misc->opt_tryoutf += 0; // Hack to force update.

  uni_misc.to_dev();

  //
```

```
// ---------------------------------------------------------------------------

if ( !pipe_plain )
  pipe_plain
    .init( vh.qs )
    .ds_uniform_use( "BIND_LIGHT_SIMPLE", uni_light_simple )
    .ds_uniform_use( "BIND_MISC", uni_misc )
    .ds_use( sampler, texture_id_syllabus )
    .shader_inputs_info_set<pCoor,pNorm,pColor,pTCoor>()
    .shader_code_set
    ("demo-03-shdr.cc", "vs_main();", nullptr, "fs_main();")
    .topology_set( vk::PrimitiveTopology::eTriangleList )
    .create();

// Reset buffers in this buffer set.
//
bset_plain.reset(pipe_plain);

// Object-space coordinates of a triangle.
//
pCoor p0 = { 0, 0,  0 };
pCoor p1 = { 9, 6, -9 };
pCoor p2 = { 0, 5, -5 };

// Find triangle normal using cross product function (in coord.h).
//
pNorm tri_norm = cross( p0, p1, p2 );
pColor color_tri( .2, .900, .1 ); // Red, Green, Blue; Values in [0,1]

// Insert vertex coordinates into buffer set (bset_plain).
//
bset_plain << p0 << p1 << p2;
bset_plain << pTCoor(0,0) << pTCoor(1,1) << pTCoor(0,1);

// Insert vertex colors and normals.
//
bset_plain << color_tri << color_red << color_blue;
bset_plain << tri_norm << tri_norm << tri_norm;
//
// All three vertices here have the same normal.


/// Problem 1: Modify texture coordinates below.

// Add a square consisting of a light gray and green triangle.
//
bset_plain << pCoor(-1,-1,-2)  << pCoor(-5,3,-2)   << pCoor(-5,-1,-2);
bset_plain << color_light_gray << color_light_gray << color_light_gray;
bset_plain << pTCoor(1,1)      << pTCoor(0,0)      << pTCoor(0,1);
//
/// SOLUTION -- Problem 1 -- Above and Below
//
//  Correctly insert texture coordinates for the light gray triangle.
//
```

```
bset_plain << pCoor(-1,-1,-2) << pCoor(-1,3,-2) << pCoor(-5,3,-2);
bset_plain << color_green     << color_green    << color_green;
bset_plain << pTCoor(1,1)     << pTCoor(1,0)    << pTCoor(0,0);


pNorm snorm = cross( pCoor(-4,0,-2), pCoor(-4,2,-2), pCoor(-2,0,-2) );
bset_plain << snorm << snorm << snorm << snorm << snorm << snorm;

bset_plain.to_dev();
pipe_plain.ds_set( transform );
pipe_plain.record_draw(cb, bset_plain);

// Update theta_0, the angle by which the turbines should be at.
//
if ( !opt_pause ) world_time += time_now - time_last_update;
time_last_update = time_now;
const float omega = 1;
const float theta_0 = world_time * omega;

Hyperboloid_Info& h = hyperb_info;

// Values used in constructing the turbines.
//

if ( true )
  {
    uni_hw05->front[0] = color_red;
    uni_hw05->back[0] = color_red;

    uni_hw05->front[1] = color_lsu_spirit_gold;
    uni_hw05->back[1] = color_lsu_spirit_purple;
    uni_hw05->front[2] = color_lsu_business_purple;
    uni_hw05->back[2] = color_gold;
    uni_hw05->front[3] = color_salmon;
    uni_hw05->back[3] = color_rosy_brown;
    uni_hw05->front[4] = color_aquamarine;
    uni_hw05->back[4] = color_peru;;

    uni_hw05->front[5] = color_navajo_white;
    uni_hw05->back[5] = color_white;

    uni_hw05.to_dev();

  }

if ( !pipe_hyperb )
  pipe_hyperb
    .init( vh.qs )
    .ds_uniform_use( "BIND_LIGHT_SIMPLE", uni_light_simple )
    .ds_uniform_use( "BIND_MISC", uni_misc )
    .ds_uniform_use( "BIND_HW05", uni_hw05 )
    .ds_use( sampler, texture_id_syllabus )
    .shader_inputs_info_set<pCoor,pNorm,int,pTCoor>()
    .shader_code_set
```

```
      ("hw04-shdr.cc", "vs_main();", "gs_main();", "fs_main();")
      .topology_set( vk::PrimitiveTopology::eTriangleList )
      .create();

  if ( !pipe_hyperb_strip )
    pipe_hyperb_strip
      .init( vh.qs )
      .ds_uniform_use( "BIND_LIGHT_SIMPLE", uni_light_simple )
      .ds_uniform_use( "BIND_MISC", uni_misc )
      .ds_uniform_use( "BIND_HW05", uni_hw05 )
      .ds_use( sampler, texture_id_syllabus )
      .shader_inputs_info_set<pCoor,pNorm,int,pTCoor>()
      .shader_code_set
      ("hw04-shdr.cc", "vs_main();", "gs_main();", "fs_main();")
      .topology_set( vk::PrimitiveTopology::eTriangleStrip )
      .create();

  if ( !pipe_hyperb_points )
    pipe_hyperb_points
      .init( vh.qs )
      .ds_uniform_use( "BIND_LIGHT_SIMPLE", uni_light_simple )
      .ds_uniform_use( "BIND_MISC", uni_misc )
      .ds_uniform_use( "BIND_HW05", uni_hw05 )
      .ds_storage_follow( "BIND_HYPERB_COORDS", sto_hyperb_coords )
      .ds_storage_follow( "BIND_HYPERB_NORMS", sto_hyperb_norms )
      .ds_use( sampler, texture_id_syllabus )
      .shader_inputs_info_set()
      .shader_code_set
      ("hw05-shdr-sol.cc", "vs_points();", "gs_points();", "fs_points();")
      .topology_set( vk::PrimitiveTopology::ePointList )
      .create();

  if ( h.n_sides_constructed != opt_n_sides ) h.serial++;

  pCoor bot_ctr = h.bot_ctr_pos;
  pCoor top_ctr = h.top_ctr_pos;
  pCoor line_0_bot_pos = h.line_0_bot_pos;
  pVect line_0_dir = hyperb_info.line_0_dir;

  pVect hyperb_vz(bot_ctr,top_ctr);
  pNorm hyperb_az(hyperb_vz);
  pVect bot_vx(bot_ctr,line_0_bot_pos);

  pMatrix_Rotation rot(hyperb_az,theta_0);
  pMatrix trot =
    pMatrix_Translate( bot_ctr ) * rot * pMatrix_Translate( -bot_ctr );

  int n_levels = opt_n_sides/3;
  int n_sides = opt_n_sides;

  if ( !bset_hyperb || hyperb_serial != h.serial )
    {
      hyperb_serial = h.serial;
      h.n_sides_constructed = n_sides;
```

```
pCoor line_0_top_pos =
  line_0_bot_pos
  + hyperb_az.magnitude / dot( hyperb_az, line_0_dir ) * line_0_dir;

float delta_theta = 2 * M_PI / n_sides;
pVect delta_z = (1.0/n_levels ) * hyperb_vz;

pVect line_v(line_0_bot_pos,line_0_top_pos);
pVect line_dv = (1.0/n_levels) * line_v;

/// SOLUTION -- Problem 2
//
//  Compute a transformation from object (global) space to a
//  hyperboloid local space. In the hyperboloid local space the
//  hyperboloid axis is the z axis and bot_vx (see above) points
//  along the x axis. The local space is described in more
//  detail where texture coordinates are computed, search for
//  "case PV_Strip:"
//
//  Compute translation that moves hyperboloid axis so that it
//  passes through the origin. (The hyperboloid axis passes
//  through bot_ctr and top_ctr, either could be used to
//  construct the translation.)
//
pMatrix_Translate trans_to_hyp_local(-bot_ctr);
//
//  Compute hyperboloid local x and y axes.
//
pNorm hyperb_ax( bot_vx );
pNorm hyperb_ay = cross( hyperb_az, hyperb_ax );
//
//  Use hyperboloid axes to construct a matrix that will rotate
//  object space to hyperboloid local space.
//
pMatrix_Rows rot_to_hyp_local(hyperb_ax,hyperb_ay,hyperb_az);
//
//  Compute the transformation to hyperboloid local space:
//
pMatrix hyperb_local_from_object = rot_to_hyp_local * trans_to_hyp_local;
//
//  This matrix will be used to compute texture coordinates, in
//  the "case PV_Strip:" of the code preparing the triangle
//  strip buffer set and in the geometry shader used by the
//  points pipeline.
//
//  SOLUTION to Problem 2 continues in the PV_Strip case further
//  below.

const int s_levels = n_levels + 1;
const int s_sides = n_sides + 1;

//  Pre-compute inverses.
//
```

```
const float n_levels_inv = 1.0 / n_levels;
const float n_sides_inv = 1.0 / n_sides;

// Homework 5: Set values here. (Add members to HW05_Uniform.)
//
/// SOLUTION -- Problem 3
//
//  Put information into uni_hw05 that will be needed by the
//  geometry shader to fetch coordinates and normals and to
//  compute texture coordinates.
//
uni_hw05->s_sides = s_sides;
uni_hw05->s_levels = s_levels;
uni_hw05->n_sides = n_sides;
uni_hw05->n_levels = n_levels;
uni_hw05->n_sides_inv = n_sides_inv;
uni_hw05->n_levels_inv = n_levels_inv;
uni_hw05->local_from_object = hyperb_local_from_object;
//
//  SOLUTION to Problem 3 continues below (at record_draw for PV_Points).

uni_hw05.to_dev();

auto idxf = [&](int lev, int side) { return lev + side * s_levels; };

sto_hyperb_coords.resize( s_levels * s_sides );
sto_hyperb_norms.resize( s_levels * s_sides );

vector<vec4>& coords = sto_hyperb_coords.vals;
vector<vec4>& norms = sto_hyperb_norms.vals;

pVect top_vx(top_ctr,line_0_top_pos);

// Compute vertex coordinates and normals and write them
// to CPU containers coords and norms.
//
for ( int side = 0; side <= n_sides; side++ )
  {
    float theta = side * delta_theta;
    pMatrix_Rotation rot(hyperb_az,theta);
    pCoor line_bot_pos = bot_ctr + rot * bot_vx;
    pVect l_dv = rot * line_dv;

    for ( int lev = 0; lev <= n_levels; lev++ )
      {
        pCoor p_line = line_bot_pos + lev * l_dv;
        coords[ idxf(lev,side) ] = p_line;
        pCoor ctr = bot_ctr + lev * delta_z;
        pVect tc = cross(hyperb_az,pVect(ctr,p_line));
        pNorm n = cross(tc,l_dv);
        norms[ idxf(lev,side) ] = n;
      }
  }
```

```
sto_hyperb_coords.to_dev();
sto_hyperb_norms.to_dev();

switch ( pipeline_variant ) {
case PV_Individ:

  bset_hyperb.reset(pipe_hyperb);
  for ( int side = 0; side < n_sides; side++ )
    for ( int lev = 0; lev < n_levels; lev++ )
      {
        pCoor p00 = coords[ idxf( lev,   side   ) ];
        pCoor p01 = coords[ idxf( lev,   side+1 ) ];
        pCoor p10 = coords[ idxf( lev+1, side   ) ];
        pCoor p11 = coords[ idxf( lev+1, side+1 ) ];

        bset_hyperb << p00 << p01 << p10;
        bset_hyperb << p01 << p11 << p10;

        bset_hyperb << pTCoor(0,0) << pTCoor(0,1) << pTCoor(1,0);
        bset_hyperb << pTCoor(0,1) << pTCoor(1,1) << pTCoor(1,0);

        pNorm n00 = norms[ idxf( lev,   side   ) ];
        pNorm n01 = norms[ idxf( lev,   side+1 ) ];
        pNorm n10 = norms[ idxf( lev+1, side   ) ];
        pNorm n11 = norms[ idxf( lev+1, side+1 ) ];

        bset_hyperb << n00 << n01 << n10;
        bset_hyperb << n01 << n11 << n10;

        int c1 = side == 0 ? ( lev%2 ? 5 : 2 ) : 1;
        int c2 = side == 0 ? ( lev%2 ? 3 : 4 ) : 1;

        bset_hyperb << c1 << c1 << c1 << c2 << c2 << c2;
      }
  break;

case PV_Strip:

  bset_hyperb.reset(pipe_hyperb_strip);

  for ( int side = 0; side < n_sides; side++ )
    for ( int lev = 0; lev <= n_levels; lev++ )
      {
        // For triangle strip just emit two vertices per iteration ..
        // .. rather than six vertices.
        // Same two triangles, but using four fewer vertices!

        pCoor p00 = coords[ idxf( lev, side ) ];
        pCoor p01 = coords[ idxf( lev, side + 1 ) ];

        bset_hyperb << p00 << p01;

        /// SOLUTION -- Problem 2
        //
```

```
//  Scale the y texture coordinate so that it is 0 when
//  at the top (lev==n_levels) and 1 at the bottom (lev==0).
//
float tex_y = 1 - lev * n_levels_inv;

//  Base the x texture coordinate on the angle around
//  the hyperboloid axis. That is *not* linearly
//  related to side, so computing tex_x won't be quite
//  as easy as computing tex_y.
//
//  To compute the angle first use
//  hyperb_local_from_object to rotate to the
//  hyperboloid local coordinate space. In the local
//  space the hyperboloid axis is the z axis. Consider
//  point bcloc = hyperb_local_from_object * bot_ctr;.
//  Because the hyperboloid axis is the z axis and the
//  hyperboloid axis passes through bot_ctr we know
//  that bcloc.x = 0 and bcloc.y = 0.
//
//  Transform p00 and p01 to the hyperboloid local space:
//
pCoor p00_l = hyperb_local_from_object * p00;
pCoor p01_l = hyperb_local_from_object * p01;
//
//  In the hyperboloid local space the angle that a point
//  makes with the hyperboloid axis can be found by
//  computing the arc tangent:
//
float theta0 = atan2f( p00_l.y, p00_l.x );
float theta1 = atan2f( p01_l.y, p01_l.x );
//
//  To compute the x texture coordinate multiply the angle
//  by some scale factor:
//
float tex_x_scale_factor = 1.5;
float tex_x0 = theta0 * tex_x_scale_factor;
float tex_x1 = theta1 * tex_x_scale_factor;
//
//  All that's left to do is to insert the texture
//  coordinates.
//
bset_hyperb
  << pTCoor( tex_x0, tex_y ) << pTCoor( tex_x1, tex_y );

pNorm n00 = norms[ idxf( lev, side ) ];
pNorm n01 = norms[ idxf( lev, side + 1 ) ];

bset_hyperb << n00 << n01;

// Choose color index.
//
int c1 = lev == 0 ? 0 : side == 0 ? ( (lev+1)%2 ? 5 : 2 ) : 1;
int c2 = lev == 0 ? 0 : side == 0 ? ( (lev+1)%2 ? 3 : 4 ) : 1;
//
```

```
              // If lev == 0, set to zero to start triangle strip;
              // if side == 0, choose from among four colors to show structure;
              // otherwise, use color index 1.

              bset_hyperb << c1 << c2;
            }
        break;

      case PV_Points:
        // Nothing to do.
        break;

      default: assert( false );
      }

      bset_hyperb.to_dev();
    }

  switch ( pipeline_variant ) {
  case PV_Individ:
    pipe_hyperb.ds_set( transform * trot );
    pipe_hyperb.record_draw(cb, bset_hyperb);
    break;
  case PV_Strip:
    pipe_hyperb_strip.ds_set( transform * trot );
    pipe_hyperb_strip.record_draw(cb, bset_hyperb);
    break;
  case PV_Points:
    {
      /// SOLUTION -- Problem 3
      //
      //  Have geometry shader emit two triangles per point, which
      //  is what one iteration of the triangle strip code above does.
      //  Set the number of "vertices" to the number of iterations, except
      //  that we won't need one extra level iteration.
      //
      const int n_vtx = n_levels * n_sides;
      //
      //  SOLUTION to Problem 3 continues in hw05-shdr-sol.cc.

      pipe_hyperb_points.ds_set( transform * trot );
      pipe_hyperb_points.record_draw(cb, n_vtx );
    }
    break;

  default: assert( false );
  }


  // Render Marker for Light Source
  //
  shapes.record_tetrahedron(cb,transform,light_location,0.2);

}
```

```
void
World::keyboard_handle()
{
  const int key = vh.keyboard_key_get();
  if ( !key ) return;
  pVect adjustment(0,0,0);
  pVect user_rot_axis(0,0,0);
  const bool kb_mod_s = vh.keyboard_shift;
  const bool kb_mod_c = vh.keyboard_control;
  const float move_amt = kb_mod_s ? 2.0 : kb_mod_c ? 0.08 : 0.4;

  switch ( key ) {
  case FB_KEY_LEFT: adjustment.x = -move_amt; break;
  case FB_KEY_RIGHT: adjustment.x = move_amt; break;
  case FB_KEY_PAGE_UP: adjustment.y = move_amt; break;
  case FB_KEY_PAGE_DOWN: adjustment.y = -move_amt; break;
  case FB_KEY_DOWN: adjustment.z = move_amt; break;
  case FB_KEY_UP: adjustment.z = -move_amt; break;
  case FB_KEY_DELETE: user_rot_axis.y = 1; break;
  case FB_KEY_INSERT: user_rot_axis.y =  -1; break;
  case FB_KEY_HOME: user_rot_axis.x = 1; break;
  case FB_KEY_END: user_rot_axis.x = -1; break;
  case 'n': bset_hyperb.reset(); break;
  case '1': scene_setup_1(); break;
  case '2': scene_setup_2(); break;
  case 'b': case 'B': opt_move_item = MI_Ball; break;
  case 'e': case 'E': opt_move_item = MI_Eye; break;
  case 'l': case 'L': opt_move_item = MI_Light; break;
  case 'p': case 'P': opt_pause = !opt_pause; break;
  case 't': case 'T': opt_texture = !opt_texture;
    uni_misc->opt_assign.x = int(opt_texture);
    break;
  case 'v': case 'V':
    hyperb_info.serial++;
    if ( ++pipeline_variant >= PV_SIZE ) pipeline_variant = 0;
    break;
  case 'y': opt_tryout1 = !opt_tryout1;
    uni_misc->opt_tryout.x = int(opt_tryout1);
    break;
  case 'Y': opt_tryout2 = !opt_tryout2;
    uni_misc->opt_tryout.y = int(opt_tryout2);
    break;
  case 'z': opt_tryout3 = !opt_tryout3;
    uni_misc->opt_tryout.z = int(opt_tryout3);
    break;
  case FB_KEY_TAB:
    if ( !kb_mod_s ) { variable_control.switch_var_right(); break; }
  case 96: variable_control.switch_var_left(); break;
  case '-':case '_': variable_control.adjust_lower(); break;
  case '+':case '=': variable_control.adjust_higher(); break;
  default: printf("Unknown key, %d\n",key); break;
```

```
  }

  // Update eye_direction based on keyboard command.
  //
  if ( user_rot_axis.x || user_rot_axis.y )
    {
      pMatrix_Rotation rotall(eye_direction,pVect(0,0,-1));
      user_rot_axis *= invert(rotall);
      eye_direction *= pMatrix_Rotation(user_rot_axis, M_PI * 0.03);
    }

  // Update eye_location based on keyboard command.
  //
  if ( adjustment.x || adjustment.y || adjustment.z )
    {
      const double angle =
        fabs(eye_direction.y) > 0.99
        ? 0 : atan2(eye_direction.x,-eye_direction.z);
      pMatrix_Rotation rotall(pVect(0,1,0),-angle);
      adjustment *= rotall;

      switch ( opt_move_item ){
      case MI_Light: light_location += adjustment; break;
      case MI_Eye: eye_location += adjustment; break;
      //  case MI_Ball: object_location += adjustment; break;
      default: break;
      }
    }
}


int
main(int argv, char **argc)
{
  pVulkan_Helper pvulkan_helper(argv,argc);
  World world(pvulkan_helper);

  world.setup_and_run();

  return 0;
}
```

# 20 Fall 2023 Solutions

```
/// LSU EE 4702-1 (Fall 2023), GPU Programming
//
//
 /// Homework 1 -- SOLUTION
 //
 //  Assignment: https://www.ece.lsu.edu/gpup/2023/hw01.pdf
 //
 //  This code is based on the solution to 2022 Homework 1.
 //  For the solution to 2022 Homework 2 look
 //  at file ../../2022/hw01/hw01-sol.cc
 //  or visit https://www.ece.lsu.edu/koppel/gpup/2022/hw01-sol.cc.html
 //

#include <gp/coord.h>
#include "frame_buffer.h"

// Definitions for iVect, iCoor
#include "hw01-misc.h"


struct HW01_Data {
  HW01_Data():inited(false){};
  bool inited;
  bool tryout0, tryout1;
  vector<uint32_t> fb_area_dup;
  iVect mag_from_diag;
  iVect mag_to_of; // Relative to mouse location.
  iVect mag_to_diag;
};

void
line_draw
( pFrame_Buffer& fb, iCoor p0, iCoor p1, uint32_t co)
{
  int x0 = p0.x, y0 = p0.y;
  int x1 = p1.x, y1 = p1.y;

  const iVect delta = p1 - p0;
  const int width = fb.width_get();

  if ( abs( delta.x ) < abs( delta.y ) )
    {
      // Iterate over y
      if ( y0 > y1 ) { swap(x0,x1); swap(y0,y1); }

      float dxdy = float(delta.x) / delta.y;
      float x = x0;

      for ( int yi = y0;  yi < y1;  yi++ )
        {
          fb[ yi * width + x ] = co;
          x += dxdy;
        }
```

```
    }
  else
    {
      // Iterate over x
      if ( x0 > x1 ) { swap(x0,x1); swap(y0,y1); }

      float dydx = float(delta.y) / delta.x;
      float y = y0;

      for ( int xi = x0;  xi < x1;  xi++ )
        {
          fb[ int(y) * width + xi ] = co;
          y += dydx;
        }
    }
}


void
parallel_lines_draw
( pFrame_Buffer& fb, iCoor p0, iCoor pb, iVect v, uint32_t co)
{
  line_draw( fb, p0, p0 + v, co );
  line_draw( fb, pb, pb + v, co );
}

void
parallelogram_draw
( pFrame_Buffer& fb, iCoor p00, iVect vx,  iVect vy, uint32_t color )
{
  parallel_lines_draw(fb, p00, p00+vy, vx, color);
  parallel_lines_draw(fb, p00, p00+vx, vy, color);
}

void
aa_rectangle_draw
( pFrame_Buffer& fb, iCoor p00, int wd, int ht, uint32_t color )
{
  // Window Size, in Pixels
  //
  const int win_width [[maybe_unused]] = fb.width_get();
  const int win_height [[maybe_unused]] = fb.height_get();

  // Compute coordinate of upper-right corner.
  //
  iCoor p11 = p00 + iVect(wd,ht);
  //
  // Coordinates p00 and p11 may be outside of the window.

  /// SOLUTION -- Problem 1

  // Note: The solution below is for cases where wd and ht are positive.
  assert( wd >= 0 && ht >= 0 );  // Raise a fatal error if they are not.
```

```
  // Check whether rectangle will be completely outside the window.
  //
  if ( p00.x >= win_width || p00.y >= win_height ) return;
  if ( p11.x <= 0 || p11.y <= 0 ) return;

  // Compute clipped rectangle lower-left coordinate.
  //
  iCoor p00c( max( 0, p00.x ), max( 0, p00.y ) );
  //
  // Note: The /clipped/ rectangle is the portion of the rectangle
  // that is within the window (frame buffer).

  // Compute clipped rectangle upper-right coordinate.
  //
  iCoor p11c( min( win_width-1, p11.x ), min( win_height-1, p11.y ) );

  // Compute the other two clipped rectangle coordinates.
  //
  iCoor p01c( p00c.x, p11c.y ), p10c( p11c.x, p00c.y );

  // Draw visible vertical lines.
  //
  if ( p00.x == p00c.x ) line_draw( fb, p00c, p01c, color );
  if ( p11.x == p11c.x ) line_draw( fb, p10c, p11c, color );

  // Draw visible horizontal lines.
  //
  if ( p00.y == p00c.y ) line_draw( fb, p00c, p10c, color );
  if ( p11.y == p11c.y ) line_draw( fb, p01c, p11c, color );

}

void kbd_input_handle(pFrame_Buffer& fb, HW01_Data& hw01_data);


void
render_hw01(pFrame_Buffer& fb, HW01_Data& hw01_data)
{
  // Window Size, in Pixels
  //
  const int win_width = fb.width_get();
  const int win_height = fb.height_get();

  // Adjust size and position of mag window based on keyboard input.
  kbd_input_handle(fb,hw01_data);

  /// Coordinates Used for Placement of Other Objects
  //
  const int sq_slen = min(win_width,win_height) * .45;
  const int sq_x0 = sq_slen * 0.1,   sq_y0 = win_height - 150;
  const int sq_x1 = sq_x0 + sq_slen, sq_y1 = sq_y0 - sq_slen;
  //
  // One vertex of the square is at (sq_x0,sq_y0) and the opposite
  // vertex is at (sq_x1,sq_y1).
```

```
//                            RRGGBB
const uint32_t color_red = 0xff0000;
const uint32_t color_blue =    0xff;
const uint32_t color_green = 0xff00;

/// Demo Code: Draw Sine Waves -- No need to modify this.
//
int num_waves = 4;
float plot_ht = sq_slen/float(num_waves);
for ( float
        yb = sq_y0 - plot_ht/2,  freq = 4 * M_PI / ( win_width - 40 - sq_x1 );
      yb > sq_y1;
      yb -= plot_ht,  freq *= 2 )
  for ( int x = 20;  x < win_width - 20;  x++ )
    {
      int y = yb + sin( x * freq ) * plot_ht * .45;
      fb[ y * win_width + x ] = 0xff00;
    }

/// Demo Code: Draw Lines Radiating From a Point -- No need to modify.
//
//  This code calls the line_draw routine.
//
int n_rects = 24;
int n_pts = n_rects * 3;
float c_r1 = sq_slen/6;  // Inner radius
float c_r2 = sq_slen/2;  // Outer radius.
pCoor c_ctr( c_r2 + 20, c_r2 + 20 ); // Center of "circle"
float delta_theta = 2 * M_PI / n_pts;
vector<pVect> vecs;
for ( int i=0; i<n_pts; i++ )
  vecs.push_back( pVect( cosf(i*delta_theta), sinf(i*delta_theta), 0 ) );
for ( int i=0; i<n_pts; i+=3 )
  {
    pVect v1 = vecs[i+1] * ( c_r2 - c_r1 );
    pVect v2 = c_r1 * ( vecs[i+2] - vecs[i] );
    pCoor p1 = c_ctr + c_r1 * vecs[i];
    parallelogram_draw(fb, p1, v1, v2, 0xffffff );
  }

fb.fb_stats_off(); // Turn off "redundancy" statistics.

/// Homework 1 Problem 2

// The code below draws two boxes in the vicinity of the mouse
// pointer, a blue box, and a green box. Write code that copies the
// area bounded by the blue box to the area bounded by the green
// box. Stretch (zoom) or shrink (unzoom) the material in the blue
// box so that it fills the entire area of the green box. Use the
// provided staging array, fb_area_dup, so that the green box shows
// the contents of the blue box *before the copy was started*.

// Coordinates of mouse pointer.
```

```
//
const iCoor mouse(fb.mouse_x,fb.mouse_y);

const iVect& mag_from_diag = hw01_data.mag_from_diag;
iVect& mag_to_of = hw01_data.mag_to_of;
iVect& mag_to_diag = hw01_data.mag_to_diag;
const int mag_to_wd = mag_to_diag.x;
const int mag_to_ht = mag_to_diag.y;

// Lower-left of area to be magnified.
const iCoor mag_from_ll = mouse - mag_from_diag / 2;
const int mag_from_wd [[maybe_unused ]] = mag_from_diag.x;
const int mag_from_ht [[maybe_unused]] = mag_from_diag.y;

// Lower-left of area showing magnification.
const iCoor mag_to_ll = mag_from_ll + mag_to_of;

/// Put solution to Problem 2 in this general region.

const bool show_starter_code = hw01_data.tryout0;
if ( show_starter_code )
  {
    // Draw a diagonal blue line of positive slope in mag from area.
    for ( int x=0; x<mag_from_wd; x++ )
      fb[ mag_from_ll.x + x + ( mag_from_ll.y + x ) * win_width ]
        = color_blue;

    // Draw a diagonal green line of negative slope in mag to area.
    for ( int x=0; x<min(mag_to_wd,mag_to_ht); x++ )
      fb[ mag_to_ll.x + x + ( mag_to_ht + mag_to_ll.y - x ) * win_width ]
        = color_green;

    // Draw a diagonal red line of positive slope in staging area.
    for ( int x=0; x<mag_from_wd; x++ )
      hw01_data.fb_area_dup[ x + x * mag_from_wd ]
        = color_red;
  }

/// SOLUTION -- Problem 2
//
// Copy the mag_from area to the interim storage area, fb_area_dup.
//
for ( int x=0; x<mag_from_wd; x++ )
  for ( int y=0; y<mag_from_ht; y++ )
    {
      // Coordinates of the pixel in the mag_from area.
      //
      iCoor from = mag_from_ll + iVect(x,y);

      // If false, the coordinate is outside the window, and so it
      // won't be copied.
      //
      bool in_win = from.x >= 0 && from.x < win_width
        && from.y >= 0 && from.y < win_height;
```

```
        // Write the interim storage with the pixel if its coordinate
        // is valid, write zero otherwise.
        //
        hw01_data.fb_area_dup[ x + y * mag_from_wd ] =
          in_win ? fb[ from.x + from.y * win_width ] : 0;
      }
  //
  // Note that the frame buffer is copied before the blue and green
  // rectangles are drawn.

  // Draw a blue rectangle around area to be magnified.
  aa_rectangle_draw
    ( fb, mag_from_ll, mag_from_diag.x, mag_from_diag.y, color_blue );

  // Compute zoom (magnification) from mag_from (blue) to mag_to (green)
  // areas along each axis.
  //
  const float zoom_x = float(mag_to_wd)/mag_from_wd;
  const float zoom_y = float(mag_to_ht)/mag_from_ht;

  for ( int x=0; x<mag_from_wd; x++ )
    for ( int y=0; y<mag_from_ht; y++ )
      {
        uint32_t pixel = hw01_data.fb_area_dup[x + y * mag_from_wd];

        // Compute range of coordinates in to_area to write into.
        //
        int xt0 = max( 0,          mag_to_ll.x + int(x * zoom_x)     );
        int xt1 = min( win_width,  mag_to_ll.x + int((x+1) * zoom_x) );
        int yt0 = max( 0,          mag_to_ll.y + int(y * zoom_y)     );
        int yt1 = min( win_height, mag_to_ll.y + int((y+1) * zoom_y) );

        for ( int xt = xt0;  xt < xt1; xt++ )
          for ( int yt = yt0; yt < yt1; yt++ )
            fb[ xt + yt * win_width ] = pixel;
      }

  // Draw a green rectangle area to show magnification.
  aa_rectangle_draw
    ( fb, mag_to_ll, mag_to_diag.x, mag_to_diag.y, color_green );

}

void
kbd_input_handle(pFrame_Buffer& fb, HW01_Data& hw01_data)
{
  iVect& mag_to_of = hw01_data.mag_to_of;
  iVect& mag_to_diag = hw01_data.mag_to_diag;

  if ( !hw01_data.inited )
    {
      hw01_data.inited = true;
      hw01_data.tryout0 = false;
```

```
      hw01_data.tryout1 = false;
      hw01_data.mag_from_diag = { 100, 100 };
      const int mag_from_wd = hw01_data.mag_from_diag.x;
      const int mag_from_ht = hw01_data.mag_from_diag.y;
      hw01_data.fb_area_dup.resize( mag_from_wd * mag_from_ht );
      mag_to_diag = iVect( mag_from_wd * 2.5, mag_from_ht * 2.5 );
      mag_to_of = iVect( mag_from_wd*.3, mag_from_ht*.7 );
    }

  const int kbd_inc = 2;

  switch ( fb.keyboard_key ) {
  case FB_KEY_RIGHT: case FB_KEY_LEFT:
    {
      const int inc = fb.keyboard_key == FB_KEY_LEFT ? -kbd_inc : kbd_inc;
      if ( fb.keyboard_shift ) mag_to_of.x += 2*inc;
      else { mag_to_diag.x += 2*inc; mag_to_of.x -= inc; }
      break;
    }
  case FB_KEY_UP: case FB_KEY_DOWN:
    {
      const int inc = fb.keyboard_key == FB_KEY_DOWN ? -kbd_inc : kbd_inc;
      if ( fb.keyboard_shift ) mag_to_of.y += 2*inc;
      else { mag_to_diag.y += 2*inc; mag_to_of.y -= inc; }
      break;
    }
  case 'y':
    hw01_data.tryout0 = !hw01_data.tryout0; break;
  case 'Y':
    hw01_data.tryout1 = !hw01_data.tryout1; break;
  default:
    break;
  };


}

int
main(int argc, char **argv)
{
  pFrame_Buffer demo_frame_buffer(argc,argv);

  HW01_Data hw01_data;
  demo_frame_buffer.show
    ( [&](pFrame_Buffer& fb){ render_hw01(fb,hw01_data); } );
  return 0;
}
```

```
/// LSU EE 4702-1 (Fall 2023), GPU Programming
//
 /// Homework 2 -- SOLUTION
 //
 //  Assignment: https://www.ece.lsu.edu/gpup/2023/hw02.pdf
 //
 //  Put solution in routine ball_setup_4, search for "Put Homework 2" to
 //  find it.

 /// Based on the Links Project Base Code


/// Purpose
//
//   Demonstrate simulation of point masses connected by springs.


/// What Code Does

// Simulates balls connected by springs over a platform. Balls and
// springs can be initialized in different arrangements (called
// scenes). Currently scene 1 is a tree-like arrangement. The platform
// consists of tiles, some are purple-tinted mirrors (showing a
// reflection of the ball), the others show the course syllabus.



///  Keyboard Commands
 //
 /// Object (Eye, Light, Ball) Location or Push
 //   Arrows (←,→,↑,↓) Page Up, Page Down
 //        Move object or push ball, depending on mode.
 //        Shift + KEY: motion is 5x faster.
 //        Ctrl + KEY : motion is 5x slower.
 //   'e': Move eye.
 //   'l': Move light.
 //   'b': Move head (first) ball. (Change position but not velocity.)
 //   'B': Push head ball. (Add velocity.)
 //
 /// Eye Direction and GUI Options
 //
 //   'Home', 'End', 'Delete', 'Insert'
 //         Turn the eye direction. Home should rotate eye direction
 //         up, End should rotate eye down, Delete should rotate eye
 //         left, Insert should rotate eye right. The eye direction
 //         vector is displayed in the upper left.
 //  'C-='  (Ctrl =) Increase green text size.
 //  'C--'  (Ctrl -) Decrease green text size.
 //  'F12'  Write screenshot to PNG file.

 /// Simulation Execution Options
 //  (Also see variables below.)
```

```
//
//  'p'    Pause simulation. (Press again to resume.)
//  ' '    (Space bar.) Advance simulation by 1/30 second.
//  'S- '  (Shift Space bar.) Advance simulation by one time step.

/// Performance Tuning, Graphics Effects, and Debugging Options
//
//  'y'    Toggle value of opt_tryout1. Intended for experiments and debugging.
//  'Y'    Toggle value of opt_tryout2. Intended for experiments and debugging.
//  'n'    Toggle use of surface normals for spheres.
//  'o'    Toggle shadow effect.
//  'r'    Toggle mirror effect.
//  '!'    Toggle rendering of spheres (balls).
//  '@'    Toggle rendering of links.
//  '#'    Toggle rendering of platform.


/// Scene and Simulated System Options
//  (Also see variables below.)
//
//  'w'    Twirl balls around axis formed by head and tail. (Prob 2 soln).
//  '1'    Set up scene 1.
//  '2'    Set up scene 2.
//  '3'    Set up scene 3.
//  '4'    Set up scene 4.
//  'h'    Freeze position of first (head) ball. (Press again to release.)
//  't'    Freeze position of last (tail) ball. (Press again to release.)
//  's'    Stop balls.
//  'g'    Turn gravity on and off.

/// Variables
//   Selected program variables can be modified using the keyboard.
//   Use "Tab" to cycle through the variable to be modified, the
//   name of the variable is displayed next to "VAR" on the bottom
//   line of green text.

//  'Tab' Cycle to next variable.
//  '`'   Cycle to previous variable.
//  '+'   Increase variable value.
//  '-'   Decrease variable value.
//
//  VAR Spring Constant - Set spring constant.
//  VAR Time Step Duration - Set physics time step.
//  VAR Air Resistance - Set air resistance.
//  VAR Light Intensity - The light intensity.
//  VAR Gravity - Gravitational acceleration. (Turn on/off using 'g'.)

#define MAIN_INCLUDE
#undef VBUFFER_STRICT
//  #define VBUFFER_STRICT_WATCH 0xd10d270000000018
#include <vhelper.h>

#include <vstroke.h>
#include <gp/coord.h>
```

```
#include <gp/pstring.h>
#include <gp/misc.h>

#include <vutil-texture.h>
#include <vutil-pipeline.h>
#include "shapes.h"

#include <cuda_runtime.h>

#include <gp/colors.h>

#include <gp/util-containers.h>
#include <gp/coord-containers.h>
#include <gp/cuda-gpuinfo.h>
#include "links.cuh"

#include "links-shdr-common.h"


///
/// Main Data Structures
///
//
// class World: All data about scene.


class World;

enum GPU_Physics_Method { GP_cpu, GP_cuda, GP_ENUM_SIZE };
const char* const gpu_physics_method_str[] = { "CPU", "CUDA" };
const char *sh_names[] = { "PLAIN", "SET 1", "SET 2" };


// Object Holding Ball State
//
class Ball {
public:
  Ball() { init(); }
  Ball(pCoor pos){ init(); position = pos; constants_update(); }
  Ball(pCoor pos, const Ball &bd)
  { *this = bd; position = pos; constants_update(); }

  void init()
    {
      velocity = pVect(0,0,0);
      omega = pVect(0,0,0);
      radius = 1;
      density = 1.00746;
      fdt_to_do = 0;
      locked = false;
      color = color_lsu_spirit_gold;
      specularity = 0;
      contact = false;
```

```
      orientation_set(pVect(0,1,0),0);
      world_time_expire = 3600.0 * 24 * 365 * 1000;
    }

  pCoor position;
  pVect velocity;
  pQuat orientation;
  pMatrix3x3p omatrix;
  pVect omega;                  // Spin rate and axis.
  double world_time_expire;

  int idx; // Position in balls_pos_rad;

  float mass;
  float mass_min; // Mass below which simulation is unstable.
  float radius;
  float radius_sq, radius_inv;  // Pre-computed based on radius.
  float density;
  float mass_inv;
  float fdt_to_do; // Radius / moment of inertia.

  bool locked;

  pVect force;
  pVect torque;
  pColor color;
  float specularity;            // Amount of shininess, in [0,1].
  bool contact;                 // When true, ball rendered in gray.
  float spring_constant_sum;    // Used to compute minimum mass.

  void orientation_set(pNorm dir, float angle)
    { orientation_set(pQuat(dir,angle)); }
  void orientation_set(pQuat ori)
    {
      orientation = ori;
      omatrix = pMatrix3x3_Rotation(orientation);
    }
  void constants_update();
  pVect point_rot_vel(pNorm tact_dir);
  pVect point_rot_vel(pVect tact_dir);
  void apply_tan_force(pNorm tact_dir, pNorm force_dir, double force_mag);
  void apply_tan_force(pNorm tact_dir, pVect force);
  void push(pVect amt);
  void translate(pVect amt);
  void stop();
  void freeze();
};


void
Ball::constants_update()
{
  assert( radius );
```

```
  radius_sq = radius * radius;
  mass = 4.0 / 3 * M_PI * radius_sq * radius * density;
  radius_inv = 1 / radius;
  mass_inv = 1 / mass;
  // FYI, notice simplifications:
  //   const float mo_inertia = 0.4 * mass * radius_sq;
  //   fdt_to_do = radius / mo_inertia;
  fdt_to_do = radius_inv * 2.5 * mass_inv;
}

pVect
Ball::point_rot_vel(pNorm direction)
{
  /// Return velocity of point on surface of ball.
  //
  return radius * cross( omega, direction );
}

pVect
Ball::point_rot_vel(pVect rel_pos)
{
  /// Return velocity of point relative to center.
  //
  return cross( omega, rel_pos );
}

void
Ball::apply_tan_force(pNorm tact_dir, pVect force)
{
  torque += cross(tact_dir, force);
}

void
Ball::apply_tan_force(pNorm tact_dir, pNorm force_dir, double force_mag)
{
  apply_tan_force(tact_dir, force_mag * force_dir);
}

class Link {
public:
  Link(Ball *b1, Ball *b2):ball1(b1),ball2(b2),
     snapped(false),
     natural_color(color_lsu_spirit_purple),color(color_lsu_spirit_purple)
  { init(); }
  Link(Ball *b1, Ball *b2, pColor colorp):ball1(b1),ball2(b2),
     snapped(false),
     natural_color(colorp),color(colorp)
  { init(); }
  Link(Link *link, pVect cb1p, pVect cb2p, float drp):
    ball1(link->ball1), ball2(link->ball2), cb1(cb1p), cb2(cb2p),
    is_surface_connection(true), is_renderable(false),
    is_simulatable(true), distance_relaxed(drp),
    snapped(false){ link->is_simulatable = false; }
  void init()
```

```
    {
      assert( ball1->radius > 0 );
      assert( ball2->radius > 0 );
      is_simulatable = true;
      is_renderable = true;
      pNorm n12(ball1->position,ball2->position);
      const float rad_sum = ball1->radius + ball2->radius;
      is_surface_connection = n12.magnitude >= rad_sum;
      if ( !is_surface_connection )
        {
          distance_relaxed = n12.magnitude;
          cb1 = cb2 = pVect(0,0,0);
          return;
        }
      distance_relaxed = n12.magnitude - rad_sum;
      cb1 = ball1->radius * mult_MTV( ball1->omatrix, n12 );
      cb2 = ball2->radius * mult_MTV( ball2->omatrix, -n12 );
    }
  int serial;
  Ball* const ball1;
  Ball* const ball2;
  int ball1_idx;
  int ball2_idx;
  pVect cb1, cb2, b1_dir, b2_dir;
  bool is_surface_connection;
  bool is_renderable, is_simulatable;
  float distance_relaxed;
  bool snapped;
  pColor natural_color;
  pColor color;

  pVect spring_force_12;
  pVect torque1,torque2;

};

// Declare containers for Balls and Links. The containers are derived
// from std::vector and use overloaded operator += for push_back. (See
// include/util-containers.h.)
//
typedef pVectorI<Link> Links;
typedef pVectorI<Ball> Balls;


enum Shader_Option { SO_Plain, SO_Instances, SO_True, SO_ENUM_SIZE };



enum HW2_Rotation { Rot_None, Rot_Roll, Rot_Spin, Rot_SIZE };
const char* const hw2_rot_str[] = { "NONE", "ROLL", "SPIN" };

struct Setup_4_Info {

  Setup_4_Info() { n_balls = 0; }
```

```
  pCoor center_pos;
  int n_balls;
  float ball_radius;
  float ring_outer_radius;
  pNorm ring_normal;

  // Points on opposite sides of inner ring. (That is, 180 deg apart.)
  pCoor p1, p2;

  int rotation;
  float omega;

};


class World {
public:
  World(pVulkan_Helper &fb, int argc, char **argv)
    :vh(fb), ff_state(fb.qs), raytrace(fb.qs),
     frame_timer(vh.frame_timer), transform(fb.qs),
     shapes(ff_state), sphere(ff_state),
     opt_tryout1(fb.qs.opt_tryout1),opt_tryout2(fb.qs.opt_tryout2)
  {init(argc,argv);}
  void init(int argc, char **argv);
  void init_graphics();
  void run();
  void frame_callback();
  void render(vk::CommandBuffer& cb);
  void render_objects(vk::CommandBuffer& cb, Render_Option render_option);
  void objects_erase();
  void cb_keyboard();
  void modelview_update();

  pVulkan_Helper& vh;
  VFixed_Function_State_Manager ff_state;
  VRaytrace raytrace;
  pFrame_Timer& frame_timer;
  VTransform transform;
  Shapes shapes;
  Sphere sphere;

  bool opt_want_raytrace;

  pVariable_Control variable_control;
  double world_time;
  double last_frame_wall_time;
  float opt_time_step_duration;
  int time_step_count;
  float opt_gravity_accel;      // Value chosen by user.
  pVect gravity_accel;          // Set to zero when opt_gravity is false;
  bool opt_gravity;
  bool opt_head_lock, opt_tail_lock;
```

```
bool &opt_tryout1, &opt_tryout2;  // For ad-hoc experiments.
float opt_tryoutf;
bool opt_sphere_true;
VBufferV<Shdr_Uni_Common> buf_uni_common;

// Tiled platform for ball.
//
float platform_xmin, platform_xmax, platform_zmin, platform_zmax;
float platform_pi_xwidth_inv;
VVertex_Buffer_Set bset_platform_tex, bset_platform_mir;
VPipeline pipe_platform_mir, pipe_platform_tex;
RT_Geo_Package rt_geo_platform_tex, rt_geo_platform_mir;

vk::Sampler sampler;
VTexture texid_hw;
VTexture texid_syl;
VTexture texid_emacs;

void platform_update();
bool platform_collision_possible(pCoor pos);

pCoor light_location;
float opt_light_intensity;
enum { MI_Eye, MI_Light, MI_Ball, MI_Ball_V, MI_COUNT } opt_move_item;
bool opt_pause;
bool opt_single_frame;      // Simulate for one frame.
bool opt_single_time_step;  // Simulate for one time step.

VBufferV<Uni_Lighting> uni_light, uni_light_ambient, uni_light_all;
VBufferV<Uni_Lighting> uni_light_reflected;
VBufferV<Uni_Lighting> *puni_light_curr;

pCoor eye_location;
pVect eye_direction;
pMatrix transform_projection;
pMatrix modelview;

pVect adj_vector;
double adj_t_prev;
double adj_t_stop;
double adj_duration;

uint opt_shader;

bool opt_mirror;
int opt_mirror_depth;
bool opt_shadows;
bool opt_shadow_volumes;      // Make shadow volumes visible.
bool opt_normal_sphere;
int opt_hide_stuff;

int last_setup; // Last scene set up.

void ball_setup_1(bool big);
```

```
void ball_setup_2();
void ball_setup_3();
void ball_setup_4(const Setup_4_Info& si);
void ball_setup_45_wrapper(bool it_4);
void ball_setup_5(const Setup_4_Info& si);
void ball_setup_6();
void setup_at_end();
void time_step_cpu(double);
void balls_stop();
void balls_freeze();
void balls_translate(pVect amt, int idx);
void balls_translate(pVect amt);
void balls_push(pVect amt, int idx);
void balls_push(pVect amt);
void balls_twirl();
void lock_update();

Ball *make_marker(pCoor pos, pColor col);

float opt_spring_constant, opt_spring_damp;
float opt_friction_dynamic;
float opt_air_resistance;
float distance_relaxed;
int chain_length;
Balls balls;
Ball *head_ball, *tail_ball;
Links links;
int opt_sphere_slices;

Links link_new(Ball *ball1, Ball *ball2, float stiffness = 0.2);

/// CUDA Stuff

int opt_physics_method;
GPU_Info gpu_info;
cudaEvent_t frame_start_ce, frame_stop_ce;
int opt_block_size;
bool gpu_const_data_stale;
bool cpu_phys_data_stale;

void data_cpu_to_gpu_constants();
void data_cpu_to_gpu_dynamic();
void data_gpu_to_cpu_dynamic();
CPU_GPU_Common c;  // c is for common.  See links.cuh.

void render_link_start();
void render_link_gather(Link *link);
void render_link_render(vk::CommandBuffer& cb, bool shadow_volumes = false);
void render_link_render_sv(vk::CommandBuffer& cb)
{ render_link_render(cb,true); };

VPipeline pipe_set_2, pipe_set_2_sv;
double world_time_link_update;
```

```
  VCompute comp_links;
  VBuffer<int> buf_links_as_idx; // Indices for links' acceleration structures.
  VBuffer<vec2> buf_links_tex_coor;
  VBuffer<vec4> buf_links_vertex_g;
  VBuffer<vec4> buf_links_normal_g;
  RT_Geo_Package rt_geo_links;

  int opt_sides, opt_segments;

  vector< VBufferVV<vec4>* > lis_bufs;
  VBufferVV<vec4> lis_pos1, lis_pos2;
  VBufferVV<vec4> lis_v1, lis_v2;
  VBufferVV<vec4> lis_b1_ydir, lis_b2_ydir, lis_tex_rad;

  /// 2023 Homework 1
  //
  float opt_omega;
  int opt_hw2_rotation;
  Setup_4_Info hw2_setup_4_info;
};

enum { OH_Links = 1, OH_Platform = 2, OH_Sphere = 4 };

void
World::init_graphics()
{
  vh.want_raytrace().init();
  vh.opt_record_every_time = true;
  vh.display_cb_set([&](){frame_callback();});
  vh.cbs_cmd_record.push_back( [&](vk::CommandBuffer& cb){ render(cb); });
  sphere.ppuni_light = &puni_light_curr;
  sphere.transform = &transform;

  opt_want_raytrace = true;

  if ( vh.have_raytrace )
    {
      raytrace.init();
      for ( auto b: { &bset_platform_mir, &bset_platform_tex } )
        b->usage_set
          ( vk::BufferUsageFlagBits::eVertexBuffer
            | vk::BufferUsageFlagBits::eStorageBuffer );
      buf_links_as_idx.init
        ( vh.qs, vk::BufferUsageFlagBits::eStorageBuffer, 0,
          vk::MemoryPropertyFlagBits::eDeviceLocal );
      buf_links_tex_coor.init
        ( vh.qs, vk::BufferUsageFlagBits::eStorageBuffer, 0,
          vk::MemoryPropertyFlagBits::eDeviceLocal );
      buf_links_vertex_g.init
        ( vh.qs, vk::BufferUsageFlagBits::eStorageBuffer, 0,
          vk::MemoryPropertyFlagBits::eDeviceLocal );
      buf_links_normal_g.init
        ( vh.qs, vk::BufferUsageFlagBits::eStorageBuffer, 0,
```

```
        vk::MemoryPropertyFlagBits::eDeviceLocal );
  }

buf_uni_common.init(vh.qs,vk::BufferUsageFlagBits::eUniformBuffer);
sphere.buf_uni_common = buf_uni_common;

lis_bufs =
  { &lis_pos1, &lis_pos2,
    &lis_v1, &lis_v2,
    &lis_b1_ydir, &lis_b2_ydir,
    &lis_tex_rad };

for ( auto b: lis_bufs )
  b->init(vh.dev_phys, vh.dev, vk::BufferUsageFlagBits::eStorageBuffer );

///
/// Graphical Model Initialization
///

opt_head_lock = false;
opt_tail_lock = false;
opt_tryout1 = opt_tryout2 = false;
opt_normal_sphere = true;
opt_sphere_true = true;
opt_tryoutf = 0;
variable_control.insert_linear(opt_tryoutf,"Tryout F",0.1);

eye_location = pCoor(24.2,11.6,-38.7);
eye_direction = pVect(-0.42,-0.09,0.9);

platform_xmin = -40; platform_xmax = 40;
platform_zmin = -40; platform_zmax = 40;

texid_syl.init
  ( vh.qs, P_Image_Read( vh.gp_root_get() / "vulkan" / "gpup.png", 255 ) );
texid_emacs.init( vh.qs, P_Image_Read("mult.png",-1) );
texid_hw.init( vh.qs, P_Image_Read("hw05-assign.png",255) );
sampler = vh.qs.dev.createSampler
  ( { {},
      vk::Filter::eLinear,  // magFilter
      vk::Filter::eLinear,  // minFilter
      vk::SamplerMipmapMode::eLinear,
      // Also: eRepeat, eMirroredRepeat, eClampToEdge, etc.
      vk::SamplerAddressMode::eRepeat,
      vk::SamplerAddressMode::eRepeat,
      vk::SamplerAddressMode::eRepeat,
      0.0f,  // mipLodBias
      false, // anisotropyEnable,
      16.0f, // maxAnisotropy
      false, vk::CompareOp::eNever,  // compareEnable, compareOp
      0.0f, VK_LOD_CLAMP_NONE,  // min and max LOD
      vk::BorderColor::eFloatOpaqueBlack,
      false // unnormalizedCoordinates
      } );
```

```
  sphere.texture_set(sampler,texid_emacs);

  opt_light_intensity = 100.2;
  light_location = pCoor(platform_xmax,platform_xmax,platform_zmin);
  uni_light.init(vh.qs,vk::BufferUsageFlagBits::eUniformBuffer);
  uni_light_all.init(vh.qs,vk::BufferUsageFlagBits::eUniformBuffer);
  uni_light_reflected.init(vh.qs,vk::BufferUsageFlagBits::eUniformBuffer);
  uni_light_ambient.init(vh.qs,vk::BufferUsageFlagBits::eUniformBuffer);

  //  variable_control.insert(opt_light_intensity,"Light Intensity");

  opt_move_item = MI_Eye;
  opt_pause = false;
  opt_single_time_step = false;
  opt_single_frame = false;

  platform_update();
  modelview_update();

  adj_t_stop = 0;
  adj_duration = 0.25;

  opt_sides = 20;
  opt_segments = 16;
  //  variable_control.insert(opt_sides,"Link Sides");
  //  variable_control.insert(opt_segments,"Link Segments",5,1);
  opt_hide_stuff = 0;

  if ( vh.have_raytrace )
    {
      raytrace.buf_light_set(uni_light_all);
    }
}

void
World::run()
{
  vh.message_loop_spin();

  uni_light.destroy();
  uni_light_ambient.destroy();
  uni_light_all.destroy();
  uni_light_reflected.destroy();
  transform.destroy();
  buf_uni_common.destroy();

  for ( auto b: lis_bufs ) b->destroy();

  pipe_platform_tex.destroy(); pipe_platform_mir.destroy();
  bset_platform_tex.destroy(); bset_platform_mir.destroy();
  pipe_set_2.destroy();
  pipe_set_2_sv.destroy();
```

```
  texid_syl.destroy();
  texid_emacs.destroy();
  texid_hw.destroy();
  vh.dev.destroySampler(sampler);
  shapes.destroy();
  sphere.destroy();

  buf_links_as_idx.destroy();
  buf_links_tex_coor.destroy();
  buf_links_normal_g.destroy();
  buf_links_vertex_g.destroy();
  rt_geo_platform_tex.destroy();
  rt_geo_platform_mir.destroy();
  rt_geo_links.destroy();
  comp_links.destroy();
  raytrace.destroy();

  vh.finish();
}

void
World::platform_update()
{
  const float tile_count = 19;
  const float ep = 1.00001;
  const float delta_x = ( platform_xmax - platform_xmin ) / tile_count * ep;
  const float zdelta = ( platform_zmax - platform_zmin ) / tile_count * ep;

  const float trmin = 0.05;
  const float trmax = 0.7;
  const float tsmin = 0;
  const float tsmax = 0.4;
  pColor color_platform( color_white * 0.5 );

  if ( !pipe_platform_mir )
    pipe_platform_mir
      .init( ff_state )
      .ds_set( color_lsu_spirit_purple )
      .ds_follow( puni_light_curr )
      .ds_follow( transform )
      .topology_set( vk::PrimitiveTopology::eTriangleList )
      .create();
  bset_platform_mir.reset( pipe_platform_mir ).tolerate_tcoords();

  if ( !pipe_platform_tex )
    pipe_platform_tex
      .init( ff_state )
      .ds_set_material(color_platform)
      .ds_follow( puni_light_curr )
      .ds_follow( transform )
      .topology_set( vk::PrimitiveTopology::eTriangleList )
      .ds_use( sampler, texid_syl )
      .create();
  bset_platform_tex.reset( pipe_platform_tex );
```

```
  const pVect platform_normal(0,1,0);
  pTCoor t00(trmin,tsmin), t01(trmin,tsmax), t10(trmax,tsmin), t11(trmax,tsmax);
  bool even = true;

  for ( int i = 0; i < tile_count; i++ )
    {
      const float x0 = platform_xmin + i * delta_x;
      const float x1 = x0 + delta_x;
      const float y = 0;
      for ( float z = platform_zmin; z < platform_zmax; z += zdelta )
        {
          auto& bset = even ? bset_platform_tex : bset_platform_mir;
          const float z1 = z+zdelta;
          pCoor p00(x0,y,z), p01(x0,y,z1), p10(x1,y,z), p11(x1,y,z1);

          bset << t11 << platform_normal << p00;
          bset << t10 << platform_normal << p01;
          bset << t00 << platform_normal << p11;

          bset << t11 << platform_normal << p00;
          bset << t00 << platform_normal << p11;
          bset << t01 << platform_normal << p10;

          even = !even;
        }
    }

  bset_platform_mir.to_dev();
  bset_platform_tex.to_dev();

  if ( vh.have_raytrace && !rt_geo_platform_mir )
    {
      pColor mirror_color(color_lsu_spirit_purple);
      mirror_color.a = -0.5;
      rt_geo_platform_mir
        .init( raytrace )
        .color_front_set( mirror_color )
        .bind( bset_platform_mir, RT_Geometry_Triangles );
      rt_geo_platform_tex
        .init( raytrace )
        .color_front_set( color_platform )
        .bind( VR_BUF_IDX_sampler, sampler, texid_syl )
        .bind( bset_platform_tex, RT_Geometry_Triangles );
    }
}

void
World::modelview_update()
{
  pMatrix_Translate center_eye(-eye_location);
  pMatrix_Rotation rotate_eye(eye_direction,pVect(0,0,-1));
  modelview = rotate_eye * center_eye;
}
```

```
void
World::render_objects(vk::CommandBuffer& cb, Render_Option option)
{
  //  const float shininess_ball = 5;
  pColor spec_color(1,1,1);
  //  pColor spec_light = color_white * opt_light_intensity;

  const bool hide_links = opt_hide_stuff & OH_Links;
  const bool hide_platform = opt_hide_stuff & OH_Platform;
  const bool hide_sphere = opt_hide_stuff & OH_Sphere;

  sphere.init(opt_sphere_slices);

  if ( option == RO_Shadow_Volumes )
    {
      if ( !hide_sphere )
        {
          sphere.light_pos = light_location;
          sphere.render_bunch_gather_sv(world_time);
          for ( Ball *ball: balls )
            sphere.render_shadow_volume(cb,ball->radius,ball->position);

          sphere.render_bunch_render_sv(cb);
        }

      if ( !hide_links )
        {
          render_link_start();
          for ( Link *link: links )
            if ( link->is_renderable ) render_link_gather(link);
          render_link_render_sv(cb);
        }
    }
  else
    {
      sphere.rt_geo_sphere.rt_wanted_set(!hide_sphere);
      if ( !hide_sphere )
        {
          sphere.render_bunch_gather(world_time);
          for ( Ball *ball: balls )
            {
              if ( ball->contact )
                sphere.color = color_gray;
              else if ( ball->mass > 0 && ball->mass < ball->mass_min )
                sphere.color = color_red;
              else if ( ball->mass > 0 && ball->locked )
                sphere.color = color_pale_green;
              else
                sphere.color = ball->color;

              sphere.color.a = ball->specularity ? -ball->specularity : 1;
              sphere.render
                (cb, ball->radius,ball->position,
```

```
                pMatrix_Rotation(ball->orientation));
            }

          sphere.render_bunch_render(cb, opt_sphere_true);
        }

      rt_geo_links.rt_wanted_set(!hide_links);
      if ( !hide_links )
        {
          render_link_start();
          for ( Link *link: links )
            if ( link->is_renderable ) render_link_gather(link);
          render_link_render(cb);
        }
    }

  rt_geo_platform_tex.rt_wanted_set(!hide_platform);
  rt_geo_platform_mir.rt_wanted_set(!hide_platform);
  if ( vh.raytrace_active() ) return;

  if ( hide_platform ) return;

  if ( option == RO_Shadow_Volumes ) return;

  //
  // Render Platform
  //
  pipe_platform_tex.record_draw(cb,bset_platform_tex);
}

void
World::render(vk::CommandBuffer& cb)
{
  const bool do_raytrace = vh.raytrace_active();

  /// Emit a Graphical Representation of Simulation State
  //

  // Understanding of the code below not required for introductory
  // lectures.

  // That said, much of the complexity of the code is to show
  // the ball shadow and reflection.


  const int win_width = vh.get_width();
  const int win_height = vh.get_height();
  const float aspect = float(win_width) / win_height;
  const float n_dist = 0.01;
  const float xr = .8 * n_dist;

  transform.eye_from_global_set( modelview );
  transform.clip_from_eye_set
    ( pMatrix_Frustum(-xr,xr,-xr/aspect,xr/aspect,n_dist,5000) );
```

```
// pMatrix_Frustum: left, right, bottom, top, near, far

vh.clearValues[0].color =
  vk::ClearColorValue( array<float, 4>( { { 0, 0, 0, 0.5 } } ) );

/// Lighting
//
auto& light_all = uni_light_all->cgl_LightSource[0];

light_all.position = transform.eye_from_global * light_location;

light_all.constantAttenuation = 0.5;
light_all.linearAttenuation = 1.0;
light_all.quadraticAttenuation = 0;

pColor ambient_color(0x555555);

uni_light_all->cgl_LightModel.ambient = ambient_color;

light_all.diffuse = opt_light_intensity * color_white;
light_all.ambient = color_black;
light_all.specular = opt_light_intensity * color_white;

uni_light_reflected = uni_light_all.get_val();
uni_light_reflected->cgl_LightSource[0].position =
  transform.eye_from_global * pMatrix_Scale(1,-1,1) * light_location;

uni_light_ambient = uni_light_all.get_val();
auto& light_amb = uni_light_ambient->cgl_LightSource[0];
light_amb.diffuse = color_black;
light_amb.specular = color_black;

uni_light = uni_light_all.get_val();
uni_light->cgl_LightModel.ambient = color_black;

uni_light.to_dev();
uni_light_ambient.to_dev();
uni_light_all.to_dev();
uni_light_reflected.to_dev();

puni_light_curr = &uni_light_all;

buf_uni_common->tryout =
  ivec4(opt_tryout1,opt_tryout2,opt_normal_sphere,false);
buf_uni_common->tryoutf = vec4(opt_tryoutf, 0, 0, 0);
buf_uni_common->light_pos_g = light_location;
if ( do_raytrace )
  {
    raytrace.buf_uni_common->opt_tryout = buf_uni_common->tryout;
    raytrace.buf_uni_common->opt_tryoutf = buf_uni_common->tryoutf;
    raytrace.buf_uni_common->opt_shadows = opt_shadows;
    raytrace.buf_uni_common->opt_mirror =
      opt_mirror ? opt_mirror_depth : 0;
    raytrace.buf_light_set(uni_light_all);
```

```
    }

  const double time_now = time_wall_fp();
  const bool blink_visible = int64_t(time_now*3) & 1;
# define BLINK(txt,pad) ( blink_visible ? txt : pad )

  vh.fbprintf("%s\n",frame_timer.frame_rate_text_get());

  vh.fbprintf
    ("Raytrace: %s ('R')   Compiled: %s\n",
     !vh.have_raytrace ? "NA" :
     opt_want_raytrace ? ( raytrace.rt_ready ? "ON " : "???" ) : "OFF",
#ifdef __OPTIMIZE__
     "WITH OPTIMIZATION"
#else
     BLINK("WITHOUT OPTIMIZATION","")
#endif
     );

  vh.fbprintf
    ("Time Step: %8d  World Time: %11.6f  %s\n",
     time_step_count, world_time,
     opt_pause ? BLINK("PAUSED, 'p' to unpause, SPC or S-SPC to step.","") :
     "Press 'p' to pause."
     );

  vh.fbprintf
    ("Eye location: [%5.1f, %5.1f, %5.1f]  "
     "Eye direction: [%+.2f, %+.2f, %+.2f]\n",
     eye_location.x, eye_location.y, eye_location.z,
     eye_direction.x, eye_direction.y, eye_direction.z);

  Ball& ball = *balls[0];

  vh.fbprintf
    ("Head Ball Pos  [%5.1f,%5.1f,%5.1f] Vel [%+5.1f,%+5.1f,%+5.1f]\n",
     ball.position.x,ball.position.y,ball.position.z,
     ball.velocity.x,ball.velocity.y,ball.velocity.z );

  vh.fbprintf
    ("%4zd Balls  %4zd Links  Physics: %s ('a' to change)  \n",
     balls.size(), links.size(), gpu_physics_method_str[opt_physics_method]);

  vh.fbprintf
    ("Hide: %c%c%c ('!@#')  Effect: %c%c ('or')  Sphere: %s  ('z')  "
     "Tryout 1: %s  ('y')  Tryout 2: %s  ('Y')  Normals: %s ('n')\n",
     opt_hide_stuff & OH_Sphere ? 'S' : '_',
     opt_hide_stuff & OH_Links ? 'L' : '_',
     opt_hide_stuff & OH_Platform ? 'P' : '_',
     opt_shadows ? 'S' : '_',
     opt_mirror ? 'M' : '_',
     opt_sphere_true ? "TRUE" : "TRI" ,
     opt_tryout1 ? BLINK("ON ","   ") : "OFF",
     opt_tryout2 ? BLINK("ON ","   ") : "OFF",
```

```
    opt_normal_sphere ? "SPHERE" : "TRI");

  vh.fbprintf("HW 2 Rotation: %s ('w')\n",
              hw2_rot_str[opt_hw2_rotation]);

  pVariable_Control_Elt* const cvar = variable_control.current;
  vh.fbprintf("VAR %s = %.5f  (TAB or '`' to change, +/- to adjust)\n",
                      cvar->name,cvar->get_val());

  // Render Marker for Light Source
  //
  shapes.record_tetrahedron(cb,transform,light_location,0.15);

  //
  // Render ball reflection.  (Will be blended with dark tiles.)
  //

  const bool hide_platform = opt_hide_stuff & OH_Platform;

  if ( do_raytrace )
    {
      raytrace.tform_update(transform);
      render_objects(cb,RO_Normally);
      raytrace.buf_uni_common.to_dev();

      return;
    }

  buf_uni_common.to_dev();

  vk::ColorComponentFlags clr_all
    ( vk::ColorComponentFlagBits::eR | vk::ColorComponentFlagBits::eG |
      vk::ColorComponentFlagBits::eB | vk::ColorComponentFlagBits::eA );

  if ( !hide_platform )
    {
      // Save current state of fixed-functionality state.
      //
      ff_state.push();

      if ( opt_mirror )
        {
          /// Mirror Effect
          //
          //  - Step 1:  Write location of mirrors into stencil buffer.
          //
          //  - Step 2a: Prepare a matrix that transforms to mirrored location.
          //  - Step 2b: Set up blending to blend mirrored objects with mirrors.
          //
          //  - Step 3:  Render all objects except mirrors.

          ///  Step 1: Write location of mirrors into stencil buffer.
          //
          //   Write stencil value 2 at location of dark (mirrored) tiles.
```

```
  //

  vk::StencilOpState s_op_apply
    ( vk::StencilOp::eReplace, vk::StencilOp::eKeep,
      vk::StencilOp::eKeep, vk::CompareOp::eNever, 2, 2, 2 );
  // sfail, spass, dfail, compare;  cmask, wmask, ref

  ff_state.pipelineDepthStencilStateCreateInfo_get()
    .setStencilTestEnable(true)
    .setFront(s_op_apply).setBack(s_op_apply);

  // Render mirrored tiles (but only to write stencil buffer).
  //
  pipe_platform_mir.record_draw(cb,bset_platform_mir);

  cb.nextSubpass(vk::SubpassContents::eInline); vh.qs.subpass++;

  // Prepare to write only stenciled locations.
  //
  vk::StencilOpState s_op_use
    ( vk::StencilOp::eKeep, vk::StencilOp::eKeep,
      vk::StencilOp::eKeep, vk::CompareOp::eEqual, 2, 2, 2 );
  // sfail, spass, dfail, compare;  cmask, wmask, ref
  //
  // Stencil test passes if stencil value = 2.
  // Don't change stencil buffer.

  ff_state.pipelineDepthStencilStateCreateInfo_get()
    .setFront(s_op_use).setBack(s_op_use);

  ///  Step 2a: Prepare a matrix that transforms to mirrored location.
  //   Use a transform that reflects objects to other side of platform.
  //
  const pMatrix prev = transform.eye_from_global;
  const pMatrix_Scale reflect(1,-1,1);
  transform.eye_from_global = prev * reflect;

  // Reflected front face should still be treated as the front face.
  //
  ff_state.pipelineRasterizationStateCreateInfo_get()
    .setFrontFace(vk::FrontFace::eClockwise);

  puni_light_curr = &uni_light_reflected;

  /// Step 3: Render all objects.

  render_objects(cb,RO_Mirrored);

  transform.eye_from_global = prev;

  ff_state.pipelineRasterizationStateCreateInfo_get()
    .setFrontFace(vk::FrontFace::eCounterClockwise);
}
```

```
    // blend operation, blend factors, blend constants
    ff_state.pipelineColorBlendStateCreateInfo_get()
      .setBlendConstants( array<float,4>{.5,.5,.5,.5} );

    ff_state.pipelineColorBlendAttachmentState_get() =
      vk::PipelineColorBlendAttachmentState
      ( true,                    // blendEnable
        vk::BlendFactor::eConstantAlpha,  // srcColorBlendFactor
        vk::BlendFactor::eConstantAlpha,  // dstColorBlendFactor
        vk::BlendOp::eAdd,       // colorBlendOp
        vk::BlendFactor::eOne,   // srcAlphaBlendFactor
        vk::BlendFactor::eZero,  // dstAlphaBlendFactor
        vk::BlendOp::eAdd,       // alphaBlendOp
        clr_all      // colorWriteMask
        );

    puni_light_curr = &uni_light_all;;
    pipe_platform_mir.record_draw(cb,bset_platform_mir);

    ff_state.pop();
  }

if ( !opt_shadows )
  {
    // Render.
    //
    render_objects(cb,RO_Normally);
  }
else
  {
    //
    // First pass, render using only ambient light.
    //
    puni_light_curr = &uni_light_ambient;

    // Send balls, tiles, and platform to opengl.
    //
    render_objects(cb,RO_Normally);

    //
    // Second pass, find un-shadowed areas and add on light0.
    //


    /// Write Shadow Volumes
    //
    vk::ClearAttachment clr_att
      ( vk::ImageAspectFlagBits::eStencil, 0,
        vk::ClearDepthStencilValue(0,0) );
    vk::ClearRect clr_rect( vk::Rect2D({}, vh.s_extent), 0, 1 );
    cb.clearAttachments( clr_att, clr_rect );

    ff_state.push();
```

```
vk::StencilOpState s_op_apply_f
  ( vk::StencilOp::eKeep, vk::StencilOp::eIncrementAndWrap,
    vk::StencilOp::eKeep, vk::CompareOp::eAlways, -1, -1, 0 );
vk::StencilOpState s_op_apply_b
  ( vk::StencilOp::eKeep, vk::StencilOp::eDecrementAndWrap,
    vk::StencilOp::eKeep, vk::CompareOp::eAlways, -1, -1, 0 );
// sfail, spass, dfail, compare; c-mask, w-mask, ref
// ref is -1 (all 1's) because we want to operate on all bits in
// stencil value.

ff_state.pipelineDepthStencilStateCreateInfo_get()
  .setStencilTestEnable(true)
  .setDepthWriteEnable(false)
  .setFront(s_op_apply_f).setBack(s_op_apply_b);

if ( !opt_shadow_volumes )
  {
    ff_state.pipelineColorBlendAttachmentState_get()
      .setColorWriteMask( {} );
  }
else
  {
    ff_state.pipelineColorBlendAttachmentState_get() =
      vk::PipelineColorBlendAttachmentState
      ( true,                    // blendEnable
        vk::BlendFactor::eConstantAlpha,  // srcColorBlendFactor
        vk::BlendFactor::eConstantAlpha,  // dstColorBlendFactor
        vk::BlendOp::eAdd,       // colorBlendOp
        vk::BlendFactor::eOne,   // srcAlphaBlendFactor
        vk::BlendFactor::eZero,  // dstAlphaBlendFactor
        vk::BlendOp::eAdd,       // alphaBlendOp
        clr_all      // colorWriteMask
        );

    ff_state.pipelineColorBlendStateCreateInfo_get()
      .setBlendConstants( array<float,4>{.5,.5,.5,.5} );

    ff_state.pipelineDepthStencilStateCreateInfo_get()
      .setDepthWriteEnable(true);
  }

// Write stencil with shadow locations based on shadow volumes
// cast by light0 (light_location).  Shadowed locations will
// have a positive stencil value.
//
render_objects(cb,RO_Shadow_Volumes);

cb.nextSubpass(vk::SubpassContents::eInline); vh.qs.subpass++;

// Use stencil test to prevent writes to shadowed areas.
//
vk::StencilOpState s_op_apply_fb
  ( vk::StencilOp::eKeep, vk::StencilOp::eKeep,
    vk::StencilOp::eKeep, vk::CompareOp::eEqual, -1, -1, 0 );
```

```
      // sfail, spass, dfail, compare;  c-mask, w-mask, ref
      //
      // Stencil test passes if stencil value == 0.

      ff_state.pipelineDepthStencilStateCreateInfo_get()
        .setFront(s_op_apply_fb).setBack(s_op_apply_fb)
        .setDepthCompareOp(vk::CompareOp::eEqual);

      // Allow pixels to be re-written.
      //
      vk::ColorComponentFlags cmask_all
        ( vk::ColorComponentFlagBits::eR | vk::ColorComponentFlagBits::eG |
          vk::ColorComponentFlagBits::eB | vk::ColorComponentFlagBits::eA );

      ff_state.pipelineColorBlendAttachmentState_get() =
        vk::PipelineColorBlendAttachmentState
        ( true,                   // blendEnable
          vk::BlendFactor::eOne,  // srcColorBlendFactor
          vk::BlendFactor::eOne,  // dstColorBlendFactor
          vk::BlendOp::eAdd,       // colorBlendOp
          vk::BlendFactor::eOne,   // srcAlphaBlendFactor
          vk::BlendFactor::eOne,  // dstAlphaBlendFactor
          vk::BlendOp::eAdd,       // alphaBlendOp
          cmask_all      // colorWriteMask
          );

      puni_light_curr = &uni_light;

      // Render.
      //
      render_objects(cb,RO_Normally);

      ff_state.pop();
    }

  puni_light_curr = &uni_light_all;
}


void
World::cb_keyboard()
{
  const int key = vh.keyboard_key_get();
  if ( !key ) return;
  pVect adjustment(0,0,0);
  pVect user_rot_axis(0,0,0);
  const bool kb_mod_s = vh.keyboard_shift;
  const bool kb_mod_c = vh.keyboard_control;
  const float move_amt = kb_mod_s ? 2.0 : kb_mod_c ? 0.08 : 0.4;

  switch ( key ) {
  case FB_KEY_LEFT: adjustment.x = -move_amt; break;
  case FB_KEY_RIGHT: adjustment.x = move_amt; break;
  case FB_KEY_PAGE_UP: adjustment.y = move_amt; break;
```

```
case FB_KEY_PAGE_DOWN: adjustment.y = -move_amt; break;
case FB_KEY_DOWN: adjustment.z = move_amt; break;
case FB_KEY_UP: adjustment.z = -move_amt; break;
case FB_KEY_DELETE: user_rot_axis.y = 1; break;
case FB_KEY_INSERT: user_rot_axis.y =  -1; break;
case FB_KEY_HOME:
  if ( opt_move_item == MI_Light ) adjustment.x = 1; else user_rot_axis.x = 1;
  break;
case FB_KEY_END: user_rot_axis.x = -1; break;
case '1': ball_setup_1(kb_mod_c); break;
case '2': ball_setup_2(); break;
case '3': ball_setup_3(); break;
case '4': ball_setup_45_wrapper(true); break;
case '5': ball_setup_45_wrapper(false); break;
case '6': ball_setup_6(); break;
case 'a':
  opt_physics_method++;
  if ( opt_physics_method == GP_ENUM_SIZE ) opt_physics_method = GP_cpu;
  break;
case 'b': opt_move_item = MI_Ball; break;
case 'B': opt_move_item = MI_Ball_V; break;
case 'e': case 'E': opt_move_item = MI_Eye; break;
case 'g': case 'G': opt_gravity = !opt_gravity; break;
case 'h': case 'H': opt_head_lock = !opt_head_lock; break;
case '!': opt_hide_stuff ^= OH_Sphere; break;
case '@': opt_hide_stuff ^= OH_Links; break;
case '#': opt_hide_stuff ^= OH_Platform; break;
case 't': case 'T': opt_tail_lock = !opt_tail_lock; break;
case 'l': case 'L': opt_move_item = MI_Light; break;
case 'r': opt_mirror = !opt_mirror; break;
case 'R':
  opt_want_raytrace = !opt_want_raytrace;
  world_time_link_update = 0;
  break;
case 'o': opt_shadows = !opt_shadows; break;
case 'O':
  opt_shadow_volumes = !opt_shadow_volumes;
  if ( opt_shadow_volumes ) opt_shadows = true;
  break;
case 'n': case 'N': opt_normal_sphere = !opt_normal_sphere; break;
case 'p': case 'P': opt_pause = !opt_pause; break;
case 's': case 'S': balls_stop(); break;
case 'v': case 'V': opt_shader++;
  if ( opt_shader >= sizeof(sh_names)/sizeof(sh_names[0]) ) opt_shader = 0;
  break;
  case 'w': case 'W':
    if ( kb_mod_c ) { balls_twirl(); break; }
    opt_hw2_rotation++;
    if ( opt_hw2_rotation == Rot_SIZE ) opt_hw2_rotation = 0;
    break;
case 'y': opt_tryout1 = !opt_tryout1; break;
case 'Y': opt_tryout2 = !opt_tryout2; break;
case 'z': opt_sphere_true = !opt_sphere_true; break;
case ' ':
```

```
  if ( kb_mod_s ) opt_single_time_step = true; else opt_single_frame = true;
  opt_pause = true;
  break;
case FB_KEY_TAB:
  if ( !kb_mod_s ) { variable_control.switch_var_right(); break; }
case 96: variable_control.switch_var_left(); break;
case '-':case '_': variable_control.adjust_lower(); break;
case '+':case '=': variable_control.adjust_higher(); break;
default: printf("Unknown key, %d\n",key); break;
}

gravity_accel.y = opt_gravity ? -opt_gravity_accel : 0;

lock_update();
sphere.bunch_invalidate();

// Update eye_direction based on keyboard command.
//
if ( user_rot_axis.x || user_rot_axis.y )
  {
    pMatrix_Rotation rotall(eye_direction,pVect(0,0,-1));
    user_rot_axis *= invert(rotall);
    eye_direction *= pMatrix_Rotation(user_rot_axis, M_PI * 0.03);
    modelview_update();
  }

// Update eye_location based on keyboard command.
//
if ( adjustment.x || adjustment.y || adjustment.z )
  {
    const double angle =
      fabs(eye_direction.y) > 0.99
      ? 0 : atan2(eye_direction.x,-eye_direction.z);
    pMatrix_Rotation rotall(pVect(0,1,0),-angle);
    adjustment *= rotall;

    switch ( opt_move_item ){
    case MI_Ball:
      adj_vector = adjustment;
      if ( adj_t_stop == 0 )
        adj_t_prev = adj_t_stop = world_time;
      adj_t_stop += adj_duration;
      break;
    case MI_Ball_V: balls_push(adjustment,0); break;
    case MI_Light:
      if ( key == FB_KEY_HOME )
        {
          pVect l_to_b(light_location,balls[0]->position);
          adjustment = 0.25 * l_to_b;
        }
      light_location += adjustment;
      break;
    case MI_Eye: eye_location += adjustment; break;
    default: break;
```

```
      }
      modelview_update();
    }

}




void
World::init(int argc, char **argv)
{
  chain_length = 14;

  opt_time_step_duration = 0.00003;
  //  variable_control.insert(opt_time_step_duration,"Time Step Duration");

  distance_relaxed = 15.0 / chain_length;
  opt_spring_constant = 2000;
  variable_control.insert(opt_spring_constant,"Spring Constant");

  opt_spring_damp = 0.7;
  variable_control.insert(opt_spring_damp,"Spring Damping").set_max(1);

  opt_friction_dynamic = 0.2;
  variable_control.insert(opt_friction_dynamic,"Dynamic Friction");

  opt_gravity_accel = 9.8;
  opt_gravity = true;
  gravity_accel = pVect(0,-opt_gravity_accel,0);
  variable_control.insert(opt_gravity_accel,"Gravity");

  opt_air_resistance = 0.01;
  variable_control.insert(opt_air_resistance,"Air Resistance");

  world_time = 0;
  time_step_count = 0;
  last_frame_wall_time = time_wall_fp();
  frame_timer.work_unit_set("Steps / s");

  world_time_link_update = 0;

  opt_mirror = true;
  opt_mirror_depth = 2;
  //  variable_control.insert(opt_mirror_depth,"Mirror Depth",1,0);
  opt_shadows = true;
  opt_shadow_volumes = false;
  opt_shader = SO_Plain;

  /// Init CUDA

  // Get information about GPU and its ability to run CUDA.
  //
  int device_count;
```

```
  cudaGetDeviceCount(&device_count); // Get number of GPUs.
  if ( device_count == 0 )
    {
      fprintf(stderr,"No GPU found, exiting.\n");
      exit(1);
    }

  // Print information about the available GPUs.
  //
  gpu_info_print();

  // Use course library routine to choose a good GPU for CUDA.
  //
  int dev = gpu_choose_index();
  CE(cudaSetDevice(dev));
  printf("Using GPU %d\n",dev);

  gpu_info.get_gpu_info(dev);
  cuda_setup(&gpu_info);

  // Print information about time_step routine.
  //
  printf("\nCUDA Routine Resource Usage:\n");

  int block_size_max = 4096;

  for ( int i=0; i<gpu_info.num_kernels; i++ )
    {
      printf("For %s:\n", gpu_info.ki[i].name);
      printf("  %6zd shared, %zd const, %zd loc, %d regs; "
             "%d max threads per block.\n",
             gpu_info.ki[i].cfa.sharedSizeBytes,
             gpu_info.ki[i].cfa.constSizeBytes,
             gpu_info.ki[i].cfa.localSizeBytes,
             gpu_info.ki[i].cfa.numRegs,
             gpu_info.ki[i].cfa.maxThreadsPerBlock);
      set_min(block_size_max,gpu_info.ki[i].cfa.maxThreadsPerBlock);
    }

  opt_physics_method = GP_cuda;
  gpu_const_data_stale = true;
  cpu_phys_data_stale = false;

  CE( cudaEventCreate(&frame_start_ce) );
  CE( cudaEventCreate(&frame_stop_ce) );

  opt_block_size = 32;

  if ( false )
  variable_control.insert_power_of_2
    (opt_block_size, "Block Size", 32, block_size_max);

  c.alloc_n_balls = 0;
  c.alloc_n_links = 0;
```

```
init_graphics();

opt_sphere_slices = 35;
sphere.opt_texture = true;
sphere.init(opt_sphere_slices);
//  variable_control.insert(opt_sphere_slices,"Sphere Slices");

const char* const links_shader_code_path = "links-shdr-links.cc";

const auto bvec_vs = vk::ShaderStageFlagBits::eVertex;
const auto bvec_vgs =
  vk::ShaderStageFlagBits::eVertex | vk::ShaderStageFlagBits::eGeometry;

pipe_set_2
  .init( ff_state )
  .ds_follow( transform )
  .ds_follow( puni_light_curr )
  .ds_uniform_use( "BIND_UNI_COMMON", buf_uni_common )
  .ds_set_material( color_dim_gray, color_salmon )
  .ds_use( sampler, texid_hw )
  .ds_storage_follow( "BIND_LINKS_POS1", lis_pos1, bvec_vs )
  .ds_storage_follow( "BIND_LINKS_POS2", lis_pos2, bvec_vs )
  .ds_storage_follow( "BIND_LINKS_V1", lis_v1, bvec_vs )
  .ds_storage_follow( "BIND_LINKS_V2", lis_v2, bvec_vs )
  .ds_storage_follow( "BIND_LINKS_B1_YDIR", lis_b1_ydir, bvec_vs )
  .ds_storage_follow( "BIND_LINKS_B2_YDIR", lis_b2_ydir, bvec_vs )
  .ds_storage_follow( "BIND_LINKS_TEX_RAD", lis_tex_rad, bvec_vgs )
  .shader_inputs_info_set( )
  .shader_code_set
  (links_shader_code_path,
   "vs_main_2(); ",     // Name of vertex shader main routine.
   "gs_main_2();",      // Name of geometry shader main routine.
   "fs_main();"         // Name of fragment shader main routine.
   )
  .topology_set( vk::PrimitiveTopology::eLineStrip )
  .create();

if ( vh.have_raytrace )
  comp_links
    .init( vh.qs )
    .ds_uniform_use( "BIND_UNI_COMMON", buf_uni_common )
    .ds_uniform_use( "BIND_TRANSFORM", raytrace.buf_uni_transform )
    .ds_storage_follow( "BIND_LINKS_POS1", lis_pos1 )
    .ds_storage_follow( "BIND_LINKS_POS2", lis_pos2 )
    .ds_storage_follow( "BIND_LINKS_V1", lis_v1 )
    .ds_storage_follow( "BIND_LINKS_V2", lis_v2 )
    .ds_storage_follow( "BIND_LINKS_B1_YDIR", lis_b1_ydir )
    .ds_storage_follow( "BIND_LINKS_B2_YDIR", lis_b2_ydir )
    .ds_storage_follow( "BIND_LINKS_TEX_RAD", lis_tex_rad )
    .ds_storage_follow( "BIND_LINKS_AS_IDX", buf_links_as_idx )
    .ds_storage_follow( "BIND_LINKS_TEX_COOR", buf_links_tex_coor )
    .ds_storage_follow( "BIND_LINKS_VERTEX_G", buf_links_vertex_g )
    .ds_storage_follow( "BIND_LINKS_NORMAL_G", buf_links_normal_g )
```

```
      .shader_code_set(links_shader_code_path, "comp_main(); ")
      .create();

  pipe_set_2_sv
    .init( ff_state ).ds_use( uni_light )
    .ds_follow( transform )
    .ds_uniform_use( "BIND_UNI_COMMON", buf_uni_common )
    .ds_storage_follow( "BIND_LINKS_POS1", lis_pos1 )
    .ds_storage_follow( "BIND_LINKS_POS2", lis_pos2 )
    .ds_storage_follow( "BIND_LINKS_V1", lis_v1 )
    .ds_storage_follow( "BIND_LINKS_V2", lis_v2 )
    .ds_storage_follow( "BIND_LINKS_B1_YDIR", lis_b1_ydir )
    .ds_storage_follow( "BIND_LINKS_B2_YDIR", lis_b2_ydir )
    .ds_storage_follow( "BIND_LINKS_TEX_RAD", lis_tex_rad )
    .shader_inputs_info_set( )
    .shader_code_set
    (links_shader_code_path,
     "vs_main_2_sv(); ",     // Name of vertex shader main routine.
     "gs_main_2_sv();",      // Name of geometry shader main routine.
     "fs_main_sv();"         // Name of fragment shader main routine.
     )
    .topology_set( vk::PrimitiveTopology::eLineStrip )
    .create();


  // 2023 Homework 2
  opt_omega = 10;
  variable_control.insert(opt_omega,"HW 2: Spin Rate");
  opt_hw2_rotation = Rot_Roll;

  srand48(4702);

  ball_setup_45_wrapper(true);
  return;
  // For benchmarking.
  opt_pause = true;
  opt_hide_stuff = OH_Links | OH_Platform;
  opt_shadows = false;
  opt_mirror = false;
}

Ball*
World::make_marker(pCoor position, pColor color)
{
  Ball* const ball = new Ball;
  ball->position = position;
  ball->locked = true;
  ball->velocity = pVect(0,0,0);
  ball->radius = 0.2;
  ball->mass = 0;
  ball->contact = false;
  ball->color = color;
  return ball;
}
```

```
void
World::lock_update()
{
  // This routine called when options like opt_head_lock might have
  // changed.

  // Update locked status.
  //
  if ( head_ball ) head_ball->locked = opt_head_lock;
  if ( tail_ball ) tail_ball->locked = opt_tail_lock;

  // Re-compute minimum mass needed for stability.
  //
  for ( Ball *ball: balls ) ball->spring_constant_sum = 0;
  const double dtis = pow( opt_time_step_duration, 2 );
  for ( Link *link: links )
    {
      Ball* const b1 = link->ball1;
      Ball* const b2 = link->ball2;
      b1->spring_constant_sum += opt_spring_constant;
      b2->spring_constant_sum += opt_spring_constant;
    }
  for ( Ball *ball: balls )
    {
      ball->mass_min = ball->spring_constant_sum * dtis;
      ball->constants_update();
    }
  gpu_const_data_stale = true;
}

void
World::balls_twirl()
{
  if ( !head_ball || !tail_ball ) return;

  pNorm axis(head_ball->position, tail_ball->position);

  for ( Ball *ball: balls )
    {
      pVect b_to_top(ball->position,head_ball->position);
      const float dist_along_axis = dot(b_to_top,axis);
      const float lsq = b_to_top.mag_sq() - dist_along_axis * dist_along_axis;
      if ( lsq <= 1e-5 ) { ball->velocity = pVect(0,0,0); continue; }
      const float dist_to_axis = sqrt(lsq);
      pNorm rot_dir = cross(b_to_top,axis);
      ball->velocity += 2 * dist_to_axis * rot_dir;
    }
}

void
World::objects_erase()
{
  balls.erase();
```

```
  links.erase();
}

Links
World::link_new(Ball *ball1, Ball *ball2, float stiffness)
{
  Links links_rv;
  assert( ball1->radius > 0 );
  assert( ball2->radius > 0 );

  Link* const rlink = new Link(ball1,ball2);
  links_rv += rlink;

  pNorm n12(ball1->position,ball2->position);
  const float rad_sum = ball1->radius + ball2->radius;
  pMatrix3x3 b1rot = ball1->omatrix;
  pMatrix3x3 b2rot = ball2->omatrix;
  pCoor ctr = ball1->position
    + ( ball1->radius + 0.5 * ( n12.magnitude - rad_sum ) ) * n12;

  pNorm b1_y = b1rot * pVect(0,1,0);
  pNorm b1_x = b1rot * pVect(1,0,0);
  bool b1_dir_is_y = fabs(dot(b1_y,n12)) < 0.999;
  pNorm b1_dir = b1_dir_is_y ? b1_y : b1_x;

  pNorm con_x = cross(b1_dir,n12);
  pVect con_y = cross(n12,con_x);
  rlink->b1_dir = mult_MTV( b1rot, con_y );
  rlink->b2_dir = mult_MTV( b2rot, con_y );

  const float lrad = stiffness * ball1->radius;

  const int n_springs = 4;
  const double delta_theta = 2 * M_PI / n_springs;

  for ( int i=0; i<n_springs; i++ )
    {
      const double theta = i * delta_theta;
      pVect convec = lrad * ( cos(theta) * con_x + sin(theta) * con_y );
      pCoor con = ctr + convec;
      pVect cb1 = mult_MTV( b1rot, con - ball1->position );
      pVect cb2 = mult_MTV( b2rot, con - ball2->position );
      links_rv += new Link(rlink, cb1, cb2, 0);
    }
  return links_rv;
}

///
/// CUDA Code
///

float4 tof4(pCoor c)
{
  float4 f4;
```

```
  f4.x = c.x;
  f4.y = c.y;
  f4.z = c.z;
  f4.w = c.w;
  return f4;
}

float4 tof4(pVect c)
{
  float4 f4;
  f4.x = c.x;
  f4.y = c.y;
  f4.z = c.z;
  f4.w = 0;
  return f4;
}

pCoor topc(float4 c)
{
  pCoor f4;
  f4.x = c.x;
  f4.y = c.y;
  f4.z = c.z;
  f4.w = c.w;
  return f4;
}

void
World::data_cpu_to_gpu_constants()
{
  c.platform_xmin = platform_xmin;
  c.platform_xmax = platform_xmax;
  c.platform_zmax = platform_zmax;
  c.platform_zmin = platform_zmin;

  c.n_balls = balls.size();
  c.n_links = links.size();
  c.opt_tryout1 = opt_tryout1;
  c.opt_tryout2 = opt_tryout2;
  c.opt_tryoutf = opt_tryoutf;
  c.opt_spring_constant = opt_spring_constant;
  c.opt_spring_damp = opt_spring_damp;
  c.opt_air_resistance = opt_air_resistance;
  c.opt_friction_dynamic = opt_friction_dynamic;
  c.opt_head_lock = opt_head_lock;
  c.opt_tail_lock = opt_tail_lock;
  c.gravity_accel = gravity_accel;

  data_cpu_to_gpu_common(&c);
}


void
World::data_cpu_to_gpu_dynamic()
```

```
{
  const int n_balls = balls.size();
  const int n_links = links.size();
  const bool need_alloc =
    n_balls > c.alloc_n_balls || n_links > c.alloc_n_links;
  const bool need_init = c.alloc_n_balls == 0;

#define ALLOC_MOVE_ZERO(n_balls,balls,memb,do_move,do_zero)                \
  {                                                                        \
    const int size_elt_bytes = sizeof(c.balls.memb[0]);                    \
    const int size_bytes = n_balls * size_elt_bytes;                       \
    if ( need_alloc )                                                      \
      {                                                                    \
        if ( !need_init )                                                  \
          {                                                                \
            CE( cudaFree( c.balls.memb ) );                                \
            free(c.h_##balls.memb);                                        \
          }                                                                \
        CE( cudaMalloc( &c.balls.memb, size_bytes ) );                     \
        c.h_##balls.memb = (decltype(c.balls.memb)) malloc( size_bytes );  \
      }                                                                    \
    if ( do_move )                                                         \
      {                                                                    \
        for ( int i=0; i<n_balls; i++ )                                    \
          c.h_##balls.memb[i] = balls[i]->memb;                            \
        CE( cudaMemcpy                                                     \
            ( c.balls.memb, c.h_##balls.memb,                              \
              size_bytes, cudaMemcpyDefault ) );                           \
      }                                                                    \
    else if ( do_zero )                                                    \
      {                                                                    \
        CE( cudaMemset( c.balls.memb, 0, size_bytes ) );                   \
      }                                                                    \
  }

#define MOVEc(num,ptr,memb) \
  ALLOC_MOVE_ZERO(num,ptr,memb,gpu_const_data_stale,0)
#define MOVE(num,ptr,memb) ALLOC_MOVE_ZERO(num,ptr,memb,1,0)
#define ALLOC(num,ptr,memb) ALLOC_MOVE_ZERO(num,ptr,memb,0,0)
#define ZERO(num,ptr,memb) ALLOC_MOVE_ZERO(num,ptr,memb,0,1)
  MOVEc(n_balls,balls,position);
  MOVEc(n_balls,balls,velocity);
  ZERO(n_balls,balls,force);

  MOVEc(n_balls,balls,orientation);
  MOVEc(n_balls,balls,omatrix);
  MOVEc(n_balls,balls,omega);
  ZERO(n_balls,balls,torque);
  MOVEc(n_balls,balls,mass);
  MOVEc(n_balls,balls,radius);
  MOVEc(n_balls,balls,fdt_to_do);
  MOVEc(n_balls,balls,locked);

  MOVEc(n_links,links,ball1_idx);
```

```
  MOVEc(n_links,links,ball2_idx);
  MOVEc(n_links,links,cb1);
  MOVEc(n_links,links,cb2);
  MOVEc(n_links,links,distance_relaxed);
  MOVEc(n_links,links,is_simulatable);
  MOVEc(n_links,links,is_surface_connection);

#undef MOVE
#undef MOVEc
#undef ALLOC
#undef ZERO
#undef ALLOC_MOVE
  if ( need_alloc )
    {
      c.alloc_n_balls = n_balls;
      c.alloc_n_links = n_links;
      data_cpu_to_gpu_common(&c);
    }
  gpu_const_data_stale = false;
}

void
World::data_gpu_to_cpu_dynamic()
{
  const int n_balls = balls.size();

#define MOVE(n_balls,balls,memb)                                        \
  {                                                                     \
    const int size_elt_fr_bytes = sizeof(c.balls.memb[0]);              \
    const int size_bytes = n_balls * size_elt_fr_bytes;                 \
    CE( cudaMemcpy                                                      \
        ( c.h_##balls.memb, c.balls.memb, size_bytes,                   \
          cudaMemcpyDeviceToHost ) );                                   \
    for ( int i=0; i<n_balls; i++ )                                     \
      balls[i]->memb = c.h_##balls.memb[i];                             \
  }

  MOVE(n_balls,balls,position);
  MOVE(n_balls,balls,velocity);
  MOVE(n_balls,balls,orientation);
  MOVE(n_balls,balls,omatrix);
  MOVE(n_balls,balls,omega);

#undef MOVE

}


///
/// Physical Simulation Code
///

 /// Initialize Simulation
//
```

```
void
World::ball_setup_1(bool big)
{
  // Arrange balls to form sort of tree.

  last_setup = 1;

  double ball_rad = 1;  // Warning: Re-assigned, changes behavior of nb.

  pCoor first_pos(17.2,ball_rad,-20.2);

  pVect dir_up(0.1,3,0.1);
  pVect dir_z(3*distance_relaxed,0,0);
  pVect dir_x(0,0,3*distance_relaxed);

  // Remove objects from the simulated objects lists, balls and links.
  // The delete operator is used on objects in the lists.
  //
  objects_erase();

  auto nb =
    [&] (pCoor pos) {
      Ball* const ball = new Ball;
      ball->position = pos;
      ball->locked = false;
      ball->velocity = pVect(0,0,0);
      ball->radius = ball_rad;
      ball->contact = false;
      balls += ball;
      return ball;
      };

  tail_ball = nb(first_pos);
  tail_ball->specularity = 0.5;

  pCoor last_pos = tail_ball->position;

  const int n_tiers = big ? 32 : 8;
  const double shrink_final = 0.17;
  const double shrink = pow(shrink_final,1.0/n_tiers);

  Balls tail_balls;
  for ( int i=1; i<n_tiers; i++ )
    {
      ball_rad *= shrink;
      Ball* const ball = nb(last_pos + (i==1?1.0:ball_rad) * dir_up);
      ball->specularity = 0.5;
      tail_balls += ball;
      links += link_new(balls[balls-2],ball,0.8);
      last_pos = ball->position;
    }

  head_ball = balls[balls-1];
```

```
  pNorm dir_zn(dir_z);
  pNorm dir_xn(dir_x);
  for ( Ball *ball: tail_balls )
    {
      if ( ball == head_ball ) break;
      pNorm to_top(ball->position,head_ball->position);
      const float r = to_top.magnitude;
      int num_hairs = max(4,int(7*r));
      ball_rad = ball->radius / 4;
      const float delta_theta = 2 * M_PI / num_hairs;
      for ( int i=0; i<num_hairs; i++ )
        {
          const float theta = i * delta_theta;
          pCoor pos = ball->position +
            r * cos(theta) * dir_xn + r * sin(theta) * dir_zn;
          Ball* const ballr = nb(pos);
          links += link_new(ballr,ball);
        }
    }

  opt_head_lock = false;
  opt_tail_lock = true;
  setup_at_end();
}

void
World::ball_setup_2()
{
  // Arrange balls to form a suspended thing.

  last_setup = 2;

  pCoor first_pos(17.2,14.8f,-20.2);

  pVect dir_dn(0.5,first_pos.y/8,0.5);
  pVect dir_z(3*distance_relaxed,0,0);
  pVect dir_x(0,0,3*distance_relaxed);

  // Remove objects from the simulated objects lists, balls and links.
  // The delete operator is used on objects in the lists.
  //
  objects_erase();

  float ball_rad = 0.3;  // Warning: Re-assigned, changes behavior of nb.

  auto nb =
    [&] (pCoor pos) {
      Ball* const ball = new Ball;
      ball->position = pos;
      ball->locked = false;
      ball->velocity = pVect(0,0,0);
      ball->radius = ball_rad;
      ball->contact = false;
```

```
      ball->specularity = 0.5;
      balls += ball;
      return ball;
      };

  head_ball = nb(first_pos);
  for ( int j: {-1,1} )
    for ( int i: {-1,0,1} )
      {
        Ball* const ball = nb(first_pos - dir_dn + i * dir_x + j * dir_z );
        if ( i != 0 ) links += link_new(head_ball,ball);
        if ( i != -1 ) links += link_new(balls[balls-2],ball);
        if ( j == 1 && i != 0 )  links += link_new(balls[balls-4],ball);
      }

  Ball* const tail_start = nb(first_pos - dir_dn);
  links += link_new(balls[balls-6],tail_start);
  links += link_new(balls[balls-3],tail_start);
  pCoor last_pos = tail_start->position;
  Balls tail_balls;
  for ( int i=1; i<5; i++ )
    {
      Ball* const ball = nb(last_pos - 0.5 * dir_dn);
      tail_balls += ball;
      links += link_new(balls[balls-2],ball);
      last_pos = ball->position;
    }

  tail_ball = balls[balls-1];

  pNorm dir_zn(dir_z);
  pNorm dir_xn(dir_x);
  int num_hairs = 20;
  ball_rad = 0.1;
  for ( Ball *ball: tail_balls )
    {
      const float delta_theta = 2 * M_PI / num_hairs;
      for ( int i=0; i<num_hairs; i++ )
        {
          const float theta = i * delta_theta;
          const float r = ball->radius * 10;
          pCoor pos = ball->position +
            r * cos(theta) * dir_xn + r * sin(theta) * dir_zn;
          Ball* const ballr = nb(pos);
          links += link_new(ballr,ball);
        }
    }

  opt_head_lock = true;    // Head ball will be frozen in space.
  opt_tail_lock = false;   // Tail ball can move freely.
  setup_at_end();
}

void
```

```
World::ball_setup_3()
{
  // Arrange links to form what I would eventually learn to be a
  // corona virus.

  const bool sarcov = true;

  last_setup = 3;
  objects_erase();

  pCoor first_pos(17.2,14.8f,-10.2);

  float ball_radius = 1;

  auto nb =
    [&] (pCoor pos) {
      Ball* const ball = new Ball;
      ball->position = pos;
      ball->locked = false;
      ball->velocity = pVect(0,0,0);
      ball->radius = ball_radius;
      ball->contact = false;
      balls += ball;
      return ball;
      };

  head_ball = nb(first_pos);
  head_ball->specularity = 0.5;
  ball_radius = ball_radius * ( sarcov ? 0.08 : .1 );
  const int n_spikes = sarcov ? 100 : 500;

  for ( int i=0; i<n_spikes; i++ )
    {
      pNorm dir(0.5-drand48(),0.5-drand48(),0.5-drand48());
      const float len = head_ball->radius * ( sarcov ? 1.5 : 3.0 );
      Ball* const ball = nb(first_pos + len * dir);
      links += link_new(ball,head_ball,2);
    }

  tail_ball = balls[1];

  opt_head_lock = false;    // Head ball will be frozen in space.
  opt_tail_lock = false;   // Tail ball can move freely.

  setup_at_end();
}

pVect vec_rand() {return pVect(drand48(),drand48(),drand48());}

void
World::ball_setup_45_wrapper(bool is_4)
{
  Setup_4_Info& si = hw2_setup_4_info;
  const bool kb_mod_c = vh.keyboard_control;
```

```
  objects_erase();

  if ( si.n_balls == 0 || !kb_mod_c )
    {
      pCoor center_pos = si.center_pos = pCoor(17.2,14.8f,-10.2) + 2*vec_rand();

      si.n_balls = 4 * ( 4 + random()%4 );
      float ro = si.ring_outer_radius = 5 + 2 * drand48();
      si.ball_radius = 0.2 * 2 * M_PI * si.ring_outer_radius / si.n_balls;

      pNorm c_to_eye(center_pos,eye_location);
      si.ring_normal = c_to_eye + 0.5 * vec_rand();
      pVect az = si.ring_normal;

      pNorm ax = fabs(az.x) < fabs(az.y)
        ? pVect(0,az.z,-az.y) : pVect(az.z,0,-az.x);
      pVect ay(az,ax);

      float alpha1 = 2 * M_PI * drand48();
      float alpha2 = 2 * M_PI * drand48();
      si.p1 = center_pos +
        ro * drand48() * ( cosf(alpha1) * ax + sinf(alpha1) * ay );
      si.p2 = center_pos +
        ro * drand48() * ( cosf(alpha2) * ax + sinf(alpha2) * ay );

    }

  si.rotation = opt_hw2_rotation;
  si.omega = opt_omega;

  if ( is_4 ) ball_setup_4(si); else ball_setup_5(si);

  Ball* const m1 = make_marker(si.p1,0.5*color_yellow);
  Ball* const m2 = make_marker(si.p2,0.5*color_yellow);
  m1->radius = m2->radius = si.ball_radius * 1.2;
  m1->world_time_expire = m2->world_time_expire = 3;
  balls += m1;
  balls += m2;
  setup_at_end();
}


void
World::ball_setup_4(const Setup_4_Info& si)
{
  /// Homework 2 SOLUTION in this routine


  last_setup = 4;

  // Default settings for balls.
  //
  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;
  //
  // For colors see gp/include/colors.h

  // 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;

  pCoor center_pos = si.center_pos;
  const int n_balls = si.n_balls;
  const float d_theta = 2 * M_PI / n_balls;

  // Place balls in a circle.
  //
  for ( int i=0; i<n_balls; i++ )
    {
      const float theta = d_theta * i;
      pCoor pos = 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;
    }

  const float link_stiffness = 10;

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

  // Place a ball in the center and connect it to all the other balls.
  if ( opt_tryout1 )
    {
      Ball* const center_ball = new Ball(center_pos,ball_default);
      balls += center_ball;

      for ( int i=0; i<n_balls; i++ )
        links += link_new( center_ball, balls[i], 2 );
```

```
  }

/// Problem 1
//
//  Construct a second ring in plane of first ring and which contains
//  points p1 and p2 (below) on opposite sides.

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

/// SOLUTION -- Problem 1

// Construct vector between two inner-ring points.
//
pNorm p1p2(p1,p2);

// Compute radius and center of inner ring.
//
float r2 = 0.5f * p1p2.magnitude;
//

pCoor c2 = p1 + r2 * p1p2;

// Determine whether inner ring intersects outer ring.
//
pNorm c_to_c2(center_pos,c2);
const bool too_big = c_to_c2.magnitude + r2 > si.ring_outer_radius;

// Construct vectors used for find points on inner ring.
//
pVect vx2 = ax * r2;
pVect vy2 = ay * r2;

// Place balls on inner ring and link to outer ring.
//
for ( int i=0; i<n_balls; i++ )
  {
    const float theta = d_theta * i;
    pCoor pos = c2 + vx2 * cosf(theta) + vy2 * sinf(theta);

    // Construct inner ring ball.
    //
    Ball* ball = new Ball(pos,ball_default);
    balls += ball;
    ball->color = too_big ? color_red : color_lsu_spirit_purple;

    // Construct a link between inner ring ball and outer ring ball.
    //
    links += link_new( balls[i], ball, link_stiffness );
    //
    // Note: balls[i] is the outer ring ball at same angle as ball.
  }

// Construct links between balls in inner ring.
```

```
  //
  for ( int i=0; i<n_balls; i++ )
    links +=
      link_new( balls[n_balls+i], balls[n_balls+(i+1)%n_balls], link_stiffness);

  /// SOLUTION -- Problem 2
  //
  // Assign velocity to balls so that wheel rotates around axis (if
  // any) specified by si.rotation.

  if ( si.rotation != Rot_None )
    {
      const float m1 = balls[0]->mass, m2 = balls[n_balls]->mass;
      //
      // Note: Masses are only needed when m1 != m2.

      // Rotation axis for Rot_Spin should connect the center of mass
      // of each ring.
      //
      pVect cp_to_c2(center_pos,c2);

      // Compute center of mass of wheel. Rotation axis for both Rot_Spin
      // and Rot_Roll pass through this point.
      //
      pCoor cg_pos = center_pos + m2/(m1+m2) * cp_to_c2;
      //
      // Note: If the density and radii of both rings' balls are the
      // same then: cg_pos = center_pos + 0.5 * cp_to_c2;

      // Choose the desired axis direction.
      //
      pNorm spin_axis = si.rotation == Rot_Roll ? az : cp_to_c2;

      for ( Ball* ball: balls )
        {
          pCoor pos = ball->position;

          // Find the point on the spin axis closest to pos.
          //
          pCoor axis_cp =
            si.rotation == Rot_Roll
            ? cg_pos
            : cg_pos + spin_axis * dot( spin_axis, pVect(cg_pos,pos) );

          // Compute the vector from the axis point to pos.
          //
          pVect cp_to_p(axis_cp,pos);

          // Compute the velocity of the ball.
          //
          ball->velocity = si.omega * cross(cp_to_p,spin_axis);
        }
    }
```

```
  tail_ball = balls.back();
  head_ball = balls.front();

  opt_head_lock = false; // If true, head ball locked in place.
  opt_tail_lock = false;
}

void
World::ball_setup_5(const Setup_4_Info& si)
{
  // Use this routine for trying things out or as a pristine copy of
  // ball_setup_4.

  last_setup = 5;

  // Default settings for balls.
  //
  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;
  //
  // For colors see gp/include/colors.h

  // 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;

  pCoor center_pos = si.center_pos;
  const int n_balls = si.n_balls;
  const float d_theta = 2 * M_PI / n_balls;

  // Place balls in a circle.
  //
  for ( int i=0; i<n_balls; i++ )
    {
      const float theta = d_theta * i;
      pCoor pos = 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;
  }

const float link_stiffness = 10;

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

// Place a ball in the center and connect it to all the other balls.
if ( opt_tryout1 )
  {
    Ball* const center_ball = new Ball(center_pos,ball_default);
    balls += center_ball;

    for ( int i=0; i<n_balls; i++ )
      links += link_new( center_ball, balls[i], 2 );
  }

/// Problem 1
//
//  Construct a second ring in plane of first ring and which contains
//  points p1 and p2 (below) on opposite sides.

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


/// Problem 2
//
//  Set the velocity of all of the balls to zero if si.rotation == Rot_None,
//  or to a velocity so that they rotate around the appropriate axis.

switch( si.rotation ) {
case Rot_None:
  break;

case Rot_Roll: // Use si.ring_normal for spin axis.
  {
    pNorm spin_axis = si.ring_normal;
    pCoor c_of_mass = center_pos;

    for ( Ball* ball: balls )
      {
        pVect cm_to_ball( c_of_mass, ball->position );

        ball->velocity = si.omega * cross( cm_to_ball, spin_axis );
      }
  }
  break;

case Rot_Spin:
  // Find an axis orthogonal to si.ring_normal that is
```

```
    // an eigenvector of the inertia tensor.

    break;

  }

  tail_ball = balls.back();
  head_ball = balls.front();

  opt_head_lock = false; // If true, head ball locked in place.
  opt_tail_lock = false;
}



void
World::ball_setup_6()
{
  last_setup = 6;
  objects_erase();

  pCoor center_pos(17.2,14.8f,-10.2);

  const int n_balls = 20;
  const float crad = 5;

  float ball_radius = 0.3 * 2 * M_PI * crad / n_balls;

  const float d_theta = 2 * M_PI / n_balls;
  //  pVect vz(drand48()*2-1,drand48()*.1,drand48()*2-1);
  pVect vz(1,-2,0);
  pNorm ax( vz.y, -vz.x, 0 );
  pNorm ay = cross(vz,ax);
  pVect vx = crad * ax;
  pVect vy = crad * ay;

  pNorm spin_axis = vz;
  float omega = opt_omega;
  opt_spring_damp = .4;

  auto ball_new =
    [&] (pCoor pos) {
      Ball* const ball = new Ball;
      ball->color = color_gray;
      ball->position = pos;
      ball->locked = false;
      ball->velocity = pVect(0,0,0);
      ball->radius = ball_radius;
      ball->contact = false;
      balls += ball;
      return ball;
      };

  for ( int i=0; i<n_balls; i++ )
```

```
    {
      const float theta = d_theta * i;
      pCoor pos = center_pos + vx * cosf(theta) + vy * sinf(theta);
      Ball* ball = ball_new(pos);
      pCoor axis_cp =
        center_pos + spin_axis * dot( spin_axis, pVect(center_pos,pos) );
      pVect cp_to_p(axis_cp,pos);
      ball->velocity = omega * cross(cp_to_p,spin_axis);
    }

  balls[0]->color = color_blue;

  for ( int i=0; i<n_balls; i++ )
    links += link_new( balls[i], balls[(i+1)%n_balls], 2 );

  pNorm az(vz);
  pCoor top_pos = center_pos + az * crad * 1.5;
  pCoor bot_pos = center_pos - az * crad * 1.5;

  if ( true )
    {
      head_ball = ball_new(top_pos);
      tail_ball = ball_new(bot_pos);

      for ( int i=0; i<n_balls; i += 3 )
        {
          links += link_new( head_ball, balls[i], 2 );
          links += link_new( tail_ball, balls[i], 2 );
        }
    }


  opt_head_lock = false;    // Head ball will be frozen in space.
  opt_tail_lock = false;   // Tail ball can move freely.

  setup_at_end();

}




void
World::setup_at_end()
{
  lock_update();
  time_step_count = 0;
  world_time = 0;
  world_time_link_update = -1;
  for ( uint i=0; i<balls.size(); i++ ) balls[i]->idx = i;
  for ( Link *l : links )
    { l->ball1_idx = l->ball1->idx; l->ball2_idx = l->ball2->idx; }
}
```

```
 /// Advance Simulation State by delta_t Seconds
//
void
World::time_step_cpu(double delta_t)
{
  const int n_balls = balls;
  const int n_links = links;

  // Smoothly move ball in response to user input.
  //
  if ( adj_t_stop )
    {
      const double dt = min(world_time,adj_t_stop) - adj_t_prev;
      pVect adj = dt/adj_duration * adj_vector;
      balls_translate(adj,0);
      adj_t_prev = world_time;
      if ( world_time >= adj_t_stop ) adj_t_stop = 0;
    }

#pragma omp parallel for
  for ( int bi=0; bi<n_balls; bi++ )
    {
      Ball* const ball = balls[bi];
      assert( ball->fdt_to_do );
      ball->force = ball->mass * gravity_accel;
      ball->torque = pVect(0,0,0);
    }

#pragma omp parallel for
  for ( int i=0; i<n_links; i++ )
    {
      Link* const link = links[i];
      if ( !link->is_simulatable ) continue;

      //
      // Spring Force from Neighbor Balls
      //

      Ball* const ball1 = link->ball1;
      Ball* const ball2 = link->ball2;

      /// Notes:
      //
      // link->cb1
      //   Coordinate where link touches ball1, in ball1's local space.
      //
      // ball1->omatrix
      //   Ball's orientation transformation ..
      //   .. which transforms vectors in ball1's local space to object space.
      //

      // Find position and velocity of the point where the link touches
```

```
// the surface of ball 1 ...
//
pVect dir1 = ball1->omatrix * link->cb1;
pCoor pos1 = ball1->position + dir1;
pVect vel1 = ball1->velocity + ball1->point_rot_vel(dir1);

// ... and ball 2.
//
pVect dir2 = ball2->omatrix * link->cb2;
pCoor pos2 = ball2->position + dir2;
pVect vel2 = ball2->velocity + ball2->point_rot_vel(dir2);

// Construct a normalized (Unit) Vector from ball to neighbor
// based on link connection points and ball centers.
//
pNorm link_dir(pos1,pos2);
pNorm c_to_c(ball1->position,ball2->position);

const float link_length = link_dir.magnitude;

// Compute the speed of ball's end of link towards neighbor's end of link.
//
pVect delta_v = vel2 - vel1;
float delta_s = dot( delta_v, link_dir );

// Compute by how much the spring is stretched (positive value)
// or compressed (negative value).
//
const float spring_stretch = link_length - link->distance_relaxed;

// Determine whether spring is gaining energy (whether its length
// is getting further from its relaxed length).
//
const bool gaining_e = ( delta_s > 0.0 ) == ( spring_stretch > 0 );

const float blend_length_factor = 0.1; // Just a guess.
const float blend_s = blend_length_factor * c_to_c.magnitude;
const float blend = min( 1.0f, fabs(delta_s) / blend_s );

// Use a smaller spring constant when spring is loosing energy,
// a quick and dirty way of simulating energy loss due to spring
// friction.
//
const float spring_constant =
  gaining_e ? opt_spring_constant :
  opt_spring_constant * ( 1.0f - blend + blend * opt_spring_damp );

const float force_mag = spring_constant * spring_stretch;
pVect spring_force_12 = force_mag * link_dir;

// Apply forces affecting linear momentum.
//
link->spring_force_12 = spring_force_12;
```

```
      if ( ! link->is_surface_connection ) continue;

      // Apply torque.
      //
      link->torque1 = cross(pNorm(dir1), spring_force_12);
      link->torque2 = cross(pNorm(dir2), spring_force_12);
    }

  // Note: Because two links can reference the same ball this should
  // not be done in parallel.
  for ( Link *link: links )
    {
      if ( !link->is_simulatable ) continue;
      Ball* const ball1 = link->ball1;
      Ball* const ball2 = link->ball2;
      ball1->force += link->spring_force_12;
      ball2->force -= link->spring_force_12;
      ball1->torque += link->torque1;
      ball2->torque -= link->torque2;
    }

  ///
  /// Update Position of Each Ball
  ///

#pragma omp parallel for
  for ( int bi=0; bi<n_balls; bi++ )
    {
      Ball* const ball = balls[bi];
      if ( ball->locked )
        {
          ball->velocity = pVect(0,0,0);
          ball->omega = pVect(0,0,0);
          continue;
        }

      // Update Velocity
      //
      // This code assumes that force on ball is constant over time
      // step. This is clearly wrong when balls are moving with
      // respect to each other because the springs are changing
      // length. This inaccuracy will make the simulation unstable
      // when spring constant is large for the time step.
      //
      const float mass = max( ball->mass, ball->mass_min );

      pVect delta_v = ( ball->force / mass ) * delta_t;
      const float r = ball->radius;

      if ( platform_collision_possible(ball->position)
           && fabs(ball->position.y) < r )
        {
          const bool above = ball->position.y >= 0;
          const float spring_constant_plat =
```

```
          ball->velocity.y < 0 == above ? 100000 : 50000;
        const float fric_coefficient = opt_friction_dynamic;
        const float force_away = ( above ? 1 : -1 )
          * ( r - fabs(ball->position.y) ) * spring_constant_plat;
        const float delta_v_away = force_away / mass * delta_t;
        const float fric_force_mag = fric_coefficient * force_away;
        pNorm surface_v(ball->velocity.x,0,ball->velocity.z);
        const float delta_v_surf = fric_force_mag / mass * delta_t;

        if ( delta_v_surf > surface_v.magnitude )
          {
            // Ignoring other forces?
            delta_v =
              pVect(-ball->velocity.x,delta_v.y,-ball->velocity.z);
          }
        else
          {
            delta_v -= delta_v_surf * surface_v;
          }
        delta_v.y += delta_v_away;
      }

    ball->velocity += delta_v;

    // Air Resistance
    //
    const pNorm ball_dir(ball->velocity);
    const float delta_s_air_raw =
      opt_air_resistance * ball_dir.mag_sq * delta_t;
    const float delta_s_air = min( ball_dir.magnitude, delta_s_air_raw );
    ball->velocity -= delta_s_air * ball_dir;

    // Update Position
    //
    // Assume that velocity is constant.
    //
    ball->position += ball->velocity * delta_t;

    ball->omega += delta_t * ball->fdt_to_do * ball->torque;

    pNorm axis(ball->omega);

    // Update Orientation
    //
    // If ball isn't spinning fast skip expensive rotation.
    //
    if ( axis.mag_sq < 0.000001 ) continue;

    // Update ball orientation.
    //
    ball->orientation_set
      ( pQuat(axis,delta_t * axis.magnitude) * ball->orientation );
  }
}
```

```
bool
World::platform_collision_possible(pCoor pos)
{
  // Assuming no motion in x or z axes.
  //
  return pos.x >= platform_xmin && pos.x <= platform_xmax
    && pos.z >= platform_zmin && pos.z <= platform_zmax;
}

 /// External Modifications to State
//
//   These allow the user to play with state while simulation
//   running.

// Move the ball.
//
void Ball::translate(pVect amt) {position += amt;}

// Add velocity to the ball.
//
void Ball::push(pVect amt) {velocity += amt;}

// Set the velocity to zero.
//
void Ball::stop() {velocity = pVect(0,0,0); }

// Set the velocity and rotation (not yet supported) to zero.
//
void Ball::freeze() {velocity = pVect(0,0,0); }



void World::balls_translate(pVect amt,int b){head_ball->translate(amt);}
void World::balls_push(pVect amt,int b){head_ball->push(amt);}
void World::balls_translate(pVect amt)
{ for ( Ball *ball: balls ) ball->translate(amt); }
void World::balls_push(pVect amt)
{ for ( Ball *ball: balls ) ball->push(amt); }
void World::balls_stop()
{ for ( Ball *ball: balls ) ball->stop(); }
void World::balls_freeze(){balls_stop();}


void
World::render_link_start()
{
  if ( world_time_link_update == world_time ) return;

  // Empty the containers in preparation for a new pass.
  for ( auto b: lis_bufs ) b->clear();
}
```

```
void
World::render_link_gather(Link *link)
{
  // Place data needed to render link (the argument to this function)
  // in arrays. Just before rendering those arrays will be placed in
  // buffer objects.


  // If we've already collected links, just return.  This happens
  // when doing multiple passes to render shadows and reflections.
  //
  if ( world_time_link_update == world_time ) return;

  Ball *const ball1 = link->ball1;
  Ball *const ball2 = link->ball2;

  // Direction of link relative to center off ball.
  //
  pVect dir1 = ball1->omatrix * link->cb1;

  // Place on surface of ball where link connects.
  //
  pCoor pos1 = ball1->position + dir1;

  // Direction and connection location for ball 2.
  //
  pVect dir2 = ball2->omatrix * link->cb2;
  pCoor pos2 = ball2->position + dir2;

  pVect p1p2(pos1,pos2);
  pNorm p1p2n(p1p2);

  // Radius of link.
  //
  const float rad = ball1->radius * 0.3;

  // Compute scale factors for texture.
  const float tex_aspect_ratio = 8.5 / 11;  // Yes, inches. Can't remember mm.
  const float page_width_o = 2.5;  // Width of texture in object-space coords.
  const float tex_t_scale = link->distance_relaxed / page_width_o;
  const float tex_angle_scale = rad * tex_aspect_ratio / page_width_o;

  pNorm dirn1(dir1);
  pNorm dirn2(dir2);

  // Vectors used to describe the cubic Hermite curve.
  //
  pVect4 v1 = p1p2n.magnitude * dirn1;
  pVect4 v2 = p1p2n.magnitude * dirn2;

  // Convert link's local x and y axes to global coordinates.
  //
  pVect4 b1_ydir = ball1->omatrix * link->b1_dir;
  pVect4 b2_ydir = ball2->omatrix * link->b2_dir;
```

```
  // Place data needed to render links in lists. Note that
  // the += operator appends the right-hand side to the container
  // on the left-hand side.
  //
  lis_pos1 << pos1;
  lis_pos2 << pos2;
  lis_v1 << v1;
  lis_v2 << v2;
  lis_b1_ydir << b1_ydir;
  lis_b2_ydir << b2_ydir;
  lis_tex_rad << vec4( tex_t_scale, tex_angle_scale, rad, 0);
}



void
World::render_link_render(vk::CommandBuffer& cb, bool shadow_volumes)
{
  // Perform a rendering pass to render all the links collected
  // by the routine render_link_2.


  // Check whether this is the first time we are rendering this data.
  //
  const bool first_render =
    world_time_link_update != world_time
    || opt_segments != buf_uni_common->links_n_segs
    || opt_sides != buf_uni_common->sides;
  world_time_link_update = world_time;
  const size_t n_instances = lis_pos1.size();
  const bool do_rt = vh.raytrace_active();
  const int segments = shadow_volumes ? max(2,opt_segments/4) : opt_segments;
  const int segments_sv = max(2,opt_segments/4);

  if ( !n_instances ) return;

  if ( first_render )
    {
      for ( auto b: lis_bufs ) b->to_dev();
      buf_uni_common->sides = opt_sides;
      buf_uni_common->links_n_segs_sv = segments_sv;
      buf_uni_common->delta_tee_sv = 1.0f / segments_sv;
      buf_uni_common->links_n_segs = opt_segments;
      buf_uni_common->delta_tee = 1.0f / opt_segments;
    }

  if ( do_rt && first_render )
    {
      const int n_tri_p_segment = 2 * opt_sides;
      const int n_tri_p_link = n_tri_p_segment * opt_segments;
      const int n_tri_p_pass = n_tri_p_link * n_instances;
```

```
      const int n_vtx_p_segment = 2 * ( opt_sides + 1 );
      const int n_vtx_p_link = n_vtx_p_segment * opt_segments;
      const int n_vtx_p_pass = n_vtx_p_link * n_instances;

      if ( !rt_geo_links )
        rt_geo_links
          .init( raytrace )
          .color_set( color_dim_gray, color_salmon )
          .bind( VR_BUF_IDX_sampler, sampler, texid_hw )
          .grouping_set( RT_Geometry_Indexed );

      buf_links_as_idx.assure(3*n_tri_p_pass);
      buf_links_tex_coor.assure(n_vtx_p_pass);
      buf_links_vertex_g.assure(n_vtx_p_pass);
      buf_links_normal_g.assure(n_vtx_p_pass);

      rt_geo_links
        .bind( VR_BUF_IDX_indices, buf_links_as_idx, 3*n_tri_p_pass )
        .bind( VR_BUF_IDX_pos, buf_links_vertex_g, n_vtx_p_pass )
        .bind( VR_BUF_IDX_normal, buf_links_normal_g )
        .bind( VR_BUF_IDX_tcoor, buf_links_tex_coor );
    }

  if ( do_rt )
    {
      QSet& qs = vh.qs;
      buf_uni_common.to_dev();
      qs.cb.begin( { vk::CommandBufferUsageFlagBits::eOneTimeSubmit } );
      comp_links.record_dispatch(qs.cb,opt_segments,n_instances);
      qs.cb.end();
      qs.q.submit( vk::SubmitInfo{ 0, nullptr, nullptr, 1, &qs.cb }, nullptr );
      qs.q.waitIdle();
      return;
    }


  VPipeline& pipe = shadow_volumes ? pipe_set_2_sv : pipe_set_2;

  pipe.record_draw_instanced(cb, segments+1, n_instances);
}


void
World::frame_callback()
{
  // This routine called whenever window needs to be updated, but
  // before commands are recorded.

  // Get any waiting keyboard commands.
  //
  cb_keyboard();

  if ( vh.have_raytrace ) vh.opt_want_raytrace_now = opt_want_raytrace;
```

```
// This is a hack for 2023 HW 2.
while ( !balls.empty() && balls.back()->world_time_expire <= world_time )
  balls.pop_back();

const double time_now = time_wall_fp();
const int time_step_count_start = time_step_count;
const int nmp = gpu_info.cuda_prop.multiProcessorCount;
int bl_p_sm_max = 1024;
for ( int i=0; i<gpu_info.num_kernels; i++ )
  set_min
    (bl_p_sm_max,
     gpu_info.get_max_active_blocks_per_mp(i, opt_block_size, 0) );
const int grid_size = bl_p_sm_max * nmp;

const bool opt_cuda = opt_physics_method != GP_cpu;
if ( opt_cuda )
  {
    CE( cudaEventRecord(frame_start_ce,0) );
    data_cpu_to_gpu_constants();
    data_cpu_to_gpu_dynamic();
  }

if ( !opt_pause || opt_single_frame || opt_single_time_step )
  {
    /// Advance simulation state.

    // Amount of time since the user saw the last frame.
    //
    const double wall_delta_t = time_now - last_frame_wall_time;

    // Compute amount by which to advance simulation state for this frame.
    //
    const double duration =
      vh.video_frame_duration() ?:
      opt_single_time_step ? opt_time_step_duration :
      opt_single_frame ? 1/30.0 :
      wall_delta_t;

    const double world_time_target = world_time + duration;
    const double wall_time_limit = time_now + 0.05;

    while ( world_time < world_time_target )
      {
        if ( opt_cuda )
          {
            launch_time_step(opt_time_step_duration,grid_size,opt_block_size);
          }
        else
          {
            time_step_cpu(opt_time_step_duration);
          }
        time_step_count++;
        world_time += opt_time_step_duration;
        const double time_right_now = time_wall_fp();
```

```
          if ( time_right_now > wall_time_limit ) break;
        }

      // Reset these, just in case they were set.
      //
      opt_single_frame = opt_single_time_step = false;
    }

  if ( opt_cuda )
    {
      data_gpu_to_cpu_dynamic();
      CE( cudaEventRecord(frame_stop_ce,0) );
    }

  frame_timer.phys_end();
  frame_timer.work_amt_set(time_step_count-time_step_count_start);

  float cuda_time = 0;
  if ( opt_cuda )
    {
      CE(cudaEventSynchronize(frame_stop_ce));
      CE(cudaEventElapsedTime(&cuda_time,frame_start_ce,frame_stop_ce));
    }
  frame_timer.cuda_frame_time_set(cuda_time);

  last_frame_wall_time = time_now;

}

int
main(int argv, char **argc)
{
  pVulkan_Helper pvulkan_helper(argv,argc);
  World world(pvulkan_helper,argv,argc);

  world.run();
}
```

```
/// LSU EE 4702-1 (Fall 2023), GPU Programming
//
 /// Homework 3 -- SOLUTION
 //
 //  Assignment: https://www.ece.lsu.edu/gpup/2023/hw03.pdf
 //
 /// Based on cpu-only/demo-06-rend-pipe.cc

#include <gp/coord.h>
#include <gp/colors.h>
#include "frame_buffer.h"

  ///
 /// Our_3D: Low-Level 3D Graphics Library.
 ///
class Our_3D {
public:
  Our_3D(pFrame_Buffer& fbp)
    :frame_buffer(fbp)
  {
    vtx_colors = nullptr;
    vtx_coors = nullptr;

    /// SOLUTION -- Problem 2
    topology_strip = false;
    /// SOLUTION -- Problem 3
    vtx_normals = nullptr;
  };

  Our_3D& transform_eye_from_object_set(const pMatrix& m)
  {
    eye_from_object = m;
    return *this;
  }
  Our_3D& transform_clip_from_eye_set(const pMatrix& m)
  {
    clip_from_eye = m;
    return *this;
  }
  Our_3D& light_location_set(const pCoor c)
  {
    light_location = c;
    return *this;
  }
  Our_3D& light_color_set(const pColor c)
  {
    light_color = c;
    return *this;
  }

  Our_3D& vtx_coors_set(vector<pCoor>& c)
  {
    vtx_coors = &c;
```

```
  return *this;
}

Our_3D& vtx_colors_set(vector<pColor>& c)
{
  vtx_colors = &c;
  return *this;
}

/// SOLUTION -- Problem 2
//
// Specify whether vertices are grouped for individual triangles or
// a triangle strip in the next call to draw_rasterization.
//
Our_3D& topology_strip_set( bool strip = true )
{
  topology_strip = strip;
  return *this;
}

/// SOLUTION -- Problem 3
//
//  Specify that no vertex normals will be provided for the next
//  draw_rasterization. Instead, lighting code will compute a
//  triangle normal using the provided triangle vertices.
//
Our_3D& vtx_normals_set()
{
  vtx_normals = nullptr;
  return *this;
}
//
//  Specify vertex normals to by used by the next draw_rasterization.
//
Our_3D& vtx_normals_set(vector<pVect4>& c)
{
  vtx_normals = c.data();
  return *this;
}

Our_3D& draw_rasterization();

pFrame_Buffer& frame_buffer;
pMatrix eye_from_object, clip_from_eye;
pCoor light_location;
pColor light_color;
vector<pCoor> *vtx_coors;
vector<pColor> *vtx_colors;
bool opt_tryout1, opt_tryout2;

/// SOLUTION -- Problem 2
//
// Provide a variable to hold vertex grouping setting.
//
```

```
  bool topology_strip;

  /// SOLUTION -- Problem 3
  //
  //  Provide a variable to hold pointer to normals, if any.
  //
  pVect4 *vtx_normals;

};

 ///
 ///  World: An application.
 ///
class World : public World_Base {
public:
  World(pFrame_Buffer& fb):World_Base(fb),gc(fb){}
  Our_3D gc;

  // Called each time a frame needs to be rendered.
  virtual void render(bool first_time)
  {
    if ( first_time ) init_scene();
    render_scene();
  }

  // Called Once
  void init_scene();

  // Called for each frame.
  void render_scene();

  pCoor eye_location, light_location;
  pCoor *adj_location;
  pVect eye_direction;
  bool opt_strip, opt_normals;
  bool opt_tryout1,opt_tryout2;

};

void
World::init_scene()
{
  eye_location = pCoor(-1.2,.5,10.2);
  light_location = pCoor(-2.2,1.5,10.2);
  eye_direction = pVect(.1,0,-1);
  adj_location = &eye_location;
  opt_tryout1 = opt_tryout2 = false;
  gc.opt_tryout1 = opt_tryout1;
  gc.opt_tryout2 = opt_tryout2;
  opt_normals = opt_strip = false;
}

void
World::render_scene()
```

```
{
  pFrame_Buffer& frame_buffer = gc.frame_buffer;

  switch ( frame_buffer.keyboard_key ) {
  case FB_KEY_LEFT: adj_location->x -= 0.1; break;
  case FB_KEY_RIGHT: adj_location->x += 0.1; break;
  case FB_KEY_PAGE_UP: adj_location->y += 0.1; break;
  case FB_KEY_PAGE_DOWN: adj_location->y -= 0.1; break;
  case FB_KEY_UP: adj_location->z -= 0.1; break;
  case FB_KEY_DOWN: adj_location->z += 0.1; break;
  case 'e': case 'E': adj_location = &eye_location; break;
  case 'l': case 'L': adj_location = &light_location; break;
  case 'y': gc.opt_tryout1 = opt_tryout1 = !opt_tryout1; break;
  case 'Y': gc.opt_tryout2 = opt_tryout2 = !opt_tryout2; break;
  case 'n': case 'N': opt_normals = !opt_normals; break;
  case 's': case 'S': opt_strip = !opt_strip; break;
  }

  gc.frame_buffer.fbprintf
    ("Strip %s ('s')  Normals %s ('n')  Tryout1 %s ('y')  Tryout2 %s ('Y') \n",
     opt_strip ? "ON " : "OFF", opt_normals ? "ON " : "OFF",
     opt_tryout1 ? "ON " : "OFF", opt_tryout2 ? "ON " : "OFF");

  gc.light_location_set( light_location ).light_color_set( color_white * 100 );

  ///
  /// Prepare Transformations
  ///

  // Specifications of User's Monitor
  //
  const uint win_width = frame_buffer.width_get();
  const uint win_height = frame_buffer.height_get();
  //
  // win_width and win_height are pixel-space coordinates.
  //
  const float aspect = 1.0 * win_width / win_height;
  const float width_m = 0.8;  // Width of user's monitor.
  const float height_m = width_m / aspect;
  //
  // width_m and height_m are in eye-space coordinates.

  // Specify Transformation from Object to Eye Space: eye_from_object.
  //
  pMatrix_Translate center_eye(-eye_location);
  pMatrix_Rotation rotate_eye(eye_direction,pVect(0,0,-1));
  pMatrix eye_from_object = rotate_eye * center_eye;

  gc.transform_eye_from_object_set(eye_from_object);

  // Transform from Eye to Clip
  //
  pMatrix_Frustum clip_from_eye
    ( -width_m/2, width_m/2,  -height_m/2, height_m/2,  1, 5000 );
```

```
  gc.transform_clip_from_eye_set(clip_from_eye);


  ///
  /// Prepare Triangles
  ///

  // List of coordinates in object space and list of colors.
  //
  vector<pCoor> coors_os;
  vector<pColor> colors;

  // Insert First Triangle
  //
  // Insert coordinates of a triangle into coordinate list,
  // and insert its color into a color list.
  //
  coors_os << pCoor( 0, 0, 0 ) << pCoor( 9, 6, -5 ) << pCoor( 0, 7, -3 );
  colors << color_white << color_white << color_white;

  // Insert Second Triangle
  //
  // Add coordinates and color of another triangle
  //
  coors_os << pCoor(7,2,1) << pCoor(-3,3,-3.5) << pCoor(9,0,0);
  colors << color_lsu_spirit_purple << color_lsu_spirit_purple
         << color_lsu_spirit_purple;

  // Insert Two More Triangles
  //
  // Add a square consisting of a gold and multicolored triangle.
  //
  coors_os << pCoor(-4,0,-3) << pCoor(-4,2,-3) << pCoor(-2,0,-3);
  colors << color_lsu_spirit_gold << color_lsu_spirit_gold
         << color_lsu_spirit_gold;

  coors_os << pCoor(-4,2,-3) << pCoor(-2,2,-3) << pCoor(-2,0,-3);
  colors << color_blue << color_green << color_red;

 // Add a strip.
  int n_segs = 20;
  pCoor center(1,-1,-3);
  pVect sz( 0, 0.75, 0.05 );
  pNorm ax( 0, -sz.z, sz.y ), ay = cross(sz,ax);
  vector<pCoor> strip_coor(n_segs+1);
  float r = 4, d_th = 2 * M_PI / n_segs;

  /// SOLUTION -- Problem 1
  //
  // Draw the red triangle. (Okay, it's orange_red. Close enough.)
  //
  coors_os << pCoor(-2,2,-3) << pCoor(-2,0,-3) << center + sz;
  colors << color_orange_red << color_orange_red << color_orange_red;
```

```
/// SOLUTION -- Problem 2
//
// Tell Our_3D that in the next draws vertices will not be grouped
// as triangle strips.
//
gc.topology_strip_set(false);

/// SOLUTION -- Problem 3
//
// Tell Our_3D that in the next draws normals will not be provided.
//
gc.vtx_normals_set();

gc.vtx_coors_set(coors_os).vtx_colors_set(colors);
gc.draw_rasterization();
coors_os.clear(); colors.clear();

/// SOLUTION -- Problem 3
//
vector<pVect4> strip_norms(n_segs+1);

for ( int i=0; i<=n_segs; i++ )
  {
    /// SOLUTION -- Problem 3
    //
    // Compute normal for segment i.
    //
    strip_norms[i] = ax * cosf(i*d_th) + ay * sinf(i*d_th);

    strip_coor[i] = center + r*strip_norms[i];
  }
vector<pVect4> norms_os;

if ( opt_strip )
  {
    for ( int i=0; i<=n_segs; i++ )
      {
        /// SOLUTION -- Problem 2
        //
        //  Insert vertex coordinates and colors in triangle-strip
        //  order. Just two!
        //
        coors_os << strip_coor[i] + sz << strip_coor[i];
        colors << color_cyan << color_cyan;

        /// SOLUTION -- Problem 3
        //
        //  Insert appropriate normal for each vertex.
        //
        norms_os << strip_norms[i] << strip_norms[i];
      }
  }
else
```

```
    {
      for ( int i=0; i<n_segs; i++ )
        {
          coors_os << strip_coor[i] << strip_coor[i+1] << strip_coor[i] + sz;
          coors_os << strip_coor[i]+sz << strip_coor[i+1]+sz << strip_coor[i+1];

          for ( int i=0; i<6; i++ ) colors << color_cyan;

          /// SOLUTION -- Problem 3
          //
          //  Insert appropriate normal for each vertex.
          //
          norms_os << strip_norms[i] << strip_norms[i+1] << strip_norms[i];
          norms_os << strip_norms[i] << strip_norms[i+1] << strip_norms[i+1];
        }
    }

  /// SOLUTION -- Problem 3
  //
  //  Provide normals to Our_3D or tell Our_3D not to expect them.
  //
  if ( opt_normals ) gc.vtx_normals_set(norms_os); else gc.vtx_normals_set();

  /// SOLUTION -- Problem 2
  //
  //  Tell Our_3D whether the vertices are grouped for triangle strips.
  //
  gc.topology_strip_set(opt_strip);

  gc.vtx_coors_set(coors_os).vtx_colors_set(colors);
  gc.draw_rasterization();
}



 ///
 /// Code Below (in Our3D) is Part of (Fictional) 3D Graphics Library
 ///

Our_3D&
Our_3D::draw_rasterization()
{
  ///
  /// Prepare Transformations
  ///

  // Specifications of User's Monitor
  //
  const uint win_width = frame_buffer.width_get();
  const uint win_height = frame_buffer.height_get();
  //
  // win_width and win_height are pixel-space coordinates.

  // Transform from Clip to Pixel
```

```
//
pMatrix_Translate recenter(pVect(1,1,0));
pMatrix_Scale scale( win_width/2, win_height/2, 1 );
pMatrix window_from_clip = scale * recenter;
pMatrix ws_from_os = window_from_clip * clip_from_eye * eye_from_object;

// Homework 3: DON'T MODIFY THIS LINE.
frame_buffer.n_vtx_frame += vtx_coors->size();

///
/// Rasterize Triangle and Write Frame Buffer
///

// Get pointer to the depth (z) buffer.
//
float* const zbuffer = frame_buffer.buffer_depth_get();
//
// The depth buffer uses the same index as the frame buffer.

auto& coors_os = *vtx_coors;
auto& colors = *vtx_colors;

/// SOLUTION -- Problem 2
//
//  Set loop stride based on indicated grouping.
//
const size_t inc = topology_strip ? 1 : 3;

for ( size_t i=0; i+2<coors_os.size(); i += inc )
  {
    frame_buffer.n_tri_frame++; // Count of number of triangles.

    // Get next triangle's object space coordinates.
    //
    pCoor o0 = coors_os[i+0],  o1 = coors_os[i+1],  o2 = coors_os[i+2];
    pCoor_Homogenized
      w0( ws_from_os * o0 ), w1( ws_from_os * o1 ), w2( ws_from_os * o2 );

    pColor c0 = colors[i+0], c1 = colors[i+1], c2 = colors[i+2];

    // Compute triangle normal (for lighting).
    //
    pNorm tn = cross(o0,o1,o2);

    /// SOLUTION -- Problem 3
    //
    //  Extract normals from list.
    //
    pVect4 n0, n1, n2;
    if ( vtx_normals )
      {
        n0 = vtx_normals[i+0];
        n1 = vtx_normals[i+1];
        n2 = vtx_normals[i+2];
```

```
  }

pVect v20(w2,w0), v21(w2,w1);
const float db0 = 1/max(fabs(v20.x),fabs(v20.y));
const float db1 = 1/max(fabs(v21.x),fabs(v21.y));

// Used for computing perspective-correct barycentric coords.
const float w0ow2z = w0.z / w2.z, w1ow2z = w1.z / w2.z;

// Rasterize the Triangle.
// Iterate over triangle using barycentric coordinates.
//
for ( float b0=0; b0<=1; b0 += db0 )
  for ( float b1=0; b1<=1-b0; b1 += db1 )
    {
      frame_buffer.n_frag_frame++; // Count of number of fragments.
      const float b2 = 1 - b0 - b1;

      // Each iteration operates on one fragment of triangle.

      pCoor wf = b0*w0 + b1*w1 + b2*w2;

      if ( uint(wf.x) >= win_width || uint(wf.y) >= win_height ) continue;

      // Compute the index into the frame and depth buffers.
      //
      const size_t idx = wf.x + int(wf.y) * win_width;

      // Depth (Z) Test
      //
      float zinv = wf.z;
      if ( zinv < 0 || zinv > 1 ) continue;
      if ( zbuffer[ idx ] < zinv ) continue;

      zbuffer[ idx ] = zinv;

      // Compute perspective-correct barycentric coordinates.
      //
      float bc0 = b0 ? 1/( w0ow2z * (1-b0)/b0 + 1 ) : 0;
      float bc1 = b1 ? 1/( w1ow2z * (1-b1)/b1 + 1 ) : 0;
      float bc2 = 1 - bc0 - bc1;

      /// SOLUTION -- Problem 3
      //
      //  Either blend provided vertex normals or use tn (the
      //  triangle normal computed above).
      //
      pVect4 n = vtx_normals ? bc0*n0 + bc1*n1 + bc2*n2 : tn;

      // Compute Lighted Color
      //

      // Blend color of three vertices together.
      //
```

```
            pColor color = bc0*c0 + bc1*c1 + bc2*c2;

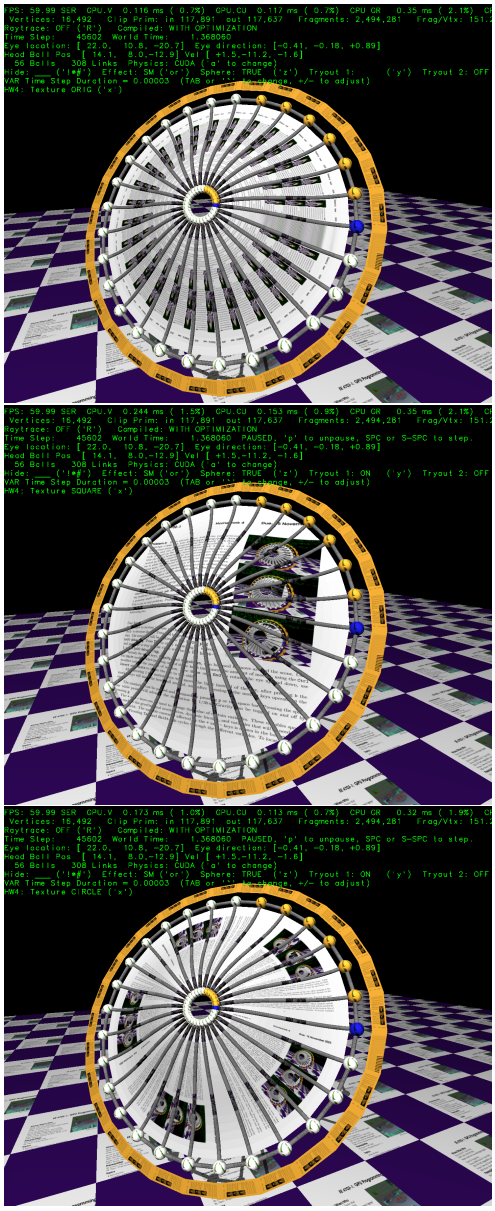
            // Find approximate object-space coordinate of fragment.
            //
            pCoor of =     bc0*o0 + bc1*o1 + bc2*o2;

            pNorm f_to_l(of,light_location);
            float phase = fabs(dot(n,f_to_l));
            pColor lighted_color = phase / f_to_l.mag_sq * light_color * color;

            //
            // Write the frame (color) buffer with the lighted color
            //
            frame_buffer[ idx ] = lighted_color.int_rgb();
            frame_buffer.n_px_frame++; // Count of number of written pixels.
          }
      }
  return *this;
}


int
main(int argc, char **argv)
{
  pFrame_Buffer demo_frame_buffer(argc,argv);
  WRender render( demo_frame_buffer );
  World our_world( demo_frame_buffer );
  WRENDER_INSERT(render, our_world);
  render.run();
  return 0;
}
```

# LSU EE 4702-1 Homework 4 Solution Due: 15 November 2023



**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 wheel from Homework 2, but this time the spokes between the inner and outer ring are connected by flexible material. In the screenshots the material is white with an image of the first page of this homework assignment handout projected in various ways. The material is simulated as triangles interacting with air, and the material is rendered as a surface following the shape of the spokes. Also there is a flat ring surrounding the wheel, that is shown in gold, also showing this homework in the screenshots.

In Problem 1 the gold ring is to be drawn. This is a straightforward triangle strip drawing problem. It's supposed to be the confidence builder so if you're having problems on it **please ask for help!** In Problem 2 a geometry shader is used to compute surface normals. This is also an easy problem. Problem 3a, finding texture coordinates, is of medium difficulty. Problem 3b is the interesting one.

## 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 `hw04.png` or `hw04-debug.png`. Press `F10` to start recording a video, and press `F10` to stop it. The video will be in file `hw04-1.ogg` or `hw04-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 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 1 through 5. For this assignment use scene 4 (which is how the code starts). 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 wheel), 5 for a top (a spinning child's toy). Scene 1 is initialized by the code in World::ball_setup_1, etc. Parts of the solution to this assignment should be put in World::ball_setup_4.

The scene for this assignment, 4, initializes randomly each time it is reset. (That is to avoid solutions that accidentally only work for a special case of positioning, etc.)

In this assignment the projection of textures on to the spoke material can be changed by pressing x. There are four possible projections NONE, meaning no textures, ORIG, meaning the one shown on the top screenshot, SQUARE, which, when solved, shows the middle screenshot, and CIRCLE, which, when solved, shows the bottom screenshot. Pressing x cycles between these projections. Variable opt_texture is set to the current projection, which can be TX_None, TX_Orig, TX_Square, or TX_Circle. (In hw04.cc those are enumeration constants in hw04-shdr.cc those are integer constants.)

### 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, hw04, and a debug-able version of the code, hw04-debug. The hw04-debug version is compiled with optimization turned off, which makes it easier to debug. When needed, you are strongly encouraged to run hw04-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`, available in CPU and shader code. 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.

Resources

A good reference for C++ is `https://en.cppreference.com/w/`. Solutions to the shader programming problems may (will) require the use of library functions. See Chapter 8 of The OpenGL Shading Language Version 4.6 specification.

**Problem 1:** Modify the code in `render_hw04`, excerpted below, so that it renders a gold ring around the wheel (see the screenshots).

```
/// Put Homework 4, Problem 1 solution below.
//

// Compute center of outer ring.
pCoor ctr(0,0,0);
const auto& balls_outer = hw04_stuff.balls_outer;
for ( Ball* b: balls_outer ) ctr += b->position;
const size_t n_balls = balls_outer.size();
ctr.w = n_balls;
ctr.homogenize();
pNorm wz [[maybe_unused]] =    // Wheel z
  cross(ctr,balls_outer[0]->position,balls_outer[n_balls/4]->position);
for ( size_t i = 0; i < n_balls; i++ )
  {
    Ball *b = balls_outer[i];
    pCoor pos = b->position;
    pNorm n(ctr,pos);
    pCoor p = pos + 2 * b->radius * n;
    bset_hw04_p1 << p << color_cyan << pNorm(0,1,0) << pTCoor(0,0);
  }
```

To help with this problem the current center of the wheel is in variable `ctr` and an approximate normal is in `wz`. Also, there is a loop that iterates over each outer-ring ball. In that loop body a vector, `n`, and coordinate, `p`, are computed that can be used to construct the ring. These are used to compute a point, but not necessarily the point(s) needed to solve the problem. Also, a placeholder color, normal and texture coordinate are inserted.

Pipeline `pipe_hw04_p1` is used to render the ring. It expects a vertex coordinate, color, normal, and a texture coordinate. Placeholder values are inserted for each of these attributes, which must be modified for the correct solution. Textures are optional for this problem. Texture coordinates might be chosen to help understand how texture coordinates work, to assist in solving Problem 3.

The Problem 1 solution code appears below. The code is entirely within `World::render_hw04`.

```
/// SOLUTION – Problem 1
```

```
//
for ( size_t i=0; i<=n_balls; i++ )
  {
    /// SOLUTION – Problem 1
    //
    // Visit the i=0 ball twice so that a triangle strip around the
    // wheel can be completed. To do that change loop end condition
    // above from "i<n_balls" to "i<=n_balls" and change the index
    // to balls_outer, below, from i to i%n_balls.

    Ball *b = balls_outer[i%n_balls];

    /// SOLUTION – Problem 1
    //
    //  Compute two coordinates (per iteration). Both points are on
    //  a line starting at the wheel center, ctr, and and passing
    //  through the ball center, pos. The first, p, is at a distance
    //  b->radius from the ball center. (This coordinate was
    //  provided in the unmodified assignment.) The second, pb, is
    //  at a distance 2 * b->radius from p.

    pCoor pos = b->position;
    pNorm n(ctr,pos);
    pCoor p = pos + 2 * b->radius * n;
    pCoor pb = p + 2 * b->radius * n;

    //  Insert those too coordinates into the buffer set.
    //
    bset_hw04_p1 << p << pb;

    //  Also insert colors.
    //
    bset_hw04_p1 << color_lsu_spirit_gold << color_lsu_spirit_gold;

    //  Problem 1 did not require that any particular texture coordinates
    //  be inserted, so it's hard to get this wrong. Here texture
    //  coordinates are inserted so that an entire copy of the texture
    //  appears between each pair of adjacent balls.
    //
    bset_hw04_p1 << pTCoor(0,i) << pTCoor(1,i);

    //  Problem 1 did not specifically require that the approximate
    //  normal, wz, had to be improved on.  If the wheel were rigid
    //  then wz would be a correct normal, but the wheel can flex,
    //  so its not. Here the approximate normal is used.
    //
    bset_hw04_p1 << wz << wz;

    // Commented out code from unmodified assignment.
    // bset_hw04_p1 << p << color_cyan << pNorm(0,1,0) << pTCoor(0,0);
  }
```

**Problem 2:** Pipeline `pipe_hw04_p2` has only two inputs, a vertex coordinate and a texture coordinate. A normal is needed to compute lighting. Modify the shaders in `hw04-shdr.cc` so that they compute a normal based on triangle coordinates. *Hint: This is easy too.*

The solution code, shown below, is entirely within shader routine `gs_main`, in file `hw04-shdr.cc`.

```
void gs_main() {
  // SOLUTION -- Problem 2
  //
  //  Compute normal by taking the cross product of vectors formed
  //  by the triangle vertices.
  //
  vec3 normal_o =
    cross( In[1].vertex_o.xyz - In[0].vertex_o.xyz,
           In[2].vertex_o.xyz - In[0].vertex_o.xyz );
  vec3 normal_e = mat3( ut.eye_from_object ) * normal_o;

  // Note: The code below is not part of the solution.

  for ( int i=0; i<3; i++ )
    {
      gl_Position = ut.clip_from_object * In[i].vertex_o;
      Out.normal_e = normal_e;
      Out.vertex_e = ut.eye_from_object * In[i].vertex_o;
      Out.tex_coor = In[i].tex_coor;
      EmitVertex();
    }
  EndPrimitive();
}
```

**Problem 3:** Texture coordinates for the spoke material is assigned in routine `World::ball_setup_4()`, search for Homework 4 in that routine. Coordinates are assigned to `Ball` members `Ball::tc.orig`, `Ball::tc.square`, and `Ball::tc.circle`.

```
  for ( int i=0; i<n_balls; i++ ) {
      const float theta = d_theta * i;
      pVect pos_n = cosf(theta) * ax + sinf(theta) * ay;
      pCoor pos_o = center_pos + ring_outer_radius * pos_n;
      pCoor pos_i = center_pos + ring_inner_radius * pos_n;

      // Construct new ball and add it to the ball list.
      Ball* ball_o = balls_outer.emplace_back( new Ball(pos_o,ballo_default) );
      Ball* ball_i = balls_inner.emplace_back( new Ball(pos_i,balli_default) );
      Link *link = new Link(ball_o,ball_i, {.spring_constant=opt_spring_constant/200});
      links.push_back(link);
      hw04_stuff.ribs.push_back(link);

      // Homework 4 -- Change these values for experimentation, if you like.
      ball_o->tc.orig = pTCoor(i,0);     ball_i->tc.orig = pTCoor(i,1);

      //  HW 4: Modify the assignments to the tc.square and tc.circle members.
```

```
ball_o->tc.square = pTCoor(i,0);   ball_i->tc.square = pTCoor(i,1);
ball_o->tc.circle = pTCoor(i,0);   ball_i->tc.circle = pTCoor(i,1);
```

In the unmodified assignment all three are assigned the same texture coordinate. Note that the code assigns texture coordinates for an outer-ring ball, `ball_o`, and an inner-ring ball, `ball_i`.

(*a*) Assign `Ball:tc.square` so that the texture is mapped as a square. The blue ball is on the middle right-hand side and the yellow balls are on the upper-left quadrant. See the middle screenshot on *the first page, not the one below*.

To find the texture coordinates imagine that a circle, the wheel, were drawn over the texture image. The texture coordinates range from $\begin{bmatrix}0\\0\end{bmatrix}$ to $\begin{bmatrix}1\\1\end{bmatrix}$ with $\begin{bmatrix}0.5\\0.5\end{bmatrix}$ in the center of the texture image. So to find the texture coordinates for the inner and outer balls of the wheel we need to determine where on the circle we are. Consider the code computing the balls' positions, `pos_o` and `pos_i`:

```
for ( int i=0; i<n_balls; i++ ) {
    const float theta = d_theta * i;
    pVect pos_n = cosf(theta) * ax + sinf(theta) * ay;
    pCoor pos_o = center_pos + ring_outer_radius * pos_n;
    pCoor pos_i = center_pos + ring_inner_radius * pos_n;
```

The balls' positions are found using vector `pos_n` which is a unit vector pointing from the center of the wheel toward the intended ball positions. We can construct a similar vector to find the texture coordinates by replacing `ax` and `ay` with $\begin{bmatrix}1\\0\end{bmatrix}$ and $\begin{bmatrix}0\\1\end{bmatrix}$. Or more simply by just computing `vec2( cosf(theta), -sinf(theta) );`. To get the actual texture coordinate compute $\begin{bmatrix}0.5\\0.5\end{bmatrix} + r\begin{bmatrix}\cos(\theta)\\-\sin(\theta)\end{bmatrix}$, where $r = 1$ for the outer ball and $r = r_i/r_o$, where $r_i$ is the inner radius, `ring_inner_radius` and $r_o$ is the outer radius, `ring_outer_radius`.

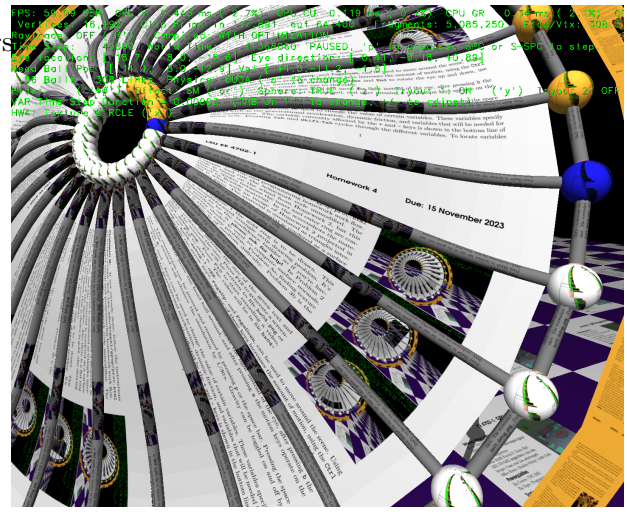
The solution code appears below, taken from `World::ball_setup_4` in file `hw04.cc`:

```
//  First, compute a vector of length 0.5 from the circle enter to the circle:
vec2 tex_sq_norm = 0.5 * vec2( cosf(theta), -sinf(theta) );

//  Prepare a coordinate of the circle center.
vec2 tex_ctr(0.5,0.5);

//  Add the vector to the circle center to get the outer-ring texture coordinate.
ball_o->tc.square = tex_ctr + tex_sq_norm;

//  Use a smaller radius for the inner ball.
float ring_io = ring_inner_radius / ring_outer_radius;
ball_i->tc.square = tex_ctr + ring_io * tex_sq_norm;
```



(*b*) Assign `Ball:tc.circle` and modify the shaders so that the texture is mapped wrapping around the ring as shown in the screenshot to the right and the bottom screenshot on the first page. In those screenshots about four copies of the image are shown.
The exact number of copies is not important, however the text away from the center must be readable (not stretched or compressed). Notice that the lines of text are parallel to the spokes (actually parallel to their centers, not the surfaces).

This is more difficult than part a because it requires more than just assigning the right texture coordinates. To solve this correctly one might need to provide additional uniform variables. Here is how one might add a new variable `factor_x`.

*Step 1 – Declare in CPU Code*
Add the variable to `struct Uni_HW04`:

```
struct Uni_HW04
{
  pColor color;
  int opt_texture;

  float factor_x; // Added a new variable
};
```

*Step 2 – Assign in CPU Code*
Assign the value, a convenient place is in `hw04_render()`.

```
  uni_hw04_needs_update = true; // Set this if x changes every frame.

  if ( uni_hw04_needs_update )
    {
      uni_hw04_needs_update = false;
      uni_hw04->color = 1.0 * color_white;
      uni_hw04->opt_texture = opt_texture;
      uni_hw04->factor_x = x; // Assign new member.
      uni_hw04.to_dev();
    }
```

*Step 3 – Declare in Shader Code*
Declare in the uniform shown below. The type and order must match the CPU code. Putting things in the wrong order or using the wrong type will result in incorrect execution but there won't be an error message.

```
layout ( binding = BIND_HW04 ) uniform HW_04
{
  vec4 color;
  int opt_texture;
  float factor_x;  // New member.
```

```
};
```

*Step 4 – Use The Variable in Shader Code*
Here the value is assigned to the red component of the lit color.

```
void fs_main()
{
  vec4 lit_color = generic_lighting(In.vertex_e, color, In.normal_e);
  lit_color.r = factor_x;
```

For 3b the texture should wrap around the ring. To achieve this effect the $x$ component of the texture coordinate should be proportional to the distance of the ball from the wheel center. If we want the inner ring to be on the left-hand edge of the texture and the outer ring on the right-hand edge of the texture we would set the $x$ component on the inner balls to 0 and the $x$ component on the outer balls to 1.

To get the wrapping effect the $y$ component should be proportional to the angle, $\theta$. Call the proportionality factor $\omega$. Then for the inner ball we would choose texture coordinate $\begin{bmatrix} 0 \\ \omega\theta \end{bmatrix}$ and for the outer ball $\begin{bmatrix} 1 \\ \omega\theta \end{bmatrix}$. We want to choose $\omega$ so that the text in the texture image does not appear squished together (condensed) or stretched out. The distance between the inner and outer edge is 1 in texture space (because of the way the $x$ component of the texture coordinate was chosen) and $r_o - r_i$ in global space, where $r_o$ is `ring_outer_radius` and $r_i$ is `ring_inner_radius`. Assuming we want to maintain the texture aspect ratio a distance of 1 in texture space along a circle between the inner- and outer-ring must correspond to a distance of $r_o - r_i$ in global space. Let $\theta_1$ be the angle at which the distance along that mid circle is the distance between the edges: $\theta_1 \frac{r_o+r_i}{2} = r_o - r_i$, solving $\theta_1 = 2\frac{r_o-r_i}{r_o+r_i}$. If we choose $\omega = 1/\theta_1$ the aspect ratio will match the original texture. In the solution code, including the fragments below, `delta_r` $= r_o - r_i$ and `r_mid` $= \frac{r_o+r_i}{2}$.

In an imperfect solution would insert the texture coordinates computed above into the pipeline:

```
const float delta_r = ring_outer_radius - ring_inner_radius;
const float r_mid = ( ring_outer_radius + ring_inner_radius ) / 2;
const float omega  = r_mid / delta_r;

ball_o->tc.circle = vec2( 1, theta * omega );
ball_i->tc.circle = vec2( 0, theta * omega );
```

The texture coordinates above are correct for the inner and outer balls, and these are the texture coordinates sent to the vertex shader. Recall that each vertex of a triangle has a texture coordinate, and that for a fragment within the triangle the texture coordinate is computed by interpolating the values at the three vertices (using perspective-correct barycentric coordinates). An invocation of the fragment shader, which is invoked for each fragment of a triangle, will get a texture coordinate interpolated from the three vertices of the triangle. This interpolation works correctly for typical mappings of textures to triangles, but not for the circular wrapping we are attempting here. As a result the text in the texture image will be slightly zig-zagy.

To see that this *will not* compute the texture coordinates we want, consider texture coordinates assigned to two adjacent outer-ring balls. The $x$ component of both are 1, which is correct. Now consider a fragment halfway between the two balls. The interpolated $x$ texture coordinate component will still be 1. But that's not correct because the balls are on a circle and the halfway point between the two balls is *inside* the circle and so the $x$ component of the texture, based on the radius, must also be less than one.

The solution to this dilemma is actually quite simple. Choose texture coordinates using a method similar to the one used for the square, except now do so with a center at $\begin{bmatrix} 0 \\ 0 \end{bmatrix}$ and with the radius of the outer ring set to 1 and the inner ring set to the inner/outer ratio, and also place $\omega$ and other useful constants in a uniform:

```
vec2 loc_n = vec2( cosf(theta), -sinf(theta) );
```

```
    float ring_io = ring_inner_radius / ring_outer_radius;
    ball_o->tc.circle = loc_n;
    ball_i->tc.circle = ring_io * loc_n;

    uni_hw04->r_scale_factor = 1 / ( 1 - ring_io );
    uni_hw04->omega = r_mid / delta_r;
```

In the fragment shader we will use these coordinates to compute the distance of the fragment from the wheel center (in fragment space) and the angle, $\theta$, and from those get texture coordinates:

```
if ( opt_texture == TX_None )
  {
    out_frag_color = lit_color;
  }
else if ( opt_texture == TX_Circle )
  {
    float dist = length(In.tex_coor);
    float angle = atan(In.tex_coor.y,In.tex_coor.x);
    vec2 tc =  vec2( ( dist - 1 ) * r_scale_factor,
                     angle * omega );

    out_frag_color = lit_color * texture(tex_unit_0,tc);
  }
else
  {
    out_frag_color = lit_color * texture(tex_unit_0,In.tex_coor);
  }
```

```
/// LSU EE 4702-1 (Fall 2023), GPU Programming
//
 /// Homework 4 -- SOLUTION
 //
 //  Modified this file and hw04.cc.
 //  Assignment: https://www.ece.lsu.edu/gpup/2023/hw04.pdf
 //  Solution:   https://www.ece.lsu.edu/gpup/2023/hw04_sol.pdf

// Specify version of OpenGL Shading Language.
//
#version 460

#extension GL_GOOGLE_include_directive : enable
#include <light.h>
#include <transform.h>
#include "links-shdr-common.h"
#include "shader-common-generic-lighting.h"

layout ( binding = BIND_UNI_COMMON ) uniform Common_Uniform
{
  Shdr_Uni_Common com;
};

bool opt_tryout1 = bool(com.tryout.x);
bool opt_tryout2 = bool(com.tryout.y);
float opt_tryoutf = com.tryoutf.x;

const int TX_None = 0, TX_Orig = 1, TX_Square = 2, TX_Circle = 3;

layout ( binding = BIND_HW04 ) uniform HW_04
{
  vec4 color;
  int opt_texture;
  /// SOLUTION -- Problem 3b
  float r_scale_factor, omega;
};

#ifdef _VERTEX_SHADER_

layout ( location = LOC_IN_POS ) in vec4 in_vertex_o;
layout ( location = LOC_IN_TCOOR ) in vec2 in_tex_coor;

// Interface block for vertex shader output / geometry shader input.
//
layout ( location = 0 ) out Data_to_GS
{
  vec4 vertex_o;
  vec2 tex_coor;
} Out;

void
vs_main()
{
```

```
  Out.vertex_o = in_vertex_o;
  Out.tex_coor = in_tex_coor;
}


#endif

#ifdef _GEOMETRY_SHADER_

layout ( location = 0 ) in Data_to_GS
{
  vec4 vertex_o;
  vec2 tex_coor;
} In[];

layout ( location = 0 ) out Data_to_FS
{
  vec3 normal_e;
  vec4 vertex_e;
  vec2 tex_coor;
} Out;

// Type of primitive at geometry shader input.
//
layout ( triangles ) in;

// Type of primitives emitted geometry shader output.
//
layout ( triangle_strip, max_vertices = 3 ) out;



void
gs_main()
{
  // Homework 4 Problem 2: Compute the correct normal.
  /// SOLUTION -- Problem 2
  //
  //  Compute normal by taking the cross product of vectors formed
  //  by the triangle vertices.
  //
  vec3 normal_o =
    cross( In[1].vertex_o.xyz - In[0].vertex_o.xyz,
           In[2].vertex_o.xyz - In[0].vertex_o.xyz );
  vec3 normal_e = mat3( ut.eye_from_object ) * normal_o;

  for ( int i=0; i<3; i++ )
    {
      gl_Position = ut.clip_from_object * In[i].vertex_o;
      Out.normal_e = normal_e;
      Out.vertex_e = ut.eye_from_object * In[i].vertex_o;
      Out.tex_coor = In[i].tex_coor;
      EmitVertex();
    }
```

```
  EndPrimitive();
}



#endif


#ifdef _FRAGMENT_SHADER_

#ifdef BIND_TEXUNIT
layout ( binding = BIND_TEXUNIT ) uniform sampler2D tex_unit_0;
#endif

layout ( location = 0 ) in Data_to_FS
{
  vec3 normal_e;
  vec4 vertex_e;
  vec2 tex_coor;
} In;

layout ( location = 0 ) out vec4 out_frag_color;


void
fs_main()
{
  vec4 lit_color = generic_lighting(In.vertex_e, color, In.normal_e);

  if ( opt_texture == TX_None )
    {
      out_frag_color = lit_color;
    }
  else if ( opt_texture == TX_Circle )
    {
      /// SOLUTION -- Problem 3b.
      //
      //  The texture coordinates in In.tex_coor are just the
      //  coordinate of the fragment relative to the wheel center and
      //  scaled so that the outer ring has a radius of 1. Use
      //  In.tex_coor to compute the angle of the ball. The angle is
      //  used to compute the y texture coordinate (assigned to tc).
      //  Compute the distance of In.tex_coor from the center (which
      //  is at the origin) and use that to compute the x texture
      //  coordinate component.

      float dist = length(In.tex_coor);
      float angle = atan(In.tex_coor.y,In.tex_coor.x);
      vec2 tc =
        vec2( ( dist - 1 ) * r_scale_factor,
              angle * omega );

      out_frag_color = lit_color * texture(tex_unit_0,tc);
    }
```

```
  else
    {
      out_frag_color = lit_color * texture(tex_unit_0,In.tex_coor);
    }

}


#endif
```

```
/// LSU EE EE 7722/ EE 4702-X, GPU Prog/MicroArch
//
 /// Homework 4 -- SOLUTION
 //
 //  Modified this file and hw04-shdr.cc
 //  Assignment: https://www.ece.lsu.edu/gpup/2023/hw04.pdf
 //  Solution:   https://www.ece.lsu.edu/gpup/2023/hw04_sol.pdf

/// Purpose
//
//   Demonstrate simulation of point masses connected by springs.


/// What Code Does

// Simulates balls connected by springs over a platform. Balls and
// springs can be initialized in different arrangements (called
// scenes). Currently scene 1 is a tree-like arrangement. The platform
// consists of tiles, some are purple-tinted mirrors (showing a
// reflection of the ball), the others show the course syllabus.



///  Keyboard Commands
 //
 /// Object (Eye, Light, Ball) Location or Push
 //   Arrows (←,→,↑,↓) Page Up, Page Down
 //        Move object or push ball, depending on mode.
 //        Shift + KEY: motion is 5x faster.
 //        Ctrl + KEY : motion is 5x slower.
 //   'e': Move eye.
 //   'l': Move light.
 //   'b': Move head (first) ball. (Change position but not velocity.)
 //   'B': Push head ball. (Add velocity.)
 //
 /// Eye Direction and GUI Options
 //
 //   'Home', 'End', 'Delete', 'Insert'
 //         Turn the eye direction. Home should rotate eye direction
 //         up, End should rotate eye down, Delete should rotate eye
 //         left, Insert should rotate eye right. The eye direction
 //         vector is displayed in the upper left.
 //  'C-='  (Ctrl =) Increase green text size.
 //  'C--'  (Ctrl -) Decrease green text size.
 //  'F12'  Write screenshot to PNG file.

 /// Simulation Execution Options
 //  (Also see variables below.)
 //
 //  'p'    Pause simulation. (Press again to resume.)
 //  ' '    (Space bar.) Advance simulation by 1/30 second.
 //  'S- '  (Shift Space bar.) Advance simulation by one time step.
```

```
/// Performance Tuning, Graphics Effects, and Debugging Options
//
//  'y'    Toggle value of opt_tryout1. Intended for experiments and debugging.
//  'Y'    Toggle value of opt_tryout2. Intended for experiments and debugging.
//  'n'    Toggle use of surface normals for spheres.
//  'o'    Toggle shadow effect.
//  'r'    Toggle mirror effect.
//  '!'    Toggle rendering of spheres (balls).
//  '@'    Toggle rendering of links.
//  '#'    Toggle rendering of platform.


/// Scene and Simulated System Options
//  (Also see variables below.)
//
//  'w'    Twirl balls around axis formed by head and tail. (Prob 2 soln).
//  '1'    Set up scene 1.
//  '2'    Set up scene 2.
//  '3'    Set up scene 3.
//  '4'    Set up scene 4.
//  'h'    Freeze position of first (head) ball. (Press again to release.)
//  't'    Freeze position of last (tail) ball. (Press again to release.)
//  's'    Stop balls.
//  'g'    Turn gravity on and off.

/// Variables
//   Selected program variables can be modified using the keyboard.
//   Use "Tab" to cycle through the variable to be modified, the
//   name of the variable is displayed next to "VAR" on the bottom
//   line of green text.

//  'Tab' Cycle to next variable.
//  '`'   Cycle to previous variable.
//  '+'   Increase variable value.
//  '-'   Decrease variable value.
//
//  VAR Spring Constant - Set spring constant.
//  VAR Time Step Duration - Set physics time step.
//  VAR Air Resistance - Set air resistance.
//  VAR Light Intensity - The light intensity.
//  VAR Gravity - Gravitational acceleration. (Turn on/off using 'g'.)

#define MAIN_INCLUDE
#undef VBUFFER_STRICT
#include <vhelper.h>

#include <vstroke.h>
#include <gp/coord.h>
#include <gp/pstring.h>
#include <gp/misc.h>

#include <vutil-texture.h>
#include <vutil-pipeline.h>
```

```
#include "shapes.h"

#include <cuda_runtime.h>

#include <gp/colors.h>

#include <gp/util-containers.h>
#include <gp/coord-containers.h>
#include <gp/cuda-gpuinfo.h>
#include "links.cuh"

#include "links-shdr-common.h"


///
/// Main Data Structures
///
//
// class World: All data about scene.


class World;
class Link;

enum GPU_Physics_Method { GP_cpu, GP_cuda, GP_ENUM_SIZE };
const char* const gpu_physics_method_str[] = { "CPU", "CUDA" };
const char *sh_names[] = { "PLAIN", "SET 1", "SET 2" };

enum HW4_Texture { TX_None, TX_Orig, TX_Square, TX_Circle, TX_ENUM_SIZE };
const char* const hw4_texture_str[] = {"NONE", "ORIG", "SQUARE", "CIRCLE" };

pTCoor
mix( const pTCoor& a, const pTCoor& b, float t )
{
  return pTCoor(lerp(a.x,b.x,t),lerp(a.y,b.y,t));
}

inline pCoor
bez(pCoor p1, pCoor p2, pVect v1, pVect v2, float t)
{
  const float t2 = t * t;
  const float t3 = t2 * t;
  return pCoor( ( 2*t3 - 3*t2 + 1 ) * pVect(p1) + (-2*t3 + 3*t2 ) * pVect(p2)
                + (t3 - 2*t2 + t ) * v1 - (t3 - t2 ) * v2 );
}



// Object Holding Ball State
//
class Ball {
public:
  Ball() { init(); }
  Ball(pCoor pos){ init(); position = pos; constants_update(); }
```

```
Ball(pCoor pos, const Ball &bd)
{ *this = bd; position = pos; constants_update(); }

void init()
  {
    velocity = pVect(0,0,0);
    omega = pVect(0,0,0);
    radius = 1;
    density = 1.00746;
    fdt_to_do = 0;
    locked = false;
    color = color_lsu_spirit_gold;
    specularity = 0;
    contact = false;
    orientation_set(pVect(0,1,0),0);
    world_time_expire = 3600.0 * 24 * 365 * 1000;
  }

pCoor position;
pVect velocity;
pQuat orientation;
pMatrix3x3p omatrix;
pVect omega;                    // Spin rate and axis.
double world_time_expire;

int idx; // Position in balls_pos_rad;

union {
  float4 m_r_fdt;
  struct {
    float mass;
    float radius;
    float fdt_to_do; // Radius / moment of inertia.
    float _;
  }; };

float mass_min; // Mass below which simulation is unstable.
float radius_sq, radius_inv;  // Pre-computed based on radius.
float density;
float mass_inv;

bool locked;

pVect force;
pVect torque;
pColor color;
float specularity;              // Amount of shininess, in [0,1].
bool contact;                   // When true, ball rendered in gray.
float spring_constant_sum;      // Used to compute minimum mass.

/// Homework 4

union {
  pTCoor tca [4];
```

```
    struct { pTCoor none, orig, square, circle; } tc; };

  void orientation_set(pNorm dir, float angle)
    { orientation_set(pQuat(dir,angle)); }
  void orientation_set(pQuat ori)
    {
      orientation = ori;
      omatrix = pMatrix3x3_Rotation(orientation);
    }
  void constants_update();
  pVect point_rot_vel(pNorm tact_dir);
  pVect point_rot_vel(pVect tact_dir);
  void apply_tan_force(pNorm tact_dir, pNorm force_dir, double force_mag);
  void apply_tan_force(pNorm tact_dir, pVect force);
  void push(pVect amt);
  void translate(pVect amt);
  void stop();
  void freeze();
};

struct HW04_Stuff
{
  vector<Ball*> balls_outer;
  vector<Ball*> balls_inner;
  vector<Link*> ribs;
};

struct Uni_HW04
{
  pColor color;
  int opt_texture;

  /// Homework 4
  //
  // This structure is copied to the GPU.
  // Adding members will be needed for Problem 3.

  /// SOLUTION -- Problem 3b
  //
  float r_scale_factor, omega;
  //
  // Provide constants needed to compute texture coordinates of wrapped
  // texture.
};

void
Ball::constants_update()
{
  assert( radius );
  radius_sq = radius * radius;
  mass = 4.0 / 3 * M_PI * radius_sq * radius * density;
  radius_inv = 1 / radius;
  mass_inv = 1 / mass;
```

```
  // FYI, notice simplifications:
  //   const float mo_inertia = 0.4 * mass * radius_sq;
  //   fdt_to_do = radius / mo_inertia;
  fdt_to_do = radius_inv * 2.5 * mass_inv;
}

pVect
Ball::point_rot_vel(pNorm direction)
{
  /// Return velocity of point on surface of ball.
  //
  return radius * cross( omega, direction );
}

pVect
Ball::point_rot_vel(pVect rel_pos)
{
  /// Return velocity of point relative to center.
  //
  return cross( omega, rel_pos );
}

void
Ball::apply_tan_force(pNorm tact_dir, pVect force)
{
  torque += cross(tact_dir, force);
}

void
Ball::apply_tan_force(pNorm tact_dir, pNorm force_dir, double force_mag)
{
  apply_tan_force(tact_dir, force_mag * force_dir);
}

struct Link_Args {
  pColor natural_color = color_red;
  pColor color = color_red;
  float distance_relaxed = 0;
  float spring_constant = 0;
};

class Link {
public:
  Link(Ball *b1, Ball *b2, Link_Args args = {}):ball1(b1),ball2(b2),
     snapped(false)
  { init(args); }
  Link(Link *link, pVect cb1p, pVect cb2p, float drp):
    ball1(link->ball1), ball2(link->ball2), cb1(cb1p), cb2(cb2p),
    is_surface_connection(true), is_renderable(false),
    is_simulatable(true), distance_relaxed(drp), spring_constant(0),
    snapped(false){link->is_simulatable = false;}

  Link(Link *link, pVect cb1p, pVect cb2p, Link_Args args = {} ):
    ball1(link->ball1), ball2(link->ball2), cb1(cb1p), cb2(cb2p),
```

```
  is_surface_connection(true), is_renderable(false),
  is_simulatable(true),
  distance_relaxed(args.distance_relaxed),
  spring_constant(args.spring_constant),
  snapped(false){link->is_simulatable = false;}

void init(Link_Args& args)
{
  assert( args.distance_relaxed == 0 );
  natural_color = args.natural_color;
  color = args.color;
  spring_constant = args.spring_constant;
  init();
}
void init()
  {
    assert( ball1->radius > 0 );
    assert( ball2->radius > 0 );
    is_simulatable = true;
    is_renderable = true;
    pNorm n12(ball1->position,ball2->position);
    const float rad_sum = ball1->radius + ball2->radius;
    is_surface_connection = n12.magnitude >= rad_sum;

    pMatrix3x3 b1rot = ball1->omatrix;
    pMatrix3x3 b2rot = ball2->omatrix;

    pVect b1_y = b1rot * pVect(0,1,0);
    pVect b1_x = b1rot * pVect(1,0,0);
    bool b1_dir_is_y = fabs(dot(b1_y,n12)) < 0.999;
    pVect b1_dir_loc = b1_dir_is_y ? b1_y : b1_x;

    con_x = pNorm(cross(b1_dir_loc,n12));
    con_y = cross(n12,con_x);
    b1_dir = mult_MTV( b1rot, con_y );
    b2_dir = mult_MTV( b2rot, con_y );

    if ( !is_surface_connection )
      {
        distance_relaxed = n12.magnitude;
        cb1 = cb2 = pVect(0,0,0);
        is_renderable = false;
        return;
      }

    distance_relaxed = n12.magnitude - rad_sum;
    cb1 = ball1->radius * mult_MTV( b1rot,  n12 );
    cb2 = ball2->radius * mult_MTV( b2rot, -n12 );
  }
int serial;
int idx;  // Position in lis_pos1, etc.
Ball* const ball1;
Ball* const ball2;
union {
```

```
    int2 ball_idx;
    struct { int ball1_idx, ball2_idx; };
  };
  pVect cb1, cb2, con_x, con_y, b1_dir, b2_dir;
  bool is_surface_connection;
  bool is_renderable, is_simulatable;
  union {
    float2 dr_sc;
    struct {
      float distance_relaxed;
      float spring_constant;
    };
  };
  bool snapped;
  pColor natural_color;
  pColor color;

  pVect spring_force_12;
  pVect torque1,torque2;

};

class Surface {
public:
  Ball *b0, *b1, *b2;
  union {
    int4 b_idx;
    struct { int b0_idx, b1_idx, b2_idx, pad; };
  };
  Surface* operator ->() { return this; }
};

// Declare containers for Balls and Links. The containers are derived
// from std::vector and use overloaded operator += for push_back. (See
// include/util-containers.h.)
//
typedef pVectorI<Link> Links;
typedef pVectorI<Ball> Balls;


enum Shader_Option { SO_Plain, SO_Instances, SO_True, SO_ENUM_SIZE };

class World {
public:
  World(pVulkan_Helper &fb, int argc, char **argv)
    :vh(fb), ff_state(fb.qs), raytrace(fb.qs),
     frame_timer(vh.frame_timer), transform(fb.qs),
     shapes(ff_state), sphere(ff_state),
     opt_tryout1(fb.qs.opt_tryout1),opt_tryout2(fb.qs.opt_tryout2)
  {init(argc,argv);}
  void init(int argc, char **argv);
  void init_graphics();
  void run();
  void frame_callback();
```

```
void render(vk::CommandBuffer& cb);
void render_objects(vk::CommandBuffer& cb, Render_Option render_option);
void objects_erase();
void cb_keyboard();
void modelview_update();

pVulkan_Helper& vh;
VFixed_Function_State_Manager ff_state;
VRaytrace raytrace;
pFrame_Timer& frame_timer;
VTransform transform;
Shapes shapes;
Sphere sphere;

bool opt_want_raytrace;

pVariable_Control variable_control;
double world_time;
double last_frame_wall_time;
float opt_time_step_duration;
int time_step_count;
float opt_gravity_accel;      // Value chosen by user.
pVect gravity_accel;          // Set to zero when opt_gravity is false;
bool opt_gravity;
bool opt_head_lock, opt_tail_lock;
bool &opt_tryout1, &opt_tryout2;  // For ad-hoc experiments.
float opt_tryoutf;
bool opt_sphere_true;
VBufferV<Shdr_Uni_Common> buf_uni_common;

// Tiled platform for ball.
//
float platform_xmin, platform_xmax, platform_zmin, platform_zmax;
float platform_pi_xwidth_inv;
VVertex_Buffer_Set bset_platform_tex, bset_platform_mir;
VPipeline pipe_platform_mir, pipe_platform_tex;
RT_Geo_Package rt_geo_platform_tex, rt_geo_platform_mir;

vk::Sampler sampler;
VTexture texid_hw;
VTexture texid_syl;
VTexture texid_emacs;

void platform_update();
bool platform_collision_possible(pCoor pos);

pCoor light_location;
float opt_light_intensity;
enum { MI_Eye, MI_Light, MI_Ball, MI_Ball_V, MI_COUNT } opt_move_item;
bool opt_pause;
bool opt_single_frame;      // Simulate for one frame.
int opt_single_time_step;  // Simulate for one time step.

VBufferV<Uni_Lighting> uni_light, uni_light_ambient, uni_light_all;
```

```
VBufferV<Uni_Lighting> uni_light_reflected;
VBufferV<Uni_Lighting> *puni_light_curr;

pCoor eye_location;
pVect eye_direction;
pMatrix transform_projection;
pMatrix modelview;

pVect adj_vector;
double adj_t_prev;
double adj_t_stop;
double adj_duration;

uint opt_shader;

bool opt_mirror;
int opt_mirror_depth;
bool opt_shadows;
bool opt_shadow_volumes;      // Make shadow volumes visible.
bool opt_normal_sphere;
int opt_hide_stuff;

int last_setup; // Last scene set up.

void ball_setup_1(bool big);
void ball_setup_2();
void ball_setup_3();
void ball_setup_4();
void ball_setup_5();
void ball_setup_6();
void setup_at_end();
void time_step_cpu(double);
void balls_stop();
void balls_freeze();
void balls_translate(pVect amt, int idx);
void balls_translate(pVect amt);
void balls_push(pVect amt, int idx);
void balls_push(pVect amt);
void balls_twirl();
void lock_update();

Ball *make_marker(pCoor pos, pColor col);

float opt_spring_constant, opt_spring_damp;
float opt_friction_dynamic;
float opt_air_resistance;
float distance_relaxed;
int chain_length;
Balls balls;
Ball *head_ball, *tail_ball;
Links links;
vector<Surface> surfaces;
int opt_sphere_slices;
```

```
float opt_omega;  // For disk and top.

Links link_new
( Ball *ball1, Ball *ball2,
  float stiffness = 0.2, float spring_constant = 0);

/// CUDA Stuff

int opt_physics_method;
GPU_Info gpu_info;
cudaEvent_t frame_start_ce, frame_stop_ce;
int opt_block_size;
bool gpu_const_data_stale;
bool cpu_phys_data_stale;

void data_cpu_to_gpu_constants();
void data_cpu_to_gpu_dynamic();
void data_gpu_to_cpu_dynamic();
CPU_GPU_Common c;  // c is for common.  See links.cuh.

void render_link_start();
void render_link_gather(Link *link);
void render_link_render(vk::CommandBuffer& cb, bool shadow_volumes = false);
void render_link_render_sv(vk::CommandBuffer& cb)
{ render_link_render(cb,true); };

VPipeline pipe_set_2, pipe_set_2_sv;
double world_time_link_update;

VCompute comp_links;
VBuffer<int> buf_links_as_idx; // Indices for links' acceleration structures.
VBuffer<vec2> buf_links_tex_coor;
VBuffer<vec4> buf_links_vertex_g;
VBuffer<vec4> buf_links_normal_g;
RT_Geo_Package rt_geo_links;

int opt_sides, opt_segments;

vector< VBufferVV<vec4>* > lis_bufs;
VBufferVV<vec4> lis_pos1, lis_pos2;
VBufferVV<vec4> lis_v1, lis_v2;
VBufferVV<vec4> lis_b1_ydir, lis_b2_ydir, lis_tex_rad;

// Homework 4
HW04_Stuff hw04_stuff;
VVertex_Buffer_Set bset_hw04_p1, bset_hw04_p2;
VPipeline pipe_hw04_p1, pipe_hw04_p2;
RT_Geo_Package rt_geo_hw04;
VBufferV<Uni_HW04> uni_hw04;
bool uni_hw04_needs_update;
int opt_texture;

void render_hw04(vk::CommandBuffer& cb);
```

```
};

enum { OH_Links = 1, OH_Platform = 2, OH_Sphere = 4 };

void
World::init_graphics()
{
  const bool many_vars = false;
  vh.want_raytrace().init();
  vh.opt_record_every_time = true;
  vh.display_cb_set([&](){frame_callback();});
  vh.cbs_cmd_record.push_back( [&](vk::CommandBuffer& cb){ render(cb); });
  sphere.ppuni_light = &puni_light_curr;
  sphere.transform = &transform;

  opt_want_raytrace = false;

  if ( vh.have_raytrace )
    {
      raytrace.init();
      for ( auto b: { &bset_platform_mir, &bset_platform_tex } )
        b->usage_set
          ( vk::BufferUsageFlagBits::eVertexBuffer
            | vk::BufferUsageFlagBits::eStorageBuffer );
      buf_links_as_idx.init
        ( vh.qs, vk::BufferUsageFlagBits::eStorageBuffer, 0,
          vk::MemoryPropertyFlagBits::eDeviceLocal );
      buf_links_tex_coor.init
        ( vh.qs, vk::BufferUsageFlagBits::eStorageBuffer, 0,
          vk::MemoryPropertyFlagBits::eDeviceLocal );
      buf_links_vertex_g.init
        ( vh.qs, vk::BufferUsageFlagBits::eStorageBuffer, 0,
          vk::MemoryPropertyFlagBits::eDeviceLocal );
      buf_links_normal_g.init
        ( vh.qs, vk::BufferUsageFlagBits::eStorageBuffer, 0,
          vk::MemoryPropertyFlagBits::eDeviceLocal );
    }

  buf_uni_common.init(vh.qs,vk::BufferUsageFlagBits::eUniformBuffer);
  sphere.buf_uni_common = buf_uni_common;

  lis_bufs =
    { &lis_pos1, &lis_pos2,
      &lis_v1, &lis_v2,
      &lis_b1_ydir, &lis_b2_ydir,
      &lis_tex_rad };

  for ( auto b: lis_bufs )
    b->init(vh.dev_phys, vh.dev, vk::BufferUsageFlagBits::eStorageBuffer );

  ///
  /// Graphical Model Initialization
  ///
```

```
opt_head_lock = false;
opt_tail_lock = false;
opt_tryout1 = opt_tryout2 = false;
opt_normal_sphere = true;
opt_sphere_true = true;
opt_tryoutf = 0;
variable_control.insert_linear(opt_tryoutf,"Tryout F",0.1);

eye_location = pCoor(24.2,11.6,-38.7);
eye_direction = pVect(-0.42,-0.09,0.9);

platform_xmin = -40; platform_xmax = 40;
platform_zmin = -40; platform_zmax = 40;

texid_syl.init
  ( vh.qs, P_Image_Read( vh.gp_root_get() / "vulkan" / "gpup.png", 255 ) );
texid_emacs.init( vh.qs, P_Image_Read("mult.png",-1) );
texid_hw.init( vh.qs, P_Image_Read("hw-assign.png",255) );
sampler = vh.qs.dev.createSampler
  ( { {},
      vk::Filter::eLinear,  // magFilter
      vk::Filter::eLinear,  // minFilter
      vk::SamplerMipmapMode::eLinear,
      // Also: eRepeat, eMirroredRepeat, eClampToEdge, etc.
      vk::SamplerAddressMode::eRepeat,
      vk::SamplerAddressMode::eRepeat,
      vk::SamplerAddressMode::eRepeat,
      0.0f,  // mipLodBias
      true, // anisotropyEnable,
      16.0f, // maxAnisotropy
      false, vk::CompareOp::eNever,  // compareEnable, compareOp
      0.0f, VK_LOD_CLAMP_NONE,  // min and max LOD
      vk::BorderColor::eFloatOpaqueBlack,
      false // unnormalizedCoordinates
      } );

sphere.texture_set(sampler,texid_emacs);

opt_light_intensity = 50;
light_location = pCoor(platform_xmax,platform_xmax,platform_zmin);
uni_light.init(vh.qs,vk::BufferUsageFlagBits::eUniformBuffer);
uni_light_all.init(vh.qs,vk::BufferUsageFlagBits::eUniformBuffer);
uni_light_reflected.init(vh.qs,vk::BufferUsageFlagBits::eUniformBuffer);
uni_light_ambient.init(vh.qs,vk::BufferUsageFlagBits::eUniformBuffer);

variable_control.insert(opt_light_intensity,"Light Intensity");

opt_move_item = MI_Eye;
opt_pause = false;
opt_single_time_step = 0;
opt_single_frame = false;

platform_update();
modelview_update();
```

```
  adj_t_stop = 0;
  adj_duration = 0.25;

  opt_sides = 20;
  opt_segments = 16;
  if ( many_vars )
    variable_control.insert(opt_sides,"Link Sides");
  variable_control.insert(opt_segments,"Link Segments",5,1);
  opt_hide_stuff = 0;

  if ( vh.have_raytrace )
    {
      raytrace.buf_light_set(uni_light_all);
    }

  // Homework 4

  uni_hw04.init( vh.qs, vk::BufferUsageFlagBits::eUniformBuffer );
  uni_hw04_needs_update = true;
  opt_texture = TX_Orig;

}

void
World::run()
{
  vh.message_loop_spin();

  uni_light.destroy();
  uni_light_ambient.destroy();
  uni_light_all.destroy();
  uni_light_reflected.destroy();
  transform.destroy();
  buf_uni_common.destroy();

  for ( auto b: lis_bufs ) b->destroy();

  pipe_platform_tex.destroy(); pipe_platform_mir.destroy();
  bset_platform_tex.destroy(); bset_platform_mir.destroy();
  pipe_set_2.destroy();
  pipe_set_2_sv.destroy();

  texid_syl.destroy();
  texid_emacs.destroy();
  texid_hw.destroy();
  vh.dev.destroySampler(sampler);
  shapes.destroy();
  sphere.destroy();

  uni_hw04.destroy();
  bset_hw04_p1.destroy();
  pipe_hw04_p1.destroy();
  bset_hw04_p2.destroy();
```

```
  pipe_hw04_p2.destroy();

  buf_links_as_idx.destroy();
  buf_links_tex_coor.destroy();
  buf_links_normal_g.destroy();
  buf_links_vertex_g.destroy();
  rt_geo_platform_tex.destroy();
  rt_geo_platform_mir.destroy();
  rt_geo_links.destroy();
  comp_links.destroy();
  raytrace.destroy();

  vh.finish();
}

void
World::render_hw04(vk::CommandBuffer& cb)
{
  if ( !pipe_hw04_p1 )
    pipe_hw04_p1
      .init( vh.qs )
      .ds_follow( puni_light_curr )
      .ds_follow( transform )
      .ds_use( sampler, texid_hw )
      .topology_set( vk::PrimitiveTopology::eTriangleStrip )
      .shader_inputs_info_set<pNorm,pCoor,pColor,pTCoor>()
      .shader_code_set
        ("links-shdr-simple.cc", "vs_main();", nullptr, "fs_main();")
      .create();

  if ( !pipe_hw04_p2 )
    pipe_hw04_p2
      .init( vh.qs )
      .ds_follow( puni_light_curr )
      .ds_follow( transform )
      .ds_uniform_use( "BIND_UNI_COMMON", buf_uni_common )
      .ds_uniform_use( "BIND_HW04", uni_hw04 )
      .ds_use( sampler, texid_hw )
      .topology_set( vk::PrimitiveTopology::eTriangleStrip )
      .shader_inputs_info_set<pCoor,pTCoor>()
      .shader_code_set
        ("hw04-shdr-sol.cc", "vs_main();", "gs_main();", "fs_main();" )
      .create();

  if ( uni_hw04_needs_update )
    {
      uni_hw04_needs_update = false;
      uni_hw04->color = 1.0 * color_white;
      uni_hw04->opt_texture = opt_texture;
      uni_hw04.to_dev();
    }


  bset_hw04_p1.reset( pipe_hw04_p1 );
```

```
/// Put Homework 4, Problem 1 solution below.
//

// Compute center of outer ring.
pCoor ctr(0,0,0);
const auto& balls_outer = hw04_stuff.balls_outer;
for ( Ball* b: balls_outer ) ctr += b->position;
const size_t n_balls = balls_outer.size();
ctr.w = n_balls;
ctr.homogenize();
// Wheel z
pNorm wz [[maybe_unused]] =
  cross(ctr,balls_outer[0]->position,balls_outer[n_balls/4]->position);

/// SOLUTION -- Problem 1
//
for ( size_t i=0; i<=n_balls; i++ )
  {
    /// SOLUTION -- Problem 1
    //
    // Visit the i=0 ball twice so that a triangle strip around the
    // wheel can be completed. To do that change loop end condition
    // above from "i<n_balls" to "i<=n_balls" and change the index
    // to balls_outer, below, from i to i%n_balls.

    Ball *b = balls_outer[i%n_balls];

    /// SOLUTION -- Problem 1
    //
    //  Compute two coordinates (per iteration). Both points are on
    //  a line starting at the wheel center, ctr, and and passing
    //  through the ball center, pos. The first, p, is at a distance
    //  b->radius from the ball center. (This coordinate was
    //  provided in the unmodified assignment.) The second, pb, is
    //  at a distance 2 * b->radius from p.

    pCoor pos = b->position;
    pNorm n(ctr,pos);
    pCoor p = pos + 2 * b->radius * n;
    pCoor pb = p + 2 * b->radius * n;

    //  Insert those too coordinates into the buffer set.
    //
    bset_hw04_p1 << p << pb;

    //  Also insert colors.
    //
    bset_hw04_p1 << color_lsu_spirit_gold << color_lsu_spirit_gold;

    //  Problem 1 did not require that any particular texture coordinates
    //  be inserted, so it's hard to get this wrong. Here texture
    //  coordinates are inserted so that an entire copy of the texture
    //  appears between each pair of adjacent balls.
```

```
      //
      bset_hw04_p1 << pTCoor(0,i) << pTCoor(1,i);

      //  Problem 1 did not specifically require that the approximate
      //  normal, wz, had to be improved on.  If the wheel were rigid
      //  then wz would be a correct normal, but the wheel can flex,
      //  so its not. Here the approximate normal is used.
      //
      bset_hw04_p1 << wz << wz;

      // Commented out code from unmodified assignment.
      // bset_hw04_p1 << p << color_cyan << pNorm(0,1,0) << pTCoor(0,0);
    }

  bset_hw04_p1.to_dev();
  pipe_hw04_p1.record_draw( cb, bset_hw04_p1 );


  /// Problems 2 and 3
  //
  // Feel free to inspect the code below, but don't change it.

  bset_hw04_p2.reset( pipe_hw04_p2 );

  const size_t n_links = hw04_stuff.ribs.size();

  for ( size_t i = 0; i < n_links; i++ )
    {
      size_t i2 = (i+1) % n_links;
      Link *link1 = hw04_stuff.ribs[i], *link2 = hw04_stuff.ribs[i2];
      const int li1 = link1->idx;
      const int li2 = link2->idx;
      pTCoor tex11 = link1->ball1->tca[opt_texture];
      pTCoor tex12 = link1->ball2->tca[opt_texture];
      pTCoor tex21 = link2->ball1->tca[opt_texture];
      pTCoor tex22 = link2->ball2->tca[opt_texture];
      pCoor pos11 = lis_pos1[li1], pos12 = lis_pos2[li1];
      pVect v11 = lis_v1[li1], v12 = lis_v2[li1];
      pCoor pos21 = lis_pos1[li2], pos22 = lis_pos2[li2];
      pVect v21 = lis_v1[li2], v22 = lis_v2[li2];

      const int n_segs = opt_segments;
      const float delta_t = 1.0/n_segs;
      for ( int s=0; s<=n_segs; s++ )
        {
          const float t = s * delta_t;
          pCoor pos1t = bez(pos11,pos12,v11,v12,t);
          pCoor pos2t = bez(pos21,pos22,v21,v22,t);
          if ( s == 0 ) bset_hw04_p2 << pos1t << pTCoor(0,0);
          bset_hw04_p2
            << pos1t << mix(tex11,tex12,t)
            << pos2t << mix(tex21,tex22,t);
          if ( s == n_segs ) bset_hw04_p2 << pos2t << pTCoor(0,0);
        }
```

```
    }

  bset_hw04_p2.to_dev();

  pipe_hw04_p2.record_draw( cb, bset_hw04_p2 );
}


void
World::platform_update()
{
  const float tile_count = 19;
  const float ep = 1.00001;
  const float delta_x = ( platform_xmax - platform_xmin ) / tile_count * ep;
  const float zdelta = ( platform_zmax - platform_zmin ) / tile_count * ep;

  const float trmin = 0.05;
  const float trmax = 0.7;
  const float tsmin = 0;
  const float tsmax = 0.4;
  pColor color_platform( color_white * 0.9 );

  if ( !pipe_platform_mir )
    pipe_platform_mir
      .init( ff_state )
      .ds_set( color_lsu_spirit_purple )
      .ds_follow( puni_light_curr )
      .ds_follow( transform )
      .topology_set( vk::PrimitiveTopology::eTriangleList )
      .create();
  bset_platform_mir.reset( pipe_platform_mir ).tolerate_tcoords();

  if ( !pipe_platform_tex )
    pipe_platform_tex
      .init( ff_state )
      .ds_set_material(color_platform)
      .ds_follow( puni_light_curr )
      .ds_follow( transform )
      .topology_set( vk::PrimitiveTopology::eTriangleList )
      .ds_use( sampler, texid_syl )
      .create();
  bset_platform_tex.reset( pipe_platform_tex );

  const pVect platform_normal(0,1,0);
  pTCoor t00(trmin,tsmin), t01(trmin,tsmax), t10(trmax,tsmin), t11(trmax,tsmax);
  bool even = true;

  for ( int i = 0; i < tile_count; i++ )
    {
      const float x0 = platform_xmin + i * delta_x;
      const float x1 = x0 + delta_x;
      const float y = 0;
      for ( float z = platform_zmin; z < platform_zmax; z += zdelta )
        {
```

```
          auto& bset = even ? bset_platform_tex : bset_platform_mir;
          const float z1 = z+zdelta;
          pCoor p00(x0,y,z), p01(x0,y,z1), p10(x1,y,z), p11(x1,y,z1);

          bset << t11 << platform_normal << p00;
          bset << t10 << platform_normal << p01;
          bset << t00 << platform_normal << p11;

          bset << t11 << platform_normal << p00;
          bset << t00 << platform_normal << p11;
          bset << t01 << platform_normal << p10;

          even = !even;
        }
    }

  bset_platform_mir.to_dev();
  bset_platform_tex.to_dev();

  if ( vh.have_raytrace && !rt_geo_platform_mir )
    {
      pColor mirror_color(color_lsu_spirit_purple);
      mirror_color.a = -0.5;
      rt_geo_platform_mir
        .init( raytrace )
        .color_front_set( mirror_color )
        .bind( bset_platform_mir, RT_Geometry_Triangles );
      rt_geo_platform_tex
        .init( raytrace )
        .color_front_set( color_platform )
        .bind( VR_BUF_IDX_sampler, sampler, texid_syl )
        .bind( bset_platform_tex, RT_Geometry_Triangles );
    }
}

void
World::modelview_update()
{
  pMatrix_Translate center_eye(-eye_location);
  pMatrix_Rotation rotate_eye(eye_direction,pVect(0,0,-1));
  modelview = rotate_eye * center_eye;
}

void
World::render_objects(vk::CommandBuffer& cb, Render_Option option)
{
  //  const float shininess_ball = 5;
  pColor spec_color(1,1,1);
  //  pColor spec_light = color_white * opt_light_intensity;

  const bool hide_links = opt_hide_stuff & OH_Links;
  const bool hide_platform = opt_hide_stuff & OH_Platform;
  const bool hide_sphere = opt_hide_stuff & OH_Sphere;
```

```
  sphere.init(opt_sphere_slices);

  if ( option == RO_Shadow_Volumes )
    {
      if ( !hide_sphere )
        {
          sphere.light_pos = light_location;
          sphere.render_bunch_gather_sv(world_time);
          for ( Ball *ball: balls )
            sphere.render_shadow_volume(cb,ball->radius,ball->position);

          sphere.render_bunch_render_sv(cb);
        }

      if ( !hide_links )
        {
          render_link_start();
          for ( Link *link: links )
            if ( link->is_renderable ) render_link_gather(link);
          render_link_render_sv(cb);
        }
    }
  else
    {
      sphere.rt_geo_sphere.rt_wanted_set(!hide_sphere);
      if ( !hide_sphere )
        {
          sphere.render_bunch_gather(world_time);
          for ( Ball *ball: balls )
            {
              if ( ball->contact )
                sphere.color = color_gray;
              else if ( ball->mass > 0 && ball->mass < ball->mass_min )
                sphere.color = color_red;
              else if ( ball->mass > 0 && ball->locked )
                sphere.color = color_pale_green;
              else
                sphere.color = ball->color;

              sphere.color.a = ball->specularity ? -ball->specularity : 1;
              sphere.render
                (cb, ball->radius,ball->position,
                 pMatrix_Rotation(ball->orientation));
            }

          sphere.render_bunch_render(cb, opt_sphere_true);
        }

      rt_geo_links.rt_wanted_set(!hide_links);
      if ( !hide_links )
        {
          render_link_start();
          for ( Link *link: links )
            if ( link->is_renderable ) render_link_gather(link);
```

```
          render_link_render(cb);
        }
    }

  rt_geo_platform_tex.rt_wanted_set(!hide_platform);
  rt_geo_platform_mir.rt_wanted_set(!hide_platform);
  if ( vh.raytrace_active() ) return;

  if ( last_setup == 4 && option != RO_Shadow_Volumes )
    render_hw04(cb);

  if ( hide_platform ) return;

  if ( option == RO_Shadow_Volumes ) return;

  //
  // Render Platform
  //
  pipe_platform_tex.record_draw(cb,bset_platform_tex);
}

void
World::render(vk::CommandBuffer& cb)
{
  const bool do_raytrace = vh.raytrace_active();

  /// Emit a Graphical Representation of Simulation State
  //

  // Understanding of the code below not required for introductory
  // lectures.

  // That said, much of the complexity of the code is to show
  // the ball shadow and reflection.


  const int win_width = vh.get_width();
  const int win_height = vh.get_height();
  const float aspect = float(win_width) / win_height;
  const float n_dist = 0.01;
  const float xr = .8 * n_dist;

  transform.eye_from_global_set( modelview );
  transform.clip_from_eye_set
    ( pMatrix_Frustum(-xr,xr,-xr/aspect,xr/aspect,n_dist,5000) );
  // pMatrix_Frustum: left, right, bottom, top, near, far

  vh.clearValues[0].color =
    vk::ClearColorValue( array<float, 4>( { { 0, 0, 0, 0.5 } } ) );

  /// Lighting
  //
  auto& light_all = uni_light_all->cgl_LightSource[0];
```

```
  light_all.position = transform.eye_from_global * light_location;

  light_all.constantAttenuation = 0.5;
  light_all.linearAttenuation = 1.0;
  light_all.quadraticAttenuation = 0;

  pColor ambient_color(0x555555);

  uni_light_all->cgl_LightModel.ambient = ambient_color;

  light_all.diffuse = opt_light_intensity * color_white;
  light_all.ambient = color_black;
  light_all.specular = opt_light_intensity * color_white;

  uni_light_reflected = uni_light_all.get_val();
  uni_light_reflected->cgl_LightSource[0].position =
    transform.eye_from_global * pMatrix_Scale(1,-1,1) * light_location;

  uni_light_ambient = uni_light_all.get_val();
  auto& light_amb = uni_light_ambient->cgl_LightSource[0];
  light_amb.diffuse = color_black;
  light_amb.specular = color_black;

  uni_light = uni_light_all.get_val();
  uni_light->cgl_LightModel.ambient = color_black;

  uni_light.to_dev();
  uni_light_ambient.to_dev();
  uni_light_all.to_dev();
  uni_light_reflected.to_dev();

  puni_light_curr = &uni_light_all;

  buf_uni_common->tryout =
    ivec4(opt_tryout1,opt_tryout2,opt_normal_sphere,false);
  buf_uni_common->tryoutf = vec4(opt_tryoutf, 0, 0, 0);
  buf_uni_common->light_pos_g = light_location;
  if ( do_raytrace )
    {
      raytrace.buf_uni_common->opt_tryout = buf_uni_common->tryout;
      raytrace.buf_uni_common->opt_tryoutf = buf_uni_common->tryoutf;
      raytrace.buf_uni_common->opt_shadows = opt_shadows;
      raytrace.buf_uni_common->opt_mirror =
        opt_mirror ? opt_mirror_depth : 0;
      raytrace.buf_light_set(uni_light_all);
    }

  const double time_now = time_wall_fp();
  const bool blink_visible = int64_t(time_now*3) & 1;
# define BLINK(txt,pad) ( blink_visible ? txt : pad )

  vh.fbprintf("%s\n",frame_timer.frame_rate_text_get());

  vh.fbprintf
```

```
    ("Raytrace: %s ('R')   Compiled: %s\n",
     !vh.have_raytrace ? "NA" :
     opt_want_raytrace ? ( raytrace.rt_ready ? "ON " : "???" ) : "OFF",
#ifdef __OPTIMIZE__
     "WITH OPTIMIZATION"
#else
     BLINK("WITHOUT OPTIMIZATION","")
#endif
     );

  vh.fbprintf
    ("Time Step: %8d  World Time: %11.6f  %s\n",
     time_step_count, world_time,
     opt_pause ? BLINK("PAUSED, 'p' to unpause, SPC or S-SPC to step.","") :
     "Press 'p' to pause."
     );

  vh.fbprintf
    ("Eye location: [%5.1f, %5.1f, %5.1f]  "
     "Eye direction: [%+.2f, %+.2f, %+.2f]\n",
     eye_location.x, eye_location.y, eye_location.z,
     eye_direction.x, eye_direction.y, eye_direction.z);

  Ball& ball = *balls[0];

  vh.fbprintf
    ("Head Ball Pos  [%5.1f,%5.1f,%5.1f] Vel [%+5.1f,%+5.1f,%+5.1f]\n",
     ball.position.x,ball.position.y,ball.position.z,
     ball.velocity.x,ball.velocity.y,ball.velocity.z );

  vh.fbprintf
    ("%4zd Balls  %4zd Links  Physics: %s ('a' to change)  \n",
     balls.size(), links.size(), gpu_physics_method_str[opt_physics_method]);

  vh.fbprintf
    ("Hide: %c%c%c ('!@#')  Effect: %c%c ('or')  Sphere: %s  ('z')  "
     "Tryout 1: %s  ('y')  Tryout 2: %s  ('Y')  Normals: %s ('n')\n",
     opt_hide_stuff & OH_Sphere ? 'S' : '_',
     opt_hide_stuff & OH_Links ? 'L' : '_',
     opt_hide_stuff & OH_Platform ? 'P' : '_',
     opt_shadows ? 'S' : '_',
     opt_mirror ? 'M' : '_',
     opt_sphere_true ? "TRUE" : "TRI" ,
     opt_tryout1 ? BLINK("ON ","   ") : "OFF",
     opt_tryout2 ? BLINK("ON ","   ") : "OFF",
     opt_normal_sphere ? "SPHERE" : "TRI");

  pVariable_Control_Elt* const cvar = variable_control.current;
  vh.fbprintf("VAR %s = %.5f  (TAB or '`' to change, +/- to adjust)\n",
                    cvar->name,cvar->get_val());

  vh.fbprintf
    ("HW4: Texture %s ('x')\n",
     hw4_texture_str[opt_texture]);
```

```
// Render Marker for Light Source
//
shapes.record_tetrahedron(cb,transform,light_location,0.15);

//
// Render ball reflection.  (Will be blended with dark tiles.)
//

const bool hide_platform = opt_hide_stuff & OH_Platform;

if ( do_raytrace )
  {
    raytrace.tform_update(transform);
    render_objects(cb,RO_Normally);
    raytrace.buf_uni_common.to_dev();

    return;
  }

buf_uni_common.to_dev();

vk::ColorComponentFlags clr_all
  ( vk::ColorComponentFlagBits::eR | vk::ColorComponentFlagBits::eG |
    vk::ColorComponentFlagBits::eB | vk::ColorComponentFlagBits::eA );

if ( !hide_platform )
  {
    // Save current state of fixed-functionality state.
    //
    ff_state.push();

    if ( opt_mirror )
      {
        /// Mirror Effect
        //
        //  - Step 1:  Write location of mirrors into stencil buffer.
        //
        //  - Step 2a: Prepare a matrix that transforms to mirrored location.
        //  - Step 2b: Set up blending to blend mirrored objects with mirrors.
        //
        //  - Step 3:  Render all objects except mirrors.

        ///  Step 1: Write location of mirrors into stencil buffer.
        //
        //   Write stencil value 2 at location of dark (mirrored) tiles.
        //

        vk::StencilOpState s_op_apply
          ( vk::StencilOp::eReplace, vk::StencilOp::eKeep,
            vk::StencilOp::eKeep, vk::CompareOp::eNever, 2, 2, 2 );
        // sfail, spass, dfail, compare;  cmask, wmask, ref

        ff_state.pipelineDepthStencilStateCreateInfo_get()
```

```
      .setStencilTestEnable(true)
      .setFront(s_op_apply).setBack(s_op_apply);

    // Render mirrored tiles (but only to write stencil buffer).
    //
    pipe_platform_mir.record_draw(cb,bset_platform_mir);

    cb.nextSubpass(vk::SubpassContents::eInline); vh.qs.subpass++;

    // Prepare to write only stenciled locations.
    //
    vk::StencilOpState s_op_use
      ( vk::StencilOp::eKeep, vk::StencilOp::eKeep,
        vk::StencilOp::eKeep, vk::CompareOp::eEqual, 2, 2, 2 );
    // sfail, spass, dfail, compare;  cmask, wmask, ref
    //
    // Stencil test passes if stencil value = 2.
    // Don't change stencil buffer.

    ff_state.pipelineDepthStencilStateCreateInfo_get()
      .setFront(s_op_use).setBack(s_op_use);

    ///   Step 2a: Prepare a matrix that transforms to mirrored location.
    //    Use a transform that reflects objects to other side of platform.
    //
    const pMatrix prev = transform.eye_from_global;
    const pMatrix_Scale reflect(1,-1,1);
    transform.eye_from_global = prev * reflect;

    // Reflected front face should still be treated as the front face.
    //
    ff_state.pipelineRasterizationStateCreateInfo_get()
      .setFrontFace(vk::FrontFace::eClockwise);

    puni_light_curr = &uni_light_reflected;

    /// Step 3: Render all objects.

    render_objects(cb,RO_Mirrored);

    transform.eye_from_global = prev;

    ff_state.pipelineRasterizationStateCreateInfo_get()
      .setFrontFace(vk::FrontFace::eCounterClockwise);
  }

// blend operation, blend factors, blend constants
ff_state.pipelineColorBlendStateCreateInfo_get()
  .setBlendConstants( array<float,4>{.5,.5,.5,.5} );

ff_state.pipelineColorBlendAttachmentState_get() =
  vk::PipelineColorBlendAttachmentState
  ( true,                    // blendEnable
    vk::BlendFactor::eConstantAlpha,  // srcColorBlendFactor
```

```
          vk::BlendFactor::eConstantAlpha,  // dstColorBlendFactor
          vk::BlendOp::eAdd,       // colorBlendOp
          vk::BlendFactor::eOne,   // srcAlphaBlendFactor
          vk::BlendFactor::eZero,  // dstAlphaBlendFactor
          vk::BlendOp::eAdd,       // alphaBlendOp
          clr_all      // colorWriteMask
          );

      puni_light_curr = &uni_light_all;;
      pipe_platform_mir.record_draw(cb,bset_platform_mir);

      ff_state.pop();
    }

  if ( !opt_shadows )
    {
      // Render.
      //
      render_objects(cb,RO_Normally);
    }
  else
    {
      //
      // First pass, render using only ambient light.
      //
      puni_light_curr = &uni_light_ambient;

      // Send balls, tiles, and platform to opengl.
      //
      render_objects(cb,RO_Normally);

      //
      // Second pass, find un-shadowed areas and add on light0.
      //


      /// Write Shadow Volumes
      //
      vk::ClearAttachment clr_att
        ( vk::ImageAspectFlagBits::eStencil, 0,
          vk::ClearDepthStencilValue(0,0) );
      vk::ClearRect clr_rect( vk::Rect2D({}, vh.s_extent), 0, 1 );
      cb.clearAttachments( clr_att, clr_rect );

      ff_state.push();

      vk::StencilOpState s_op_apply_f
        ( vk::StencilOp::eKeep, vk::StencilOp::eIncrementAndWrap,
          vk::StencilOp::eKeep, vk::CompareOp::eAlways, -1, -1, 0 );
      vk::StencilOpState s_op_apply_b
        ( vk::StencilOp::eKeep, vk::StencilOp::eDecrementAndWrap,
          vk::StencilOp::eKeep, vk::CompareOp::eAlways, -1, -1, 0 );
      // sfail, spass, dfail, compare; c-mask, w-mask, ref
      // ref is -1 (all 1's) because we want to operate on all bits in
```

```
// stencil value.

ff_state.pipelineDepthStencilStateCreateInfo_get()
  .setStencilTestEnable(true)
  .setDepthWriteEnable(false)
  .setFront(s_op_apply_f).setBack(s_op_apply_b);

if ( !opt_shadow_volumes )
  {
    ff_state.pipelineColorBlendAttachmentState_get()
      .setColorWriteMask( {} );
  }
else
  {
    ff_state.pipelineColorBlendAttachmentState_get() =
      vk::PipelineColorBlendAttachmentState
      ( true,                    // blendEnable
        vk::BlendFactor::eConstantAlpha,  // srcColorBlendFactor
        vk::BlendFactor::eConstantAlpha,  // dstColorBlendFactor
        vk::BlendOp::eAdd,       // colorBlendOp
        vk::BlendFactor::eOne,   // srcAlphaBlendFactor
        vk::BlendFactor::eZero,  // dstAlphaBlendFactor
        vk::BlendOp::eAdd,       // alphaBlendOp
        clr_all      // colorWriteMask
        );

    ff_state.pipelineColorBlendStateCreateInfo_get()
      .setBlendConstants( array<float,4>{.5,.5,.5,.5} );

    ff_state.pipelineDepthStencilStateCreateInfo_get()
      .setDepthWriteEnable(true);
  }

// Write stencil with shadow locations based on shadow volumes
// cast by light0 (light_location).  Shadowed locations will
// have a positive stencil value.
//
render_objects(cb,RO_Shadow_Volumes);

cb.nextSubpass(vk::SubpassContents::eInline); vh.qs.subpass++;

// Use stencil test to prevent writes to shadowed areas.
//
vk::StencilOpState s_op_apply_fb
  ( vk::StencilOp::eKeep, vk::StencilOp::eKeep,
    vk::StencilOp::eKeep, vk::CompareOp::eEqual, -1, -1, 0 );
// sfail, spass, dfail, compare;  c-mask, w-mask, ref
//
// Stencil test passes if stencil value == 0.

ff_state.pipelineDepthStencilStateCreateInfo_get()
  .setFront(s_op_apply_fb).setBack(s_op_apply_fb)
  .setDepthCompareOp(vk::CompareOp::eEqual);
```

```
      // Allow pixels to be re-written.
      //
      vk::ColorComponentFlags cmask_all
        ( vk::ColorComponentFlagBits::eR | vk::ColorComponentFlagBits::eG |
          vk::ColorComponentFlagBits::eB | vk::ColorComponentFlagBits::eA );

      ff_state.pipelineColorBlendAttachmentState_get() =
        vk::PipelineColorBlendAttachmentState
        ( true,                   // blendEnable
          vk::BlendFactor::eOne,  // srcColorBlendFactor
          vk::BlendFactor::eOne,  // dstColorBlendFactor
          vk::BlendOp::eAdd,       // colorBlendOp
          vk::BlendFactor::eOne,   // srcAlphaBlendFactor
          vk::BlendFactor::eOne,  // dstAlphaBlendFactor
          vk::BlendOp::eAdd,       // alphaBlendOp
          cmask_all      // colorWriteMask
          );

      puni_light_curr = &uni_light;

      // Render.
      //
      render_objects(cb,RO_Normally);

      ff_state.pop();
    }

  puni_light_curr = &uni_light_all;
}


void
World::cb_keyboard()
{
  const int key = vh.keyboard_key_get();
  if ( !key ) return;
  pVect adjustment(0,0,0);
  pVect user_rot_axis(0,0,0);
  const bool kb_mod_s = vh.keyboard_shift;
  const bool kb_mod_c = vh.keyboard_control;
  const float move_amt = kb_mod_s ? 2.0 : kb_mod_c ? 0.08 : 0.4;

  switch ( key ) {
  case FB_KEY_LEFT: adjustment.x = -move_amt; break;
  case FB_KEY_RIGHT: adjustment.x = move_amt; break;
  case FB_KEY_PAGE_UP: adjustment.y = move_amt; break;
  case FB_KEY_PAGE_DOWN: adjustment.y = -move_amt; break;
  case FB_KEY_DOWN: adjustment.z = move_amt; break;
  case FB_KEY_UP: adjustment.z = -move_amt; break;
  case FB_KEY_DELETE: user_rot_axis.y = 1; break;
  case FB_KEY_INSERT: user_rot_axis.y =  -1; break;
  case FB_KEY_HOME:
    if ( opt_move_item == MI_Light ) adjustment.x = 1; else user_rot_axis.x = 1;
    break;
```

```
case FB_KEY_END: user_rot_axis.x = -1; break;
case '1': ball_setup_1(kb_mod_c); break;
case '2': ball_setup_2(); break;
case '3': ball_setup_3(); break;
case '4': ball_setup_4(); break;
case '5': ball_setup_5(); break;
case '6': ball_setup_6(); break;
case 'a':
  opt_physics_method++;
  if ( opt_physics_method == GP_ENUM_SIZE ) opt_physics_method = GP_cpu;
  break;
case 'b': opt_move_item = MI_Ball; break;
case 'B': opt_move_item = MI_Ball_V; break;
case 'e': case 'E': opt_move_item = MI_Eye; break;
case 'g': case 'G': opt_gravity = !opt_gravity; break;
case 'h': case 'H': opt_head_lock = !opt_head_lock; break;
case '!': opt_hide_stuff ^= OH_Sphere; break;
case '@': opt_hide_stuff ^= OH_Links; break;
case '#': opt_hide_stuff ^= OH_Platform; break;
case 't': case 'T': opt_tail_lock = !opt_tail_lock; break;
case 'l': case 'L': opt_move_item = MI_Light; break;
case 'r': opt_mirror = !opt_mirror; break;
case 'R':
  opt_want_raytrace = !opt_want_raytrace;
  world_time_link_update = 0;
  break;
case 'o': opt_shadows = !opt_shadows; break;
case 'O':
  opt_shadow_volumes = !opt_shadow_volumes;
  if ( opt_shadow_volumes ) opt_shadows = true;
  break;
case 'n': case 'N': opt_normal_sphere = !opt_normal_sphere; break;
case 'p': case 'P': opt_pause = !opt_pause; break;
case 's': case 'S': balls_stop(); break;
case 'v': case 'V': opt_shader++;
  if ( opt_shader >= sizeof(sh_names)/sizeof(sh_names[0]) ) opt_shader = 0;
  break;
case 'w': case 'W': balls_twirl(); break;
case 'x': case 'X':
  uni_hw04_needs_update = true;
  opt_texture++; if ( opt_texture == TX_ENUM_SIZE ) opt_texture = 0;
  break;
case 'y': opt_tryout1 = !opt_tryout1; break;
case 'Y': opt_tryout2 = !opt_tryout2; break;
case 'z': opt_sphere_true = !opt_sphere_true; break;
case ' ':
  if ( kb_mod_s && kb_mod_c ) opt_single_time_step = 10;
  else if ( kb_mod_s ) opt_single_time_step = 100;
  else if ( kb_mod_c ) opt_single_time_step = 1; else opt_single_frame = true;
  opt_pause = true;
  break;
case FB_KEY_TAB:
  if ( !kb_mod_s ) { variable_control.switch_var_right(); break; }
case 96: variable_control.switch_var_left(); break;
```

```
  case '-':case '_': variable_control.adjust_lower(); break;
  case '+':case '=': variable_control.adjust_higher(); break;
  default: printf("Unknown key, %d\n",key); break;
  }

  gravity_accel.y = opt_gravity ? -opt_gravity_accel : 0;

  lock_update();
  sphere.bunch_invalidate();

  // Update eye_direction based on keyboard command.
  //
  if ( user_rot_axis.x || user_rot_axis.y )
    {
      pMatrix_Rotation rotall(eye_direction,pVect(0,0,-1));
      user_rot_axis *= invert(rotall);
      eye_direction *= pMatrix_Rotation(user_rot_axis, M_PI * 0.03);
      modelview_update();
    }

  // Update eye_location based on keyboard command.
  //
  if ( adjustment.x || adjustment.y || adjustment.z )
    {
      const double angle =
        fabs(eye_direction.y) > 0.99
        ? 0 : atan2(eye_direction.x,-eye_direction.z);
      pMatrix_Rotation rotall(pVect(0,1,0),-angle);
      adjustment *= rotall;

      switch ( opt_move_item ){
      case MI_Ball:
        adj_vector = adjustment;
        if ( adj_t_stop == 0 )
          adj_t_prev = adj_t_stop = world_time;
        adj_t_stop += adj_duration;
        break;
      case MI_Ball_V: balls_push(adjustment,0); break;
      case MI_Light:
        if ( key == FB_KEY_HOME )
          {
            pVect l_to_b(light_location,balls[0]->position);
            adjustment = 0.25 * l_to_b;
          }
        light_location += adjustment;
        break;
      case MI_Eye: eye_location += adjustment; break;
      default: break;
      }
      modelview_update();
    }

}
```

```
void
World::init(int argc, char **argv)
{
  chain_length = 14;
  const bool many_vars = false;

  opt_time_step_duration = 0.00003;
  if ( many_vars )
    variable_control.insert(opt_time_step_duration,"Time Step Duration");

  distance_relaxed = 15.0 / chain_length;
  opt_spring_constant = 500;
  variable_control.insert(opt_spring_constant,"Spring Constant");

  opt_spring_damp = 0.7;
  if ( many_vars )
    variable_control.insert(opt_spring_damp,"Spring Damping").set_max(1);

  opt_friction_dynamic = 0.2;
  if ( many_vars )
    variable_control.insert(opt_friction_dynamic,"Dynamic Friction");

  opt_omega = 10;
  if ( many_vars )
    variable_control.insert(opt_omega,"Initial Rotation Rate");

  opt_gravity_accel = 9.8;
  opt_gravity = true;
  gravity_accel = pVect(0,-opt_gravity_accel,0);
  if ( many_vars )
    variable_control.insert(opt_gravity_accel,"Gravity");

  opt_air_resistance = 0.01;
  variable_control.insert(opt_air_resistance,"Air Resistance");

  world_time = 0;
  time_step_count = 0;
  last_frame_wall_time = time_wall_fp();
  frame_timer.work_unit_set("Steps / s");

  world_time_link_update = 0;

  opt_mirror = true;
  opt_mirror_depth = 2;
  if ( many_vars )
    variable_control.insert(opt_mirror_depth,"Mirror Depth",1,0);
  opt_shadows = true;
  opt_shadow_volumes = false;
  opt_shader = SO_Plain;

  /// Init CUDA
```

```
// Get information about GPU and its ability to run CUDA.
//
int device_count;
cudaGetDeviceCount(&device_count); // Get number of GPUs.
if ( device_count == 0 )
  {
    fprintf(stderr,"No GPU found, exiting.\n");
    exit(1);
  }

// Print information about the available GPUs.
//
gpu_info_print();

// Use course library routine to choose a good GPU for CUDA.
//
int dev = gpu_choose_index();
CE(cudaSetDevice(dev));
printf("Using GPU %d\n",dev);

gpu_info.get_gpu_info(dev);
cuda_setup(&gpu_info);

// Print information about time_step routine.
//
printf("\nCUDA Routine Resource Usage:\n");

int block_size_max = 4096;

for ( int i=0; i<gpu_info.num_kernels; i++ )
  {
    printf("For %s:\n", gpu_info.ki[i].name);
    printf("  %6zd shared, %zd const, %zd loc, %d regs; "
           "%d max threads per block.\n",
           gpu_info.ki[i].cfa.sharedSizeBytes,
           gpu_info.ki[i].cfa.constSizeBytes,
           gpu_info.ki[i].cfa.localSizeBytes,
           gpu_info.ki[i].cfa.numRegs,
           gpu_info.ki[i].cfa.maxThreadsPerBlock);
    set_min(block_size_max,gpu_info.ki[i].cfa.maxThreadsPerBlock);
  }

opt_physics_method = GP_cuda;
gpu_const_data_stale = true;
cpu_phys_data_stale = false;

CE( cudaEventCreate(&frame_start_ce) );
CE( cudaEventCreate(&frame_stop_ce) );

opt_block_size = 32;

if ( many_vars )
  variable_control.insert_power_of_2
```

```
    (opt_block_size, "Block Size", 32, block_size_max);

c.alloc_n_balls = 0;
c.alloc_n_links = 0;

init_graphics();

opt_sphere_slices = 35;
sphere.opt_texture = true;
sphere.init(opt_sphere_slices);
if ( many_vars )
  variable_control.insert(opt_sphere_slices,"Sphere Slices");

const char* const links_shader_code_path = "links-shdr-links.cc";

const auto bvec_vs = vk::ShaderStageFlagBits::eVertex;
const auto bvec_vgs =
  vk::ShaderStageFlagBits::eVertex | vk::ShaderStageFlagBits::eGeometry;

pipe_set_2
  .init( ff_state )
  .ds_follow( transform )
  .ds_follow( puni_light_curr )
  .ds_uniform_use( "BIND_UNI_COMMON", buf_uni_common )
  .ds_set_material( color_dim_gray, color_salmon )
  .ds_use( sampler, texid_hw )
  .ds_storage_follow( "BIND_LINKS_POS1", lis_pos1, bvec_vs )
  .ds_storage_follow( "BIND_LINKS_POS2", lis_pos2, bvec_vs )
  .ds_storage_follow( "BIND_LINKS_V1", lis_v1, bvec_vs )
  .ds_storage_follow( "BIND_LINKS_V2", lis_v2, bvec_vs )
  .ds_storage_follow( "BIND_LINKS_B1_YDIR", lis_b1_ydir, bvec_vs )
  .ds_storage_follow( "BIND_LINKS_B2_YDIR", lis_b2_ydir, bvec_vs )
  .ds_storage_follow( "BIND_LINKS_TEX_RAD", lis_tex_rad, bvec_vgs )
  .shader_inputs_info_set( )
  .shader_code_set
  (links_shader_code_path,
   "vs_main_2(); ",     // Name of vertex shader main routine.
   "gs_main_2();",      // Name of geometry shader main routine.
   "fs_main();"         // Name of fragment shader main routine.
   )
  .topology_set( vk::PrimitiveTopology::eLineStrip )
  .create();

if ( vh.have_raytrace )
  comp_links
    .init( vh.qs )
    .ds_uniform_use( "BIND_UNI_COMMON", buf_uni_common )
    .ds_uniform_use( "BIND_TRANSFORM", raytrace.buf_uni_transform )
    .ds_storage_follow( "BIND_LINKS_POS1", lis_pos1 )
    .ds_storage_follow( "BIND_LINKS_POS2", lis_pos2 )
    .ds_storage_follow( "BIND_LINKS_V1", lis_v1 )
    .ds_storage_follow( "BIND_LINKS_V2", lis_v2 )
    .ds_storage_follow( "BIND_LINKS_B1_YDIR", lis_b1_ydir )
    .ds_storage_follow( "BIND_LINKS_B2_YDIR", lis_b2_ydir )
```

```
      .ds_storage_follow( "BIND_LINKS_TEX_RAD", lis_tex_rad )
      .ds_storage_follow( "BIND_LINKS_AS_IDX", buf_links_as_idx )
      .ds_storage_follow( "BIND_LINKS_TEX_COOR", buf_links_tex_coor )
      .ds_storage_follow( "BIND_LINKS_VERTEX_G", buf_links_vertex_g )
      .ds_storage_follow( "BIND_LINKS_NORMAL_G", buf_links_normal_g )
      .shader_code_set(links_shader_code_path, "comp_main(); ")
      .create();

  pipe_set_2_sv
    .init( ff_state ).ds_use( uni_light )
    .ds_follow( transform )
    .ds_uniform_use( "BIND_UNI_COMMON", buf_uni_common )
    .ds_storage_follow( "BIND_LINKS_POS1", lis_pos1 )
    .ds_storage_follow( "BIND_LINKS_POS2", lis_pos2 )
    .ds_storage_follow( "BIND_LINKS_V1", lis_v1 )
    .ds_storage_follow( "BIND_LINKS_V2", lis_v2 )
    .ds_storage_follow( "BIND_LINKS_B1_YDIR", lis_b1_ydir )
    .ds_storage_follow( "BIND_LINKS_B2_YDIR", lis_b2_ydir )
    .ds_storage_follow( "BIND_LINKS_TEX_RAD", lis_tex_rad )
    .shader_inputs_info_set( )
    .shader_code_set
    (links_shader_code_path,
     "vs_main_2_sv(); ",     // Name of vertex shader main routine.
     "gs_main_2_sv();",      // Name of geometry shader main routine.
     "fs_main_sv();"         // Name of fragment shader main routine.
     )
    .topology_set( vk::PrimitiveTopology::eLineStrip )
    .create();

  ball_setup_4();
  return;
  // For benchmarking.
  opt_pause = true;
  opt_hide_stuff = OH_Links | OH_Platform;
  opt_shadows = false;
  opt_mirror = false;
}

Ball*
World::make_marker(pCoor position, pColor color)
{
  Ball* const ball = new Ball;
  ball->position = position;
  ball->locked = true;
  ball->velocity = pVect(0,0,0);
  ball->radius = 0.2;
  ball->mass = 0;
  ball->contact = false;
  ball->color = color;
  return ball;
}

void
World::lock_update()
```

```
{
  // This routine called when options like opt_head_lock might have
  // changed.

  // Update locked status.
  //
  if ( head_ball ) head_ball->locked = opt_head_lock;
  if ( tail_ball ) tail_ball->locked = opt_tail_lock;

  // Re-compute minimum mass needed for stability.
  //
  for ( Ball *ball: balls ) ball->spring_constant_sum = 0;
  const double dtis = pow( opt_time_step_duration, 2 );
  for ( Link *link: links )
    {
      Ball* const b1 = link->ball1;
      Ball* const b2 = link->ball2;
      b1->spring_constant_sum += opt_spring_constant;
      b2->spring_constant_sum += opt_spring_constant;
    }
  for ( Ball *ball: balls )
    {
      ball->mass_min = ball->spring_constant_sum * dtis;
      ball->constants_update();
    }
  gpu_const_data_stale = true;
}

void
World::balls_twirl()
{
  if ( !head_ball || !tail_ball ) return;

  pNorm axis(head_ball->position, tail_ball->position);
  const float omega = 0.1 * opt_omega;

  for ( Ball *ball: balls )
    {
      pVect b_to_top(ball->position,head_ball->position);
      const float dist_along_axis = dot(b_to_top,axis);
      const float lsq = b_to_top.mag_sq() - dist_along_axis * dist_along_axis;
      ball->omega += omega * axis;
      if ( lsq <= 1e-5 ) continue;
      const float dist_to_axis = sqrt(lsq);
      pNorm rot_dir = cross(b_to_top,axis);
      ball->velocity += omega * dist_to_axis * rot_dir;
    }
}

void
World::objects_erase()
{
  balls.erase();
  links.erase();
```

```
  surfaces.clear();

  hw04_stuff.balls_inner.clear();
  hw04_stuff.balls_outer.clear();
  hw04_stuff.ribs.clear();
}

Links
World::link_new
(Ball *ball1, Ball *ball2, float stiffness, float spring_constant_p)
{
  Links links_rv;
  assert( ball1->radius > 0 );
  assert( ball2->radius > 0 );
  const float spring_constant = spring_constant_p ?: opt_spring_constant;

  Link* const rlink = new Link(ball1,ball2);
  links_rv += rlink;

  pNorm n12(ball1->position,ball2->position);
  const float rad_sum = ball1->radius + ball2->radius;
  pMatrix3x3 b1rot = ball1->omatrix;
  pMatrix3x3 b2rot = ball2->omatrix;
  pCoor ctr = ball1->position
    + ( ball1->radius + 0.5 * ( n12.magnitude - rad_sum ) ) * n12;

  pVect con_x = rlink->con_x;
  pVect con_y = rlink->con_y;

  const float lrad = stiffness * ball1->radius;

  const int n_springs = 4;
  const double delta_theta = 2 * M_PI / n_springs;

  for ( int i=0; i<n_springs; i++ )
    {
      const double theta = i * delta_theta;
      pVect convec = lrad * ( cos(theta) * con_x + sin(theta) * con_y );
      pCoor con = ctr + convec;
      pVect cb1 = mult_MTV( b1rot, con - ball1->position );
      pVect cb2 = mult_MTV( b2rot, con - ball2->position );
      links_rv +=
        new Link(rlink, cb1, cb2,
                 { .distance_relaxed=0, .spring_constant=spring_constant } );
    }
  return links_rv;
}

///
/// CUDA Code
///

float4 tof4(pCoor c)
{
```

```
  float4 f4;
  f4.x = c.x;
  f4.y = c.y;
  f4.z = c.z;
  f4.w = c.w;
  return f4;
}

float4 tof4(pVect c)
{
  float4 f4;
  f4.x = c.x;
  f4.y = c.y;
  f4.z = c.z;
  f4.w = 0;
  return f4;
}

pCoor topc(float4 c)
{
  pCoor f4;
  f4.x = c.x;
  f4.y = c.y;
  f4.z = c.z;
  f4.w = c.w;
  return f4;
}

void
World::data_cpu_to_gpu_constants()
{
  c.platform_xmin = platform_xmin;
  c.platform_xmax = platform_xmax;
  c.platform_zmax = platform_zmax;
  c.platform_zmin = platform_zmin;

  c.n_balls = balls.size();
  c.n_links = links.size();
  c.n_surfaces = surfaces.size();
  c.opt_tryout1 = opt_tryout1;
  c.opt_tryout2 = opt_tryout2;
  c.opt_tryoutf = opt_tryoutf;
  c.opt_spring_constant = opt_spring_constant;
  c.opt_spring_damp = opt_spring_damp;
  c.opt_air_resistance = opt_air_resistance;
  c.opt_friction_dynamic = opt_friction_dynamic;
  c.opt_head_lock = opt_head_lock;
  c.opt_tail_lock = opt_tail_lock;
  c.gravity_accel = gravity_accel;

  data_cpu_to_gpu_common(&c);
}
```

```
void
World::data_cpu_to_gpu_dynamic()
{
  const int n_balls = balls.size();
  const int n_links = links.size();
  const int n_surfaces = surfaces.size();
  const bool need_alloc =
    n_balls > c.alloc_n_balls || n_links > c.alloc_n_links
    || n_surfaces > c.alloc_n_surfaces;
  const bool need_init = c.alloc_n_balls == 0;

#define ALLOC_MOVE_ZERO(n_balls,balls,memb,do_move,do_zero)                \
  {                                                                        \
    const int size_elt_bytes = sizeof(c.balls.memb[0]);                    \
    const int size_bytes = n_balls * size_elt_bytes;                       \
    if ( need_alloc )                                                      \
      {                                                                    \
        if ( !need_init )                                                  \
          {                                                                \
            CE( cudaFree( c.balls.memb ) );                                \
            free(c.h_##balls.memb);                                        \
          }                                                                \
        CE( cudaMalloc( &c.balls.memb, size_bytes ) );                     \
        c.h_##balls.memb = (decltype(c.balls.memb)) malloc( size_bytes );  \
      }                                                                    \
    if ( do_move )                                                         \
      {                                                                    \
        for ( int i=0; i<n_balls; i++ )                                    \
          c.h_##balls.memb[i] = balls[i]->memb;                            \
        CE( cudaMemcpy                                                     \
            ( c.balls.memb, c.h_##balls.memb,                              \
              size_bytes, cudaMemcpyDefault ) );                           \
      }                                                                    \
    else if ( do_zero )                                                    \
      {                                                                    \
        CE( cudaMemset( c.balls.memb, 0, size_bytes ) );                   \
      }                                                                    \
  }

#define MOVEc(num,ptr,memb) \
  ALLOC_MOVE_ZERO(num,ptr,memb,gpu_const_data_stale,0)
#define MOVE(num,ptr,memb) ALLOC_MOVE_ZERO(num,ptr,memb,1,0)
#define ALLOC(num,ptr,memb) ALLOC_MOVE_ZERO(num,ptr,memb,0,0)
#define ZERO(num,ptr,memb) ALLOC_MOVE_ZERO(num,ptr,memb,0,1)
  MOVEc(n_balls,balls,position);
  MOVEc(n_balls,balls,velocity);
  ZERO(n_balls,balls,force);

  MOVEc(n_balls,balls,orientation);
  MOVEc(n_balls,balls,omatrix);
  MOVEc(n_balls,balls,omega);
  ZERO(n_balls,balls,torque);
  MOVEc(n_balls,balls,m_r_fdt);
  MOVEc(n_balls,balls,locked);
```

```
  MOVEc(n_links,links,ball_idx);
  MOVEc(n_links,links,cb1);
  MOVEc(n_links,links,cb2);
  MOVEc(n_links,links,dr_sc);
  MOVEc(n_links,links,is_simulatable);
  MOVEc(n_links,links,is_surface_connection);

  MOVEc(n_surfaces,surfaces,b_idx);

#undef MOVE
#undef MOVEc
#undef ALLOC
#undef ZERO
#undef ALLOC_MOVE
  if ( need_alloc )
    {
      c.alloc_n_balls = n_balls;
      c.alloc_n_links = n_links;
      c.alloc_n_surfaces = n_surfaces;
      data_cpu_to_gpu_common(&c);
    }
  gpu_const_data_stale = false;
}

void
World::data_gpu_to_cpu_dynamic()
{
  const int n_balls = balls.size();

#define MOVE(n_balls,balls,memb)                                          \
  {                                                                       \
    const int size_elt_fr_bytes = sizeof(c.balls.memb[0]);                \
    const int size_bytes = n_balls * size_elt_fr_bytes;                   \
    CE( cudaMemcpy                                                        \
        ( c.h_##balls.memb, c.balls.memb, size_bytes,                     \
          cudaMemcpyDeviceToHost ) );                                     \
    for ( int i=0; i<n_balls; i++ )                                       \
      balls[i]->memb = c.h_##balls.memb[i];                               \
  }

  MOVE(n_balls,balls,position);
  MOVE(n_balls,balls,velocity);
  MOVE(n_balls,balls,orientation);
  MOVE(n_balls,balls,omatrix);
  MOVE(n_balls,balls,omega);

#undef MOVE

}


///
/// Physical Simulation Code
```

```
///

 /// Initialize Simulation
//

void
World::ball_setup_1(bool big)
{
  // Arrange balls to form sort of tree.

  last_setup = 1;

  double ball_rad = 1;  // Warning: Re-assigned, changes behavior of nb.

  pCoor first_pos(17.2,ball_rad,-20.2);

  pVect dir_up(0.1,3,0.1);
  pVect dir_z(3*distance_relaxed,0,0);
  pVect dir_x(0,0,3*distance_relaxed);

  // Remove objects from the simulated objects lists, balls and links.
  // The delete operator is used on objects in the lists.
  //
  objects_erase();

  auto nb =
    [&] (pCoor pos) {
      Ball* const ball = new Ball;
      ball->position = pos;
      ball->locked = false;
      ball->velocity = pVect(0,0,0);
      ball->radius = ball_rad;
      ball->contact = false;
      balls += ball;
      return ball;
      };

  tail_ball = nb(first_pos);
  tail_ball->specularity = 0.5;

  pCoor last_pos = tail_ball->position;

  const int n_tiers = big ? 32 : 8;
  const double shrink_final = 0.17;
  const double shrink = pow(shrink_final,1.0/n_tiers);

  Balls tail_balls;
  for ( int i=1; i<n_tiers; i++ )
    {
      ball_rad *= shrink;
      Ball* const ball = nb(last_pos + (i==1?1.0:ball_rad) * dir_up);
      ball->specularity = 0.5;
      tail_balls += ball;
      links += link_new(balls[balls-2],ball,0.8);
```

```
      last_pos = ball->position;
    }

  head_ball = balls[balls-1];

  pNorm dir_zn(dir_z);
  pNorm dir_xn(dir_x);
  for ( Ball *ball: tail_balls )
    {
      if ( ball == head_ball ) break;
      pNorm to_top(ball->position,head_ball->position);
      const float r = to_top.magnitude;
      int num_hairs = max(4,int(7*r));
      ball_rad = ball->radius / 4;
      const float delta_theta = 2 * M_PI / num_hairs;
      for ( int i=0; i<num_hairs; i++ )
        {
          const float theta = i * delta_theta;
          pCoor pos = ball->position +
            r * cos(theta) * dir_xn + r * sin(theta) * dir_zn;
          Ball* const ballr = nb(pos);
          links += link_new(ballr,ball);
        }
    }

  opt_head_lock = false;
  opt_tail_lock = true;
  setup_at_end();
}

void
World::ball_setup_2()
{
  // Arrange balls to form a suspended thing.

  last_setup = 2;

  pCoor first_pos(17.2,14.8f,-20.2);

  pVect dir_dn(0.5,first_pos.y/8,0.5);
  pVect dir_z(3*distance_relaxed,0,0);
  pVect dir_x(0,0,3*distance_relaxed);

  // Remove objects from the simulated objects lists, balls and links.
  // The delete operator is used on objects in the lists.
  //
  objects_erase();

  float ball_rad = 0.3;  // Warning: Re-assigned, changes behavior of nb.

  auto nb =
    [&] (pCoor pos) {
      Ball* const ball = new Ball;
      ball->position = pos;
```

```
    ball->locked = false;
    ball->velocity = pVect(0,0,0);
    ball->radius = ball_rad;
    ball->contact = false;
    ball->specularity = 0.5;
    balls += ball;
    return ball;
    };

head_ball = nb(first_pos);
for ( int j: {-1,1} )
  for ( int i: {-1,0,1} )
    {
      Ball* const ball = nb(first_pos - dir_dn + i * dir_x + j * dir_z );
      if ( i != 0 ) links += link_new(head_ball,ball);
      if ( i != -1 ) links += link_new(balls[balls-2],ball);
      if ( j == 1 && i != 0 )  links += link_new(balls[balls-4],ball);
    }

Ball* const tail_start = nb(first_pos - dir_dn);
links += link_new(balls[balls-6],tail_start);
links += link_new(balls[balls-3],tail_start);
pCoor last_pos = tail_start->position;
Balls tail_balls;
for ( int i=1; i<5; i++ )
  {
    Ball* const ball = nb(last_pos - 0.5 * dir_dn);
    tail_balls += ball;
    links += link_new(balls[balls-2],ball);
    last_pos = ball->position;
  }

tail_ball = balls[balls-1];

pNorm dir_zn(dir_z);
pNorm dir_xn(dir_x);
int num_hairs = 20;
ball_rad = 0.1;
for ( Ball *ball: tail_balls )
  {
    const float delta_theta = 2 * M_PI / num_hairs;
    for ( int i=0; i<num_hairs; i++ )
      {
        const float theta = i * delta_theta;
        const float r = ball->radius * 10;
        pCoor pos = ball->position +
          r * cos(theta) * dir_xn + r * sin(theta) * dir_zn;
        Ball* const ballr = nb(pos);
        links += link_new(ballr,ball);
      }
  }

opt_head_lock = true;    // Head ball will be frozen in space.
opt_tail_lock = false;   // Tail ball can move freely.
```

```
  setup_at_end();
}

void
World::ball_setup_3()
{
  // Arrange links to form what I would eventually learn to be a
  // corona virus.

  const bool sarcov = true;

  last_setup = 3;
  objects_erase();

  pCoor first_pos(17.2,14.8f,-10.2);

  float ball_radius = 1;

  auto nb =
    [&] (pCoor pos) {
      Ball* const ball = new Ball;
      ball->position = pos;
      ball->locked = false;
      ball->velocity = pVect(0,0,0);
      ball->radius = ball_radius;
      ball->contact = false;
      balls += ball;
      return ball;
      };

  head_ball = nb(first_pos);
  head_ball->specularity = 0.5;
  ball_radius = ball_radius * ( sarcov ? 0.08 : .1 );
  const int n_spikes = sarcov ? 100 : 500;

  for ( int i=0; i<n_spikes; i++ )
    {
      pNorm dir(0.5-drand48(),0.5-drand48(),0.5-drand48());
      const float len = head_ball->radius * ( sarcov ? 1.5 : 3.0 );
      Ball* const ball = nb(first_pos + len * dir);
      links += link_new(ball,head_ball,2);
    }

  tail_ball = balls[1];

  opt_head_lock = false;    // Head ball will be frozen in space.
  opt_tail_lock = false;   // Tail ball can move freely.

  setup_at_end();
}

pVect vec_rand() {return pVect(drand48(),drand48(),drand48());}

void
```

```
World::ball_setup_4()
{
  // For Homework 4
  // Render a wheel (disk of balls with one in the center).

  last_setup = 4;
  objects_erase();

  pCoor center_pos = pCoor(17.2,14.8f,-10.2) + 2*vec_rand();

  const int n_balls = 4 * ( 4 + random()%4 );
  float ring_outer_radius = 5 + 2 * drand48();
  float ring_inner_radius = ring_outer_radius * ( 0.1 + drand48() * 0.4 );
  float ball_radius = 0.2 * 2 * M_PI * ring_outer_radius / n_balls;

  pNorm c_to_eye(center_pos,eye_location);
  pNorm az = c_to_eye + 0.5 * vec_rand() + pVect(0,0.5,0);

  pNorm ax = fabs(az.x) < fabs(az.y)
    ? pVect(0,az.z,-az.y) : pVect(az.z,0,-az.x);
  pVect ay(az,ax);

  // Default settings for balls.
  //
  Ball ballo_default;
  ballo_default.specularity = 0.5;
  ballo_default.velocity = pVect(0,0,0);
  ballo_default.radius = ball_radius;
  ballo_default.color = color_light_slate_gray;
  Ball balli_default = ballo_default;
  balli_default.radius = ballo_default.radius * 0.6;

  const float d_theta = 2 * M_PI / n_balls;

  const float link_stiffness = 10;

  auto& balls_outer = hw04_stuff.balls_outer;
  auto& balls_inner = hw04_stuff.balls_inner;

  // Place balls in a circle.
  //
  for ( int i=0; i<n_balls; i++ )
    {
      const float theta = d_theta * i;
      pVect pos_n = cosf(theta) * ax + sinf(theta) * ay;
      pCoor pos_o = center_pos + ring_outer_radius * pos_n;
      pCoor pos_i = center_pos + ring_inner_radius * pos_n;

      // Construct new ball and add it to the ball list.
      //
      Ball* ball_o = balls_outer.emplace_back( new Ball(pos_o,ballo_default) );
      Ball* ball_i = balls_inner.emplace_back( new Ball(pos_i,balli_default) );
      Link *link =
        new Link( ball_o, ball_i,
```

```
            { .spring_constant = opt_spring_constant/200 } );
links.push_back(link);
hw04_stuff.ribs.push_back(link);

// Homework 4 -- Change these values for experimentation, if you like.
ball_o->tc.orig = pTCoor(i,0);
ball_i->tc.orig = pTCoor(i,1);

/// Homework 4 -- Problem 3 solution here.
//
//  Modify the assignments to the tc.square and tc.circle members.
//
/// SOLUTION -- Problem 3a
//
//  To find the texture coordinates imagine that a circle, the
//  wheel, were drawn over the texture image. The texture
//  coordinates range from (0,0) to (1,1) with (0.5,0.5) in the
//  center of the texture image. So to find the texture
//  coordinates for the inner and outer balls of the wheel we
//  need to determine where on the circle those balls are.
//
//  At each iteration an angle, theta, is computed and used to
//  find the ball coordinate. That angle can also be used to
//  find the texture coordinates. First, consider the outer
//  ring. The outer ring texture coordinates should form a
//  circle with the center at x=0.5, y=0.5 (the center of
//  texture space), and with a radius of 0.5. With that center
//  and radius texture coordinates x and y components will range
//  from 0 to 1.

//  First, compute a vector of length 0.5 from the circle enter
//  to the circle:
//
vec2 tex_sq_norm = 0.5 * vec2( cosf(theta), -sinf(theta) );

//  Prepare a coordinate of the circle center.
//
vec2 tex_ctr(0.5,0.5);

//  Add the vector to the circle center to get the outer-ring texture
//  coordinate.
//
ball_o->tc.square = tex_ctr + tex_sq_norm;

//   For the inner ring texture coordinate we need to shorten
//   vector tex_sq_norm by the ratio of the inner to outer
//   radii.
//
float ring_io = ring_inner_radius / ring_outer_radius;
//
ball_i->tc.square = tex_ctr + ring_io * tex_sq_norm;

/// SOLUTION -- Problem 3b
//
```

```
        //  For 3b the texture should wrap around the ring. To achieve
        //  this effect the x component of the texture coordinate should
        //  be proportional to the distance of the ball from the wheel
        //  center. The effect will be achieved if the texture x
        //  component were 1 for the inner ball and 0 for the inner
        //  ball. To get the wrapping effect the y component should be
        //  proportional to the angle, theta. Call the proportionality
        //  factor omega. Then we can naively construct texture
        //  coordinates:
        //
#if 0
        const float delta_r = ring_outer_radius - ring_inner_radius;
        const float r_mid = ( ring_outer_radius + ring_inner_radius ) / 2;
        const float omega  = r_mid / delta_r;

        ball_o->tc.circle = vec2( 1, theta * omega );
        ball_i->tc.circle = vec2( 0, theta * omega );
#endif
        //
        //  These texture coordinates are in fact correct for the two
        //  balls' coordinates. They won't be however for fragments
        //  inside the triangles. (Only the fragments at the triangle
        //  vertices will be correct.) Recall that each vertex of a
        //  triangle has a texture coordinate, and that for a fragment
        //  the three vertices texture coordinates are interpolated
        //  (using perspective-correct barycentric coordinates). To see
        //  that consider coordinates assigned to two adjacent
        //  outer-ring balls. The x component of both are 1, which is
        //  correct. Now consider a fragment halfway between the two
        //  balls. The interpolated x texture coordinate component will
        //  still be 1. But that's not correct because the balls are on
        //  a circle and the halfway point between the two balls is
        //  *inside* the circle and so the x component of the texture,
        //  based on the radius, must also be less than one.
        //
        //  The solution to this dilemma is actually quite simple.
        //  Choose coordinates similar to as we have for the square,
        //  except now do so with a center of (0,0) an with the radius
        //  of the outer ring set to 1 and the inner ring set to the
        //  inner/outer ratio:
        //
        vec2 loc_n = vec2( cosf(theta), -sinf(theta) );
        ball_o->tc.circle = loc_n;
        ball_i->tc.circle = ring_io * loc_n;
        //
        //  In the fragment shader we will use these coordinates to
        //  compute r and theta, and from that get texture coordinates.
        //
        //  To get theta take the arctangent of the x and y components.
        //  See the shader code for details.
        //
        const float delta_r = ring_outer_radius - ring_inner_radius;
        const float r_mid = ( ring_outer_radius + ring_inner_radius ) / 2;
        uni_hw04->r_scale_factor = 1 / ( 1 - ring_io );
```

```
      uni_hw04->omega = r_mid / delta_r;
      uni_hw04_needs_update = true;

      // Set color of first ball to blue, others to white.
      ball_i->color = ball_o->color =
        i == 0 ? color_blue :
        theta <= M_PI/2 ? color_lsu_spirit_gold :
        color_white;
    }

  // Connect each ball to its neighbor with a link.
  //
  for ( auto bc: { &balls_outer, &balls_inner } )
    for ( int i=0; i<n_balls; i++ )
      {
        balls.push_back( bc[0][i] );
        links += link_new( bc[0][i], bc[0][(i+1)%n_balls], link_stiffness );
      }

  for ( int i=0; i<n_balls; i++ )
    {
      const int j = ( i + 1 ) % n_balls;
      Ball* b00 = balls_inner[i];
      Ball* b01 = balls_outer[i];
      Ball* b10 = balls_inner[j];
      Ball* b11 = balls_outer[j];
      surfaces.emplace_back(b00,b01,b11);
      surfaces.emplace_back(b10,b00,b11);
    }

  pNorm spin_axis = ay;
  pCoor c_of_mass = center_pos;
  float omega_init = opt_omega * 0.1;

  if ( 0 )
  for ( Ball* ball: balls )
    {
      pVect cm_to_ball( c_of_mass, ball->position );

      ball->velocity = omega_init * cross( spin_axis, cm_to_ball );
      ball->omega = omega_init * spin_axis;
    }

  tail_ball = balls.back();
  head_ball = balls.front();

  opt_head_lock = false; // If true, head ball locked in place.
  opt_tail_lock = false;

  setup_at_end();
}

void
World::ball_setup_5()
```

```
{
  // Construct a top. (Top, as in a child's toy.)

  last_setup = 5;
  objects_erase();

  pCoor center_pos(17.2,14.8f,-10.2);

  const int n_balls = 20;
  const float crad = 5;

  const float ball_radius = 0.3 * 2 * M_PI * crad / n_balls;

  // Default settings for balls.
  //
  Ball ball_default;
  ball_default.specularity = 0.5;
  ball_default.velocity = pVect(0,0,0);
  ball_default.radius = ball_radius;
  ball_default.color = pColor(0xffbbff); // Plum 1. That's the color's name.

  const float d_theta = 2 * M_PI / n_balls;

  pVect vz = pVect(1,6,1) - 2*vec_rand();
  pNorm ax( vz.y, -vz.x, 0 );
  pNorm ay = cross(vz,ax);
  pVect vx = crad * ax;
  pVect vy = crad * ay;

  pNorm spin_axis = vz;
  float omega = opt_omega;
  opt_spring_damp = .4;

  // Construct Top's Main Ring
  //
  for ( int i=0; i<n_balls; i++ )
    {
      const float theta = d_theta * i;
      pCoor pos = center_pos + vx * cosf(theta) + vy * sinf(theta);
      Ball* ball = balls.emplace_back( new Ball(pos,ball_default) );

      // Find the coordinate on the spin axis closest to the new ball.
      //
      pCoor axis_cp =
        center_pos + spin_axis * dot( spin_axis, pVect(center_pos,pos) );
      //
      // Use this coordinate to compute a vector giving the ball's
      // motion when the top is spinning around axis spin_axis.
      //
      pVect cp_to_p(axis_cp,pos);
      ball->velocity = omega * cross(spin_axis,cp_to_p);
    }

  // Make one of the balls on the top's main ring a different color.
```

```
  balls[0]->color = color_blue;

  // Connect the balls on the main ring with links.
  //
  for ( int i=0; i<n_balls; i++ )
    links += link_new( balls[i], balls[(i+1)%n_balls], 2 );

  // Complete the top with balls above and below the ring, and one in
  // the center.
  //
  pNorm az(vz);
  pCoor top_pos = center_pos + az * crad * 1.5;
  pCoor bot_pos = center_pos - az * crad * 1.5;

  head_ball = balls.emplace_back( new Ball(top_pos,ball_default) );
  tail_ball = balls.emplace_back( new Ball(bot_pos,ball_default) );
  Ball* ctr_ball = balls.emplace_back( new Ball(center_pos,ball_default) );

  for ( int i=0; i<n_balls; i += 2 )
    {
      links += link_new( head_ball, balls[i], 2 );
      links += link_new( tail_ball, balls[i], 2 );
    }

  for ( int i=0; i<n_balls; i++ )
    links += link_new( ctr_ball, balls[i], 2 );

  for ( auto b: balls ) b->omega = omega * spin_axis;

  opt_head_lock = false;    // Head ball will be frozen in space.
  opt_tail_lock = false;   // Tail ball can move freely.

  setup_at_end();
}

void
World::ball_setup_6()
{
  last_setup = 6;
}


void
World::setup_at_end()
{
  lock_update();
  time_step_count = 0;
  world_time = 0;
  world_time_link_update = -1;
  for ( uint i=0; i<balls.size(); i++ ) balls[i]->idx = i;
  for ( Link *l : links )
    { l->ball1_idx = l->ball1->idx; l->ball2_idx = l->ball2->idx; }
  for ( auto& s: surfaces )
    { s.b0_idx = s.b0->idx;  s.b1_idx = s.b1->idx;  s.b2_idx = s.b2->idx; }
```

```
}


 /// Advance Simulation State by delta_t Seconds
//
void
World::time_step_cpu(double delta_t)
{
  const int n_balls = balls;
  const int n_links = links;

  // Smoothly move ball in response to user input.
  //
  if ( adj_t_stop )
    {
      const double dt = min(world_time,adj_t_stop) - adj_t_prev;
      pVect adj = dt/adj_duration * adj_vector;
      balls_translate(adj,0);
      adj_t_prev = world_time;
      if ( world_time >= adj_t_stop ) adj_t_stop = 0;
    }

#pragma omp parallel for
  for ( int bi=0; bi<n_balls; bi++ )
    {
      Ball* const ball = balls[bi];
      assert( ball->fdt_to_do );
      ball->force = ball->mass * gravity_accel;
      ball->torque = pVect(0,0,0);
    }

  const int n_surfaces = surfaces.size();
  for ( int i=0; i<n_surfaces; i++ )
    {
      Surface& s = surfaces[i];
      Ball& b0 = *s.b0;
      Ball& b1 = *s.b1;
      Ball& b2 = *s.b2;
      pNorm snorm = cross(b0.position,b1.position,b2.position);
      pVect vel = (1.0/3) * ( b0.velocity + b1.velocity + b2.velocity );
      float speed = dot(snorm,vel);
      float fmag =
        -speed * fabs(speed) * snorm.magnitude * 0.5f * opt_air_resistance;
      pVect f = fmag * snorm;
      b0.force += f;
      b1.force += f;
      b2.force += f;
    }

#pragma omp parallel for
  for ( int i=0; i<n_links; i++ )
    {
      Link* const link = links[i];
      if ( !link->is_simulatable ) continue;
```

```
//
// Spring Force from Neighbor Balls
//

Ball* const ball1 = link->ball1;
Ball* const ball2 = link->ball2;

/// Notes:
//
// link->cb1
//   Coordinate where link touches ball1, in ball1's local space.
//
// ball1->omatrix
//   Ball's orientation transformation ..
//   .. which transforms vectors in ball1's local space to object space.
//

// Find position and velocity of the point where the link touches
// the surface of ball 1 ...
//
pVect dir1 = ball1->omatrix * link->cb1;
pCoor pos1 = ball1->position + dir1;
pVect vel1 = ball1->velocity + ball1->point_rot_vel(dir1);

// ... and ball 2.
//
pVect dir2 = ball2->omatrix * link->cb2;
pCoor pos2 = ball2->position + dir2;
pVect vel2 = ball2->velocity + ball2->point_rot_vel(dir2);

// Construct a normalized (Unit) Vector from ball to neighbor
// based on link connection points and ball centers.
//
pNorm link_dir(pos1,pos2);
pNorm c_to_c(ball1->position,ball2->position);

const float link_length = link_dir.magnitude;

// Compute the speed of ball's end of link towards neighbor's end of link.
//
pVect delta_v = vel2 - vel1;
float delta_s = dot( delta_v, link_dir );

// Compute by how much the spring is stretched (positive value)
// or compressed (negative value).
//
const float spring_stretch = link_length - link->distance_relaxed;

// Determine whether spring is gaining energy (whether its length
// is getting further from its relaxed length).
//
const bool gaining_e = ( delta_s > 0.0 ) == ( spring_stretch > 0 );
```

```
      const float blend_length_factor = 0.1; // Just a guess.
      const float blend_s = blend_length_factor * c_to_c.magnitude;
      const float blend = min( 1.0f, fabs(delta_s) / blend_s );

      // Use a smaller spring constant when spring is loosing energy,
      // a quick and dirty way of simulating energy loss due to spring
      // friction.
      //
      const float sc_base = link->spring_constant ?: opt_spring_constant;
      const float spring_constant =
        gaining_e ? sc_base :
        sc_base * ( 1.0f - blend + blend * opt_spring_damp );

      const float force_mag = spring_constant * spring_stretch;
      pVect spring_force_12 = force_mag * link_dir;

      // Apply forces affecting linear momentum.
      //
      link->spring_force_12 = spring_force_12;

      if ( ! link->is_surface_connection ) continue;

      // Apply torque.
      //
      link->torque1 = cross(pNorm(dir1), spring_force_12);
      link->torque2 = cross(pNorm(dir2), spring_force_12);
    }

  // Note: Because two links can reference the same ball this should
  // not be done in parallel.
  for ( Link *link: links )
    {
      if ( !link->is_simulatable ) continue;
      Ball* const ball1 = link->ball1;
      Ball* const ball2 = link->ball2;
      ball1->force += link->spring_force_12;
      ball2->force -= link->spring_force_12;
      ball1->torque += link->torque1;
      ball2->torque -= link->torque2;
    }

  ///
  /// Update Position of Each Ball
  ///

#pragma omp parallel for
  for ( int bi=0; bi<n_balls; bi++ )
    {
      Ball* const ball = balls[bi];
      if ( ball->locked )
        {
          ball->velocity = pVect(0,0,0);
          ball->omega = pVect(0,0,0);
          continue;
```

```
  }

// Update Velocity
//
// This code assumes that force on ball is constant over time
// step. This is clearly wrong when balls are moving with
// respect to each other because the springs are changing
// length. This inaccuracy will make the simulation unstable
// when spring constant is large for the time step.
//
const float mass = max( ball->mass, ball->mass_min );

pVect delta_v = ( ball->force / mass ) * delta_t;
const float r = ball->radius;

if ( platform_collision_possible(ball->position)
     && fabs(ball->position.y) < r )
  {
    const bool above = ball->position.y >= 0;
    const float spring_constant_plat =
      ball->velocity.y < 0 == above ? 100000 : 50000;
    const float fric_coefficient = opt_friction_dynamic;
    const float force_away = ( above ? 1 : -1 )
      * ( r - fabs(ball->position.y) ) * spring_constant_plat;
    const float delta_v_away = force_away / mass * delta_t;
    const float fric_force_mag = fric_coefficient * force_away;
    pNorm surface_v(ball->velocity.x,0,ball->velocity.z);
    const float delta_v_surf = fric_force_mag / mass * delta_t;

    if ( delta_v_surf > surface_v.magnitude )
      {
        // Ignoring other forces?
        delta_v =
          pVect(-ball->velocity.x,delta_v.y,-ball->velocity.z);
      }
    else
      {
        delta_v -= delta_v_surf * surface_v;
      }
    delta_v.y += delta_v_away;
  }

ball->velocity += delta_v;

// Air Resistance
//
const pNorm ball_dir(ball->velocity);
const float delta_s_air_raw =
  opt_air_resistance * ball_dir.mag_sq * delta_t;
const float delta_s_air = min( ball_dir.magnitude, delta_s_air_raw );
ball->velocity -= delta_s_air * ball_dir;

// Update Position
//
```

```
      // Assume that velocity is constant.
      //
      ball->position += ball->velocity * delta_t;

      ball->omega += delta_t * ball->fdt_to_do * ball->torque;

      pNorm axis(ball->omega);

      // Update Orientation
      //
      // If ball isn't spinning fast skip expensive rotation.
      //
      if ( axis.mag_sq < 0.000001 ) continue;

      // Update ball orientation.
      //
      ball->orientation_set
        ( pQuat(axis,delta_t * axis.magnitude) * ball->orientation );
    }
}

bool
World::platform_collision_possible(pCoor pos)
{
  // Assuming no motion in x or z axes.
  //
  return pos.x >= platform_xmin && pos.x <= platform_xmax
    && pos.z >= platform_zmin && pos.z <= platform_zmax;
}

 /// External Modifications to State
//
//   These allow the user to play with state while simulation
//   running.

// Move the ball.
//
void Ball::translate(pVect amt) {position += amt;}

// Add velocity to the ball.
//
void Ball::push(pVect amt) {velocity += amt;}

// Set the velocity to zero.
//
void Ball::stop() {velocity = pVect(0,0,0); }

// Set the velocity and rotation (not yet supported) to zero.
//
void Ball::freeze() {velocity = pVect(0,0,0); }



void World::balls_translate(pVect amt,int b){head_ball->translate(amt);}
```

```
void World::balls_push(pVect amt,int b){head_ball->push(amt);}
void World::balls_translate(pVect amt)
{ for ( Ball *ball: balls ) ball->translate(amt); }
void World::balls_push(pVect amt)
{ for ( Ball *ball: balls ) ball->push(amt); }
void World::balls_stop()
{ for ( Ball *ball: balls ) ball->stop(); }
void World::balls_freeze(){balls_stop();}


void
World::render_link_start()
{
  if ( world_time_link_update == world_time ) return;

  // Empty the containers in preparation for a new pass.
  for ( auto b: lis_bufs ) b->clear();
}

void
World::render_link_gather(Link *link)
{
  // Place data needed to render link (the argument to this function)
  // in arrays. Just before rendering those arrays will be placed in
  // buffer objects.


  // If we've already collected links, just return.  This happens
  // when doing multiple passes to render shadows and reflections.
  //
  if ( world_time_link_update == world_time ) return;

  Ball *const ball1 = link->ball1;
  Ball *const ball2 = link->ball2;

  // Direction of link relative to center off ball.
  //
  pVect dir1 = ball1->omatrix * link->cb1;

  // Place on surface of ball where link connects.
  //
  pCoor pos1 = ball1->position + dir1;

  // Direction and connection location for ball 2.
  //
  pVect dir2 = ball2->omatrix * link->cb2;
  pCoor pos2 = ball2->position + dir2;

  pVect p1p2(pos1,pos2);
  pNorm p1p2n(p1p2);

  // Radius of link.
  //
  const float rad = ball1->radius * 0.3;
```

```
  // Compute scale factors for texture.
  const float tex_aspect_ratio = 8.5 / 11;  // Yes, inches. Can't remember mm.
  const float page_width_o = 2.5;  // Width of texture in object-space coords.
  const float tex_t_scale = link->distance_relaxed / page_width_o;
  const float tex_angle_scale = rad * tex_aspect_ratio / page_width_o;

  pNorm dirn1(dir1);
  pNorm dirn2(dir2);

  // Vectors used to describe the cubic Hermite curve.
  //
  pVect4 v1 = p1p2n.magnitude * dirn1;
  pVect4 v2 = p1p2n.magnitude * dirn2;

  // Convert link's local x and y axes to global coordinates.
  //
  pVect4 b1_ydir = ball1->omatrix * link->b1_dir;
  pVect4 b2_ydir = ball2->omatrix * link->b2_dir;


  // Place data needed to render links in lists. Note that
  // the += operator appends the right-hand side to the container
  // on the left-hand side.
  //
  link->idx = lis_pos1.size();
  lis_pos1 << pos1;
  lis_pos2 << pos2;
  lis_v1 << v1;
  lis_v2 << v2;
  lis_b1_ydir << b1_ydir;
  lis_b2_ydir << b2_ydir;
  lis_tex_rad << vec4( tex_t_scale, tex_angle_scale, rad, 0);
}



void
World::render_link_render(vk::CommandBuffer& cb, bool shadow_volumes)
{
  // Perform a rendering pass to render all the links collected
  // by the routine render_link_2.


  // Check whether this is the first time we are rendering this data.
  //
  const bool first_render =
    world_time_link_update != world_time
    || opt_segments != buf_uni_common->links_n_segs
    || opt_sides != buf_uni_common->sides;
  world_time_link_update = world_time;
  const size_t n_instances = lis_pos1.size();
  const bool do_rt = vh.raytrace_active();
  const int segments = shadow_volumes ? max(2,opt_segments/4) : opt_segments;
```

```
  const int segments_sv = max(2,opt_segments/4);

  if ( !n_instances ) return;

  if ( first_render )
    {
      for ( auto b: lis_bufs ) b->to_dev();
      buf_uni_common->sides = opt_sides;
      buf_uni_common->links_n_segs_sv = segments_sv;
      buf_uni_common->delta_tee_sv = 1.0f / segments_sv;
      buf_uni_common->links_n_segs = opt_segments;
      buf_uni_common->delta_tee = 1.0f / opt_segments;
    }

  if ( do_rt && first_render )
    {
      const int n_tri_p_segment = 2 * opt_sides;
      const int n_tri_p_link = n_tri_p_segment * opt_segments;
      const int n_tri_p_pass = n_tri_p_link * n_instances;

      const int n_vtx_p_segment = 2 * ( opt_sides + 1 );
      const int n_vtx_p_link = n_vtx_p_segment * opt_segments;
      const int n_vtx_p_pass = n_vtx_p_link * n_instances;

      if ( !rt_geo_links )
        rt_geo_links
          .init( raytrace )
          .color_set( color_dim_gray, color_salmon )
          .bind( VR_BUF_IDX_sampler, sampler, texid_hw )
          .grouping_set( RT_Geometry_Indexed );

      buf_links_as_idx.assure(3*n_tri_p_pass);
      buf_links_tex_coor.assure(n_vtx_p_pass);
      buf_links_vertex_g.assure(n_vtx_p_pass);
      buf_links_normal_g.assure(n_vtx_p_pass);

      rt_geo_links
        .bind( VR_BUF_IDX_indices, buf_links_as_idx, 3*n_tri_p_pass )
        .bind( VR_BUF_IDX_pos, buf_links_vertex_g, n_vtx_p_pass )
        .bind( VR_BUF_IDX_normal, buf_links_normal_g )
        .bind( VR_BUF_IDX_tcoor, buf_links_tex_coor );
    }

  if ( do_rt )
    {
      QSet& qs = vh.qs;
      buf_uni_common.to_dev();
      qs.cb.begin( { vk::CommandBufferUsageFlagBits::eOneTimeSubmit } );
      comp_links.record_dispatch(qs.cb,opt_segments,n_instances);
      qs.cb.end();
      qs.q.submit( vk::SubmitInfo{ 0, nullptr, nullptr, 1, &qs.cb }, nullptr );
      qs.q.waitIdle();
      return;
    }
```

```
  VPipeline& pipe = shadow_volumes ? pipe_set_2_sv : pipe_set_2;

  pipe.record_draw_instanced(cb, segments+1, n_instances);
}


void
World::frame_callback()
{
  // This routine called whenever window needs to be updated, but
  // before commands are recorded.

  // Get any waiting keyboard commands.
  //
  cb_keyboard();

  if ( vh.have_raytrace ) vh.opt_want_raytrace_now = opt_want_raytrace;

  // This is a hack for 2023 HW 2.
  while ( !balls.empty() && balls.back()->world_time_expire <= world_time )
    balls.pop_back();
  //
  // Balls must be last in list and not part of a link.

  const double time_now = time_wall_fp();
  const int time_step_count_start = time_step_count;
  const int nmp = gpu_info.cuda_prop.multiProcessorCount;
  int bl_p_sm_max = 1024;
  for ( int i=0; i<gpu_info.num_kernels; i++ )
    set_min
      (bl_p_sm_max,
       gpu_info.get_max_active_blocks_per_mp(i, opt_block_size, 0) );
  const int grid_size = bl_p_sm_max * nmp;

  const bool opt_cuda = opt_physics_method != GP_cpu;
  if ( opt_cuda )
    {
      CE( cudaEventRecord(frame_start_ce,0) );
      data_cpu_to_gpu_constants();
      data_cpu_to_gpu_dynamic();
    }

  if ( !opt_pause || opt_single_frame || opt_single_time_step )
    {
      /// Advance simulation state.

      // Amount of time since the user saw the last frame.
      //
      const double wall_delta_t = time_now - last_frame_wall_time;

      // Compute amount by which to advance simulation state for this frame.
      //
```

```
      const double duration =
        vh.video_frame_duration() ?:
        opt_single_time_step ? opt_single_time_step * opt_time_step_duration :
        opt_single_frame ? 1/60.0 :
        wall_delta_t;

      const double world_time_target = world_time + duration;
      const double wall_time_limit = time_now + 0.05;

      while ( world_time < world_time_target )
        {
          if ( opt_cuda )
            {
              launch_time_step(opt_time_step_duration,grid_size,opt_block_size);
            }
          else
            {
              time_step_cpu(opt_time_step_duration);
            }
          time_step_count++;
          world_time += opt_time_step_duration;
          const double time_right_now = time_wall_fp();
          if ( time_right_now > wall_time_limit ) break;
        }

      // Reset these, just in case they were set.
      //
      opt_single_frame = false;
      opt_single_time_step = 0;
    }

  if ( opt_cuda )
    {
      data_gpu_to_cpu_dynamic();
      CE( cudaEventRecord(frame_stop_ce,0) );
    }

  frame_timer.phys_end();
  frame_timer.work_amt_set(time_step_count-time_step_count_start);

  float cuda_time = 0;
  if ( opt_cuda )
    {
      CE(cudaEventSynchronize(frame_stop_ce));
      CE(cudaEventElapsedTime(&cuda_time,frame_start_ce,frame_stop_ce));
    }
  frame_timer.cuda_frame_time_set(cuda_time);

  last_frame_wall_time = time_now;

}

int
main(int argv, char **argc)
```

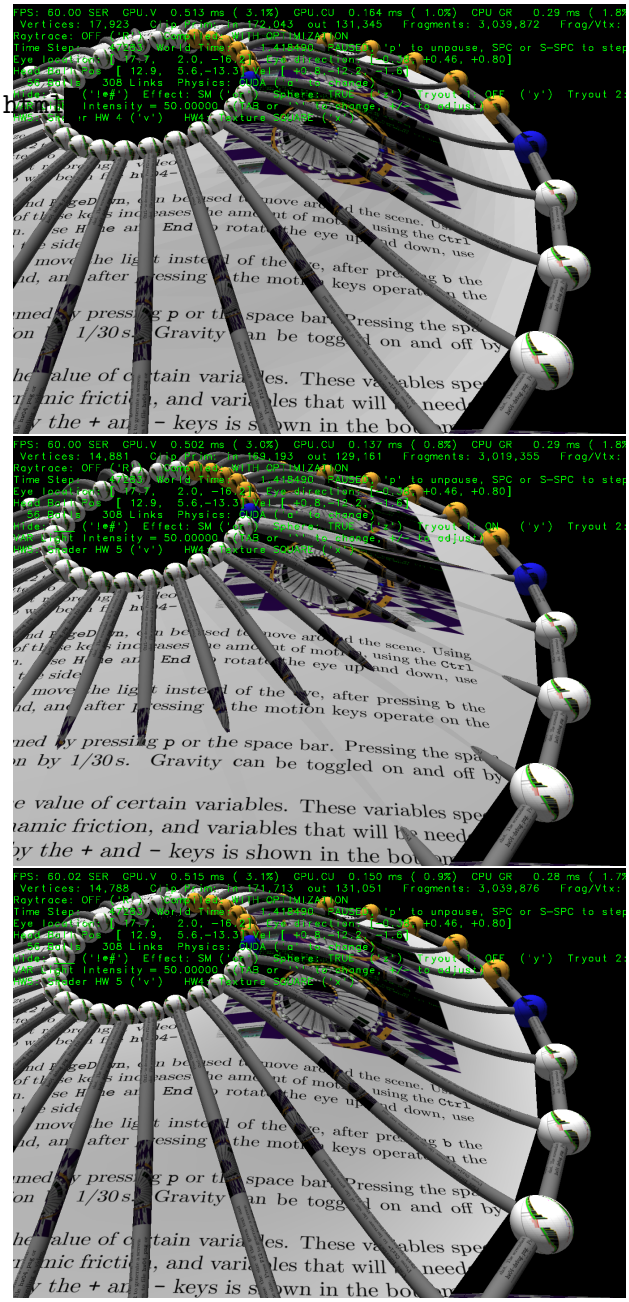
```
{
  pVulkan_Helper pvulkan_helper(argv,argc);
  World world(pvulkan_helper,argv,argc);

  world.run();
}
```

# LSU EE 4702-1 Homework 5 Solution Due: 22 November 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/hw05.cc.html` *(CPU code) and* `https://www.ece.lsu.edu/koppel/gpup/2023/hw05-shdr.cc.html` *(shader code). The solution code is at* `https://www.ece.lsu.edu/koppel/gpup/2023/hw05-sol.cc.html` *(CPU code) and* `https://www.ece.lsu.edu/koppel/gpup/2023/hw05-shdr-sol.cc.html` *(shader code).*

**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 solution to Homework 4.



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 `hw05.png` or `hw05-debug.png`. Press `F10` to start recording a video, and press `F10` to stop it. The video will be in file `hw05-1.ogg` or `hw05-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 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 1 through 5. For this assignment use scene 4 (which is how the code starts). 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 wheel), 5 for a top (a spinning child's toy). Scene 1 is initialized by the code in World::ball_setup_1, etc. Parts of the solution to this assignment should be put in World::ball_setup_4.

The scene for this assignment, 4, initializes randomly each time it is reset. (That is to avoid solutions that accidentally only work for a special case of positioning, etc.)

The code for Scene 4 can be run with two shaders, called HW 4 and HW 5. The shader in use is shown on the bottom line of green text and can be switched by pressing v. The HW 4 shader is based on the solution to Homework 4. The HW 5 shader is to be modified in this assignment. Initially it shows a cruder version of the wheel, in which the surface does closely follow the shape of spokes when the spokes are bent. The code setting up the HW 4 shader is in hw05.cc:World::render_hw04 and the shaders themselves are in hw04-shdr.cc. The code setting up the HW 5 shader is in hw05.cc:World::render_hw05 and the shaders themselves are in hw05-shdr.cc.

The projection of textures on to the surface can be changed by pressing x. There are three possible projections NONE, meaning no textures, SQUARE, which shows one copy of the texture mapped over the entire wheel, and CIRCLE, which shows the texture wrapped around the wheel, perhaps requiring several copies of the texture. Pressing x cycles between these projections. Variable opt_texture is set to the current projection, which can be TX_None, TX_Orig, TX_Square, or TX_Circle. (In hw05.cc those are enumeration constants in hw05-shdr.cc those are integer constants.)

### 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, `hw05`, and a debug-able version of the code, `hw05-debug`. The `hw05-debug` version is compiled with optimization turned off, which makes it easier to debug. When needed, you are strongly encouraged to run `hw05-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`, available in CPU and shader code. 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.

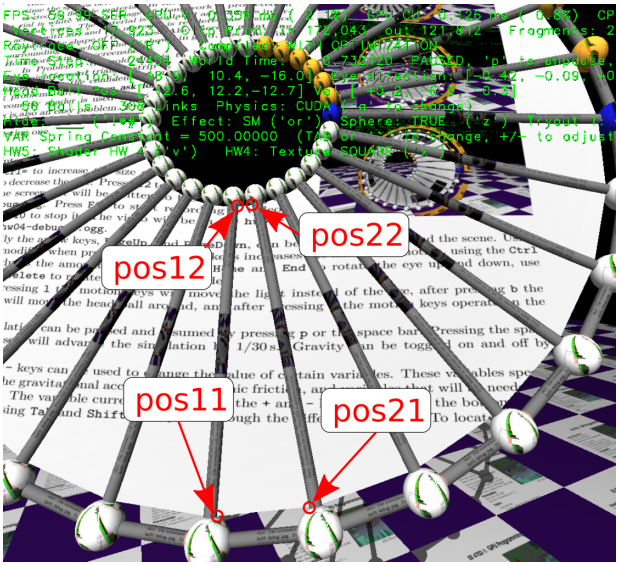
### Resources

A good reference for C++ is `https://en.cppreference.com/w/`. Solutions to the shader programming problems may (will) require the use of library functions. See Chapter 8 of The OpenGL Shading Language Version 4.6 specification. Also see 2014 Homework 4-6 (especially 6), the spiral, to see examples of how the pipeline input topology can be chosen to provide data needed by the geometry shader, even though the topology class (point, line, triangle) does not geometrically correspond to what is being drawn.

*The bottom half of this page intentionally left blank (other than this message).*

## Code Overview

The `render_hw04` routine draws the surface of the wheel using a pipeline set up for a triangle strip. The outer loop in that routine, extracted below, iterates over spokes. In each iteration it uses information from two adjacent spokes, `spoke1` and `spoke2`, to construct the strip. Each spoke has an index, such as `li1`, and that index can be used to get information about the spoke (or link). Array `lis_pos1` holds the coordinate of one spoke endpoint and `lis_pos2` holds the coordinate of the other endpoint. In the code below the two coordinates from each of the two spokes are extracted and placed in variables `pos11`, `pos12`, `pos21`, and `pos22`. The diagram to the right shows the relationship of the coordinates to the spokes and wheel.




```
const size_t n_spokes = hw05_stuff.spokes.size();
for ( size_t i = 0; i < n_spokes; i++ )
  {
    size_t i2 = (i+1) % n_spokes;
    Link *spoke1 = hw05_stuff.spokes[i], *spoke2 = hw05_stuff.spokes[i2];
    const int link_idx1 = spoke1->idx;
    const int link_idx2 = spoke2->idx;
    pCoor pos11 = lis_pos1[link_idx1];
    pCoor pos12 = lis_pos2[link_idx1];
    pCoor pos21 = lis_pos1[link_idx2];
    pCoor pos22 = lis_pos2[link_idx2]; }
```

It would be possible to use these four coordinates to draw two triangles, but those triangles would not match the shape of the spokes, which are cubic curves. An inner loop, shown below, finds points between the endpoints using routine `bez`. Point `pos1t` is between `pos11` and `pos12`, following the curved path defined by the endpoints and vectors `v11` and `v12`. It is points `pos1t` and `pos2t` that are inserted into the buffer set for use in drawing the surface. In addition to the coordinates, texture coordinates are also inserted. The `mix` routine blends the endpoint texture coordinates (not shown in the exceprt) using parameter `t`.

```
    pVect v11 = lis_v1[link_idx1], v12 = lis_v2[link_idx1];
    pVect v21 = lis_v1[link_idx2], v22 = lis_v2[link_idx2];
    const float delta_t = 1.0/opt_segments;
    for ( int s=0; s<=opt_segments; s++ ) {
        const float t = s * delta_t;
        pCoor pos1t = bez(pos11,pos12,v11,v12,t);
        pCoor pos2t = bez(pos21,pos22,v21,v22,t);
        bset_hw04 << pos1t << mix(tex11,tex12,t)
                  << pos2t << mix(tex21,tex22,t);
   }
```

The code in `render_hw05` prepares coordinates for pipeline `pipe_hw05`. The pipe has two inputs, an `int` and a texture coordinate. The pipeline also binds four arrays, `lis_pos1`, `lis_pos2`,

lis_v1, and lis_v2. Theses are the arrays holding link data used above, and they can be accessed in shader code using C array syntax.

The main loop in render_hw05 inserts just two vertices (there is no s loop) in the pipeline for each spoke (and does the first spoke again at the end):

```
  const size_t n_spokes = hw05_stuff.spokes.size();
  for ( size_t i = 0; i <= n_spokes; i++ ) {
      Link* const spoke1 = hw05_stuff.spokes[i%n_spokes];
      const int link_idx = spoke1->idx;
      pTCoor tex11 = spoke1->ball1->tca[opt_texture];
      pTCoor tex12 = spoke1->ball2->tca[opt_texture];

      bset_hw05 << link_idx << tex11;
      bset_hw05 << link_idx << tex12;
    }
```

Each vertex consists of an integer, link_idx, and a texture coordinate. Note that the same index is used twice. The integer is used to index the lis arrays. A placeholder vertex shader uses the index, named in_link_idx in the shader code, to extract a position:

```
void vs_hw05() {  // Vertex Shader in hw05-shdr.cc
  vec4 p1 = pos1[in_link_idx];  // Outer ring.
  vec4 p2 = pos2[in_link_idx];  // Inner ring.
  vec4 p = ( gl_VertexIndex & 1 ) == 0 ? p1 : p2;
  Out.vertex_o = p;
  Out.vertex_e = ut.eye_from_object * p;
  Out.tex_coor = in_tex_coor;
}
```

The code above actually finds two coordinates. It uses p1 for even-numbered vertices, and p2 for odd-numbered vertices. Built-in variable gl_VertexIndex provides the vertex number. (Zero for the first vertex in a draw.)

The pipeline is set to group vertices for a triangle strip topology, so the input to the geometry shader is a triangle. The placeholder geometry shader computes the triangle normal and a clip-space coordinate, and emits the triangle:

```
void gs_hw05() {
  vec3 normal_o =
    cross( In[1].vertex_o.xyz - In[0].vertex_o.xyz,
           In[2].vertex_o.xyz - In[0].vertex_o.xyz );
  vec3 normal_e = mat3( ut.eye_from_object ) * normal_o;

  for ( int i=0; i<3; i++ ) {
      gl_Position = ut.clip_from_object * In[i].vertex_o;
      Out.normal_e = normal_e;
      Out.vertex_e = In[i].vertex_e;
      Out.tex_coor = In[i].tex_coor;
      EmitVertex();
    }
  EndPrimitive();
}
```

To help in solving the assignment the shader file has a bez routine that can be used for finding points along the spoke. It also has a routine, bez_dt, that computes the derivative of the curve

between the two points (with respect to the parametric parameter, `t`). Routine `bez_dt` can be used for finding surface normals.

**Problem 1:** Modify code in `hw05.cc:World::render_05` and the shaders in `hw05-shdr.cc` so that the geometry shader computes the triangles between two spokes. Each invocation of a geometry shader should do the same work as the `s` loop from `render_hw04` described above.
Here is what needs to be done:

*Modify Pipeline Input Topology*
The geometry shader needs information on two adjacent spokes (links) so that it can find the four points. That's not easily possible using a triangle strip topology. Modify the topology used by `pipe_hw05` to better provide this information. This can be done by modifying the argument to `.topology_set` called where the pipeline is set up in `render_hw05`. Consider the approach used in 2014 Homework 6, in which a spiral was constructed using a line list.

*Modify Pipeline Inputs*
In the unmodified assignment `pipe_hw05` is set for two inputs, an int and a texture coordinate. Those are set by the `.shader_inputs_info_set` call by the types between the angle brackets. A limited number of types are accepted and they can't be repeated. The valid types are `int`, `ivec2`, `ivec4`, `pCoor`, `pColor`, and `pVect`. (This is a limitation of the course library, not Vulkan.) An input type does not have to be used for the purpose implied by its name. So a `pColor`, for example, can be used to hold four floats used for any purpose. Note that `ivec2` is a 2-element integer vector.

*Modify Shader Inputs*
For any change made to pipeline inputs (by modifying `.shader_inputs_info_set`) a corresponding change must be made in the shader code. For your convenience some inputs are pre-defined, guarded by ifdefs, such as `in_int4` for an `ivec4` type.

*Modify the Shader Interface Blocks*
A correct solution to this assignment will require changing the information sent from the vertex to the geometry shader. Modify the interface blocks for that. (These are the structure-like constructs with tags like `Data_to_GS`.) Remember that the block describing the output of the vertex shader must match the block at the input to the geometry shader.

*Modify the Geometry Shader Input and Output Layouts*
The geometry shader output topology should remain a triangle strip, but the `max_vertices` setting should change. Compute `max_vertices` from `opt_segments`, which is available as a constant in the shader file (trust me on that). The input layout to the geometry shader must match what was set on `pipe_hw05`.

*Modify the Geometry Shader to Emit the Triangles*
Modify the geometry shader so that it computes and emits the triangles between the spokes. There should be something like the `s` loop, but adapted for a geometry shader.

*Compute the Correct Surface Normal Based on the Shape of the Curve*
Use `bez_dt` to help get the normal. Note that `bez_dt` does not provide the surface normal, instead it is a vector in a direction of the curve from `posx1` to `posx2` at point `t`.

The solution code is in the repo, and is on the Web at
`https://www.ece.lsu.edu/koppel/gpup/2023/hw05-sol.cc.html` (CPU code) and
`https://www.ece.lsu.edu/koppel/gpup/2023/hw05-shdr-sol.cc.html` (shader code). In the discussion below solution code fragments are shown in various states of development, in some cases a code fragment is first shown in some tentative or incomplete way, and then again with improvements. After this discussion more complete versions of the code are shown.

The description of the solution will start with the geometry shader. Each invocation of the geometry shader must emit the `2*opt_segments` triangles emitted by an `s` loop iteration (described in the Code Overview section above).

Consider the simplified CPU `s` loop from the Code Overview section:

```
pCoor pos11 = lis_pos1[link_idx1];
pCoor pos12 = lis_pos2[link_idx1];
pCoor pos21 = lis_pos1[link_idx2];
pCoor pos22 = lis_pos2[link_idx2];
pVect v11 = lis_v1[link_idx1], v12 = lis_v2[link_idx1];
pVect v21 = lis_v1[link_idx2], v22 = lis_v2[link_idx2];
const float delta_t = 1.0/opt_segments;
for ( int s=0; s<=opt_segments; s++ ) {
    const float t = s * delta_t;
    pCoor pos1t = bez(pos11,pos12,v11,v12,t);
    pCoor pos2t = bez(pos21,pos22,v21,v22,t);
    bset_hw04 << pos1t << mix(tex11,tex12,t)
              << pos2t << mix(tex21,tex22,t);
 }
```

To adapt this to the geometry shader we need to make changes to data types and such, and to also plan for the needed input data.

As stated in the Code Overview, arrays `lis_pos1`, `lis_pos2`, `lis_v1`, and `lis_v2` used by the CPU code are also available to the shader (as Vulkan storage buffers). Their names are shorter, just `pos1`, `pos2`, etc. As stated in the Code Overview, the element data types for all those arrays are `vec4` (not `pCoor` and `pVect`). So the statement `pCoor pos11 = lis_pos1[link_idx1]` should be changed to `vec4 pos11 = pos1[link_idx1]`.

Also stated in the Code Overview, is that `bez` is provided in the shader code file and `mix` is a built-in GLSL function. The qualifier `const` has a different meaning in GLSL than it does in C++, so it needs to be removed. Making all those changes we get:

```
// Partially translated s loop. Still doesn't work.
vec4 pos11 = pos1[link_idx1];
vec4 pos12 = pos2[link_idx1];
vec4 pos21 = pos1[link_idx2];
vec4 pos22 = pos2[link_idx2];
vec4 v11 = v1[link_idx1], v12 = v2[link_idx1];
vec4 v21 = v1[link_idx2], v22 = v2[link_idx2];
float delta_t = 1.0/opt_segments;
for ( int s=0; s<=opt_segments; s++ ) {
    float t = s * delta_t;
    vec4 pos1t = bez(pos11,pos12,v11,v12,t);
    vec4 pos2t = bez(pos21,pos22,v21,v22,t);
    // WARNING: The lines below have no meaning in GLSL.
    bset_hw04 << pos1t << mix(tex11,tex12,t)
              << pos2t << mix(tex21,tex22,t);
 }
```

Here's what remains to be done with the code above: the buffer set insertion line, `bset_hw04 <<`, has no meaning in shader code, it must be replaced with writes to geometry shader output variables. Also, we need to provide values for `link_idx1`, `link_idx2`, `tex11`, `tex12`, `tex21`, and `tex22`. These can come from geometry shader inputs.

To start, lets replace the buffer set insertion line. That line is writing two vertex coordinates, **pos1t** and **pos2t**, which are supposed to be grouped using a triangle strip topology. The output of the geometry shader is already set for a triangle strip (that's the only kind of triangles available). So, we need to just emit the two vertices.

To emit a vertex from a geometry shader we first write the geometry shader output variables and then call **EmitVertex()**. The geometry shader has built-in output **gl_Position**, which must be written with the clip-space coordinate of the vertex. Our shader code has declared three other outputs (which are expected by our fragment shader which remember is not part of this assignment):

```
layout ( location = 0 ) out Data_to_FS
{
  vec3 normal_e;
  vec4 vertex_e;
  vec2 tex_coor;
} Out;
```

So, for each vertex we need to write these four output variables with appropriate values. The eye- and clip-space coordinates are easy to compute from the object-space coordinates in **pos1t** and **pos2t**. The only part that requires thought is the normal, which we'll get to later. Here is code emitting the two vertices:

```
vec4 pos11 = pos1[link_idx1], pos12 = pos2[link_idx1];
vec4 pos21 = pos1[link_idx2], pos22 = pos2[link_idx2];
vec4 v11 = v1[link_idx1], v12 = v2[link_idx1];
vec4 v21 = v1[link_idx2], v22 = v2[link_idx2];
float delta_t = 1.0/opt_segments;
for ( int s=0; s<=opt_segments; s++ ) {
    float t = s * delta_t;
    vec4 pos1t = bez(pos11,pos12,v11,v12,t);
    vec4 pos2t = bez(pos21,pos22,v21,v22,t);

    // Emit Vertex for link_idx1
    gl_Position = ut.clip_from_object * pos1t;
    Out.vertex_e = ut.eye_from_object * pos1t;
    Out.tex_coor = mix(tex11,tex12,t);
    // Out.normal_e = // Save for later.
    EmitVertex();

    // Emit Vertex for link_idx2
    gl_Position = ut.clip_from_object * pos2t;
    Out.vertex_e = ut.eye_from_object * pos2t;
    Out.tex_coor = mix(tex21,tex22,t);
    // Out.normal_e = // Save for later.
    EmitVertex();
 }
 EndPrimitive();
```

For the code above to work we need to remember to change the **max_vertices** attribute on the geometry shader output layout declaration. The loop iterates **opt_segments+1** times, and each iteration emits two vertices. So change the output to:

```
layout ( lines ) in;
layout ( triangle_strip, max_vertices = 2 * ( opt_segments + 1 ) ) out;
```

Note that `opt_segments` is provided as a constant, so the declaration above is correct. It would not be correct if `opt_segments` were a uniform variable or storage buffer. The change input topology is also shown.

At this point we still haven't described how the geometry shader gets `link_idx1` and `link_idx2` and the four texture coordinates. The code above, remember, emits the surface between two spokes, one associated with `link_idx1` and the other with `link_idx2`.

The solution is to treat each spoke as a vertex. The CPU code will insert a link index and two texture coordinates for each vertex. The vertex shader will read one such vertex, and emit it. The pipeline will be set up with a line strip topology, `vk::PrimitiveTopology::eLineStrip`, and the geometry shader will be set up to expect lines.

On the CPU side, other than changing the topology, we need to insert just one vertex per spoke, but we need to provide two texture coordinates for that vertex. This is done by packing two texture coordinates into a `pCoor` type. (One texture coordinate is for the inner end of the spoke, and one is for the outer end of the spoke.)

```
// Code from hw05-sol.cc:World::render_hw05:
for ( size_t i = 0; i <= n_spokes; i++ ) {
    Link* const spoke1 = hw05_stuff.spokes[i%n_spokes];
    const int link_idx = spoke1->idx;
    bset_hw05 << link_idx;

    pTCoor tex11 = spoke1->ball1->tca[opt_texture];
    pTCoor tex12 = spoke1->ball2->tca[opt_texture];
    pCoor pack;
    pack.x = tex11.x;  pack.y = tex11.y;
    pack.z = tex12.x;  pack.w = tex12.y;
    bset_hw05 << pack;
  }
```

Packing a data type intended for one purpose with data meant for another purpose is stylistically sloppy but is sometimes done for performance reasons.

The vertex shader in the solution code unpacks the textures, arriving at location `LOC_IN_POS`, into two `vec2`. The vertex shader code is shown below:

```
layout ( location = LOC_IN_INT1 ) in int in_link_idx;
layout ( location = LOC_IN_POS ) in vec4 in_pack;

layout ( location = 0 ) out Data_to_GS
{
  vec2 tc1, tc2;
  int link_idx;
} Out;

void vs_hw05() {
  Out.tc1 = in_pack.xy;
  Out.tc2 = in_pack.zw;
  Out.link_idx = in_link_idx;
}
```

In the solution shown so far, it is the geometry shader that uses the link index to retrieve the `pos1`, `pos2`, `v1`, and `v2` values. The solution checked into the repo uses the vertex shader to do that:

```
layout ( location = LOC_IN_INT1 ) in int in_link_idx;
layout ( location = LOC_IN_POS ) in vec4 in_pack;

layout ( location = 0 ) out Data_to_GS
{
  vec4 p1, p2, v1, v2;
  vec2 tc1, tc2;
} Out;

void vs_hw05() {
  Out.tc1 = in_pack.xy;
  Out.tc2 = in_pack.zw;

  Out.p1 = pos1[in_link_idx];
  Out.p2 = pos2[in_link_idx];
  Out.v1 = v1[in_link_idx];
  Out.v2 = v2[in_link_idx];
}
```

The advantage of the approach immediately above is that each data item, such as `pos1[123]` for link index value 123 is retrieved just once. In the version of the code where the geometry shader retrieves `pos1` a particular element, such as `pos1[123]` would be retrieved twice because each vertex (except the first and the last) is an input to two geometry shader invocations. There is also less code in the version immediately above. A possible disadvantage is lower performance because more data must be moved from the vertex shader to the geometry shader, but that depends on what is done with the vertex shader output data.

With the vertex shader immediately above the geometry shader simplifies to:

```
void gs_hw05() {
  float delta_t = 1.0 / opt_segments;
  for ( int s=0; s<=opt_segments; s++ )
    {
      float t = s * delta_t;
      vec4 pos0t_o = bez( In[0].p1, In[0].p2, In[0].v1, In[0].v2, t );
      //                  pos11     pos12     v11       v12
      vec4 pos1t_o = bez( In[1].p1, In[1].p2, In[1].v1, In[1].v2, t );
      //                  pos21     pos22     v21       v22

      //  Emit the vertex along spoke 0  ( In[0] )
      gl_Position = ut.clip_from_object * pos0t_o;
      Out.vertex_e = ut.eye_from_object * pos0t_o;
      Out.tex_coor = mix( In[0].tc1, In[0].tc2, t );
      // Out.normal_e = // later.
      EmitVertex();

      /// Emit the vertex along spoke 1  ( In[1] ).
      gl_Position = ut.clip_from_object * pos1t_o;
      Out.vertex_e = ut.eye_from_object * pos1t_o;
      Out.tex_coor = mix( In[1].tc1, In[1].tc2, t );
      // Out.normal_e = // later.
      EmitVertex();
    }
```

```
  EndPrimitive();
}
```

In the code above notice that **pos1t** has been renamed to **pos0t_o** to match the geometry shader input index, **In[0]**, of the former spoke 1, now called spoke 0.

Finally, the normals. The normal is the vector orthogonal to the surface. A normal can be found by taking the cross product of two vectors on the surface. One such vector is easy to compute, the one from **pos0_o** to **pos1_o**. That vector goes from one spoke to the other. Note that **pos0_o** is computed using function **bez** which takes parameter **t**, which varies from 0 to 1. To find the other vector on the surface we can take the derivative of **bez** with respect to **t**. To make things easy, that is precomputed in **bez_dt**. So by taking the cross product of those two vectors we get the normal:

```
      //  Compute direction along each spoke. Needed to find normals.
      vec3 tan0 = bez_dt( In[0].p1, In[0].p2, In[0].v1, In[0].v2, t );

      //  Compute a vector from point on spoke 0 to spoke 1. For normals.
      vec3 vx = pos1t_o.xyz - pos0t_o.xyz;

      //  Compute and emit surface normal.
      vec3 normal_1_o = cross( tan0, vx );
      Out.normal_e = normalize( mat3(ut.eye_from_object) * normal_1_o );
```

Larger excerpts of the solution are shown below, including with comments and other formatting removed above.

```
void
World::render_hw05(vk::CommandBuffer& cb)
{
  if ( opt_segments != opt_segments_pipe_hw05 )
    {
      if ( pipe_hw05 ) pipe_hw05.destroy();
      opt_segments_pipe_hw05 = opt_segments;
    }

  if ( !pipe_hw05 )
    pipe_hw05
      .init( vh.qs )
      .ds_follow( puni_light_curr )
      .ds_follow( transform )
      .ds_uniform_use( "BIND_UNI_COMMON", buf_uni_common )
      .ds_uniform_use( "BIND_HW05", uni_hw05 )
      .ds_storage_follow( "BIND_LINKS_POS1", lis_pos1 )
      .ds_storage_follow( "BIND_LINKS_POS2", lis_pos2 )
      .ds_storage_follow( "BIND_LINKS_V1", lis_v1 )
      .ds_storage_follow( "BIND_LINKS_V2", lis_v2 )
      .ds_use( sampler, texid_hw )

      /// SOLUTION
      //
      //  Change topology from eTriangleStrip to eLineStrip.
      //
```

```
    .topology_set( vk::PrimitiveTopology::eLineStrip )
    //
    //  Change shader inputs from <int,pTCoor> to <int,pCoor>
    //
    .shader_inputs_info_set<int,pCoor>()
    //
    .shader_code_set
      ("hw05-shdr-sol.cc", "vs_hw05();", "gs_hw05();", "fs_main();",
       pStringF("const int opt_segments = %d;\n", opt_segments) )
    .create();

if ( uni_hw05_needs_update )
  {
    uni_hw05_needs_update = false;
    uni_hw05->color = 1.0 * color_white;
    uni_hw05->opt_texture = opt_texture;
    uni_hw05.to_dev();
  }

bset_hw05.reset( pipe_hw05 );

const size_t n_spokes = hw05_stuff.spokes.size();

for ( size_t i = 0; i <= n_spokes; i++ )
  {
    Link* const spoke1 = hw05_stuff.spokes[i%n_spokes];
    const int link_idx = spoke1->idx;
    pTCoor tex11 = spoke1->ball1->tca[opt_texture];
    pTCoor tex12 = spoke1->ball2->tca[opt_texture];

    /// SOLUTION
    //
    //  Insert just one "vertex", corresponding to a spoke.
    //
    //  The "vertex" consists of a link index, the texture
    //  coordinate at the inner ball, and the texture coordinate at
    //  the outer ball.

    //  Insert the link index (corresponding to this spoke).
    //
    bset_hw05 << link_idx;
    //
    //  The geometry shader will retrieve the coordinates from arrays
    //  pos1 and pos2.

    //  Send two texture coordinates, one for the inner ball and
    //  one for the outer ball. Since a texture coordinate has just
    //  two components a pCoor can be used to hold two texture coordinates,
    //  and so pack the two texture coordinates into a pCoor.
    //
    pCoor pack;
```

```
      pack.x = tex11.x;  pack.y = tex11.y;
      pack.z = tex12.x;  pack.w = tex12.y;
      bset_hw05 << pack;
    }

  bset_hw05.to_dev();

  pipe_hw05.record_draw( cb, bset_hw05 );
}
```

```
/// LSU EE 4702-1 (Fall 2023), GPU Programming
//
 /// Homework 5 – SOLUTION
 //
 //  Modified this file and hw05-sol.cc.
 //  Assignment: https://www.ece.lsu.edu/gpup/2023/hw05.pdf


// Specify version of OpenGL Shading Language.
//
#version 460

#extension GL_GOOGLE_include_directive : enable
#include <light.h>
#include <transform.h>
#include "links-shdr-common.h"
#include "shader-common-generic-lighting.h"

layout ( binding = BIND_UNI_COMMON ) uniform Common_Uniform
{
  Shdr_Uni_Common com;
};

bool opt_tryout1 = bool(com.tryout.x);
bool opt_tryout2 = bool(com.tryout.y);
float opt_tryoutf = com.tryoutf.x;

const int TX_None = 0, TX_Square = 1, TX_Circle = 2;

layout ( binding = BIND_HW05 ) uniform HW_05
{
  vec4 color;
  int opt_texture;
  float ring_hole_frac, one_over_delta_r, dy_d_theta;
};

#ifdef BIND_LINKS_POS1
layout ( binding = BIND_LINKS_POS1 ) buffer sr { vec4 pos1[]; };
layout ( binding = BIND_LINKS_POS2 ) buffer sr2 { vec4 pos2[]; };
layout ( binding = BIND_LINKS_V1 ) buffer spr { vec4 v1[]; };
layout ( binding = BIND_LINKS_V2 ) buffer sc { vec4 v2[]; };
#endif


#ifdef _VERTEX_SHADER_

 /// SOLUTION
//
//   Changed declarations of vertex shader inputs.
//
//   Unsolved vertex shader inputs:
```

```
//       int   For link_idx.
//       vec2  For texture coordinate, type pTCoor on CPU.
//
//   Solution vertex shader inputs:
//       int   For link_idx.
//       vec4  For two texture coordinates. Packed into a pCoor.

layout ( location = LOC_IN_INT1 ) in int in_link_idx;

layout ( location = LOC_IN_POS ) in vec4 in_pack;

// Interface block for vertex shader output / geometry shader input.
//
layout ( location = 0 ) out Data_to_GS
{
  /// SOLUTION
  //
  vec4 p1, p2, v1, v2;
  vec2 tc1, tc2;
} Out;

void
vs_hw05()
{
  /// SOLUTION
  //
  // Unpack in_pack into a 2-element array of texture coordinates.
  Out.tc1 = in_pack.xy;
  Out.tc2 = in_pack.zw;

  // Retrieve positions and vectors in vertex shader.
  //
  Out.p1 = pos1[in_link_idx];
  Out.p2 = pos2[in_link_idx];
  Out.v1 = v1[in_link_idx];
  Out.v2 = v2[in_link_idx];
  //
  // It would also be correct to just send in_link_idx to the geometry
  // shader and have the geometry shader retrieve the positions and
  // vectors.
}


#endif

#ifdef _GEOMETRY_SHADER_

layout ( location = 0 ) in Data_to_GS
{
  /// SOLUTION
  //
```

```
  vec4 p1, p2, v1, v2;
  vec2 tc1, tc2;
} In[];

layout ( location = 0 ) out Data_to_FS
{
  vec3 normal_e;
  vec4 vertex_e;
  vec2 tex_coor;
} Out;

// Type of primitive at geometry shader input.
//

 /// SOLUTION
//
// Change layout from triangles to lines.
//
layout ( lines ) in;

// Type of primitives emitted geometry shader output.
//

 /// SOLUTION
//
//   Increase max vertices from 3 to 2 + 2*opt_segments.
//
layout ( triangle_strip, max_vertices = 2 * ( opt_segments + 1 ) ) out;

vec4
bez(vec4 p1, vec4 p2, vec4 v1, vec4 v2, float t)
{
  float t2 = t * t;
  float t3 = t2 * t;
  return
    vec4( ( 2*t3 - 3*t2 + 1 ) * p1.xyz
          + ( -2*t3 + 3*t2 ) * p2.xyz
          + ( t3 - 2*t2 + t ) * v1.xyz
          - ( t3 - t2 ) * v2.xyz,
          1 );
}

vec3
bez_dt(vec4 p1, vec4 p2, vec4 v1, vec4 v2, float t)
{
  float t2 = t * t;
  float t3 = t2 * t;
  return
    ( 6*t2 - 6*t ) * p1.xyz
    + ( -6*t2 + 6*t ) * p2.xyz
    + ( 3*t2 - 4*t + 1 ) * v1.xyz
```

```
    - ( 3*t2 - 2*t ) * v2.xyz;
}

void
gs_hw05()
{
  float delta_t = 1.0 / opt_segments;

  /// SOLUTION
  //
  //  The input to the geometry shader is a line to GLSL with one end
  //  point described by In[0] and the other endpoint described by
  //  In[1]. But to us, In[0] is spoke 0 and In[1] is spoke 1. (These
  //  can be any pair of adjacent spokes.)
  //
  //  In the loop below the geometry shader emits a triangle strip
  //  to form a surface between the two spokes. The loop below computes
  //  the same values as the s loop in hw05.cc:World::render_hw04.
  //
  //  Note that In[0].p1 in the code below corresponds to pos11 in
  //  render_hw04, In[1].p1 corresponds to pos21 in render_hw04, etc.

  for ( int s=0; s<=opt_segments; s++ )
    {
      float t = s * delta_t;

      // Compute interpolated coordinate along spokes 0 and 1 at position t.
      //
      vec4 pos0t_o = bez( In[0].p1, In[0].p2, In[0].v1, In[0].v2, t );
      //                  pos11     pos12     v11       v12
      //
      vec4 pos1t_o = bez( In[1].p1, In[1].p2, In[1].v1, In[1].v2, t );
      //                  pos21     pos22     v21       v22
      //
      // Note: Curve from In[0].p1 to In[0].p2 is a 3rd degree polynomial,
      // bez finds a point on this curve.

      //  Compute direction along each spoke. Needed to find normals.
      //
      vec3 tan0 = bez_dt( In[0].p1, In[0].p2, In[0].v1, In[0].v2, t );
      vec3 tan1 = bez_dt( In[1].p1, In[1].p2, In[1].v1, In[1].v2, t );

      //  Compute a vector from point on spoke 0 to spoke 1. For normals.
      //
      vec3 vx = pos1t_o.xyz - pos0t_o.xyz;

      //
      ///  Emit the vertex along spoke 0  ( In[0] )
      //

      //  Emit clip-space coordinate and eye-space coordinate.
```

```
      //
      gl_Position = ut.clip_from_object * pos0t_o;
      Out.vertex_e = ut.eye_from_object * pos0t_o;

      //  Compute and emit interpolated texture coordinate.
      //
      Out.tex_coor = mix( In[0].tc1, In[0].tc2, t );
      //
      //  Simple linear interpolation is sufficient for textures.

      //  Compute and emit surface normal.
      //
      vec3 normal_1_o = cross( tan0, vx );
      Out.normal_e = normalize( mat3(ut.eye_from_object) * normal_1_o );

      EmitVertex();

      //
      /// Emit the vertex along spoke 1  ( In[1] ).
      //
      gl_Position = ut.clip_from_object * pos1t_o;
      Out.vertex_e = ut.eye_from_object * pos1t_o;
      Out.tex_coor = mix( In[1].tc1, In[1].tc2, t );
      vec3 normal_2_o = cross( tan1, vx );
      Out.normal_e = normalize( mat3(ut.eye_from_object) * normal_2_o );

      EmitVertex();
    }

  EndPrimitive();
}



#endif


#ifdef _FRAGMENT_SHADER_

#ifdef BIND_TEXUNIT
layout ( binding = BIND_TEXUNIT ) uniform sampler2D tex_unit_0;
#endif

layout ( location = 0 ) in Data_to_FS
{
  vec3 normal_e;
  vec4 vertex_e;
  vec2 tex_coor;
} In;

layout ( location = 0 ) out vec4 out_frag_color;
```

```
void
fs_main()
{
  vec4 lit_color = generic_lighting(In.vertex_e, color, In.normal_e);

  if ( opt_texture == TX_None )
    {
      out_frag_color = lit_color;
    }
  else if ( opt_texture == TX_Circle )
    {
      float dist = length(In.tex_coor);
      float angle = atan(In.tex_coor.y,In.tex_coor.x);
      vec2 tc =
        vec2( ( dist - ring_hole_frac ) * one_over_delta_r,
              angle * dy_d_theta );
      out_frag_color = lit_color * texture(tex_unit_0,tc);
    }
  else
    {
      out_frag_color = lit_color * texture(tex_unit_0,In.tex_coor);
    }
}


#endif
```

```
/// LSU EE 4702-1 (Fall 2023), GPU Programming
//
 /// Homework 5 -- SOLUTION
 //
 //  Modified this file and hw05-sol.cc.
 //  Assignment: https://www.ece.lsu.edu/gpup/2023/hw05.pdf
 //  Solution Discussion: https://www.ece.lsu.edu/gpup/2023/hw05_sol.pdf


// Specify version of OpenGL Shading Language.
//
#version 460

#extension GL_GOOGLE_include_directive : enable
#include <light.h>
#include <transform.h>
#include "links-shdr-common.h"
#include "shader-common-generic-lighting.h"

layout ( binding = BIND_UNI_COMMON ) uniform Common_Uniform
{
  Shdr_Uni_Common com;
};

bool opt_tryout1 = bool(com.tryout.x);
bool opt_tryout2 = bool(com.tryout.y);
float opt_tryoutf = com.tryoutf.x;

const int TX_None = 0, TX_Square = 1, TX_Circle = 2;

layout ( binding = BIND_HW05 ) uniform HW_05
{
  vec4 color;
  int opt_texture;
  float ring_hole_frac, one_over_delta_r, dy_d_theta;
};

#ifdef BIND_LINKS_POS1
layout ( binding = BIND_LINKS_POS1 ) buffer sr { vec4 pos1[]; };
layout ( binding = BIND_LINKS_POS2 ) buffer sr2 { vec4 pos2[]; };
layout ( binding = BIND_LINKS_V1 ) buffer spr { vec4 v1[]; };
layout ( binding = BIND_LINKS_V2 ) buffer sc { vec4 v2[]; };
#endif


#ifdef _VERTEX_SHADER_

 /// SOLUTION
//
//   Changed declarations of vertex shader inputs.
//
//   Unsolved vertex shader inputs:
//       int   For link_idx.
```

```
//       vec2  For texture coordinate, type pTCoor on CPU.
//
//   Solution vertex shader inputs:
//       int   For link_idx.
//       vec4  For two texture coordinates. Packed into a pCoor.

layout ( location = LOC_IN_INT1 ) in int in_link_idx;

layout ( location = LOC_IN_POS ) in vec4 in_pack;

// Interface block for vertex shader output / geometry shader input.
//
layout ( location = 0 ) out Data_to_GS
{
  /// SOLUTION
  //
  vec4 p1, p2, v1, v2;
  vec2 tc1, tc2;
} Out;

void
vs_hw05()
{
  /// SOLUTION
  //
  // Unpack in_pack into a 2-element array of texture coordinates.
  Out.tc1 = in_pack.xy;
  Out.tc2 = in_pack.zw;

  // Retrieve positions and vectors in vertex shader.
  //
  Out.p1 = pos1[in_link_idx];
  Out.p2 = pos2[in_link_idx];
  Out.v1 = v1[in_link_idx];
  Out.v2 = v2[in_link_idx];
  //
  // It would also be correct to just send in_link_idx to the geometry
  // shader and have the geometry shader retrieve the positions and
  // vectors.
}


#endif

#ifdef _GEOMETRY_SHADER_

layout ( location = 0 ) in Data_to_GS
{
  /// SOLUTION
  //
  vec4 p1, p2, v1, v2;
  vec2 tc1, tc2;
} In[];
```

```
layout ( location = 0 ) out Data_to_FS
{
  vec3 normal_e;
  vec4 vertex_e;
  vec2 tex_coor;
} Out;

// Type of primitive at geometry shader input.
//

 /// SOLUTION
//
// Change layout from triangles to lines.
//
layout ( lines ) in;

// Type of primitives emitted geometry shader output.
//

 /// SOLUTION
//
//   Increase max vertices from 3 to 2 + 2*opt_segments.
//
layout ( triangle_strip, max_vertices = 2 * ( opt_segments + 1 ) ) out;

vec4
bez(vec4 p1, vec4 p2, vec4 v1, vec4 v2, float t)
{
  float t2 = t * t;
  float t3 = t2 * t;
  return
    vec4( ( 2*t3 - 3*t2 + 1 ) * p1.xyz
          + ( -2*t3 + 3*t2 ) * p2.xyz
          + ( t3 - 2*t2 + t ) * v1.xyz
          - ( t3 - t2 ) * v2.xyz,
          1 );
}

vec3
bez_dt(vec4 p1, vec4 p2, vec4 v1, vec4 v2, float t)
{
  float t2 = t * t;
  float t3 = t2 * t;
  return
    ( 6*t2 - 6*t ) * p1.xyz
    + ( -6*t2 + 6*t ) * p2.xyz
    + ( 3*t2 - 4*t + 1 ) * v1.xyz
    - ( 3*t2 - 2*t ) * v2.xyz;
}

void
gs_hw05()
{
  float delta_t = 1.0 / opt_segments;
```

```
/// SOLUTION
//
//  The input to the geometry shader is a line to GLSL with one end
//  point described by In[0] and the other endpoint described by
//  In[1]. But to us, In[0] is spoke 0 and In[1] is spoke 1. (These
//  can be any pair of adjacent spokes.)
//
//  In the loop below the geometry shader emits a triangle strip
//  to form a surface between the two spokes. The loop below computes
//  the same values as the s loop in hw05.cc:World::render_hw04.
//
//  Note that In[0].p1 in the code below corresponds to pos11 in
//  render_hw04, In[1].p1 corresponds to pos21 in render_hw04, etc.

for ( int s=0; s<=opt_segments; s++ )
  {
    float t = s * delta_t;

    // Compute interpolated coordinate along spokes 0 and 1 at position t.
    //
    vec4 pos0t_o = bez( In[0].p1, In[0].p2, In[0].v1, In[0].v2, t );
    //                  pos11     pos12     v11       v12
    //
    vec4 pos1t_o = bez( In[1].p1, In[1].p2, In[1].v1, In[1].v2, t );
    //                  pos21     pos22     v21       v22
    //
    // Note: Curve from In[0].p1 to In[0].p2 is a 3rd degree polynomial,
    // bez finds a point on this curve.

    //  Compute direction along each spoke. Needed to find normals.
    //
    vec3 tan0 = bez_dt( In[0].p1, In[0].p2, In[0].v1, In[0].v2, t );
    vec3 tan1 = bez_dt( In[1].p1, In[1].p2, In[1].v1, In[1].v2, t );

    //  Compute a vector from point on spoke 0 to spoke 1. For normals.
    //
    vec3 vx = pos1t_o.xyz - pos0t_o.xyz;

    //
    ///  Emit the vertex along spoke 0 ( In[0] )
    //

    //  Emit clip-space coordinate and eye-space coordinate.
    //
    gl_Position = ut.clip_from_object * pos0t_o;
    Out.vertex_e = ut.eye_from_object * pos0t_o;

    //  Compute and emit interpolated texture coordinate.
    //
    Out.tex_coor = mix( In[0].tc1, In[0].tc2, t );
    //
    //  Simple linear interpolation is sufficient for textures.
```

```
      //  Compute and emit surface normal.
      //
      vec3 normal_1_o = cross( tan0, vx );
      Out.normal_e = normalize( mat3(ut.eye_from_object) * normal_1_o );

      EmitVertex();

      //
      /// Emit the vertex along spoke 1 ( In[1] ).
      //
      gl_Position = ut.clip_from_object * pos1t_o;
      Out.vertex_e = ut.eye_from_object * pos1t_o;
      Out.tex_coor = mix( In[1].tc1, In[1].tc2, t );
      vec3 normal_2_o = cross( tan1, vx );
      Out.normal_e = normalize( mat3(ut.eye_from_object) * normal_2_o );

      EmitVertex();
    }

  EndPrimitive();
}



#endif


#ifdef _FRAGMENT_SHADER_

#ifdef BIND_TEXUNIT
layout ( binding = BIND_TEXUNIT ) uniform sampler2D tex_unit_0;
#endif

layout ( location = 0 ) in Data_to_FS
{
  vec3 normal_e;
  vec4 vertex_e;
  vec2 tex_coor;
} In;

layout ( location = 0 ) out vec4 out_frag_color;


void
fs_main()
{
  vec4 lit_color = generic_lighting(In.vertex_e, color, In.normal_e);

  if ( opt_texture == TX_None )
    {
      out_frag_color = lit_color;
    }
  else if ( opt_texture == TX_Circle )
    {
```

```
      float dist = length(In.tex_coor);
      float angle = atan(In.tex_coor.y,In.tex_coor.x);
      vec2 tc =
        vec2( ( dist - ring_hole_frac ) * one_over_delta_r,
              angle * dy_d_theta );
      out_frag_color = lit_color * texture(tex_unit_0,tc);
    }
  else
    {
      out_frag_color = lit_color * texture(tex_unit_0,In.tex_coor);
    }
}


#endif
```

```
/// LSU EE 4702-1 (Fall 2023) GPU Programming
//
 /// Homework 5 -- SOLUTION
 //
 //  Modified this file and hw05-shdr-sol.cc
 //  Assignment: https://www.ece.lsu.edu/gpup/2023/hw05.pdf
 //  Solution Discussion: https://www.ece.lsu.edu/gpup/2023/hw05_sol.pdf

/// Purpose
//
//   Demonstrate simulation of point masses connected by springs.


/// What Code Does

// Simulates balls connected by springs over a platform. Balls and
// springs can be initialized in different arrangements (called
// scenes). Currently scene 1 is a tree-like arrangement. The platform
// consists of tiles, some are purple-tinted mirrors (showing a
// reflection of the ball), the others show the course syllabus.



///  Keyboard Commands
 //
 /// Object (Eye, Light, Ball) Location or Push
 //   Arrows (←,→,↑,↓) Page Up, Page Down
 //        Move object or push ball, depending on mode.
 //        Shift + KEY: motion is 5x faster.
 //        Ctrl + KEY : motion is 5x slower.
 //   'e': Move eye.
 //   'l': Move light.
 //   'b': Move head (first) ball. (Change position but not velocity.)
 //   'B': Push head ball. (Add velocity.)
 //
 /// Eye Direction and GUI Options
 //
 //   'Home', 'End', 'Delete', 'Insert'
 //         Turn the eye direction. Home should rotate eye direction
 //         up, End should rotate eye down, Delete should rotate eye
 //         left, Insert should rotate eye right. The eye direction
 //         vector is displayed in the upper left.
 //  'C-='  (Ctrl =) Increase green text size.
 //  'C--'  (Ctrl -) Decrease green text size.
 //  'F12'  Write screenshot to PNG file.

 /// Simulation Execution Options
 //  (Also see variables below.)
 //
 //  'p'    Pause simulation. (Press again to resume.)
 //  ' '    (Space bar.) Advance simulation by 1/30 second.
 //  'S- '  (Shift Space bar.) Advance simulation by one time step.
```

```
/// Performance Tuning, Graphics Effects, and Debugging Options
//
//  'y'    Toggle value of opt_tryout1. Intended for experiments and debugging.
//  'Y'    Toggle value of opt_tryout2. Intended for experiments and debugging.
//  'n'    Toggle use of surface normals for spheres.
//  'o'    Toggle shadow effect.
//  'r'    Toggle mirror effect.
//  '!'    Toggle rendering of spheres (balls).
//  '@'    Toggle rendering of links.
//  '#'    Toggle rendering of platform.


/// Scene and Simulated System Options
//  (Also see variables below.)
//
//  'w'    Twirl balls around axis formed by head and tail. (Prob 2 soln).
//  '1'    Set up scene 1.
//  '2'    Set up scene 2.
//  '3'    Set up scene 3.
//  '4'    Set up scene 4.
//  'h'    Freeze position of first (head) ball. (Press again to release.)
//  't'    Freeze position of last (tail) ball. (Press again to release.)
//  's'    Stop balls.
//  'g'    Turn gravity on and off.

/// Variables
//   Selected program variables can be modified using the keyboard.
//   Use "Tab" to cycle through the variable to be modified, the
//   name of the variable is displayed next to "VAR" on the bottom
//   line of green text.

//  'Tab' Cycle to next variable.
//  '`'   Cycle to previous variable.
//  '+'   Increase variable value.
//  '-'   Decrease variable value.
//
//  VAR Spring Constant - Set spring constant.
//  VAR Time Step Duration - Set physics time step.
//  VAR Air Resistance - Set air resistance.
//  VAR Light Intensity - The light intensity.
//  VAR Gravity - Gravitational acceleration. (Turn on/off using 'g'.)

#define MAIN_INCLUDE
#undef VBUFFER_STRICT
#include <vhelper.h>

#include <vstroke.h>
#include <gp/coord.h>
#include <gp/pstring.h>
#include <gp/misc.h>

#include <vutil-texture.h>
#include <vutil-pipeline.h>
```

```
#include "shapes.h"

#include <cuda_runtime.h>

#include <gp/colors.h>

#include <gp/util-containers.h>
#include <gp/coord-containers.h>
#include <gp/cuda-gpuinfo.h>
#include "links.cuh"

#include "links-shdr-common.h"


///
/// Main Data Structures
///
//
// class World: All data about scene.


class World;
class Link;

enum GPU_Physics_Method { GP_cpu, GP_cuda, GP_ENUM_SIZE };
const char* const gpu_physics_method_str[] = { "CPU", "CUDA" };
const char *sh_names[] = { "HW 4", "HW 5" };

enum HW4_Texture { TX_None, TX_Square, TX_Circle, TX_ENUM_SIZE };
const char* const hw4_texture_str[] = {"NONE", "SQUARE", "CIRCLE" };

pTCoor
mix( const pTCoor& a, const pTCoor& b, float t )
{
  return pTCoor(lerp(a.x,b.x,t),lerp(a.y,b.y,t));
}

inline pCoor
bez(pCoor p1, pCoor p2, pVect v1, pVect v2, float t)
{
  const float t2 = t * t;
  const float t3 = t2 * t;
  return pCoor( ( 2*t3 - 3*t2 + 1 ) * pVect(p1) + (-2*t3 + 3*t2 ) * pVect(p2)
                + (t3 - 2*t2 + t ) * v1 - (t3 - t2 ) * v2 );
}



// Object Holding Ball State
//
class Ball {
public:
  Ball() { init(); }
  Ball(pCoor pos){ init(); position = pos; constants_update(); }
```

```
Ball(pCoor pos, const Ball &bd)
{ *this = bd; position = pos; constants_update(); }

void init()
  {
    velocity = pVect(0,0,0);
    omega = pVect(0,0,0);
    radius = 1;
    density = 1.00746;
    fdt_to_do = 0;
    locked = false;
    color = color_lsu_spirit_gold;
    specularity = 0;
    contact = false;
    orientation_set(pVect(0,1,0),0);
    world_time_expire = 3600.0 * 24 * 365 * 1000;
  }

pCoor position;
pVect velocity;
pQuat orientation;
pMatrix3x3p omatrix;
pVect omega;                    // Spin rate and axis.
double world_time_expire;

int idx; // Position in balls_pos_rad;

union {
  float4 m_r_fdt;
  struct {
    float mass;
    float radius;
    float fdt_to_do; // Radius / moment of inertia.
    float _;
  }; };

float mass_min; // Mass below which simulation is unstable.
float radius_sq, radius_inv;  // Pre-computed based on radius.
float density;
float mass_inv;

bool locked;

pVect force;
pVect torque;
pColor color;
float specularity;              // Amount of shininess, in [0,1].
bool contact;                   // When true, ball rendered in gray.
float spring_constant_sum;      // Used to compute minimum mass.

union {
  pTCoor tca [3];
  struct { pTCoor none, square, circle; } tc; };
```

```
  void orientation_set(pNorm dir, float angle)
    { orientation_set(pQuat(dir,angle)); }
  void orientation_set(pQuat ori)
    {
      orientation = ori;
      omatrix = pMatrix3x3_Rotation(orientation);
    }
  void constants_update();
  pVect point_rot_vel(pNorm tact_dir);
  pVect point_rot_vel(pVect tact_dir);
  void apply_tan_force(pNorm tact_dir, pNorm force_dir, double force_mag);
  void apply_tan_force(pNorm tact_dir, pVect force);
  void push(pVect amt);
  void translate(pVect amt);
  void stop();
  void freeze();
};

struct HW05_Stuff
{
  vector<Ball*> balls_outer;
  vector<Ball*> balls_inner;
  vector<Link*> spokes;
};

struct Uni_HW05
{
  pColor color;
  int opt_texture;

  /// Homework 5
  //
  // This structure is copied to the GPU.
  //
  float ring_hole_frac, one_over_delta_r, dy_d_theta;
};

void
Ball::constants_update()
{
  assert( radius );
  radius_sq = radius * radius;
  mass = 4.0 / 3 * M_PI * radius_sq * radius * density;
  radius_inv = 1 / radius;
  mass_inv = 1 / mass;
  // FYI, notice simplifications:
  //   const float mo_inertia = 0.4 * mass * radius_sq;
  //   fdt_to_do = radius / mo_inertia;
  fdt_to_do = radius_inv * 2.5 * mass_inv;
}

pVect
Ball::point_rot_vel(pNorm direction)
```

```
{
  /// Return velocity of point on surface of ball.
  //
  return radius * cross( omega, direction );
}

pVect
Ball::point_rot_vel(pVect rel_pos)
{
  /// Return velocity of point relative to center.
  //
  return cross( omega, rel_pos );
}

void
Ball::apply_tan_force(pNorm tact_dir, pVect force)
{
  torque += cross(tact_dir, force);
}

void
Ball::apply_tan_force(pNorm tact_dir, pNorm force_dir, double force_mag)
{
  apply_tan_force(tact_dir, force_mag * force_dir);
}

struct Link_Args {
  pColor natural_color = color_red;
  pColor color = color_red;
  float distance_relaxed = 0;
  float spring_constant = 0;
};

class Link {
public:
  Link(Ball *b1, Ball *b2, Link_Args args = {}):ball1(b1),ball2(b2),
     snapped(false)
  { init(args); }
  Link(Link *link, pVect cb1p, pVect cb2p, float drp):
    ball1(link->ball1), ball2(link->ball2), cb1(cb1p), cb2(cb2p),
    is_surface_connection(true), is_renderable(false),
    is_simulatable(true), distance_relaxed(drp), spring_constant(0),
    snapped(false){link->is_simulatable = false;}

  Link(Link *link, pVect cb1p, pVect cb2p, Link_Args args = {} ):
    ball1(link->ball1), ball2(link->ball2), cb1(cb1p), cb2(cb2p),
    is_surface_connection(true), is_renderable(false),
    is_simulatable(true),
    distance_relaxed(args.distance_relaxed),
    spring_constant(args.spring_constant),
    snapped(false){link->is_simulatable = false;}

  void init(Link_Args& args)
  {
```

```
  assert( args.distance_relaxed == 0 );
  natural_color = args.natural_color;
  color = args.color;
  spring_constant = args.spring_constant;
  init();
}
void init()
  {
    assert( ball1->radius > 0 );
    assert( ball2->radius > 0 );
    is_simulatable = true;
    is_renderable = true;
    pNorm n12(ball1->position,ball2->position);
    const float rad_sum = ball1->radius + ball2->radius;
    is_surface_connection = n12.magnitude >= rad_sum;

    pMatrix3x3 b1rot = ball1->omatrix;
    pMatrix3x3 b2rot = ball2->omatrix;

    pVect b1_y = b1rot * pVect(0,1,0);
    pVect b1_x = b1rot * pVect(1,0,0);
    bool b1_dir_is_y = fabs(dot(b1_y,n12)) < 0.999;
    pVect b1_dir_loc = b1_dir_is_y ? b1_y : b1_x;

    con_x = pNorm(cross(b1_dir_loc,n12));
    con_y = cross(n12,con_x);
    b1_dir = mult_MTV( b1rot, con_y );
    b2_dir = mult_MTV( b2rot, con_y );

    if ( !is_surface_connection )
      {
        distance_relaxed = n12.magnitude;
        cb1 = cb2 = pVect(0,0,0);
        is_renderable = false;
        return;
      }

    distance_relaxed = n12.magnitude - rad_sum;
    cb1 = ball1->radius * mult_MTV( b1rot,  n12 );
    cb2 = ball2->radius * mult_MTV( b2rot, -n12 );
  }
int serial;
int idx;  // Position in lis_pos1, etc.
Ball* const ball1;
Ball* const ball2;
union {
  int2 ball_idx;
  struct { int ball1_idx, ball2_idx; };
};
pVect cb1, cb2, con_x, con_y, b1_dir, b2_dir;
bool is_surface_connection;
bool is_renderable, is_simulatable;
union {
  float2 dr_sc;
```

```
    struct {
      float distance_relaxed;
      float spring_constant;
    };
  };
  bool snapped;
  pColor natural_color;
  pColor color;

  pVect spring_force_12;
  pVect torque1,torque2;

};

class Surface {
public:
  Ball *b0, *b1, *b2;
  union {
    int4 b_idx;
    struct { int b0_idx, b1_idx, b2_idx, pad; };
  };
  Surface* operator ->() { return this; }
};

// Declare containers for Balls and Links. The containers are derived
// from std::vector and use overloaded operator += for push_back. (See
// include/util-containers.h.)
//
typedef pVectorI<Link> Links;
typedef pVectorI<Ball> Balls;


enum Shader_Option { SO_HW04, SO_HW05, SO_ENUM_SIZE };

class World {
public:
  World(pVulkan_Helper &fb, int argc, char **argv)
    :vh(fb), ff_state(fb.qs), raytrace(fb.qs),
     frame_timer(vh.frame_timer), transform(fb.qs),
     shapes(ff_state), sphere(ff_state),
     opt_tryout1(fb.qs.opt_tryout1),opt_tryout2(fb.qs.opt_tryout2)
  {init(argc,argv);}
  void init(int argc, char **argv);
  void init_graphics();
  void run();
  void frame_callback();
  void render(vk::CommandBuffer& cb);
  void render_objects(vk::CommandBuffer& cb, Render_Option render_option);
  void objects_erase();
  void cb_keyboard();
  void modelview_update();

  pVulkan_Helper& vh;
```

```
VFixed_Function_State_Manager ff_state;
VRaytrace raytrace;
pFrame_Timer& frame_timer;
VTransform transform;
Shapes shapes;
Sphere sphere;

bool opt_want_raytrace;

pVariable_Control variable_control;
double world_time;
double last_frame_wall_time;
float opt_time_step_duration;
int time_step_count;
float opt_gravity_accel;      // Value chosen by user.
pVect gravity_accel;          // Set to zero when opt_gravity is false;
bool opt_gravity;
bool opt_head_lock, opt_tail_lock;
bool &opt_tryout1, &opt_tryout2;  // For ad-hoc experiments.
float opt_tryoutf;
bool opt_sphere_true;
VBufferV<Shdr_Uni_Common> buf_uni_common;

// Tiled platform for ball.
//
float platform_xmin, platform_xmax, platform_zmin, platform_zmax;
float platform_pi_xwidth_inv;
VVertex_Buffer_Set bset_platform_tex, bset_platform_mir;
VPipeline pipe_platform_mir, pipe_platform_tex;
RT_Geo_Package rt_geo_platform_tex, rt_geo_platform_mir;

vk::Sampler sampler;
VTexture texid_hw;
VTexture texid_syl;
VTexture texid_emacs;

void platform_update();
bool platform_collision_possible(pCoor pos);

pCoor light_location;
float opt_light_intensity;
enum { MI_Eye, MI_Light, MI_Ball, MI_Ball_V, MI_COUNT } opt_move_item;
bool opt_pause;
bool opt_single_frame;      // Simulate for one frame.
int opt_single_time_step;  // Simulate for one time step.

VBufferV<Uni_Lighting> uni_light, uni_light_ambient, uni_light_all;
VBufferV<Uni_Lighting> uni_light_reflected;
VBufferV<Uni_Lighting> *puni_light_curr;

pCoor eye_location;
pVect eye_direction;
pMatrix transform_projection;
pMatrix modelview;
```

```
pVect adj_vector;
double adj_t_prev;
double adj_t_stop;
double adj_duration;

uint opt_shader;

bool opt_mirror;
int opt_mirror_depth;
bool opt_shadows;
bool opt_shadow_volumes;      // Make shadow volumes visible.
bool opt_normal_sphere;
int opt_hide_stuff;

int last_setup; // Last scene set up.

void ball_setup_1(bool big);
void ball_setup_2();
void ball_setup_3();
void ball_setup_4();
void ball_setup_5();
void ball_setup_6();
void setup_at_end();
void time_step_cpu(double);
void balls_stop();
void balls_freeze();
void balls_translate(pVect amt, int idx);
void balls_translate(pVect amt);
void balls_push(pVect amt, int idx);
void balls_push(pVect amt);
void balls_twirl();
void lock_update();

Ball *make_marker(pCoor pos, pColor col);

float opt_spring_constant, opt_spring_damp;
float opt_friction_dynamic;
float opt_air_resistance;
float distance_relaxed;
int chain_length;
Balls balls;
Ball *head_ball, *tail_ball;
Links links;
vector<Surface> surfaces;
int opt_sphere_slices;
float opt_omega;  // For disk and top.

Links link_new
( Ball *ball1, Ball *ball2,
  float stiffness = 0.2, float spring_constant = 0);

/// CUDA Stuff
```

```
  int opt_physics_method;
  GPU_Info gpu_info;
  cudaEvent_t frame_start_ce, frame_stop_ce;
  int opt_block_size;
  bool gpu_const_data_stale;
  bool cpu_phys_data_stale;

  void data_cpu_to_gpu_constants();
  void data_cpu_to_gpu_dynamic();
  void data_gpu_to_cpu_dynamic();
  CPU_GPU_Common c;  // c is for common.  See links.cuh.

  void render_link_start();
  void render_link_gather(Link *link);
  void render_link_render(vk::CommandBuffer& cb, bool shadow_volumes = false);
  void render_link_render_sv(vk::CommandBuffer& cb)
  { render_link_render(cb,true); };

  VPipeline pipe_set_2, pipe_set_2_sv;
  double world_time_link_update;

  VCompute comp_links;
  VBuffer<int> buf_links_as_idx; // Indices for links' acceleration structures.
  VBuffer<vec2> buf_links_tex_coor;
  VBuffer<vec4> buf_links_vertex_g;
  VBuffer<vec4> buf_links_normal_g;
  RT_Geo_Package rt_geo_links;

  int opt_sides, opt_segments;

  vector< VBufferVV<vec4>* > lis_bufs;
  VBufferVV<vec4> lis_pos1, lis_pos2;
  VBufferVV<vec4> lis_v1, lis_v2;
  VBufferVV<vec4> lis_b1_ydir, lis_b2_ydir, lis_tex_rad;

  // Homework 5 (Including Homework 4 material.)
  HW05_Stuff hw05_stuff;
  VVertex_Buffer_Set bset_hw04, bset_hw05;
  VPipeline pipe_hw04, pipe_hw05;
  RT_Geo_Package rt_geo_hw05;
  VBufferV<Uni_HW05> uni_hw05;
  bool uni_hw05_needs_update;
  int opt_texture;
  int opt_segments_pipe_hw05;

  void render_hw04(vk::CommandBuffer& cb);
  void render_hw05(vk::CommandBuffer& cb);

};

enum { OH_Links = 1, OH_Platform = 2, OH_Sphere = 4 };

void
```

```
World::init_graphics()
{
  const bool many_vars = false;
  vh.want_raytrace().init();
  vh.opt_record_every_time = true;
  vh.display_cb_set([&](){frame_callback();});
  vh.cbs_cmd_record.push_back( [&](vk::CommandBuffer& cb){ render(cb); });
  sphere.ppuni_light = &puni_light_curr;
  sphere.transform = &transform;

  opt_want_raytrace = false;

  if ( vh.have_raytrace )
    {
      raytrace.init();
      for ( auto b: { &bset_platform_mir, &bset_platform_tex } )
        b->usage_set
          ( vk::BufferUsageFlagBits::eVertexBuffer
            | vk::BufferUsageFlagBits::eStorageBuffer );
      buf_links_as_idx.init
        ( vh.qs, vk::BufferUsageFlagBits::eStorageBuffer, 0,
          vk::MemoryPropertyFlagBits::eDeviceLocal );
      buf_links_tex_coor.init
        ( vh.qs, vk::BufferUsageFlagBits::eStorageBuffer, 0,
          vk::MemoryPropertyFlagBits::eDeviceLocal );
      buf_links_vertex_g.init
        ( vh.qs, vk::BufferUsageFlagBits::eStorageBuffer, 0,
          vk::MemoryPropertyFlagBits::eDeviceLocal );
      buf_links_normal_g.init
        ( vh.qs, vk::BufferUsageFlagBits::eStorageBuffer, 0,
          vk::MemoryPropertyFlagBits::eDeviceLocal );
    }

  buf_uni_common.init(vh.qs,vk::BufferUsageFlagBits::eUniformBuffer);
  sphere.buf_uni_common = buf_uni_common;

  lis_bufs =
    { &lis_pos1, &lis_pos2,
      &lis_v1, &lis_v2,
      &lis_b1_ydir, &lis_b2_ydir,
      &lis_tex_rad };

  for ( auto b: lis_bufs )
    b->init(vh.dev_phys, vh.dev, vk::BufferUsageFlagBits::eStorageBuffer );

  ///
  /// Graphical Model Initialization
  ///

  opt_head_lock = false;
  opt_tail_lock = false;
  opt_tryout1 = opt_tryout2 = false;
  opt_normal_sphere = true;
  opt_sphere_true = true;
```

```
opt_tryoutf = 0;
variable_control.insert_linear(opt_tryoutf,"Tryout F",0.1);

eye_location = pCoor(24.2,11.6,-38.7);
eye_direction = pVect(-0.42,-0.09,0.9);

platform_xmin = -40; platform_xmax = 40;
platform_zmin = -40; platform_zmax = 40;

texid_syl.init
  ( vh.qs, P_Image_Read( vh.gp_root_get() / "vulkan" / "gpup.png", 255 ) );
texid_emacs.init( vh.qs, P_Image_Read("mult.png",-1) );
texid_hw.init( vh.qs, P_Image_Read("hw-assign.png",255) );
sampler = vh.qs.dev.createSampler
  ( { {},
      vk::Filter::eLinear,  // magFilter
      vk::Filter::eLinear,  // minFilter
      vk::SamplerMipmapMode::eLinear,
      // Also: eRepeat, eMirroredRepeat, eClampToEdge, etc.
      vk::SamplerAddressMode::eRepeat,
      vk::SamplerAddressMode::eRepeat,
      vk::SamplerAddressMode::eRepeat,
      0.0f,  // mipLodBias
      true, // anisotropyEnable,
      16.0f, // maxAnisotropy
      false, vk::CompareOp::eNever,  // compareEnable, compareOp
      0.0f, VK_LOD_CLAMP_NONE,  // min and max LOD
      vk::BorderColor::eFloatOpaqueBlack,
      false // unnormalizedCoordinates
      } );

sphere.texture_set(sampler,texid_emacs);

opt_light_intensity = 50;
light_location = pCoor(platform_xmax,platform_xmax,platform_zmin);
uni_light.init(vh.qs,vk::BufferUsageFlagBits::eUniformBuffer);
uni_light_all.init(vh.qs,vk::BufferUsageFlagBits::eUniformBuffer);
uni_light_reflected.init(vh.qs,vk::BufferUsageFlagBits::eUniformBuffer);
uni_light_ambient.init(vh.qs,vk::BufferUsageFlagBits::eUniformBuffer);

variable_control.insert(opt_light_intensity,"Light Intensity");

opt_move_item = MI_Eye;
opt_pause = false;
opt_single_time_step = 0;
opt_single_frame = false;

platform_update();
modelview_update();

adj_t_stop = 0;
adj_duration = 0.25;

opt_sides = 20;
```

```
  opt_segments = 16;
  if ( many_vars )
    variable_control.insert(opt_sides,"Link Sides");
  variable_control.insert(opt_segments,"Link Segments",5,1,41);
  opt_hide_stuff = 0;

  if ( vh.have_raytrace )
    {
      raytrace.buf_light_set(uni_light_all);
    }

  // Homework 5

  uni_hw05.init( vh.qs, vk::BufferUsageFlagBits::eUniformBuffer );
  uni_hw05_needs_update = true;
  opt_texture = TX_Square;

}

void
World::run()
{
  vh.message_loop_spin();

  uni_light.destroy();
  uni_light_ambient.destroy();
  uni_light_all.destroy();
  uni_light_reflected.destroy();
  transform.destroy();
  buf_uni_common.destroy();

  for ( auto b: lis_bufs ) b->destroy();

  pipe_platform_tex.destroy(); pipe_platform_mir.destroy();
  bset_platform_tex.destroy(); bset_platform_mir.destroy();
  pipe_set_2.destroy();
  pipe_set_2_sv.destroy();

  texid_syl.destroy();
  texid_emacs.destroy();
  texid_hw.destroy();
  vh.dev.destroySampler(sampler);
  shapes.destroy();
  sphere.destroy();

  uni_hw05.destroy();
  bset_hw04.destroy();
  pipe_hw04.destroy();
  bset_hw05.destroy();
  pipe_hw05.destroy();

  buf_links_as_idx.destroy();
  buf_links_tex_coor.destroy();
  buf_links_normal_g.destroy();
```

```
  buf_links_vertex_g.destroy();
  rt_geo_platform_tex.destroy();
  rt_geo_platform_mir.destroy();
  rt_geo_links.destroy();
  comp_links.destroy();
  raytrace.destroy();

  vh.finish();
}

void
World::render_hw04(vk::CommandBuffer& cb)
{
  if ( !pipe_hw04 )
    pipe_hw04
      .init( vh.qs )
      .ds_follow( puni_light_curr )
      .ds_follow( transform )
      .ds_uniform_use( "BIND_UNI_COMMON", buf_uni_common )
      .ds_uniform_use( "BIND_HW05", uni_hw05 )
      .ds_use( sampler, texid_hw )
      .topology_set( vk::PrimitiveTopology::eTriangleStrip )
      .shader_inputs_info_set<pCoor,pTCoor>()
      .shader_code_set
        ("hw04-shdr.cc", "vs_main();", "gs_main();", "fs_main();" )
      .create();


  if ( uni_hw05_needs_update )
    {
      uni_hw05_needs_update = false;
      uni_hw05->color = 1.0 * color_white;
      uni_hw05->opt_texture = opt_texture;
      uni_hw05.to_dev();
    }

  bset_hw04.reset( pipe_hw04 );

  const size_t n_spokes = hw05_stuff.spokes.size();

  for ( size_t i = 0; i < n_spokes; i++ )
    {
      size_t i2 = (i+1) % n_spokes;
      Link *spoke1 = hw05_stuff.spokes[i], *spoke2 = hw05_stuff.spokes[i2];
      const int link_idx1 = spoke1->idx;
      const int link_idx2 = spoke2->idx;
      pCoor pos11 = lis_pos1[link_idx1];
      pCoor pos12 = lis_pos2[link_idx1];
      pCoor pos21 = lis_pos1[link_idx2];
      pCoor pos22 = lis_pos2[link_idx2];

      pVect v11 = lis_v1[link_idx1], v12 = lis_v2[link_idx1];
      pVect v21 = lis_v1[link_idx2], v22 = lis_v2[link_idx2];
      pTCoor tex11 = spoke1->ball1->tca[opt_texture];
```

```
      pTCoor tex12 = spoke1->ball2->tca[opt_texture];
      pTCoor tex21 = spoke2->ball1->tca[opt_texture];
      pTCoor tex22 = spoke2->ball2->tca[opt_texture];

      const float delta_t = 1.0/opt_segments;
      for ( int s=0; s<=opt_segments; s++ )
        {
          const float t = s * delta_t;
          pCoor pos1t = bez( pos11, pos12, v11, v12, t );
          pCoor pos2t = bez( pos21, pos22, v21, v22, t );
          if ( s == 0 ) bset_hw04 << pos1t << pTCoor(0,0);
          bset_hw04
            << pos1t << mix(tex11,tex12,t)
            << pos2t << mix(tex21,tex22,t);
          if ( s == opt_segments ) bset_hw04 << pos2t << pTCoor(0,0);
        }
    }

  bset_hw04.to_dev();

  pipe_hw04.record_draw( cb, bset_hw04 );
}

void
World::render_hw05(vk::CommandBuffer& cb)
{
  if ( opt_segments != opt_segments_pipe_hw05 )
    {
      if ( pipe_hw05 ) pipe_hw05.destroy();
      opt_segments_pipe_hw05 = opt_segments;
    }

  if ( !pipe_hw05 )
    pipe_hw05
      .init( vh.qs )
      .ds_follow( puni_light_curr )
      .ds_follow( transform )
      .ds_uniform_use( "BIND_UNI_COMMON", buf_uni_common )
      .ds_uniform_use( "BIND_HW05", uni_hw05 )
      .ds_storage_follow( "BIND_LINKS_POS1", lis_pos1 )
      .ds_storage_follow( "BIND_LINKS_POS2", lis_pos2 )
      .ds_storage_follow( "BIND_LINKS_V1", lis_v1 )
      .ds_storage_follow( "BIND_LINKS_V2", lis_v2 )
      .ds_use( sampler, texid_hw )

      /// SOLUTION
      //
      //  Change topology from eTriangleStrip to eLineStrip.
      //
      .topology_set( vk::PrimitiveTopology::eLineStrip )
      //
      //  Change shader inputs from <int,pTCoor> to <int,pCoor>
      //
      .shader_inputs_info_set<int,pCoor>()
```

```
      //
      .shader_code_set
        ("hw05-shdr-sol.cc", "vs_hw05();", "gs_hw05();", "fs_main();",
         pStringF("const int opt_segments = %d;\n", opt_segments) )
      .create();

  if ( uni_hw05_needs_update )
    {
      uni_hw05_needs_update = false;
      uni_hw05->color = 1.0 * color_white;
      uni_hw05->opt_texture = opt_texture;
      uni_hw05.to_dev();
    }

  bset_hw05.reset( pipe_hw05 );

  const size_t n_spokes = hw05_stuff.spokes.size();

  for ( size_t i = 0; i <= n_spokes; i++ )
    {
      Link* const spoke1 = hw05_stuff.spokes[i%n_spokes];
      const int link_idx = spoke1->idx;
      pTCoor tex11 = spoke1->ball1->tca[opt_texture];
      pTCoor tex12 = spoke1->ball2->tca[opt_texture];

      /// SOLUTION
      //
      //  Insert just one "vertex", corresponding to a spoke.
      //
      //  The "vertex" consists of a link index, the texture
      //  coordinate at the inner ball, and the texture coordinate at
      //  the outer ball.

      //  Insert the link index (corresponding to this spoke).
      //
      bset_hw05 << link_idx;
      //
      //  The geometry shader will retrieve the coordinates from arrays
      //  pos1 and pos2.

      //  Send two texture coordinates, one for the inner ball and
      //  one for the outer ball. Since a texture coordinate has just
      //  two components a pCoor can be used to hold two texture coordinates,
      //  and so pack the two texture coordinates into a pCoor.
      //
      pCoor pack;
      pack.x = tex11.x;  pack.y = tex11.y;
      pack.z = tex12.x;  pack.w = tex12.y;
      bset_hw05 << pack;
    }

  bset_hw05.to_dev();

  pipe_hw05.record_draw( cb, bset_hw05 );
```

```
}



void
World::platform_update()
{
  const float tile_count = 19;
  const float ep = 1.00001;
  const float delta_x = ( platform_xmax - platform_xmin ) / tile_count * ep;
  const float zdelta = ( platform_zmax - platform_zmin ) / tile_count * ep;

  const float trmin = 0.05;
  const float trmax = 0.7;
  const float tsmin = 0;
  const float tsmax = 0.4;
  pColor color_platform( color_white * 0.9 );

  if ( !pipe_platform_mir )
    pipe_platform_mir
      .init( ff_state )
      .ds_set( color_lsu_spirit_purple )
      .ds_follow( puni_light_curr )
      .ds_follow( transform )
      .topology_set( vk::PrimitiveTopology::eTriangleList )
      .create();
  bset_platform_mir.reset( pipe_platform_mir ).tolerate_tcoords();

  if ( !pipe_platform_tex )
    pipe_platform_tex
      .init( ff_state )
      .ds_set_material(color_platform)
      .ds_follow( puni_light_curr )
      .ds_follow( transform )
      .topology_set( vk::PrimitiveTopology::eTriangleList )
      .ds_use( sampler, texid_syl )
      .create();
  bset_platform_tex.reset( pipe_platform_tex );

  const pVect platform_normal(0,1,0);
  pTCoor t00(trmin,tsmin), t01(trmin,tsmax), t10(trmax,tsmin), t11(trmax,tsmax);
  bool even = true;

  for ( int i = 0; i < tile_count; i++ )
    {
      const float x0 = platform_xmin + i * delta_x;
      const float x1 = x0 + delta_x;
      const float y = 0;
      for ( float z = platform_zmin; z < platform_zmax; z += zdelta )
        {
          auto& bset = even ? bset_platform_tex : bset_platform_mir;
          const float z1 = z+zdelta;
          pCoor p00(x0,y,z), p01(x0,y,z1), p10(x1,y,z), p11(x1,y,z1);
```

```
          bset << t11 << platform_normal << p00;
          bset << t10 << platform_normal << p01;
          bset << t00 << platform_normal << p11;

          bset << t11 << platform_normal << p00;
          bset << t00 << platform_normal << p11;
          bset << t01 << platform_normal << p10;

          even = !even;
        }
    }

  bset_platform_mir.to_dev();
  bset_platform_tex.to_dev();

  if ( vh.have_raytrace && !rt_geo_platform_mir )
    {
      pColor mirror_color(color_lsu_spirit_purple);
      mirror_color.a = -0.5;
      rt_geo_platform_mir
        .init( raytrace )
        .color_front_set( mirror_color )
        .bind( bset_platform_mir, RT_Geometry_Triangles );
      rt_geo_platform_tex
        .init( raytrace )
        .color_front_set( color_platform )
        .bind( VR_BUF_IDX_sampler, sampler, texid_syl )
        .bind( bset_platform_tex, RT_Geometry_Triangles );
    }
}

void
World::modelview_update()
{
  pMatrix_Translate center_eye(-eye_location);
  pMatrix_Rotation rotate_eye(eye_direction,pVect(0,0,-1));
  modelview = rotate_eye * center_eye;
}

void
World::render_objects(vk::CommandBuffer& cb, Render_Option option)
{
  //  const float shininess_ball = 5;
  pColor spec_color(1,1,1);
  //  pColor spec_light = color_white * opt_light_intensity;

  const bool hide_links = opt_hide_stuff & OH_Links;
  const bool hide_platform = opt_hide_stuff & OH_Platform;
  const bool hide_sphere = opt_hide_stuff & OH_Sphere;

  sphere.init(opt_sphere_slices);

  if ( option == RO_Shadow_Volumes )
    {
```

```
      if ( !hide_sphere )
        {
          sphere.light_pos = light_location;
          sphere.render_bunch_gather_sv(world_time);
          for ( Ball *ball: balls )
            sphere.render_shadow_volume(cb,ball->radius,ball->position);

          sphere.render_bunch_render_sv(cb);
        }

      if ( !hide_links )
        {
          render_link_start();
          for ( Link *link: links )
            if ( link->is_renderable ) render_link_gather(link);
          render_link_render_sv(cb);
        }
    }
  else
    {
      sphere.rt_geo_sphere.rt_wanted_set(!hide_sphere);
      if ( !hide_sphere )
        {
          sphere.render_bunch_gather(world_time);
          for ( Ball *ball: balls )
            {
              if ( ball->contact )
                sphere.color = color_gray;
              else if ( ball->mass > 0 && ball->mass < ball->mass_min )
                sphere.color = color_red;
              else if ( ball->mass > 0 && ball->locked )
                sphere.color = color_pale_green;
              else
                sphere.color = ball->color;

              sphere.color.a = ball->specularity ? -ball->specularity : 1;
              sphere.render
                (cb, ball->radius,ball->position,
                 pMatrix_Rotation(ball->orientation));
            }

          sphere.render_bunch_render(cb, opt_sphere_true);
        }

      rt_geo_links.rt_wanted_set(!hide_links);
      if ( !hide_links )
        {
          render_link_start();
          for ( Link *link: links )
            if ( link->is_renderable ) render_link_gather(link);
          render_link_render(cb);
        }
    }
```

```
  rt_geo_platform_tex.rt_wanted_set(!hide_platform);
  rt_geo_platform_mir.rt_wanted_set(!hide_platform);
  if ( vh.raytrace_active() ) return;

  if ( last_setup == 4 && option != RO_Shadow_Volumes )
    {
      switch ( opt_shader ) {
      case SO_HW04: render_hw04(cb); break;
      case SO_HW05: render_hw05(cb); break;
      default: assert( false );
      }
    }

  if ( hide_platform ) return;

  if ( option == RO_Shadow_Volumes ) return;

  //
  // Render Platform
  //
  pipe_platform_tex.record_draw(cb,bset_platform_tex);
}

void
World::render(vk::CommandBuffer& cb)
{
  const bool do_raytrace = vh.raytrace_active();

  /// Emit a Graphical Representation of Simulation State
  //

  // Understanding of the code below not required for introductory
  // lectures.

  // That said, much of the complexity of the code is to show
  // the ball shadow and reflection.


  const int win_width = vh.get_width();
  const int win_height = vh.get_height();
  const float aspect = float(win_width) / win_height;
  const float n_dist = 0.01;
  const float xr = .8 * n_dist;

  transform.eye_from_global_set( modelview );
  transform.clip_from_eye_set
    ( pMatrix_Frustum(-xr,xr,-xr/aspect,xr/aspect,n_dist,5000) );
  // pMatrix_Frustum: left, right, bottom, top, near, far

  vh.clearValues[0].color =
    vk::ClearColorValue( array<float, 4>( { { 0, 0, 0, 0.5 } } ) );

  /// Lighting
```

```
  //
  auto& light_all = uni_light_all->cgl_LightSource[0];

  light_all.position = transform.eye_from_global * light_location;

  light_all.constantAttenuation = 0.5;
  light_all.linearAttenuation = 1.0;
  light_all.quadraticAttenuation = 0;

  pColor ambient_color(0x555555);

  uni_light_all->cgl_LightModel.ambient = ambient_color;

  light_all.diffuse = opt_light_intensity * color_white;
  light_all.ambient = color_black;
  light_all.specular = opt_light_intensity * color_white;

  uni_light_reflected = uni_light_all.get_val();
  uni_light_reflected->cgl_LightSource[0].position =
    transform.eye_from_global * pMatrix_Scale(1,-1,1) * light_location;

  uni_light_ambient = uni_light_all.get_val();
  auto& light_amb = uni_light_ambient->cgl_LightSource[0];
  light_amb.diffuse = color_black;
  light_amb.specular = color_black;

  uni_light = uni_light_all.get_val();
  uni_light->cgl_LightModel.ambient = color_black;

  uni_light.to_dev();
  uni_light_ambient.to_dev();
  uni_light_all.to_dev();
  uni_light_reflected.to_dev();

  puni_light_curr = &uni_light_all;

  buf_uni_common->tryout =
    ivec4(opt_tryout1,opt_tryout2,opt_normal_sphere,false);
  buf_uni_common->tryoutf = vec4(opt_tryoutf, 0, 0, 0);
  buf_uni_common->light_pos_g = light_location;
  if ( do_raytrace )
    {
      raytrace.buf_uni_common->opt_tryout = buf_uni_common->tryout;
      raytrace.buf_uni_common->opt_tryoutf = buf_uni_common->tryoutf;
      raytrace.buf_uni_common->opt_shadows = opt_shadows;
      raytrace.buf_uni_common->opt_mirror =
        opt_mirror ? opt_mirror_depth : 0;
      raytrace.buf_light_set(uni_light_all);
    }

  const double time_now = time_wall_fp();
  const bool blink_visible = int64_t(time_now*3) & 1;
# define BLINK(txt,pad) ( blink_visible ? txt : pad )
```

```
  vh.fbprintf("%s\n",frame_timer.frame_rate_text_get());

  vh.fbprintf
    ("Raytrace: %s ('R')   Compiled: %s\n",
     !vh.have_raytrace ? "NA" :
     opt_want_raytrace ? ( raytrace.rt_ready ? "ON " : "???" ) : "OFF",
#ifdef __OPTIMIZE__
     "WITH OPTIMIZATION"
#else
     BLINK("WITHOUT OPTIMIZATION","")
#endif
     );

  vh.fbprintf
    ("Time Step: %8d  World Time: %11.6f  %s\n",
     time_step_count, world_time,
     opt_pause ? BLINK("PAUSED, 'p' to unpause, SPC or S-SPC to step.","") :
     "Press 'p' to pause."
     );

  vh.fbprintf
    ("Eye location: [%5.1f, %5.1f, %5.1f]  "
     "Eye direction: [%+.2f, %+.2f, %+.2f]\n",
     eye_location.x, eye_location.y, eye_location.z,
     eye_direction.x, eye_direction.y, eye_direction.z);

  Ball& ball = *balls[0];

  vh.fbprintf
    ("Head Ball Pos  [%5.1f,%5.1f,%5.1f] Vel [%+5.1f,%+5.1f,%+5.1f]\n",
     ball.position.x,ball.position.y,ball.position.z,
     ball.velocity.x,ball.velocity.y,ball.velocity.z );

  vh.fbprintf
    ("%4zd Balls  %4zd Links  Physics: %s ('a' to change)  \n",
     balls.size(), links.size(), gpu_physics_method_str[opt_physics_method]);

  vh.fbprintf
    ("Hide: %c%c%c ('!@#')  Effect: %c%c ('or')  Sphere: %s  ('z')  "
     "Tryout 1: %s  ('y')  Tryout 2: %s  ('Y')  Normals: %s ('n')\n",
     opt_hide_stuff & OH_Sphere ? 'S' : '_',
     opt_hide_stuff & OH_Links ? 'L' : '_',
     opt_hide_stuff & OH_Platform ? 'P' : '_',
     opt_shadows ? 'S' : '_',
     opt_mirror ? 'M' : '_',
     opt_sphere_true ? "TRUE" : "TRI" ,
     opt_tryout1 ? BLINK("ON ","   ") : "OFF",
     opt_tryout2 ? BLINK("ON ","   ") : "OFF",
     opt_normal_sphere ? "SPHERE" : "TRI");

  pVariable_Control_Elt* const cvar = variable_control.current;
  vh.fbprintf("VAR %s = %.5f  (TAB or '`' to change, +/- to adjust)\n",
                     cvar->name,cvar->get_val());
```

```
vh.fbprintf
  ("HW5: Shader %s ('v')   HW4: Texture %s ('x')\n",
   sh_names[opt_shader], hw4_texture_str[opt_texture]);

// Render Marker for Light Source
//
shapes.record_tetrahedron(cb,transform,light_location,0.15);

//
// Render ball reflection.  (Will be blended with dark tiles.)
//

const bool hide_platform = opt_hide_stuff & OH_Platform;

if ( do_raytrace )
  {
    raytrace.tform_update(transform);
    render_objects(cb,RO_Normally);
    raytrace.buf_uni_common.to_dev();

    return;
  }

buf_uni_common.to_dev();

vk::ColorComponentFlags clr_all
  ( vk::ColorComponentFlagBits::eR | vk::ColorComponentFlagBits::eG |
    vk::ColorComponentFlagBits::eB | vk::ColorComponentFlagBits::eA );

if ( !hide_platform )
  {
    // Save current state of fixed-functionality state.
    //
    ff_state.push();

    if ( opt_mirror )
      {
        /// Mirror Effect
        //
        //  - Step 1:  Write location of mirrors into stencil buffer.
        //
        //  - Step 2a: Prepare a matrix that transforms to mirrored location.
        //  - Step 2b: Set up blending to blend mirrored objects with mirrors.
        //
        //  - Step 3:  Render all objects except mirrors.

        ///  Step 1: Write location of mirrors into stencil buffer.
        //
        //   Write stencil value 2 at location of dark (mirrored) tiles.
        //

        vk::StencilOpState s_op_apply
          ( vk::StencilOp::eReplace, vk::StencilOp::eKeep,
            vk::StencilOp::eKeep, vk::CompareOp::eNever, 2, 2, 2 );
```

```
    // sfail, spass, dfail, compare;  cmask, wmask, ref

    ff_state.pipelineDepthStencilStateCreateInfo_get()
      .setStencilTestEnable(true)
      .setFront(s_op_apply).setBack(s_op_apply);

    // Render mirrored tiles (but only to write stencil buffer).
    //
    pipe_platform_mir.record_draw(cb,bset_platform_mir);

    cb.nextSubpass(vk::SubpassContents::eInline); vh.qs.subpass++;

    // Prepare to write only stenciled locations.
    //
    vk::StencilOpState s_op_use
      ( vk::StencilOp::eKeep, vk::StencilOp::eKeep,
        vk::StencilOp::eKeep, vk::CompareOp::eEqual, 2, 2, 2 );
    // sfail, spass, dfail, compare;  cmask, wmask, ref
    //
    // Stencil test passes if stencil value = 2.
    // Don't change stencil buffer.

    ff_state.pipelineDepthStencilStateCreateInfo_get()
      .setFront(s_op_use).setBack(s_op_use);

    ///   Step 2a: Prepare a matrix that transforms to mirrored location.
    //    Use a transform that reflects objects to other side of platform.
    //
    const pMatrix prev = transform.eye_from_global;
    const pMatrix_Scale reflect(1,-1,1);
    transform.eye_from_global = prev * reflect;

    // Reflected front face should still be treated as the front face.
    //
    ff_state.pipelineRasterizationStateCreateInfo_get()
      .setFrontFace(vk::FrontFace::eClockwise);

    puni_light_curr = &uni_light_reflected;

    /// Step 3: Render all objects.

    render_objects(cb,RO_Mirrored);

    transform.eye_from_global = prev;

    ff_state.pipelineRasterizationStateCreateInfo_get()
      .setFrontFace(vk::FrontFace::eCounterClockwise);
  }

// blend operation, blend factors, blend constants
ff_state.pipelineColorBlendStateCreateInfo_get()
  .setBlendConstants( array<float,4>{.5,.5,.5,.5} );

ff_state.pipelineColorBlendAttachmentState_get() =
```

```
        vk::PipelineColorBlendAttachmentState
        ( true,                  // blendEnable
          vk::BlendFactor::eConstantAlpha,  // srcColorBlendFactor
          vk::BlendFactor::eConstantAlpha,  // dstColorBlendFactor
          vk::BlendOp::eAdd,       // colorBlendOp
          vk::BlendFactor::eOne,   // srcAlphaBlendFactor
          vk::BlendFactor::eZero,  // dstAlphaBlendFactor
          vk::BlendOp::eAdd,       // alphaBlendOp
          clr_all      // colorWriteMask
          );

      puni_light_curr = &uni_light_all;;
      pipe_platform_mir.record_draw(cb,bset_platform_mir);

      ff_state.pop();
    }

  if ( !opt_shadows )
    {
      // Render.
      //
      render_objects(cb,RO_Normally);
    }
  else
    {
      //
      // First pass, render using only ambient light.
      //
      puni_light_curr = &uni_light_ambient;

      // Send balls, tiles, and platform to opengl.
      //
      render_objects(cb,RO_Normally);

      //
      // Second pass, find un-shadowed areas and add on light0.
      //


      /// Write Shadow Volumes
      //
      vk::ClearAttachment clr_att
        ( vk::ImageAspectFlagBits::eStencil, 0,
          vk::ClearDepthStencilValue(0,0) );
      vk::ClearRect clr_rect( vk::Rect2D({}, vh.s_extent), 0, 1 );
      cb.clearAttachments( clr_att, clr_rect );

      ff_state.push();

      vk::StencilOpState s_op_apply_f
        ( vk::StencilOp::eKeep, vk::StencilOp::eIncrementAndWrap,
          vk::StencilOp::eKeep, vk::CompareOp::eAlways, -1, -1, 0 );
      vk::StencilOpState s_op_apply_b
        ( vk::StencilOp::eKeep, vk::StencilOp::eDecrementAndWrap,
```

```
    vk::StencilOp::eKeep, vk::CompareOp::eAlways, -1, -1, 0 );
// sfail, spass, dfail, compare; c-mask, w-mask, ref
// ref is -1 (all 1's) because we want to operate on all bits in
// stencil value.

ff_state.pipelineDepthStencilStateCreateInfo_get()
  .setStencilTestEnable(true)
  .setDepthWriteEnable(false)
  .setFront(s_op_apply_f).setBack(s_op_apply_b);

if ( !opt_shadow_volumes )
  {
    ff_state.pipelineColorBlendAttachmentState_get()
      .setColorWriteMask( {} );
  }
else
  {
    ff_state.pipelineColorBlendAttachmentState_get() =
      vk::PipelineColorBlendAttachmentState
      ( true,                    // blendEnable
        vk::BlendFactor::eConstantAlpha,  // srcColorBlendFactor
        vk::BlendFactor::eConstantAlpha,  // dstColorBlendFactor
        vk::BlendOp::eAdd,       // colorBlendOp
        vk::BlendFactor::eOne,   // srcAlphaBlendFactor
        vk::BlendFactor::eZero,  // dstAlphaBlendFactor
        vk::BlendOp::eAdd,       // alphaBlendOp
        clr_all      // colorWriteMask
        );

    ff_state.pipelineColorBlendStateCreateInfo_get()
      .setBlendConstants( array<float,4>{.5,.5,.5,.5} );

    ff_state.pipelineDepthStencilStateCreateInfo_get()
      .setDepthWriteEnable(true);
  }

// Write stencil with shadow locations based on shadow volumes
// cast by light0 (light_location).  Shadowed locations will
// have a positive stencil value.
//
render_objects(cb,RO_Shadow_Volumes);

cb.nextSubpass(vk::SubpassContents::eInline); vh.qs.subpass++;

// Use stencil test to prevent writes to shadowed areas.
//
vk::StencilOpState s_op_apply_fb
  ( vk::StencilOp::eKeep, vk::StencilOp::eKeep,
    vk::StencilOp::eKeep, vk::CompareOp::eEqual, -1, -1, 0 );
// sfail, spass, dfail, compare;  c-mask, w-mask, ref
//
// Stencil test passes if stencil value == 0.

ff_state.pipelineDepthStencilStateCreateInfo_get()
```

```
        .setFront(s_op_apply_fb).setBack(s_op_apply_fb)
        .setDepthCompareOp(vk::CompareOp::eEqual);

      // Allow pixels to be re-written.
      //
      vk::ColorComponentFlags cmask_all
        ( vk::ColorComponentFlagBits::eR | vk::ColorComponentFlagBits::eG |
          vk::ColorComponentFlagBits::eB | vk::ColorComponentFlagBits::eA );

      ff_state.pipelineColorBlendAttachmentState_get() =
        vk::PipelineColorBlendAttachmentState
        ( true,                   // blendEnable
          vk::BlendFactor::eOne,  // srcColorBlendFactor
          vk::BlendFactor::eOne,  // dstColorBlendFactor
          vk::BlendOp::eAdd,       // colorBlendOp
          vk::BlendFactor::eOne,   // srcAlphaBlendFactor
          vk::BlendFactor::eOne,  // dstAlphaBlendFactor
          vk::BlendOp::eAdd,       // alphaBlendOp
          cmask_all      // colorWriteMask
          );

      puni_light_curr = &uni_light;

      // Render.
      //
      render_objects(cb,RO_Normally);

      ff_state.pop();
    }

  puni_light_curr = &uni_light_all;
}


void
World::cb_keyboard()
{
  const int key = vh.keyboard_key_get();
  if ( !key ) return;
  pVect adjustment(0,0,0);
  pVect user_rot_axis(0,0,0);
  const bool kb_mod_s = vh.keyboard_shift;
  const bool kb_mod_c = vh.keyboard_control;
  const float move_amt = kb_mod_s ? 2.0 : kb_mod_c ? 0.08 : 0.4;

  switch ( key ) {
  case FB_KEY_LEFT: adjustment.x = -move_amt; break;
  case FB_KEY_RIGHT: adjustment.x = move_amt; break;
  case FB_KEY_PAGE_UP: adjustment.y = move_amt; break;
  case FB_KEY_PAGE_DOWN: adjustment.y = -move_amt; break;
  case FB_KEY_DOWN: adjustment.z = move_amt; break;
  case FB_KEY_UP: adjustment.z = -move_amt; break;
  case FB_KEY_DELETE: user_rot_axis.y = 1; break;
  case FB_KEY_INSERT: user_rot_axis.y =  -1; break;
```

```
  case FB_KEY_HOME:
    if ( opt_move_item == MI_Light ) adjustment.x = 1; else user_rot_axis.x = 1;
    break;
  case FB_KEY_END: user_rot_axis.x = -1; break;
  case '1': ball_setup_1(kb_mod_c); break;
  case '2': ball_setup_2(); break;
  case '3': ball_setup_3(); break;
  case '4': ball_setup_4(); break;
  case '5': ball_setup_5(); break;
  case '6': ball_setup_6(); break;
  case 'a':
    opt_physics_method++;
    if ( opt_physics_method == GP_ENUM_SIZE ) opt_physics_method = GP_cpu;
    break;
  case 'b': opt_move_item = MI_Ball; break;
  case 'B': opt_move_item = MI_Ball_V; break;
  case 'e': case 'E': opt_move_item = MI_Eye; break;
  case 'g': case 'G': opt_gravity = !opt_gravity; break;
  case 'h': case 'H': opt_head_lock = !opt_head_lock; break;
  case '!': opt_hide_stuff ^= OH_Sphere; break;
  case '@': opt_hide_stuff ^= OH_Links; break;
  case '#': opt_hide_stuff ^= OH_Platform; break;
  case 't': case 'T': opt_tail_lock = !opt_tail_lock; break;
  case 'l': case 'L': opt_move_item = MI_Light; break;
  case 'r': opt_mirror = !opt_mirror; break;
  case 'R':
    opt_want_raytrace = !opt_want_raytrace;
    world_time_link_update = 0;
    break;
  case 'o': opt_shadows = !opt_shadows; break;
  case 'O':
    opt_shadow_volumes = !opt_shadow_volumes;
    if ( opt_shadow_volumes ) opt_shadows = true;
    break;
  case 'n': case 'N': opt_normal_sphere = !opt_normal_sphere; break;
  case 'p': case 'P': opt_pause = !opt_pause; break;
  case 's': case 'S': balls_stop(); break;
  case 'v': case 'V': opt_shader++;
    if ( opt_shader == SO_ENUM_SIZE ) opt_shader = 0;
    break;
  case 'w': case 'W': balls_twirl(); break;
  case 'x': case 'X':
    uni_hw05_needs_update = true;
    opt_texture++; if ( opt_texture == TX_ENUM_SIZE ) opt_texture = 0;
    break;
  case 'y': opt_tryout1 = !opt_tryout1; break;
  case 'Y': opt_tryout2 = !opt_tryout2; break;
  case 'z': opt_sphere_true = !opt_sphere_true; break;
  case ' ':
    if ( kb_mod_s && kb_mod_c ) opt_single_time_step = 10;
    else if ( kb_mod_s ) opt_single_time_step = 100;
    else if ( kb_mod_c ) opt_single_time_step = 1; else opt_single_frame = true;
    opt_pause = true;
    break;
```

```
case FB_KEY_TAB:
  if ( !kb_mod_s ) { variable_control.switch_var_right(); break; }
case 96: variable_control.switch_var_left(); break;
case '-':case '_': variable_control.adjust_lower(); break;
case '+':case '=': variable_control.adjust_higher(); break;
default: printf("Unknown key, %d\n",key); break;
}

gravity_accel.y = opt_gravity ? -opt_gravity_accel : 0;

lock_update();
sphere.bunch_invalidate();

// Update eye_direction based on keyboard command.
//
if ( user_rot_axis.x || user_rot_axis.y )
  {
    pMatrix_Rotation rotall(eye_direction,pVect(0,0,-1));
    user_rot_axis *= invert(rotall);
    eye_direction *= pMatrix_Rotation(user_rot_axis, M_PI * 0.03);
    modelview_update();
  }

// Update eye_location based on keyboard command.
//
if ( adjustment.x || adjustment.y || adjustment.z )
  {
    const double angle =
      fabs(eye_direction.y) > 0.99
      ? 0 : atan2(eye_direction.x,-eye_direction.z);
    pMatrix_Rotation rotall(pVect(0,1,0),-angle);
    adjustment *= rotall;

    switch ( opt_move_item ){
    case MI_Ball:
      adj_vector = adjustment;
      if ( adj_t_stop == 0 )
        adj_t_prev = adj_t_stop = world_time;
      adj_t_stop += adj_duration;
      break;
    case MI_Ball_V: balls_push(adjustment,0); break;
    case MI_Light:
      if ( key == FB_KEY_HOME )
        {
          pVect l_to_b(light_location,balls[0]->position);
          adjustment = 0.25 * l_to_b;
        }
      light_location += adjustment;
      break;
    case MI_Eye: eye_location += adjustment; break;
    default: break;
    }
    modelview_update();
  }
```

```
}



void
World::init(int argc, char **argv)
{
  chain_length = 14;
  const bool many_vars = false;

  opt_time_step_duration = 0.00003;
  if ( many_vars )
    variable_control.insert(opt_time_step_duration,"Time Step Duration");

  distance_relaxed = 15.0 / chain_length;
  opt_spring_constant = 500;
  variable_control.insert(opt_spring_constant,"Spring Constant");

  opt_spring_damp = 0.7;
  if ( many_vars )
    variable_control.insert(opt_spring_damp,"Spring Damping").set_max(1);

  opt_friction_dynamic = 0.2;
  if ( many_vars )
    variable_control.insert(opt_friction_dynamic,"Dynamic Friction");

  opt_omega = 10;
  if ( many_vars )
    variable_control.insert(opt_omega,"Initial Rotation Rate");

  opt_gravity_accel = 9.8;
  opt_gravity = true;
  gravity_accel = pVect(0,-opt_gravity_accel,0);
  if ( many_vars )
    variable_control.insert(opt_gravity_accel,"Gravity");

  opt_air_resistance = 0.01;
  variable_control.insert(opt_air_resistance,"Air Resistance");

  world_time = 0;
  time_step_count = 0;
  last_frame_wall_time = time_wall_fp();
  frame_timer.work_unit_set("Steps / s");

  world_time_link_update = 0;

  opt_mirror = true;
  opt_mirror_depth = 2;
  if ( many_vars )
    variable_control.insert(opt_mirror_depth,"Mirror Depth",1,0);
  opt_shadows = true;
  opt_shadow_volumes = false;
```

```
  opt_shader = SO_HW05;

  /// Init CUDA

  // Get information about GPU and its ability to run CUDA.
  //
  int device_count;
  cudaGetDeviceCount(&device_count); // Get number of GPUs.
  if ( device_count == 0 )
    {
      fprintf(stderr,"No GPU found, exiting.\n");
      exit(1);
    }

  // Print information about the available GPUs.
  //
  gpu_info_print();

  // Use course library routine to choose a good GPU for CUDA.
  //
  int dev = gpu_choose_index();
  CE(cudaSetDevice(dev));
  printf("Using GPU %d\n",dev);

  gpu_info.get_gpu_info(dev);
  cuda_setup(&gpu_info);

  // Print information about time_step routine.
  //
  printf("\nCUDA Routine Resource Usage:\n");

  int block_size_max = 4096;

  for ( int i=0; i<gpu_info.num_kernels; i++ )
    {
      printf("For %s:\n", gpu_info.ki[i].name);
      printf("  %6zd shared, %zd const, %zd loc, %d regs; "
             "%d max threads per block.\n",
             gpu_info.ki[i].cfa.sharedSizeBytes,
             gpu_info.ki[i].cfa.constSizeBytes,
             gpu_info.ki[i].cfa.localSizeBytes,
             gpu_info.ki[i].cfa.numRegs,
             gpu_info.ki[i].cfa.maxThreadsPerBlock);
      set_min(block_size_max,gpu_info.ki[i].cfa.maxThreadsPerBlock);
    }

  opt_physics_method = GP_cuda;
  gpu_const_data_stale = true;
  cpu_phys_data_stale = false;

  CE( cudaEventCreate(&frame_start_ce) );
  CE( cudaEventCreate(&frame_stop_ce) );

  opt_block_size = 32;
```

```
  if ( many_vars )
    variable_control.insert_power_of_2
      (opt_block_size, "Block Size", 32, block_size_max);

  c.alloc_n_balls = 0;
  c.alloc_n_links = 0;

  init_graphics();

  opt_sphere_slices = 35;
  sphere.opt_texture = true;
  sphere.init(opt_sphere_slices);
  if ( many_vars )
    variable_control.insert(opt_sphere_slices,"Sphere Slices");

  const char* const links_shader_code_path = "links-shdr-links.cc";

  const auto bvec_vs = vk::ShaderStageFlagBits::eVertex;
  const auto bvec_vgs =
    vk::ShaderStageFlagBits::eVertex | vk::ShaderStageFlagBits::eGeometry;

  pipe_set_2
    .init( ff_state )
    .ds_follow( transform )
    .ds_follow( puni_light_curr )
    .ds_uniform_use( "BIND_UNI_COMMON", buf_uni_common )
    .ds_set_material( color_dim_gray, color_salmon )
    .ds_use( sampler, texid_hw )
    .ds_storage_follow( "BIND_LINKS_POS1", lis_pos1, bvec_vs )
    .ds_storage_follow( "BIND_LINKS_POS2", lis_pos2, bvec_vs )
    .ds_storage_follow( "BIND_LINKS_V1", lis_v1, bvec_vs )
    .ds_storage_follow( "BIND_LINKS_V2", lis_v2, bvec_vs )
    .ds_storage_follow( "BIND_LINKS_B1_YDIR", lis_b1_ydir, bvec_vs )
    .ds_storage_follow( "BIND_LINKS_B2_YDIR", lis_b2_ydir, bvec_vs )
    .ds_storage_follow( "BIND_LINKS_TEX_RAD", lis_tex_rad, bvec_vgs )
    .shader_inputs_info_set( )
    .shader_code_set
    (links_shader_code_path,
     "vs_main_2(); ",     // Name of vertex shader main routine.
     "gs_main_2();",      // Name of geometry shader main routine.
     "fs_main();"         // Name of fragment shader main routine.
     )
    .topology_set( vk::PrimitiveTopology::eLineStrip )
    .create();

  if ( vh.have_raytrace )
    comp_links
      .init( vh.qs )
      .ds_uniform_use( "BIND_UNI_COMMON", buf_uni_common )
      .ds_uniform_use( "BIND_TRANSFORM", raytrace.buf_uni_transform )
      .ds_storage_follow( "BIND_LINKS_POS1", lis_pos1 )
      .ds_storage_follow( "BIND_LINKS_POS2", lis_pos2 )
      .ds_storage_follow( "BIND_LINKS_V1", lis_v1 )
```

```
      .ds_storage_follow( "BIND_LINKS_V2", lis_v2 )
      .ds_storage_follow( "BIND_LINKS_B1_YDIR", lis_b1_ydir )
      .ds_storage_follow( "BIND_LINKS_B2_YDIR", lis_b2_ydir )
      .ds_storage_follow( "BIND_LINKS_TEX_RAD", lis_tex_rad )
      .ds_storage_follow( "BIND_LINKS_AS_IDX", buf_links_as_idx )
      .ds_storage_follow( "BIND_LINKS_TEX_COOR", buf_links_tex_coor )
      .ds_storage_follow( "BIND_LINKS_VERTEX_G", buf_links_vertex_g )
      .ds_storage_follow( "BIND_LINKS_NORMAL_G", buf_links_normal_g )
      .shader_code_set(links_shader_code_path, "comp_main(); ")
      .create();

  pipe_set_2_sv
    .init( ff_state ).ds_use( uni_light )
    .ds_follow( transform )
    .ds_uniform_use( "BIND_UNI_COMMON", buf_uni_common )
    .ds_storage_follow( "BIND_LINKS_POS1", lis_pos1 )
    .ds_storage_follow( "BIND_LINKS_POS2", lis_pos2 )
    .ds_storage_follow( "BIND_LINKS_V1", lis_v1 )
    .ds_storage_follow( "BIND_LINKS_V2", lis_v2 )
    .ds_storage_follow( "BIND_LINKS_B1_YDIR", lis_b1_ydir )
    .ds_storage_follow( "BIND_LINKS_B2_YDIR", lis_b2_ydir )
    .ds_storage_follow( "BIND_LINKS_TEX_RAD", lis_tex_rad )
    .shader_inputs_info_set( )
    .shader_code_set
    (links_shader_code_path,
     "vs_main_2_sv(); ",     // Name of vertex shader main routine.
     "gs_main_2_sv();",      // Name of geometry shader main routine.
     "fs_main_sv();"         // Name of fragment shader main routine.
     )
    .topology_set( vk::PrimitiveTopology::eLineStrip )
    .create();

  ball_setup_4();
  return;
  // For benchmarking.
  opt_pause = true;
  opt_hide_stuff = OH_Links | OH_Platform;
  opt_shadows = false;
  opt_mirror = false;
}

Ball*
World::make_marker(pCoor position, pColor color)
{
  Ball* const ball = new Ball;
  ball->position = position;
  ball->locked = true;
  ball->velocity = pVect(0,0,0);
  ball->radius = 0.2;
  ball->mass = 0;
  ball->contact = false;
  ball->color = color;
  return ball;
}
```

```
void
World::lock_update()
{
  // This routine called when options like opt_head_lock might have
  // changed.

  // Update locked status.
  //
  if ( head_ball ) head_ball->locked = opt_head_lock;
  if ( tail_ball ) tail_ball->locked = opt_tail_lock;

  // Re-compute minimum mass needed for stability.
  //
  for ( Ball *ball: balls ) ball->spring_constant_sum = 0;
  const double dtis = pow( opt_time_step_duration, 2 );
  for ( Link *link: links )
    {
      Ball* const b1 = link->ball1;
      Ball* const b2 = link->ball2;
      b1->spring_constant_sum += opt_spring_constant;
      b2->spring_constant_sum += opt_spring_constant;
    }
  for ( Ball *ball: balls )
    {
      ball->mass_min = ball->spring_constant_sum * dtis;
      ball->constants_update();
    }
  gpu_const_data_stale = true;
}

void
World::balls_twirl()
{
  if ( !head_ball || !tail_ball ) return;

  pNorm axis(head_ball->position, tail_ball->position);
  const float omega = 0.1 * opt_omega;

  for ( Ball *ball: balls )
    {
      pVect b_to_top(ball->position,head_ball->position);
      const float dist_along_axis = dot(b_to_top,axis);
      const float lsq = b_to_top.mag_sq() - dist_along_axis * dist_along_axis;
      ball->omega += omega * axis;
      if ( lsq <= 1e-5 ) continue;
      const float dist_to_axis = sqrt(lsq);
      pNorm rot_dir = cross(b_to_top,axis);
      ball->velocity += omega * dist_to_axis * rot_dir;
    }
}

void
World::objects_erase()
```

```
{
  balls.erase();
  links.erase();
  surfaces.clear();

  hw05_stuff.balls_inner.clear();
  hw05_stuff.balls_outer.clear();
  hw05_stuff.spokes.clear();
}

Links
World::link_new
(Ball *ball1, Ball *ball2, float stiffness, float spring_constant_p)
{
  Links links_rv;
  assert( ball1->radius > 0 );
  assert( ball2->radius > 0 );
  const float spring_constant = spring_constant_p ?: opt_spring_constant;

  Link* const rlink = new Link(ball1,ball2);
  links_rv += rlink;

  pNorm n12(ball1->position,ball2->position);
  const float rad_sum = ball1->radius + ball2->radius;
  pMatrix3x3 b1rot = ball1->omatrix;
  pMatrix3x3 b2rot = ball2->omatrix;
  pCoor ctr = ball1->position
    + ( ball1->radius + 0.5 * ( n12.magnitude - rad_sum ) ) * n12;

  pVect con_x = rlink->con_x;
  pVect con_y = rlink->con_y;

  const float lrad = stiffness * ball1->radius;

  const int n_springs = 4;
  const double delta_theta = 2 * M_PI / n_springs;

  for ( int i=0; i<n_springs; i++ )
    {
      const double theta = i * delta_theta;
      pVect convec = lrad * ( cos(theta) * con_x + sin(theta) * con_y );
      pCoor con = ctr + convec;
      pVect cb1 = mult_MTV( b1rot, con - ball1->position );
      pVect cb2 = mult_MTV( b2rot, con - ball2->position );
      links_rv +=
        new Link(rlink, cb1, cb2,
                 { .distance_relaxed=0, .spring_constant=spring_constant } );
    }
  return links_rv;
}

///
/// CUDA Code
///
```

```
float4 tof4(pCoor c)
{
  float4 f4;
  f4.x = c.x;
  f4.y = c.y;
  f4.z = c.z;
  f4.w = c.w;
  return f4;
}

float4 tof4(pVect c)
{
  float4 f4;
  f4.x = c.x;
  f4.y = c.y;
  f4.z = c.z;
  f4.w = 0;
  return f4;
}

pCoor topc(float4 c)
{
  pCoor f4;
  f4.x = c.x;
  f4.y = c.y;
  f4.z = c.z;
  f4.w = c.w;
  return f4;
}

void
World::data_cpu_to_gpu_constants()
{
  c.platform_xmin = platform_xmin;
  c.platform_xmax = platform_xmax;
  c.platform_zmax = platform_zmax;
  c.platform_zmin = platform_zmin;

  c.n_balls = balls.size();
  c.n_links = links.size();
  c.n_surfaces = surfaces.size();
  c.opt_tryout1 = opt_tryout1;
  c.opt_tryout2 = opt_tryout2;
  c.opt_tryoutf = opt_tryoutf;
  c.opt_spring_constant = opt_spring_constant;
  c.opt_spring_damp = opt_spring_damp;
  c.opt_air_resistance = opt_air_resistance;
  c.opt_friction_dynamic = opt_friction_dynamic;
  c.opt_head_lock = opt_head_lock;
  c.opt_tail_lock = opt_tail_lock;
  c.gravity_accel = gravity_accel;

  data_cpu_to_gpu_common(&c);
```

```
}


void
World::data_cpu_to_gpu_dynamic()
{
  const int n_balls = balls.size();
  const int n_links = links.size();
  const int n_surfaces = surfaces.size();
  const bool need_alloc =
    n_balls > c.alloc_n_balls || n_links > c.alloc_n_links
    || n_surfaces > c.alloc_n_surfaces;
  const bool need_init = c.alloc_n_balls == 0;

#define ALLOC_MOVE_ZERO(n_balls,balls,memb,do_move,do_zero)                  \
  {                                                                          \
    const int size_elt_bytes = sizeof(c.balls.memb[0]);                      \
    const int size_bytes = n_balls * size_elt_bytes;                         \
    if ( need_alloc )                                                        \
      {                                                                      \
        if ( !need_init )                                                    \
          {                                                                  \
            CE( cudaFree( c.balls.memb ) );                                  \
            free(c.h_##balls.memb);                                          \
          }                                                                  \
        CE( cudaMalloc( &c.balls.memb, size_bytes ) );                       \
        c.h_##balls.memb = (decltype(c.balls.memb)) malloc( size_bytes );    \
      }                                                                      \
    if ( do_move )                                                           \
      {                                                                      \
        for ( int i=0; i<n_balls; i++ )                                      \
          c.h_##balls.memb[i] = balls[i]->memb;                              \
        CE( cudaMemcpy                                                       \
            ( c.balls.memb, c.h_##balls.memb,                                \
              size_bytes, cudaMemcpyDefault ) );                             \
      }                                                                      \
    else if ( do_zero )                                                      \
      {                                                                      \
        CE( cudaMemset( c.balls.memb, 0, size_bytes ) );                     \
      }                                                                      \
  }

#define MOVEc(num,ptr,memb) \
  ALLOC_MOVE_ZERO(num,ptr,memb,gpu_const_data_stale,0)
#define MOVE(num,ptr,memb) ALLOC_MOVE_ZERO(num,ptr,memb,1,0)
#define ALLOC(num,ptr,memb) ALLOC_MOVE_ZERO(num,ptr,memb,0,0)
#define ZERO(num,ptr,memb) ALLOC_MOVE_ZERO(num,ptr,memb,0,1)
  MOVEc(n_balls,balls,position);
  MOVEc(n_balls,balls,velocity);
  ZERO(n_balls,balls,force);

  MOVEc(n_balls,balls,orientation);
  MOVEc(n_balls,balls,omatrix);
  MOVEc(n_balls,balls,omega);
```

```
  ZERO(n_balls,balls,torque);
  MOVEc(n_balls,balls,m_r_fdt);
  MOVEc(n_balls,balls,locked);

  MOVEc(n_links,links,ball_idx);
  MOVEc(n_links,links,cb1);
  MOVEc(n_links,links,cb2);
  MOVEc(n_links,links,dr_sc);
  MOVEc(n_links,links,is_simulatable);
  MOVEc(n_links,links,is_surface_connection);

  MOVEc(n_surfaces,surfaces,b_idx);

#undef MOVE
#undef MOVEc
#undef ALLOC
#undef ZERO
#undef ALLOC_MOVE
  if ( need_alloc )
    {
      c.alloc_n_balls = n_balls;
      c.alloc_n_links = n_links;
      c.alloc_n_surfaces = n_surfaces;
      data_cpu_to_gpu_common(&c);
    }
  gpu_const_data_stale = false;
}

void
World::data_gpu_to_cpu_dynamic()
{
  const int n_balls = balls.size();

#define MOVE(n_balls,balls,memb)                                        \
  {                                                                     \
    const int size_elt_fr_bytes = sizeof(c.balls.memb[0]);              \
    const int size_bytes = n_balls * size_elt_fr_bytes;                 \
    CE( cudaMemcpy                                                      \
        ( c.h_##balls.memb, c.balls.memb, size_bytes,                   \
          cudaMemcpyDeviceToHost ) );                                   \
    for ( int i=0; i<n_balls; i++ )                                     \
      balls[i]->memb = c.h_##balls.memb[i];                             \
  }

  MOVE(n_balls,balls,position);
  MOVE(n_balls,balls,velocity);
  MOVE(n_balls,balls,orientation);
  MOVE(n_balls,balls,omatrix);
  MOVE(n_balls,balls,omega);

#undef MOVE

}
```

```
///
/// Physical Simulation Code
///

 /// Initialize Simulation
//

void
World::ball_setup_1(bool big)
{
  // Arrange balls to form sort of tree.

  last_setup = 1;

  double ball_rad = 1;  // Warning: Re-assigned, changes behavior of nb.

  pCoor first_pos(17.2,ball_rad,-20.2);

  pVect dir_up(0.1,3,0.1);
  pVect dir_z(3*distance_relaxed,0,0);
  pVect dir_x(0,0,3*distance_relaxed);

  // Remove objects from the simulated objects lists, balls and links.
  // The delete operator is used on objects in the lists.
  //
  objects_erase();

  auto nb =
    [&] (pCoor pos) {
      Ball* const ball = new Ball;
      ball->position = pos;
      ball->locked = false;
      ball->velocity = pVect(0,0,0);
      ball->radius = ball_rad;
      ball->contact = false;
      balls += ball;
      return ball;
      };

  tail_ball = nb(first_pos);
  tail_ball->specularity = 0.5;

  pCoor last_pos = tail_ball->position;

  const int n_tiers = big ? 32 : 8;
  const double shrink_final = 0.17;
  const double shrink = pow(shrink_final,1.0/n_tiers);

  Balls tail_balls;
  for ( int i=1; i<n_tiers; i++ )
    {
      ball_rad *= shrink;
      Ball* const ball = nb(last_pos + (i==1?1.0:ball_rad) * dir_up);
```

```
      ball->specularity = 0.5;
      tail_balls += ball;
      links += link_new(balls[balls-2],ball,0.8);
      last_pos = ball->position;
    }

  head_ball = balls[balls-1];

  pNorm dir_zn(dir_z);
  pNorm dir_xn(dir_x);
  for ( Ball *ball: tail_balls )
    {
      if ( ball == head_ball ) break;
      pNorm to_top(ball->position,head_ball->position);
      const float r = to_top.magnitude;
      int num_hairs = max(4,int(7*r));
      ball_rad = ball->radius / 4;
      const float delta_theta = 2 * M_PI / num_hairs;
      for ( int i=0; i<num_hairs; i++ )
        {
          const float theta = i * delta_theta;
          pCoor pos = ball->position +
            r * cos(theta) * dir_xn + r * sin(theta) * dir_zn;
          Ball* const ballr = nb(pos);
          links += link_new(ballr,ball);
        }
    }

  opt_head_lock = false;
  opt_tail_lock = true;
  setup_at_end();
}

void
World::ball_setup_2()
{
  // Arrange balls to form a suspended thing.

  last_setup = 2;

  pCoor first_pos(17.2,14.8f,-20.2);

  pVect dir_dn(0.5,first_pos.y/8,0.5);
  pVect dir_z(3*distance_relaxed,0,0);
  pVect dir_x(0,0,3*distance_relaxed);

  // Remove objects from the simulated objects lists, balls and links.
  // The delete operator is used on objects in the lists.
  //
  objects_erase();

  float ball_rad = 0.3;  // Warning: Re-assigned, changes behavior of nb.

  auto nb =
```

```
    [&] (pCoor pos) {
      Ball* const ball = new Ball;
      ball->position = pos;
      ball->locked = false;
      ball->velocity = pVect(0,0,0);
      ball->radius = ball_rad;
      ball->contact = false;
      ball->specularity = 0.5;
      balls += ball;
      return ball;
      };

  head_ball = nb(first_pos);
  for ( int j: {-1,1} )
    for ( int i: {-1,0,1} )
      {
        Ball* const ball = nb(first_pos - dir_dn + i * dir_x + j * dir_z );
        if ( i != 0 ) links += link_new(head_ball,ball);
        if ( i != -1 ) links += link_new(balls[balls-2],ball);
        if ( j == 1 && i != 0 )  links += link_new(balls[balls-4],ball);
      }

  Ball* const tail_start = nb(first_pos - dir_dn);
  links += link_new(balls[balls-6],tail_start);
  links += link_new(balls[balls-3],tail_start);
  pCoor last_pos = tail_start->position;
  Balls tail_balls;
  for ( int i=1; i<5; i++ )
    {
      Ball* const ball = nb(last_pos - 0.5 * dir_dn);
      tail_balls += ball;
      links += link_new(balls[balls-2],ball);
      last_pos = ball->position;
    }

  tail_ball = balls[balls-1];

  pNorm dir_zn(dir_z);
  pNorm dir_xn(dir_x);
  int num_hairs = 20;
  ball_rad = 0.1;
  for ( Ball *ball: tail_balls )
    {
      const float delta_theta = 2 * M_PI / num_hairs;
      for ( int i=0; i<num_hairs; i++ )
        {
          const float theta = i * delta_theta;
          const float r = ball->radius * 10;
          pCoor pos = ball->position +
            r * cos(theta) * dir_xn + r * sin(theta) * dir_zn;
          Ball* const ballr = nb(pos);
          links += link_new(ballr,ball);
        }
    }
```

```
  opt_head_lock = true;    // Head ball will be frozen in space.
  opt_tail_lock = false;   // Tail ball can move freely.
  setup_at_end();
}

void
World::ball_setup_3()
{
  // Arrange links to form what I would eventually learn to be a
  // corona virus.

  const bool sarcov = true;

  last_setup = 3;
  objects_erase();

  pCoor first_pos(17.2,14.8f,-10.2);

  float ball_radius = 1;

  auto nb =
    [&] (pCoor pos) {
      Ball* const ball = new Ball;
      ball->position = pos;
      ball->locked = false;
      ball->velocity = pVect(0,0,0);
      ball->radius = ball_radius;
      ball->contact = false;
      balls += ball;
      return ball;
      };

  head_ball = nb(first_pos);
  head_ball->specularity = 0.5;
  ball_radius = ball_radius * ( sarcov ? 0.08 : .1 );
  const int n_spikes = sarcov ? 100 : 500;

  for ( int i=0; i<n_spikes; i++ )
    {
      pNorm dir(0.5-drand48(),0.5-drand48(),0.5-drand48());
      const float len = head_ball->radius * ( sarcov ? 1.5 : 3.0 );
      Ball* const ball = nb(first_pos + len * dir);
      links += link_new(ball,head_ball,2);
    }

  tail_ball = balls[1];

  opt_head_lock = false;    // Head ball will be frozen in space.
  opt_tail_lock = false;   // Tail ball can move freely.

  setup_at_end();
}
```

```
pVect vec_rand() {return pVect(drand48(),drand48(),drand48());}

void
World::ball_setup_4()
{
  // Render a wheel (disk of balls with one in the center).

  last_setup = 4;
  objects_erase();

  pCoor center_pos = pCoor(17.2,14.8f,-10.2) + 2*vec_rand();

  const int n_balls = 4 * ( 4 + random()%4 );
  float ring_outer_radius = 5 + 2 * drand48();
  float ring_inner_radius = ring_outer_radius * ( 0.1 + drand48() * 0.4 );
  float ball_radius = 0.2 * 2 * M_PI * ring_outer_radius / n_balls;

  pNorm c_to_eye(center_pos,eye_location);
  pNorm az = c_to_eye + 0.5 * vec_rand() + pVect(0,0.5,0);

  pNorm ax = fabs(az.x) < fabs(az.y)
    ? pVect(0,az.z,-az.y) : pVect(az.z,0,-az.x);
  pVect ay(az,ax);

  // Default settings for balls.
  //
  Ball ballo_default;
  ballo_default.specularity = 0.5;
  ballo_default.velocity = pVect(0,0,0);
  ballo_default.radius = ball_radius;
  ballo_default.color = color_light_slate_gray;
  Ball balli_default = ballo_default;
  balli_default.radius = ballo_default.radius * 0.6;

  const float d_theta = 2 * M_PI / n_balls;

  const float link_stiffness = 10;

  auto& balls_outer = hw05_stuff.balls_outer;
  auto& balls_inner = hw05_stuff.balls_inner;

  // Place balls in a circle.
  //
  for ( int i=0; i<n_balls; i++ )
    {
      const float theta = d_theta * i;
      pVect pos_n = cosf(theta) * ax + sinf(theta) * ay;
      pCoor pos_o = center_pos + ring_outer_radius * pos_n;
      pCoor pos_i = center_pos + ring_inner_radius * pos_n;

      // Construct new ball and add it to the ball list.
      //
      Ball* ball_o = balls_outer.emplace_back( new Ball(pos_o,ballo_default) );
      Ball* ball_i = balls_inner.emplace_back( new Ball(pos_i,balli_default) );
```

```
    Link *link =
      new Link( ball_o, ball_i,
                { .spring_constant = opt_spring_constant/200 } );
    links.push_back(link);
    hw05_stuff.spokes.push_back(link);

    // Compute a 2D unit vector from the wheel center to the current ball.
    //
    glm::vec2 loc_n = glm::vec2( cosf(theta), -sinf(theta) );

    glm::vec2 ctr_t(0.5,0.5);

    glm::vec2 loc_h = 0.5 * loc_n;
    float ring_io = ring_inner_radius / ring_outer_radius;

    ball_o->tc.square = glm::vec2(0.5,0.5) + loc_h;
    ball_i->tc.square = glm::vec2(0.5,0.5) + ring_io * loc_h;

    const float delta_r = ring_outer_radius - ring_inner_radius;
    const float r_mid = ring_inner_radius + delta_r/2;
    uni_hw05->ring_hole_frac = ring_io;
    uni_hw05->one_over_delta_r = 1/(1-ring_io);
    uni_hw05->dy_d_theta = 0.5 * delta_r / r_mid;
    uni_hw05_needs_update = true;

    ball_o->tc.circle = loc_n;
    ball_i->tc.circle = ring_io * loc_n;


    // Set color of first ball to blue, others to white.
    ball_i->color = ball_o->color =
      i == 0 ? color_blue :
      theta <= M_PI/2 ? color_lsu_spirit_gold :
      color_white;
  }

// Connect each ball to its neighbor with a link.
//
for ( auto bc: { &balls_outer, &balls_inner } )
  for ( int i=0; i<n_balls; i++ )
    {
      balls.push_back( bc[0][i] );
      links += link_new( bc[0][i], bc[0][(i+1)%n_balls], link_stiffness );
    }

for ( int i=0; i<n_balls; i++ )
  {
    const int j = ( i + 1 ) % n_balls;
    Ball* b00 = balls_inner[i];
    Ball* b01 = balls_outer[i];
    Ball* b10 = balls_inner[j];
    Ball* b11 = balls_outer[j];
    surfaces.emplace_back(b00,b01,b11);
    surfaces.emplace_back(b10,b00,b11);
```

```
    }

  pNorm spin_axis = ay;
  pCoor c_of_mass = center_pos;
  float omega_init = opt_omega * 0.1;

  if ( 0 )
  for ( Ball* ball: balls )
    {
      pVect cm_to_ball( c_of_mass, ball->position );

      ball->velocity = omega_init * cross( spin_axis, cm_to_ball );
      ball->omega = omega_init * spin_axis;
    }

  tail_ball = balls.back();
  head_ball = balls.front();

  opt_head_lock = false; // If true, head ball locked in place.
  opt_tail_lock = false;

  setup_at_end();
}

void
World::ball_setup_5()
{
  // Construct a top. (Top, as in a child's toy.)

  last_setup = 5;
  objects_erase();

  pCoor center_pos(17.2,14.8f,-10.2);

  const int n_balls = 20;
  const float crad = 5;

  const float ball_radius = 0.3 * 2 * M_PI * crad / n_balls;

  // Default settings for balls.
  //
  Ball ball_default;
  ball_default.specularity = 0.5;
  ball_default.velocity = pVect(0,0,0);
  ball_default.radius = ball_radius;
  ball_default.color = pColor(0xffbbff); // Plum 1. That's the color's name.

  const float d_theta = 2 * M_PI / n_balls;

  pVect vz = pVect(1,6,1) - 2*vec_rand();
  pNorm ax( vz.y, -vz.x, 0 );
  pNorm ay = cross(vz,ax);
  pVect vx = crad * ax;
  pVect vy = crad * ay;
```

```
pNorm spin_axis = vz;
float omega = opt_omega;
opt_spring_damp = .4;

// Construct Top's Main Ring
//
for ( int i=0; i<n_balls; i++ )
  {
    const float theta = d_theta * i;
    pCoor pos = center_pos + vx * cosf(theta) + vy * sinf(theta);
    Ball* ball = balls.emplace_back( new Ball(pos,ball_default) );

    // Find the coordinate on the spin axis closest to the new ball.
    //
    pCoor axis_cp =
      center_pos + spin_axis * dot( spin_axis, pVect(center_pos,pos) );
    //
    // Use this coordinate to compute a vector giving the ball's
    // motion when the top is spinning around axis spin_axis.
    //
    pVect cp_to_p(axis_cp,pos);
    ball->velocity = omega * cross(spin_axis,cp_to_p);
  }

// Make one of the balls on the top's main ring a different color.
balls[0]->color = color_blue;

// Connect the balls on the main ring with links.
//
for ( int i=0; i<n_balls; i++ )
  links += link_new( balls[i], balls[(i+1)%n_balls], 2 );

// Complete the top with balls above and below the ring, and one in
// the center.
//
pNorm az(vz);
pCoor top_pos = center_pos + az * crad * 1.5;
pCoor bot_pos = center_pos - az * crad * 1.5;

head_ball = balls.emplace_back( new Ball(top_pos,ball_default) );
tail_ball = balls.emplace_back( new Ball(bot_pos,ball_default) );
Ball* ctr_ball = balls.emplace_back( new Ball(center_pos,ball_default) );

for ( int i=0; i<n_balls; i += 2 )
  {
    links += link_new( head_ball, balls[i], 2 );
    links += link_new( tail_ball, balls[i], 2 );
  }

for ( int i=0; i<n_balls; i++ )
  links += link_new( ctr_ball, balls[i], 2 );

for ( auto b: balls ) b->omega = omega * spin_axis;
```

```
  opt_head_lock = false;    // Head ball will be frozen in space.
  opt_tail_lock = false;   // Tail ball can move freely.

  setup_at_end();
}

void
World::ball_setup_6()
{
  last_setup = 6;
}


void
World::setup_at_end()
{
  lock_update();
  time_step_count = 0;
  world_time = 0;
  world_time_link_update = -1;
  for ( uint i=0; i<balls.size(); i++ ) balls[i]->idx = i;
  for ( Link *l : links )
    { l->ball1_idx = l->ball1->idx; l->ball2_idx = l->ball2->idx; }
  for ( auto& s: surfaces )
    { s.b0_idx = s.b0->idx;  s.b1_idx = s.b1->idx;  s.b2_idx = s.b2->idx; }
}


 /// Advance Simulation State by delta_t Seconds
//
void
World::time_step_cpu(double delta_t)
{
  const int n_balls = balls;
  const int n_links = links;

  // Smoothly move ball in response to user input.
  //
  if ( adj_t_stop )
    {
      const double dt = min(world_time,adj_t_stop) - adj_t_prev;
      pVect adj = dt/adj_duration * adj_vector;
      balls_translate(adj,0);
      adj_t_prev = world_time;
      if ( world_time >= adj_t_stop ) adj_t_stop = 0;
    }

#pragma omp parallel for
  for ( int bi=0; bi<n_balls; bi++ )
    {
      Ball* const ball = balls[bi];
      assert( ball->fdt_to_do );
      ball->force = ball->mass * gravity_accel;
```

```
      ball->torque = pVect(0,0,0);
    }

  const int n_surfaces = surfaces.size();
  for ( int i=0; i<n_surfaces; i++ )
    {
      Surface& s = surfaces[i];
      Ball& b0 = *s.b0;
      Ball& b1 = *s.b1;
      Ball& b2 = *s.b2;
      pNorm snorm = cross(b0.position,b1.position,b2.position);
      pVect vel = (1.0/3) * ( b0.velocity + b1.velocity + b2.velocity );
      float speed = dot(snorm,vel);
      float fmag =
        -speed * fabs(speed) * snorm.magnitude * 0.5f * opt_air_resistance;
      pVect f = fmag * snorm;
      b0.force += f;
      b1.force += f;
      b2.force += f;
    }

#pragma omp parallel for
  for ( int i=0; i<n_links; i++ )
    {
      Link* const link = links[i];
      if ( !link->is_simulatable ) continue;

      //
      // Spring Force from Neighbor Balls
      //

      Ball* const ball1 = link->ball1;
      Ball* const ball2 = link->ball2;

      /// Notes:
      //
      // link->cb1
      //   Coordinate where link touches ball1, in ball1's local space.
      //
      // ball1->omatrix
      //   Ball's orientation transformation ..
      //   .. which transforms vectors in ball1's local space to object space.
      //

      // Find position and velocity of the point where the link touches
      // the surface of ball 1 ...
      //
      pVect dir1 = ball1->omatrix * link->cb1;
      pCoor pos1 = ball1->position + dir1;
      pVect vel1 = ball1->velocity + ball1->point_rot_vel(dir1);

      // ... and ball 2.
      //
      pVect dir2 = ball2->omatrix * link->cb2;
```

```
    pCoor pos2 = ball2->position + dir2;
    pVect vel2 = ball2->velocity + ball2->point_rot_vel(dir2);

    // Construct a normalized (Unit) Vector from ball to neighbor
    // based on link connection points and ball centers.
    //
    pNorm link_dir(pos1,pos2);
    pNorm c_to_c(ball1->position,ball2->position);

    const float link_length = link_dir.magnitude;

    // Compute the speed of ball's end of link towards neighbor's end of link.
    //
    pVect delta_v = vel2 - vel1;
    float delta_s = dot( delta_v, link_dir );

    // Compute by how much the spring is stretched (positive value)
    // or compressed (negative value).
    //
    const float spring_stretch = link_length - link->distance_relaxed;

    // Determine whether spring is gaining energy (whether its length
    // is getting further from its relaxed length).
    //
    const bool gaining_e = ( delta_s > 0.0 ) == ( spring_stretch > 0 );

    const float blend_length_factor = 0.1; // Just a guess.
    const float blend_s = blend_length_factor * c_to_c.magnitude;
    const float blend = min( 1.0f, fabs(delta_s) / blend_s );

    // Use a smaller spring constant when spring is loosing energy,
    // a quick and dirty way of simulating energy loss due to spring
    // friction.
    //
    const float sc_base = link->spring_constant ?: opt_spring_constant;
    const float spring_constant =
      gaining_e ? sc_base :
      sc_base * ( 1.0f - blend + blend * opt_spring_damp );

    const float force_mag = spring_constant * spring_stretch;
    pVect spring_force_12 = force_mag * link_dir;

    // Apply forces affecting linear momentum.
    //
    link->spring_force_12 = spring_force_12;

    if ( ! link->is_surface_connection ) continue;

    // Apply torque.
    //
    link->torque1 = cross(pNorm(dir1), spring_force_12);
    link->torque2 = cross(pNorm(dir2), spring_force_12);
  }
```

```
  // Note: Because two links can reference the same ball this should
  // not be done in parallel.
  for ( Link *link: links )
    {
      if ( !link->is_simulatable ) continue;
      Ball* const ball1 = link->ball1;
      Ball* const ball2 = link->ball2;
      ball1->force += link->spring_force_12;
      ball2->force -= link->spring_force_12;
      ball1->torque += link->torque1;
      ball2->torque -= link->torque2;
    }

  ///
  /// Update Position of Each Ball
  ///

#pragma omp parallel for
  for ( int bi=0; bi<n_balls; bi++ )
    {
      Ball* const ball = balls[bi];
      if ( ball->locked )
        {
          ball->velocity = pVect(0,0,0);
          ball->omega = pVect(0,0,0);
          continue;
        }

      // Update Velocity
      //
      // This code assumes that force on ball is constant over time
      // step. This is clearly wrong when balls are moving with
      // respect to each other because the springs are changing
      // length. This inaccuracy will make the simulation unstable
      // when spring constant is large for the time step.
      //
      const float mass = max( ball->mass, ball->mass_min );

      pVect delta_v = ( ball->force / mass ) * delta_t;
      const float r = ball->radius;

      if ( platform_collision_possible(ball->position)
           && fabs(ball->position.y) < r )
        {
          const bool above = ball->position.y >= 0;
          const float spring_constant_plat =
            ball->velocity.y < 0 == above ? 100000 : 50000;
          const float fric_coefficient = opt_friction_dynamic;
          const float force_away = ( above ? 1 : -1 )
            * ( r - fabs(ball->position.y) ) * spring_constant_plat;
          const float delta_v_away = force_away / mass * delta_t;
          const float fric_force_mag = fric_coefficient * force_away;
          pNorm surface_v(ball->velocity.x,0,ball->velocity.z);
          const float delta_v_surf = fric_force_mag / mass * delta_t;
```

```
          if ( delta_v_surf > surface_v.magnitude )
            {
              // Ignoring other forces?
              delta_v =
                pVect(-ball->velocity.x,delta_v.y,-ball->velocity.z);
            }
          else
            {
              delta_v -= delta_v_surf * surface_v;
            }
          delta_v.y += delta_v_away;
        }

      ball->velocity += delta_v;

      // Air Resistance
      //
      const pNorm ball_dir(ball->velocity);
      const float delta_s_air_raw =
        opt_air_resistance * ball_dir.mag_sq * delta_t;
      const float delta_s_air = min( ball_dir.magnitude, delta_s_air_raw );
      ball->velocity -= delta_s_air * ball_dir;

      // Update Position
      //
      // Assume that velocity is constant.
      //
      ball->position += ball->velocity * delta_t;

      ball->omega += delta_t * ball->fdt_to_do * ball->torque;

      pNorm axis(ball->omega);

      // Update Orientation
      //
      // If ball isn't spinning fast skip expensive rotation.
      //
      if ( axis.mag_sq < 0.000001 ) continue;

      // Update ball orientation.
      //
      ball->orientation_set
        ( pQuat(axis,delta_t * axis.magnitude) * ball->orientation );
    }
}

bool
World::platform_collision_possible(pCoor pos)
{
  // Assuming no motion in x or z axes.
  //
  return pos.x >= platform_xmin && pos.x <= platform_xmax
    && pos.z >= platform_zmin && pos.z <= platform_zmax;
```

```
}

 /// External Modifications to State
//
//   These allow the user to play with state while simulation
//   running.

// Move the ball.
//
void Ball::translate(pVect amt) {position += amt;}

// Add velocity to the ball.
//
void Ball::push(pVect amt) {velocity += amt;}

// Set the velocity to zero.
//
void Ball::stop() {velocity = pVect(0,0,0); }

// Set the velocity and rotation (not yet supported) to zero.
//
void Ball::freeze() {velocity = pVect(0,0,0); }



void World::balls_translate(pVect amt,int b){head_ball->translate(amt);}
void World::balls_push(pVect amt,int b){head_ball->push(amt);}
void World::balls_translate(pVect amt)
{ for ( Ball *ball: balls ) ball->translate(amt); }
void World::balls_push(pVect amt)
{ for ( Ball *ball: balls ) ball->push(amt); }
void World::balls_stop()
{ for ( Ball *ball: balls ) ball->stop(); }
void World::balls_freeze(){balls_stop();}


void
World::render_link_start()
{
  if ( world_time_link_update == world_time ) return;

  // Empty the containers in preparation for a new pass.
  for ( auto b: lis_bufs ) b->clear();
}

void
World::render_link_gather(Link *link)
{
  // Place data needed to render link (the argument to this function)
  // in arrays. Just before rendering those arrays will be placed in
  // buffer objects.
```

```
// If we've already collected links, just return.  This happens
// when doing multiple passes to render shadows and reflections.
//
if ( world_time_link_update == world_time ) return;

Ball *const ball1 = link->ball1;
Ball *const ball2 = link->ball2;

// Direction of link relative to center off ball.
//
pVect dir1 = ball1->omatrix * link->cb1;

// Place on surface of ball where link connects.
//
pCoor pos1 = ball1->position + dir1;

// Direction and connection location for ball 2.
//
pVect dir2 = ball2->omatrix * link->cb2;
pCoor pos2 = ball2->position + dir2;

pVect p1p2(pos1,pos2);
pNorm p1p2n(p1p2);

// Radius of link.
//
const float rad = ball1->radius * 0.3;

// Compute scale factors for texture.
const float tex_aspect_ratio = 8.5 / 11;  // Yes, inches. Can't remember mm.
const float page_width_o = 2.5;  // Width of texture in object-space coords.
const float tex_t_scale = link->distance_relaxed / page_width_o;
const float tex_angle_scale = rad * tex_aspect_ratio / page_width_o;

pNorm dirn1(dir1);
pNorm dirn2(dir2);

// Vectors used to describe the cubic Hermite curve.
//
pVect4 v1 = p1p2n.magnitude * dirn1;
pVect4 v2 = p1p2n.magnitude * dirn2;

// Convert link's local x and y axes to global coordinates.
//
pVect4 b1_ydir = ball1->omatrix * link->b1_dir;
pVect4 b2_ydir = ball2->omatrix * link->b2_dir;


// Place data needed to render links in lists. Note that
// the += operator appends the right-hand side to the container
// on the left-hand side.
//
link->idx = lis_pos1.size();
lis_pos1 << pos1;
```

```
  lis_pos2 << pos2;
  lis_v1 << v1;
  lis_v2 << v2;
  lis_b1_ydir << b1_ydir;
  lis_b2_ydir << b2_ydir;
  lis_tex_rad << vec4( tex_t_scale, tex_angle_scale, rad, 0);
}



void
World::render_link_render(vk::CommandBuffer& cb, bool shadow_volumes)
{
  // Perform a rendering pass to render all the links collected
  // by the routine render_link_2.


  // Check whether this is the first time we are rendering this data.
  //
  const bool first_render =
    world_time_link_update != world_time
    || opt_segments != buf_uni_common->links_n_segs
    || opt_sides != buf_uni_common->sides;
  world_time_link_update = world_time;
  const size_t n_instances = lis_pos1.size();
  const bool do_rt = vh.raytrace_active();
  const int segments = shadow_volumes ? max(2,opt_segments/4) : opt_segments;
  const int segments_sv = max(2,opt_segments/4);

  if ( !n_instances ) return;

  if ( first_render )
    {
      for ( auto b: lis_bufs ) b->to_dev();
      buf_uni_common->sides = opt_sides;
      buf_uni_common->links_n_segs_sv = segments_sv;
      buf_uni_common->delta_tee_sv = 1.0f / segments_sv;
      buf_uni_common->links_n_segs = opt_segments;
      buf_uni_common->delta_tee = 1.0f / opt_segments;
    }

  if ( do_rt && first_render )
    {
      const int n_tri_p_segment = 2 * opt_sides;
      const int n_tri_p_link = n_tri_p_segment * opt_segments;
      const int n_tri_p_pass = n_tri_p_link * n_instances;

      const int n_vtx_p_segment = 2 * ( opt_sides + 1 );
      const int n_vtx_p_link = n_vtx_p_segment * opt_segments;
      const int n_vtx_p_pass = n_vtx_p_link * n_instances;

      if ( !rt_geo_links )
        rt_geo_links
          .init( raytrace )
```

```
          .color_set( color_dim_gray, color_salmon )
          .bind( VR_BUF_IDX_sampler, sampler, texid_hw )
          .grouping_set( RT_Geometry_Indexed );

      buf_links_as_idx.assure(3*n_tri_p_pass);
      buf_links_tex_coor.assure(n_vtx_p_pass);
      buf_links_vertex_g.assure(n_vtx_p_pass);
      buf_links_normal_g.assure(n_vtx_p_pass);

      rt_geo_links
        .bind( VR_BUF_IDX_indices, buf_links_as_idx, 3*n_tri_p_pass )
        .bind( VR_BUF_IDX_pos, buf_links_vertex_g, n_vtx_p_pass )
        .bind( VR_BUF_IDX_normal, buf_links_normal_g )
        .bind( VR_BUF_IDX_tcoor, buf_links_tex_coor );
    }

  if ( do_rt )
    {
      QSet& qs = vh.qs;
      buf_uni_common.to_dev();
      qs.cb.begin( { vk::CommandBufferUsageFlagBits::eOneTimeSubmit } );
      comp_links.record_dispatch(qs.cb,opt_segments,n_instances);
      qs.cb.end();
      qs.q.submit( vk::SubmitInfo{ 0, nullptr, nullptr, 1, &qs.cb }, nullptr );
      qs.q.waitIdle();
      return;
    }


  VPipeline& pipe = shadow_volumes ? pipe_set_2_sv : pipe_set_2;

  pipe.record_draw_instanced(cb, segments+1, n_instances);
}


void
World::frame_callback()
{
  // This routine called whenever window needs to be updated, but
  // before commands are recorded.

  // Get any waiting keyboard commands.
  //
  cb_keyboard();

  if ( vh.have_raytrace ) vh.opt_want_raytrace_now = opt_want_raytrace;

  // This is a hack for 2023 HW 2.
  while ( !balls.empty() && balls.back()->world_time_expire <= world_time )
    balls.pop_back();
  //
  // Balls must be last in list and not part of a link.

  const double time_now = time_wall_fp();
```

```
const int time_step_count_start = time_step_count;
const int nmp = gpu_info.cuda_prop.multiProcessorCount;
int bl_p_sm_max = 1024;
for ( int i=0; i<gpu_info.num_kernels; i++ )
  set_min
    (bl_p_sm_max,
     gpu_info.get_max_active_blocks_per_mp(i, opt_block_size, 0) );
const int grid_size = bl_p_sm_max * nmp;

const bool opt_cuda = opt_physics_method != GP_cpu;
if ( opt_cuda )
  {
    CE( cudaEventRecord(frame_start_ce,0) );
    data_cpu_to_gpu_constants();
    data_cpu_to_gpu_dynamic();
  }

if ( !opt_pause || opt_single_frame || opt_single_time_step )
  {
    /// Advance simulation state.

    // Amount of time since the user saw the last frame.
    //
    const double wall_delta_t = time_now - last_frame_wall_time;

    // Compute amount by which to advance simulation state for this frame.
    //
    const double duration =
      vh.video_frame_duration() ?:
      opt_single_time_step ? opt_single_time_step * opt_time_step_duration :
      opt_single_frame ? 1/60.0 :
      wall_delta_t;

    const double world_time_target = world_time + duration;
    const double wall_time_limit = time_now + 0.05;

    while ( world_time < world_time_target )
      {
        if ( opt_cuda )
          {
            launch_time_step(opt_time_step_duration,grid_size,opt_block_size);
          }
        else
          {
            time_step_cpu(opt_time_step_duration);
          }
        time_step_count++;
        world_time += opt_time_step_duration;
        const double time_right_now = time_wall_fp();
        if ( time_right_now > wall_time_limit ) break;
      }

    // Reset these, just in case they were set.
    //
```

```
      opt_single_frame = false;
      opt_single_time_step = 0;
    }

  if ( opt_cuda )
    {
      data_gpu_to_cpu_dynamic();
      CE( cudaEventRecord(frame_stop_ce,0) );
    }

  frame_timer.phys_end();
  frame_timer.work_amt_set(time_step_count-time_step_count_start);

  float cuda_time = 0;
  if ( opt_cuda )
    {
      CE(cudaEventSynchronize(frame_stop_ce));
      CE(cudaEventElapsedTime(&cuda_time,frame_start_ce,frame_stop_ce));
    }
  frame_timer.cuda_frame_time_set(cuda_time);

  last_frame_wall_time = time_now;

}

int
main(int argv, char **argc)
{
  pVulkan_Helper pvulkan_helper(argv,argc);
  World world(pvulkan_helper,argv,argc);

  world.run();
}
```

# 21 Fall 2022 Solutions

```
/// LSU EE 4702-1 (Fall 2022), GPU Programming
//
//
 /// Homework 1 -- SOLUTION
 //
 //  Assignment: https://www.ece.lsu.edu/koppel/gpup/2022/hw01.pdf
 //
 //  This code is based on cpu-only/demo-01-frame-buffer.cc

#include <gp/coord.h>
#include "frame_buffer.h"

void
line_draw
( pFrame_Buffer& fb, int x0, int y0, int x1, int y1, uint32_t co)
{
  /// Problem 2
  //
  //  Write the frame buffer so that it shows a line from (x0,y0) to
  //  (x1,y1) of color co.
  //
  //  [✔] There should not be gaps between the pixels.
  //  [✔] Try to avoid writing the same pixel multiple times.

  /// SOLUTION -- Problem 2

  const int dx = x1 - x0;
  const int dy = y1 - y0;
  const int width = fb.width_get();

  if ( abs( dx ) < abs( dy ) )
    {
      // Iterate over y

      // Ensure that y0 < y1.
      if ( y0 > y1 ) { swap(x0,x1); swap(y0,y1); }

      float dxdy = float(dx) / dy;
      float x = x0;

      for ( int yi = y0;  yi < y1;  yi++ )
        {
          fb[ yi * width + x ] = co;
          x += dxdy;
        }
    }
  else
    {
      // Iterate over x

      // Ensure that x0 < x1.
      if ( x0 > x1 ) { swap(x0,x1); swap(y0,y1); }
```

```
      float dydx = float(dy) / dx;
      float y = y0;

      for ( int xi = x0;  xi < x1;  xi++ )
        {
          fb[ int(y) * width + xi ] = co;
          y += dydx;
        }
    }
}


void
line_draw_b
( pFrame_Buffer& fb, int x0, int y0, int x1, int y1, uint32_t co)
{
  /// SOLUTION -- Problem 2, Alternative Solution: Use Bresenham's Algorithm
  //
  // The advantage of Bresenham's algorithm is that it draws a line
  // using only integer arithmetic.

  // Width of window.
  const int width = fb.width_get();

  const int dx = abs(x1-x0), dy = abs(y1-y0);
  const bool steep = dy > dx;
  const int inc_x = x1 >= x0 ? 1     : -1;
  const int inc_y = y1 >= y0 ? width : -width;
  int da = dx,       db = dy;
  int a = x0,        b = y0*width;
  int inc_a = inc_x, inc_b = inc_y;
  if ( steep ) { swap(da,db); swap(a,b); swap(inc_a,inc_b); }

  for ( int i = 0,  D = 2*db - da;  i <= da;  i++ )
    {
      fb[ a + b ] = co;
      if ( D > 0 ) { b += inc_b;  D += 2 * ( db - da); } else { D += 2 * db; }
      a += inc_a;
    }
}


void
render_hw01(pFrame_Buffer& fb)
{
  // Window Size, in Pixels
  //
  const int win_width = fb.width_get();
  const int win_height = fb.height_get();

  //                           RRGGBB
  const uint32_t color_red = 0xff0000;

  /// Compute square coordinates based on window size.
```

```
//
const int sq_slen = min(win_width,win_height) * .45;
const int sq_x0 = sq_slen * 0.1,   sq_y0 = win_height - 150;
const int sq_x1 = sq_x0 + sq_slen, sq_y1 = sq_y0 - sq_slen;
//
// One vertex of the square is at (sq_x0,sq_y0) and the opposite
// vertex is at (sq_x1,sq_y1).


/// Problem 1
//
// Modify the code below so that the square properly uses the
// coordinates in variables sq_x0, etc.
//

// Note: This is the original code.
if ( false )
for ( int x=100; x<500; x++ )
  {
    fb[ 100 * win_width + x ] = color_red;
    fb[ 500 * win_width + x ] = 0xffff;
    fb[ x * win_width + 100 ] = 0xff00ff;
    fb[ x * win_width + 500 ] = 0xff00;
  }

/// SOLUTION -- Problem 1
//
// Draw the horizontal lines.
//
for ( int x=sq_x0; x<=sq_x1; x++ )
  {
    fb[ sq_y1 * win_width + x ] = color_red;
    fb[ sq_y0 * win_width + x ] = 0xffff00;
  }
//
// Draw the vertical lines.
//
for ( int y=sq_y0; y>=sq_y1; y-- )
  {
    fb[ y * win_width + sq_x0 ] = 0xff00ff;
    fb[ y * win_width + sq_x1 ] = 0xff00;
  }

/// Demo Code: Draw Sine Waves -- No need to modify this.
//
int num_waves = 4;
float plot_ht = sq_slen/float(num_waves);
for ( float yb = sq_y0 - plot_ht/2,
        freq = 4 * M_PI / float( win_width - 40 - sq_x1 );
      yb > sq_y1; yb -= plot_ht, freq *= 2 )
  for ( int x = sq_x1 + 20;  x < win_width - 20;  x++ )
    {
      int y = yb + sin( x * freq ) * plot_ht * .45;
      fb[ y * win_width + x ] = 0xff00;
```

```
    }


/// Demo Code: Draw Lines Radiating From a Point -- No need to modify.
//
//  This code calls the line_draw routine.
//
int n_lines = 24;
float c_r1 = sq_slen/6;  // Inner radius
float c_r2 = sq_slen/2;  // Outer radius.
pCoor c_ctr( win_width - c_r2 - 20, c_r2 + 20 ); // Center of lines.
float delta_theta = 2 * M_PI / n_lines;
for ( int i=0; i<n_lines; i++ )
  {
    float theta = i * delta_theta;
    pVect dir( cosf(theta), sinf(theta), 0 );
    pCoor p1 = c_ctr + c_r1 * dir;  // Point on inner radius.
    pCoor p2 = c_ctr + c_r2 * dir;  // Point on outer radius.
    line_draw(fb, p1.x, p1.y, p2.x, p2.y, 0xffffff);
  }


/// Problem 3
//
//  Copy the pixels around the mouse pointer into the square,
//  (sq_x0+1,sq_y0-1) to (sq_x1-1,sq_y1+1), so that the square shows
//  a zoomed (magnified) version of the area around the mouse
//  pointer. (Do not overwrite the lines drawn above.) Use variable
//  zoom as the factor, so that each pixel under the mouse pointer
//  is copied to 4*4=16 pixels in the square.
//
fb.fb_stats_off(); // Turn off "redundancy" statistics.

// Coordinates of mouse pointer.
//
const int mouse_x = fb.mouse_x;
const int mouse_y = fb.mouse_y;
const int zoom = 4;

/// SOLUTION -- Problem 3, Copy/Zoom Frame Buffer
///

// Compute coordinates of the center of the big box.
//
const int x_mid = ( sq_x0 + sq_x1 ) / 2;
const int y_mid = ( sq_y0 + sq_y1 ) / 2;

// Compute 1/2 the length of the mouse box.
//
const int src_hwid = sq_slen / ( zoom * 2 );

// Iterate using relative coordinates.
//
// Note: x=0,y=0  is at the center of the big box and mouse box.
```

```
  //
  for ( int y = -src_hwid;  y < src_hwid;  y++ )
    for ( int x = -src_hwid;  x < src_hwid;  x++ )
      {
        // Read the pixel from the mouse area.
        //
        uint32_t pixel = fb[ mouse_x + x + ( mouse_y + y ) * win_width ];

        // Compute the index of the lower-left part of the big box that
        // the pixel above will be written to.
        //
        int fb_idx = x_mid + x * zoom + ( y_mid + y * zoom ) * win_width;

        // Write the pixel zoom * zoom times.
        //
        for ( int yy = 0;  yy < zoom;  yy++ )
          for ( int xx = 0;  xx < zoom;  xx++ )
            fb[ fb_idx + xx + yy * win_width ] = pixel;
      }

  /// Problem 3 -- Put code to draw the blue square here (below).

  /// SOLUTION -- Problem 3, Blue Box Around Mouse Pointer
  //
  //  Use our line drawing routines to draw the box.

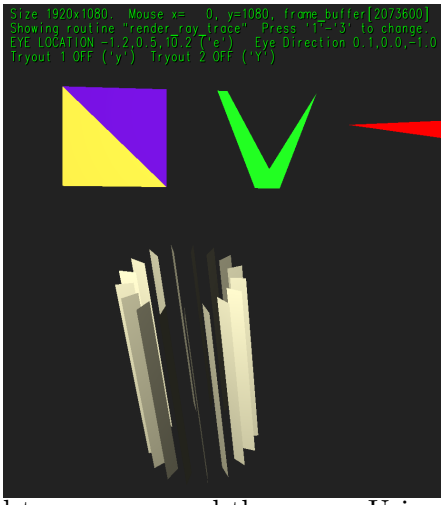
  line_draw( fb,
             mouse_x - src_hwid,  mouse_y + src_hwid,
             mouse_x + src_hwid,  mouse_y + src_hwid,  0xff );
  line_draw( fb,
             mouse_x - src_hwid,  mouse_y - src_hwid,
             mouse_x + src_hwid,  mouse_y - src_hwid,  0xff );
  line_draw( fb,
             mouse_x - src_hwid,  mouse_y + src_hwid,
             mouse_x - src_hwid,  mouse_y - src_hwid,  0xff );
  line_draw( fb,
             mouse_x + src_hwid,  mouse_y + src_hwid,
             mouse_x + src_hwid,  mouse_y - src_hwid,  0xff );

}


int
main(int argc, char **argv)
{
  pFrame_Buffer demo_frame_buffer(argc,argv);
  demo_frame_buffer.show(render_hw01);
  return 0;
}
```

# LSU EE 4702-1 Homework 2 Solution Due: 4 October 2022

Important parts of the solution are described below the complete solution code is in the repository and an html version is available at `https://www.ece.lsu.edu/koppel/gpup/2022/hw02-sol.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 the square and triangles from the `cpu-only/demo-05-ray-tracing.cc` code used in class. See the screenshot to the right, which shows the correctly solved assignment. Solving this adds the green V and the paddle wheel (the object below the square and green V).

## 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.png`.

Initially the arrow keys, `PageUp`, and `PageDown` can be used to move around the scene. Using the `Shift` modifier 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. Press `l` to move the light around and `e` to move the eye (which is what the arrow keys do when the program starts).

## Assignment-Specific User Interface

The scene is rendered by using a cpu-only ray tracing routine. (So don't complain if it's slow.) There are two ray-tracing routines, the one in use is shown in the green text to the right of `Showing routine`. Pressing `1` switches to `render_ray_trace`, this should be used for Problems 1 and 2. Pressing `2` switches to `render_ray_trace_os` (the `os` is for object space). Routine `render_ray_trace_os` won't work until Problem 3 is solved correctly.

## Code Generation and Debug Support

The compiler generates two versions of the code, `hw02` and `hw02-debug`. Use `hw02` to measure performance, but use `hw02-debug` for debugging. The `hw02-debug` version is compiled with optimization turned off. 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.

To help you debug your code and experiment in one way or another, the user interface lets you change variables. In this assignment these include the variables `hw02_info.opt_tryout1` and `hw02_info.opt_tryout2`. You can use these variables in your code (for example, `if ( hw02_info.opt_tryout1 ) { x += 5; }`) to help debug, to help familiarize yourself with how the code works, or to experiment with new ideas. Keys `y` and `Y` toggle the value of these variables. Their values are shown in the green text.

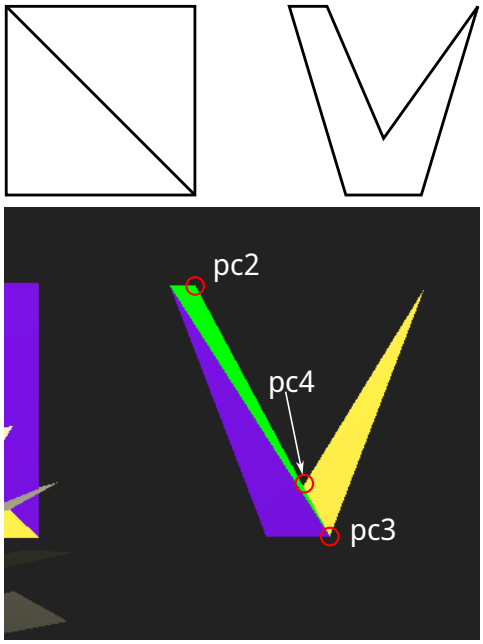
## Display of Performance-Related Data

The top green text line shows performance-related and other information. `Size` refers to the size of the window. `Mouse` refers to the coordinates of the mouse pointer. Coordinate $(0, 0)$ is at the **lower** left of the window. Text `frame_buffer[N]` shows the index of the frame buffer corresponding to

the point under the mouse pointer. (In the assignment file `frame_buffer` is abbreviated to `fb`, for convenience.)

`Render Time` and `Potential Frame Rate` show the CPU time needed to write the frame buffer. They can be ignored for this assignment. See Problem 2 for the second line, the one that starts `Frame buffer writes..`

**Problem 1:** The code in routine `prep_secene` includes a fragment that draws a square (by writing the frame buffer) consisting of a purple and gold triangle. Add code beneath that that draws the V-like figure illustrated to the right. Draw the figure using only a few triangles, and do so without overlapping triangles. It might help to make each triangle a different color. The screenshot at the beginning of the assignment shows the V-like figure in green.



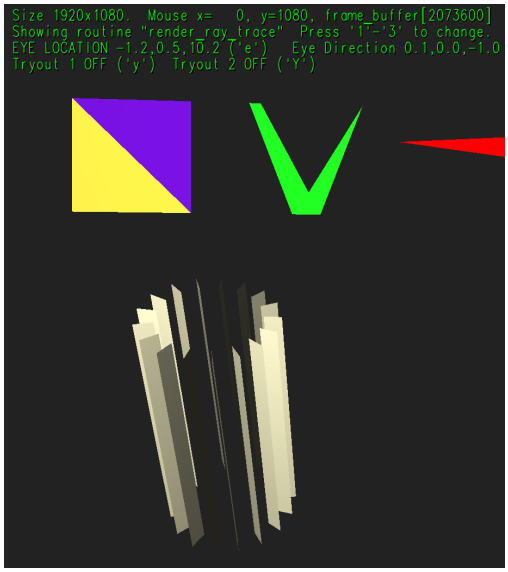

See 2016 Midterm Exam Problem 1 and 2018 Final Exam Problem 1 for roughly similar problems.

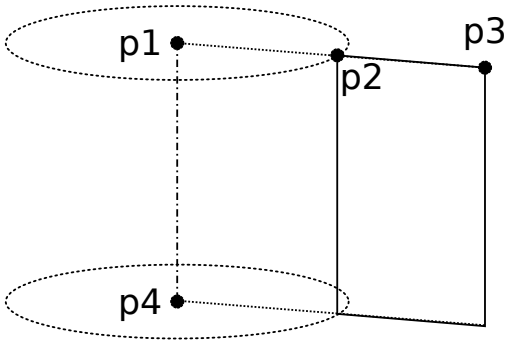
The solution code is shown below (along with some preceding code, and the resulting figure appears in the screenshot to the right. Notice that point `pc4` was computed so that the gold triangle does not overlap the green triangle. In most submissions the corresponding point was hand-computed and then its value placed in a `pCoor` constructor. Having the code compute it is more reliable and easier—once you get used to working with coordinates and vectors.

```
// Add a square consisting of a red and green triangle.
colors << color_lsu_spirit_gold << color_lsu_spirit_purple;
coors_os << pCoor(-7,0,-2) << pCoor(-7,2,-2) << pCoor(-5,0,-2);
coors_os << pCoor(-7,2,-2) << pCoor(-5,2,-2) << pCoor(-5,0,-2);

// SOLUTION -- Problem 1
pCoor pc2(-3.8,2,-2), pc3(-2.75,0,-2);
pCoor pc4 = pc3 + 0.2 * pVect(pc3,pc2);
coors_os << pCoor(-4,2,-2) << pc2                << pc3;
coors_os << pCoor(-4,2,-2) << pCoor(-3.25,0,-2) << pc3;
coors_os << pc4            << pCoor(-2,2,-2)    << pc3;
colors << color_green << color_lsu_spirit_purple << color_lsu_spirit_gold;
```

**Problem 2:** Modify the code in `prep_scene` below the comment reading `Problem 2` so that it renders a paddle-wheel-like object, as shown in the lower part of the screenshot to the right. The position and shape of the paddle wheel is determined by variables `p1`, `p2`, `p3`, `p4`, and `n_pieces`, see the diagram below the screenshot. Points `p1` and `p4` define the axis of a cylinder (actually two cylinders sharing the same axis but of different radii). Point `p2` is on the surface of one cylinder and point `p3` is on the surface of the other, larger, cylinder. Points `p1`, `p2`, and `p3` will always form a line, and that line is orthogonal to the cylinder axis. The first paddle wheel blade is positioned as shown in the diagram (the rectangle with solid lines). A complete paddle wheel should have `n_pieces-1` additional blades equally spaced around the cylinder axis.




Each time `1` (or `2` when Problem 3 is solved) is pressed new values for the variables will be chosen. Pressing those keys might help in debugging.

*Hint: Review the code for drawing a circle from the Circles section of the math slides. For this problem there will be four circles, two at the top of the cylinder, and two at the bottom. At each iteration of an* `n_pieces` *iteration loop find a point on each circle and connect them to form a blade. The illustration shows just one top circle and one bottom circle. Point* `p2` *is on the illustrated top circle, and point* `p3` *is on a top circle that's not shown.*

The solution appears below. Note that only one sine and cosine are computed per iteration. They are used to construct a vector `v`. From `v` a longer version, `v2` is computed which requires just 3 multiplications. The only remaining operation needed to compute the four points on the blade is coordinate/vector addition.

*Grading Note:* Many solutions unnecessarily computed a separate `ax` and `ay` for the upper and lower circles. If the two circles have the same normal and are not rotated with respect to each other then the same `ax` and `ay` can be used for both. Also, its less expensive to compute `vx=r*ax` and `vy=r*ay` outside the loop, and then use `vx` and `vy` inside the loop.

```
pVect vx(p1,p2), vz(p1,p4);
pNorm ax(vx), ay( cross(vz,ax) );
pVect vy = ay * ax.magnitude;
float r2or1 = pNorm(p1,p3).magnitude / ax.magnitude;
float delta_theta = 2 * M_PI / n_pieces;
for ( int i=0; i<n_pieces; i++ ) {
    float theta = i * delta_theta;
    pVect v = vx * cosf(theta) + vy * sinf(theta);
    pVect v2 = v * r2or1;
    coors_os <<  p1 + v   <<  p1 + v2  <<  p4 + v;
    coors_os <<  p1 + v2  <<  p4 + v2  <<  p4 + v;
    colors << color_lemon_chiffon << color_lemon_chiffon;  }
```

**Problem 3:** Modify routine `render_ray_trace_os` so that checks for ray/triangle intercepts using object-space coordinates. This routine is used after `2` is pressed. Routine `render_ray_trace_os` starts out by computing the pixel coordinate (`px_e`) and `ray` in eye space:

```
for (uint yw=0;  yw < win_height; yw++ ) for (uint xw=0;  xw < win_width; xw++ )
    {
      // Eye-Space Coordinate of Pixel.
      pCoor px_e = window_ll_e + window_dx_e * xw + window_dy_e * yw;

      // Ray From Eye to Pixel in Eye Space.
      pVect ray( pCoor(0,0,0), px_e );
```

But the routine iterates over the object space coordinates of triangles:

```
      for ( auto it = coors_os.begin(); it != coors_os.end(); ) {
          pCoor o0 = *it++,  o1 = *it++,  o2 = *it++;
          uint32_t color = *ic++;

          pVect tn(o0,o1,o2);  // Triangle normal.
          float t = dot( pVect(pCoor(0,0,0),o0), tn ) / dot( ray, tn ); /// WRONG!
          pCoor s = t * ray; /// WRONG!
```

This code is incorrect because the ray is in eye space but the triangle normal and vertex `o0` are in object space. Also, `s` is not computed correctly.

Fix these problems by computing a `px_o` and `ray`, both in object space. **Do not** fix this by using a transformation matrix within the loop nest. Instead directly compute a `px_o` in object space, consider using variables `window_ll_o`, and others to compute `px_o`. Also, please remove the code for `px_e`.

When this problem is correctly solved there should be no difference between rendering with `1` and `2`.

Please review the material on line/plain intercepts to help with this problem.

There are two sets of changes that need to be made: computing the pixel coordinate in object space, and modifying the parts of the code that refer to the eye location.

To compute the pixel location in object space we need the window corners in object space, which are given (`window_ll_o`, ...), and the object space `dx` and `dy` (referred to as derivatives in this context), which are not given. Like their eye-space counterparts, they are computed by dividing the object-space width and height by the window-space width and height:

```
pVect window_dx_o = pVect( window_ll_o, window_lr_o ) / win_width;
pVect window_dy_o = pVect( window_ll_o, window_ul_o ) / win_height;
```

This is correct because object space is mapped to eye space only by a rotation and translation.

The eye location in eye space is at the origin. In object space the eye location is kept in variable `hw02_info.eye_location`. Places that refer to the eye location implicitly, need to be updated to use the variable. There are two such places, the one which computes the `ray` vector, and one that computes the intercept. For the ray the eye-space pixel coordinate is replaced with the object-space pixel coordinate and the object-space eye location is used:

```
      // Ray From Eye to Pixel.
      // pVect ray( pCoor(0,0,0), px_e );  // <- Before change.
      pVect ray( eye_location, px_o );     // <- After change.
```

*(In the original assignment the ray was confusingly computed using* `pVect ray(px_e)`*. This form of the* `pVect` *constructor in effect computes* `ray = px_e - pCoor(0,0,0)`*, which forms a vector as a difference between two coordinates. Of course, computationally subtracting zero does nothing and the constructor doesn't try to subtract zero. The problem is that beginners looking at* `pVect ray(px_e)` *might start to forget the difference between a vector and coordinate.)*

The code computing the ray triangle intercept computes a vector from a triangle vertex, `o0`, and the eye. That too needs the eye-space eye coordinate, `pCoor(0,0,0)`, replaced with the variable:

```
// float t = dot( pVect( pCoor(0,0,0), o0 ), tn ) / dot( ray, tn ); // <- Beffore
float t = dot( pVect( eye_location, o0 ), tn ) / dot( ray, tn );
```

Finally, the intercept point needs to be computed from the `eye_location`:

```
pCoor s = eye_location + t * ray;
```

```
/// LSU EE 4702-1 (Fall 2022), GPU Programming
//
 /// Homework 2 --- SOLUTION
//
//   Modify this file and no others.
//   Assignment: https://www.ece.lsu.edu/koppel/gpup/2022/hw02.pdf
//   Solution: https://www.ece.lsu.edu/koppel/gpup/2022/hw02_sol.pdf
//

#include <gp/coord.h>
#include <gp/colors.h>
#include "frame_buffer.h"

struct Scene {
  vector<pCoor> coors_os;
  vector<uint32_t> colors;
};


void
prep_scene(Scene& s, const HW02_Info& hw02_info)
{
  ///
  /// Prepare Triangles -- Put Problem 1 and 2 Solutions Below
  ///

  // List of coordinates in object space and list of colors.
  //
  vector<pCoor>& coors_os = s.coors_os;
  vector<uint32_t>& colors = s.colors;

  // A starter selection of colors. For some, a lifetime collection.
  //
  uint32_t color_white = 0xffffff;
  uint32_t color_red [[maybe_unused]] = 0xff0000;
  uint32_t color_blue [[maybe_unused]] = 0xff;
  uint32_t color_green [[maybe_unused]] = 0xff00;
  //
  // For more colors see ../../../../include/colors.h
  // Or visit https://www.w3schools.com/colors/colors_picker.asp

  /// Insert First Triangle
  //
  // Insert coordinates of a triangle into coordinate list,
  // and insert its color into a color list.
  //
  coors_os << pCoor( 0, 0, 0 ) << pCoor( 9, 6, -9 ) << pCoor( 0, 7, -5 );
  colors << color_white;

  /// Insert Second Triangle
  //
  // Add coordinates and color of another triangle
  //
```

```
coors_os << pCoor(7,4,-2) << pCoor(-3,5,-9) << pCoor(9,2,-2);
colors << color_blue;

/// Insert Two More Triangles
//
// Add a square consisting of a red and green triangle.
//
colors << color_lsu_spirit_gold << color_lsu_spirit_purple;
coors_os << pCoor(-7,0,-2) << pCoor(-7,2,-2) << pCoor(-5,0,-2);
coors_os << pCoor(-7,2,-2) << pCoor(-5,2,-2) << pCoor(-5,0,-2);

/// Problem 1
//
//  Draw the v-like figure.

/// SOLUTION -- Problem 1
//
pCoor pc2(-3.8,2,-2), pc3(-2.75,0,-2);
pCoor pc4 = pc3 + 0.2 * pVect(pc3,pc2);
coors_os << pCoor(-4,2,-2) << pc2                << pc3;
coors_os << pCoor(-4,2,-2) << pCoor(-3.25,0,-2) << pc3;
coors_os << pc4            << pCoor(-2,2,-2)    << pc3;
colors << color_green << color_lsu_spirit_purple << color_lsu_spirit_gold;


/// Problem 2
//
//  Draw the paddle wheel.

// Use, but do not modify the variables below.
const pCoor p1 [[maybe_unused]] = hw02_info.p1;
const pCoor p2 [[maybe_unused]] = hw02_info.p2;
const pCoor p3 [[maybe_unused]] = hw02_info.p3;
const pCoor p4 [[maybe_unused]] = hw02_info.p4;
const int n_pieces [[maybe_unused]] = hw02_info.n_pieces;
// Use, but do not modify the variables above.


/// SOLUTION -- Problem 2

pVect vx(p1,p2), vz(p1,p4);
pNorm ax(vx), ay( cross(vz,ax) );
pVect vy = ay * ax.magnitude;
float r2or1 = pNorm(p1,p3).magnitude / ax.magnitude;
float delta_theta = 2 * M_PI / n_pieces;

for ( int i=0; i<n_pieces; i++ )
  {
    float theta = i * delta_theta;
    pVect v = vx * cosf(theta) + vy * sinf(theta);
    pVect v2 = v * r2or1;
    coors_os <<  p1 + v   <<  p1 + v2  <<  p4 + v;
    coors_os <<  p1 + v2  <<  p4 + v2  <<  p4 + v;
    colors << color_lemon_chiffon << color_lemon_chiffon;
```

```
    }
}

void
render_ray_trace_os
(pFrame_Buffer& demo_frame_buffer, const HW02_Info& hw02_info)
{
  Scene s;
  vector<pCoor>& coors_os = s.coors_os;
  vector<uint32_t>& colors = s.colors;
  prep_scene(s,hw02_info);

  ///
  /// Put Problem 3 Solution in this Routine, Further Below
  ///

  ///
  /// Transform Triangle Coordinates from Object Space to Eye Space
  ///

  // Specifications of User's Monitor (Or Window)
  //
  const uint win_width = demo_frame_buffer.width_get();
  const uint win_height = demo_frame_buffer.height_get();
  const float aspect = 1.0 * win_width / win_height;
  //
  // win_width and win_height are in window (pixel) space.

  // These are the size of the monitor in eye space.
  //
  const float width_m = 1.6;
  const float height_m = width_m / aspect;
  const float qn = 1; // Distance from eye to monitor along z axis.
  //
  // width_m and height_m are in eye space.

  // Prepare a transformation matrix that will transform object-space
  // coordinates to eye-space coordinates.
  //
  pMatrix_Translate center_eye(-hw02_info.eye_location);
  pMatrix_Rotation rotate_eye(hw02_info.eye_direction,pVect(0,0,-1));
  pMatrix eye_from_object = rotate_eye * center_eye;
  pMatrix object_from_eye = pMatrix_Translate(hw02_info.eye_location)
    * pMatrix_Rotation(pVect(0,0,-1),hw02_info.eye_direction);

  // Convert triangle coordinates from object space to eye space.
  //
  vector<pCoor> coors_es = eye_from_object * coors_os;
  pCoor light_location_o = object_from_eye*hw02_info.light_location;

  // Coordinates of Window Corners in Eye Space
  //
  pCoor window_ll_e( -width_m/2, -height_m/2, -qn);  // Lower-Left
  pCoor window_ul_e( -width_m/2,  height_m/2, -qn);  // Upper-Left
```

```
  pCoor window_lr_e( +width_m/2, -height_m/2, -qn);  // Lower-Right

  /// Problem 3 -- These may be helpful.
  //
  pCoor window_ll_o = object_from_eye * window_ll_e;
  pCoor window_ul_o = object_from_eye * window_ul_e;
  pCoor window_lr_o = object_from_eye * window_lr_e;

  /// Problem 3 -- Maybe add some code here.
  //
  /// SOLUTION -- A SMALL PART OF Problem 3

  pCoor eye_location = hw02_info.eye_location; // For convenience.
  pVect window_dx_o = pVect( window_ll_o, window_lr_o ) / win_width;
  pVect window_dy_o = pVect( window_ll_o, window_ul_o ) / win_height;
  //
  // Compute object-space versions of dx and dy.


  ///
  /// Render Triangles
  ///
#pragma omp parallel for
  // Outer Loops (yw, xw): Iterate over each pixel.
  //
  for ( uint yw = 0;  yw < win_height; yw++ )
    for ( uint xw = 0;  xw < win_width; xw++ )
      {
        /// Problem 3 -- Compute px_o and ray in object space.
        //
        //  [ ] Do not use transformation matrices here.

        /// SOLUTION -- Problem 3

        // Object-Space Coordinate of Pixel.
        //
        pCoor px_o = window_ll_o + window_dx_o * xw + window_dy_o * yw;
        //
        // Note that object space coordinate is now computed using
        // object-space window positions and derivatives (dx, dy).

        // Ray From Eye to Pixel.
        //
        pVect ray( eye_location, px_o );
        //         ^^^^^^^^^^^^
        /// SOLUTION -- Problem 3
        //  Compute ray from eye_location rather than the origin.
        //  Change is underlined above.

        float tmin = 1e10;  // Distance to closest triangle found so far.
        auto ic = colors.begin();

        // Inner Loop (it): Iterate over each triangle.
        //
```

```
        for ( auto it = coors_os.begin(); it != coors_os.end(); )
          {
            // Get Triangle and its Color
            //
            pCoor o0 = *it++,  o1 = *it++,  o2 = *it++;
            uint32_t color = *ic++;

            /// Problem 3 -- Fix computation of t and s.

            // Find where ray intercepts plane defined by triangle.
            //
            pVect tn(o0,o1,o2);  // Triangle normal.

            float t = dot(  pVect( eye_location, o0 ),  tn ) / dot( ray, tn );
            //                     ^^^^^^^^^^^^
            /// SOLUTION -- Problem 3 underlined above.
            //
            //  Change vector from pVect(o0) to pVect(eye_location,o0).

            // t indicates distance from eye to intercept point in
            // units of ray lengths.

            // Skip this triangle if a closer triangle already found ..
            // ..or if it is too close to the eye.
            //
            if ( t >= tmin || t < 1 ) continue;

            /// Problem 3 -- Fix computation of s.

            // Compute the coordinate of the line/plane intercept.
            //
            /// SOLUTION -- Problem 3
            //  Change s = t * ray; to s = eye_location + t * ray.
            //  That change is underlined below.
            //
            pCoor s = eye_location + t * ray;
            //        ^^^^^^^^^^^^^^
            // Is s inside the triangle? We don't know yet.

            // Compute a barycentric coordinate of s.
            //
            pVect v01(o0,o1), v02(o0,o2), v0s(o0,s);
            pVect v01i = cross(tn,v01);  // Vector normal to edge o0,o1
            float b2 = dot(v0s,v01i) / dot( v02,v01i );
            if ( b2 < 0 || b2 > 1 ) continue;
            //
            // Note: If b2 < 0 then s is on the wrong side of edge o0,o1.
            // If b2 > 1 then s is on the right side, but too far away.

            // Compute another barycentric coordinate of s.
            //
            pVect v02i = cross(tn,v02);
            float b1 = dot( v0s, v02i) / dot(v01,v02i);
            if ( b1 < 0 || b1 + b2 > 1 ) continue;
```

```
            // At this point we know that s is in the triangle.

            // Compute lighting.
            pNorm s_to_l(s,light_location_o);
            float ill = max( 0.1f, fabs( dot(s_to_l,pNorm(tn)) ) );
            pColor ci = pColor(color) * ill * hw02_info.light_intensity;

            // Update the minimum distance and write the frame buffer.
            //
            tmin = t;
            demo_frame_buffer[ xw + yw * win_width ] = ci.int_rgb();
          }
      }
}

void
render_ray_trace
(pFrame_Buffer& demo_frame_buffer, const HW02_Info& hw02_info)
{
  Scene s;
  vector<pCoor>& coors_os = s.coors_os;
  vector<uint32_t>& colors = s.colors;
  prep_scene(s,hw02_info);

  /// Do not modify this routine. Modify render_trace_os for Problem 3.

  ///
  /// Transform Triangle Coordinates from Object Space to Eye Space
  ///

  // Specifications of User's Monitor (Or Window)
  //
  const uint win_width = demo_frame_buffer.width_get();
  const uint win_height = demo_frame_buffer.height_get();
  const float aspect = 1.0 * win_width / win_height;
  //
  // win_width and win_height are in window (pixel) space.

  // These are the size of the monitor in eye space.
  //
  const float width_m = 1.6;
  const float height_m = width_m / aspect;
  const float qn = 1; // Distance from eye to monitor along z axis.
  //
  // width_m and height_m are in eye space.

  // Prepare a transformation matrix that will transform object-space
  // coordinates to eye-space coordinates.
  //
  pMatrix_Translate center_eye(-hw02_info.eye_location);
  pMatrix_Rotation rotate_eye(hw02_info.eye_direction,pVect(0,0,-1));
  pMatrix eye_from_object = rotate_eye * center_eye;
```

```
  // Convert triangle coordinates from object space to eye space.
  //
  vector<pCoor> coors_es = eye_from_object * coors_os;

  // Coordinates of Corners of User Monitor Window in Eye Space
  //
  pCoor window_ll_e( -width_m/2, -height_m/2, -qn);  // Lower-Left
  pCoor window_ul_e( -width_m/2,  height_m/2, -qn);  // Upper-Left
  pCoor window_lr_e( +width_m/2, -height_m/2, -qn);  // Lower-Right

  pVect window_dx_e = pVect( window_ll_e, window_lr_e ) / win_width;
  pVect window_dy_e = pVect( window_ll_e, window_ul_e ) / win_height;


  ///
  /// Render Triangles
  ///

#pragma omp parallel for
  // Outer Loops (yw, xw): Iterate over each pixel.
  //
  for ( uint yw = 0;  yw < win_height; yw++ )
    for ( uint xw = 0;  xw < win_width; xw++ )
      {
        /// Do not modify this routine! Modify render_trace_os for Problem 3.

        // Eye-Space Coordinate of Pixel.
        //
        pCoor px_e = window_ll_e + window_dx_e * xw + window_dy_e * yw;

        // Ray From Eye to Pixel.
        //
        pVect ray( pCoor(0,0,0), px_e );

        float tmin = 1e10;  // Distance to closest triangle found so far.
        auto ic = colors.begin();

        /// Do not modify this routine! Modify render_trace_os for Problem 3.

        // Inner Loop (it): Iterate over each triangle.
        //
        for ( auto it = coors_es.begin(); it != coors_es.end(); )
          {
            // Get Triangle and its Color
            //
            pCoor e0 = *it++,  e1 = *it++,  e2 = *it++;
            uint32_t color = *ic++;

            // Find where ray intercepts plane defined by triangle.
            //
            pVect tn(e0,e1,e2);  // Triangle normal.
            float t = dot( pVect(pCoor(0,0,0),e0), tn ) / dot( ray, tn );
            //
            // t indicates distance from eye to intercept point in
```

```
            // units of ray lengths.

            /// Do not modify this routine! Modify render_trace_os for Problem 3.

            // Skip this triangle if a closer triangle already found ..
            // ..or if it is too close to the eye.
            //
            if ( t >= tmin || t < 1 ) continue;

            // Compute the coordinate of the line/plane intercept.
            //
            pCoor s = t * ray;
            //
            // Is s inside the triangle? We don't know yet.

            // Compute a barycentric coordinate of s.
            //
            pVect v01(e0,e1), v02(e0,e2), v0s(e0,s);
            pVect v01i = cross(tn,v01);  // Vector normal to edge e0,e1
            float b2 = dot(v0s,v01i) / dot( v02,v01i );
            if ( b2 < 0 || b2 > 1 ) continue;
            //
            // Note: If b2 < 0 then s is on the wrong side of edge e0,e1.
            // If b2 > 1 then s is on the right side, but too far away.

            // Compute another barycentric coordinate of s.
            //
            pVect v02i = cross(tn,v02);
            float b1 = dot( v0s, v02i) / dot(v01,v02i);
            if ( b1 < 0 || b1 + b2 > 1 ) continue;

            // At this point we know that s is in the triangle.

            // Compute lighting.
            pNorm s_to_l( s, hw02_info.light_location );
            float ill = max( 0.1f, fabs( dot(s_to_l,pNorm(tn)) ) );
            pColor ci = pColor(color) * ill * hw02_info.light_intensity;

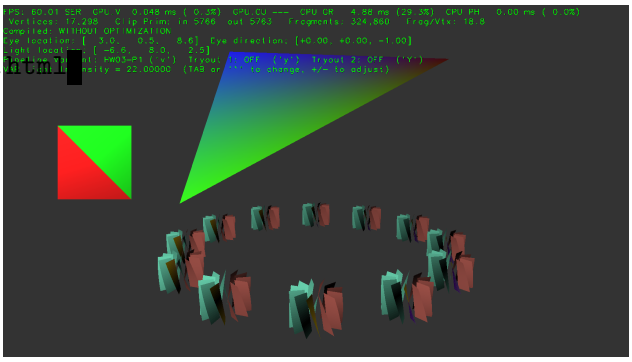
            // Update the minimum distance and write the frame buffer.
            //
            tmin = t;
            demo_frame_buffer[ xw + yw * win_width ] = ci.int_rgb();
          }
        }
}


int
main(int argc, char **argv)
{
  pFrame_Buffer demo_frame_buffer(argc,argv);
  Render render( demo_frame_buffer );
  RENDER_INSERT(render, render_ray_trace);
  RENDER_INSERT(render, render_ray_trace_os);
```

```
  render.run();
  return 0;
}
```

# LSU EE 4702-1 Homework 3 Solution Due: 17 October 2022

**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 the square and triangle from the `cpu-only/demo-03-vulkun-one.cc` code used in class, but with spinning turbines. See the screenshot to the right, which shows the correctly solved assignment. In the unsolved code all the turbines are gray.



## 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 `hw03.png` or `hw03-debug.png`.

Initially the arrow keys, `PageUp`, and `PageDown` can be used to move around the scene. Using the `Shift` modifier 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. Press `l` to move the light around and `e` to move the eye (which is what the arrow keys do when the program starts).

## Assignment-Specific User Interface

The scene can be reset by pressing `1` and `2`. When `1` is pressed the number of turbines and their positions is randomly chosen. When `2` is pressed the number and positions are the same each time. Pressing `p` will stop and start the turbines from spinning.

The scene can be rendered using three different pipelines, *Plain*, *HW03-P1*, and *HW03-P2*. The particular set of code in use is shown by the green text to the right of `Pipeline Variant`. Pressing `v` switches between the variants.

## Code Generation and Debug Support

The compiler generates two versions of the code, `hw03` and `hw03-debug`. Use `hw03` to measure performance, but use `hw03-debug` for debugging. The `hw03-debug` version is compiled with optimization turned off. You are strongly encouraged to run `hw03-debug` under the GNU debugger, `gdb`. See the material under "Running and Debugging the Assignment" on the course procedures page.

To help you debug your code and experiment in one way or another, the user interface lets you change variables. In this assignment these include the variables `opt_tryout1` and `opt_tryout2`, which are available both in CPU code and in shader code. 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` and `Y` toggle the value of these variables. Their values are shown in the green text.

## 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 rasterization. (Other assignments will use ray tracing.) 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.

The `+` and `-` keys can be used to change the value of certain variables. These variables specify things such as the light intensity, sphere radius, 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 Background

The key element of this assignment is the rendering of will be called turbines here. (They may not be shaped like real turbines.) Structure `Fan_Info` describes the shape of a turbine and how the blades should be colored. The description is similar though not identical to the way the paddle wheel from Homework 2 was described.

```
struct Fan_Info {
  pCoor pos_axis_top, pos_axis_bottom;
  pVect axis_to_cyl;
  float outer_radius;
  int n_blades;
  pColor blade_0_front_upper, blade_0_front_lower;
  pColor blade_0_back_upper, blade_0_back_lower;
  pColor blade_i_front_upper, blade_i_front_lower;
  pColor blade_i_back_upper, blade_i_back_lower;
  int serial;
};
```

The values of coordinates and vectors used above are in a local coordinate space. A transformation matrix is used to move the turbine to its intended position. In this assignment there is one `Fan_Info` object `fan_info`, and a array of transformation matrices, `fans_xforms`. (The code to generate triangles corresponding to the turbines has already been written. Look for `case PV_Plain` in the code.)

Routine `fan_setup` writes both `fan_info` and `fans_xforms`, it is called when `1` or `2` is pressed. Each time `fan_setup` executes it increments `fan_info.serial`. The purpose of `fan_info.serial` is to make it possible for other code to detect when `fan_info` has changed. (That is relevant to this assignment.)

In this assignment the turbines are to be rendered by the three pipeline variants, plain, hw03p1,

and hw03p2. (See the assignment-specific user interface section, above.) The code for the plain variant is complete (except maybe for the colors, see Problem 1).

The plain variant uses `VPipeline pipe_plain` and buffer set `bset_plain`. Here is an excerpt of the code, omitting the normals for brevity (the complete code is in `hw03.cc`):

```
case PV_Plain:
  for ( auto& m: fans_xforms )
    for ( int i=0; i<f.n_blades; i++ ) {
        float theta = theta_0 + i * delta_theta;
        pCoor p1_last, p2_last;
        for ( int j=0; j<=n_slices; j++ ) {
            float eta = theta + j * delta_eta;
            pVect v = ax * cosf(eta) + ay * sinf(eta);
            pCoor pa = f.pos_axis_top + delta_vz * j;
            pCoor p3 = m * ( pa + r1 * v );
            pCoor p4 = m * ( pa + f.outer_radius * v );
            if ( j ) {
                bset_plain << p1_last << p3 << p2_last;
                bset_plain << p2_last << p3 << p4;
                bset_plain << gray << gray << gray << gray << gray << gray;
              }
            p1_last = p3; p2_last = p4;
          }
      }
```

The `j` loop computes triangles for one blade, the `i` loop computes triangles for one turbine, and the `m` loop computes the set of turbines. Those concerned about computational waste will be appalled by the loop nest above. Why? First, the same color is used over and over, surely there must be a better way of specifying the same color a zillion times. That will be fixed in Problem 1, and in a way that allows both front and back colors to be used. Another wasteful issue is the fact that each iteration of the `m` loop does almost exactly the same thing, the only difference is the value of `m`. That will be fixed in Problem 2.

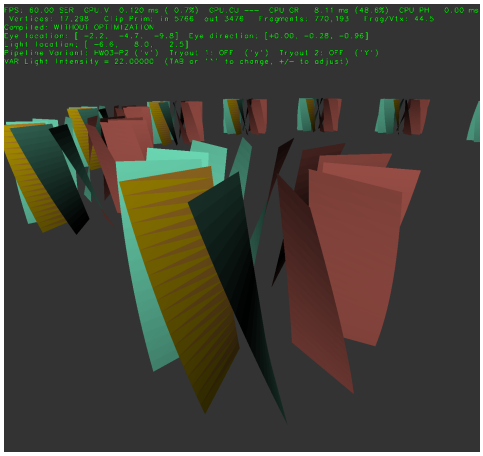
## Code Overview

Routine `World::render` is called each time the frame buffer is to be prepared. It is similar to other render routines used in class, the parts relevant to this assignment will be discussed here.

Objects `pipe_plain` and `pipe_hw03` are `VPipeline` instances. `World::render` initializes both of them, though in the solution to this assignment the initialization of `pipe_hw03` will need to be modified.

Some notes on how to work with the `VPipeline`, `VVertex_Buffer_Set`, and the buffer types can be found in this under-construction Vulkan course library note set.

**Problem 1:** Modify the code in `render` so that the turbine uses colors specified in `fan_info` and helpfully written to variables `b0_fl`, etc. The `blade_0` colors are to be used by the blade emitted when `i=0` in the `case PV_Plain` loop nest. The `blade_i` colors are to be used when `i>0`. The front color is to be used for the front of the blade, and back for the back. (See the fragment shader routine, `fs_main`, in `hw03-shdr.cc`.) Each iteration of the `j` loop emits two triangles. The first is the upper triangle and the second is the lower triangle. See the screenshot to the right. In the screenshot blade 0 has distinct upper- and lower-triangle colors, but in the other blades they are the same. Don't assume that this will always be the case.



There are three parts to this problem, only the last, part c, really needs to be solved. But solving them in order may be helpful.

(*a*) Modify the code for the `PV_Plain` case so that it uses the front colors from the `fan_info` structure, as described above. Just the front colors for this part because the `PV_Plain` pipeline variant can't take back colors. Note that this part is much easier than the others.

To solve this the code in `case PV_Plain` that inserts colors into the buffer set must be changed so that it inserts the chosen colors. Before the change that code is:

```
for ( int j=0; j<=n_slices; j++ ) {
    float eta = theta + j * delta_eta;
    pVect v = ax * cosf(eta) + ay * sinf(eta);
    pCoor pa = f.pos_axis_top + delta_vz * j;
    pCoor p3 = m * ( pa + r1 * v );
    pCoor p4 = m * ( pa + f.outer_radius * v );
    if ( j ) {
        pNorm n = cross(v,pVect(p3,p1_last));
        bset_plain << p1_last << p3 << p2_last;
        bset_plain << p2_last << p3 << p4;
        bset_plain << n << n << n   << n << n << n;
        bset_plain << color_light_gray << color_light_gray;
        bset_plain << color_light_gray << color_light_gray;
        bset_plain << color_light_gray << color_light_gray;
      }
    p1_last = p3;  p2_last = p4;
  }
```

Notice that the code inserts six copies of `color_light_gray` into the buffer set, that colors two triangles, an upper triangle and a lower triangle. As a first step we can change those to the provided front colors for blade `i`:

```
    if ( j ) {
        pNorm n = cross(v,pVect(p3,p1_last));
        bset_plain << p1_last << p3 << p2_last;
        bset_plain << p2_last << p3 << p4;
        bset_plain << n << n << n   << n << n << n;
        bset_plain
```

```
        << f.blade_i_front_upper << f.blade_i_front_upper
        << f.blade_i_front_upper
        << f.blade_i_front_lower << f.blade_i_front_lower
        << f.blade_i_front_lower;
    }
```

Notice that two colors are used, one for the upper and one for the lower triangle. To use separate colors for blade 0 and the other blades use an `if` statement with the condition checking `i`, the blade number:

```
if ( j ) {
    pNorm n = cross(v,pVect(p3,p1_last));
    bset_plain << p1_last << p3 << p2_last;
    bset_plain << p2_last << p3 << p4;
    bset_plain << n << n << n   << n << n << n;

    if ( !i )
      bset_plain
        << f.blade_0_front_upper << f.blade_0_front_upper
        << f.blade_0_front_upper
        << f.blade_0_front_lower << f.blade_0_front_lower
        << f.blade_0_front_lower;
    else
      bset_plain
        << f.blade_i_front_upper << f.blade_i_front_upper
        << f.blade_i_front_upper
        << f.blade_i_front_lower << f.blade_i_front_lower
        << f.blade_i_front_lower;
  }
```

The shaders used for the `PV_Plain` case use the same color for front and back so to get the back colors a new shader is needed. That's parts b and c of this problem.

(*b*) For this part (and the next one) write a new set of shaders in `hw03_shdr.cc` and modify the CPU code (in many places in `hw03.cc`) so that instead of using a color as a vertex attribute, an integer is used instead. Call this integer a *color set index*. The idea is to replace the code that uses colors:

```
bset_hw03 << color_red << color_blue << color_red;
```

with code that uses indices

```
bset_hw03 << 1 << 2 << 1;
```

The indices refers to colors that will have been placed in a uniform object as part of the solution. The example above implies that 1 refers to position 1 somewhere in the uniform (maybe an array element), which presumable holds red. But for this problem the color index should be used to refer to a pair of colors, front and back. For this subproblem use an existing uniform object to store the colors, but in the next part declare your own. To make things easier, the shader code already expects that there might be an integer input and there might not be a color input:

```
// Vertex Shader Inputs
layout (location = LOC_IN_POS) in vec4 in_vertex_o;
layout (location = LOC_IN_NORMAL) in vec3 in_normal_o;
#ifdef LOC_IN_COLOR
layout (location = LOC_IN_COLOR) in vec4 in_color;
#endif
#ifdef LOC_IN_INT1
layout (location = LOC_IN_INT1) in int in_color_idx;
#endif
```

The following changes need to be made to the shader code:

- A color array(s) must be added to an existing uniform. Note that a color can be declared pColor in CPU code but must be declared `vec4` in shader code.
- The changes to the uniform in the shader code must be made to the corresponding structure in the CPU code.
- Modify the shaders so that they get the correct color using the color set index, named `in_color_idx` in the shader code.
- The shaders should no longer use the `in_color` input.
- Compute a front and back lighted color in the vertex shader.
- Modify the vertex shader `out` declaration and the fragment shader `in` declaration to accommodate the back color.
- Modify the fragment shader to use the back color when appropriate.

On the CPU side the following changes are needed:

- Add the color declarations to the struct used for colors (corresponding to the uniform used in the shader code).
- Populate the uniform with the `fan_info` colors.
- Modify the code in the `PV_HW03_P1` case so that it uses color set indices rather than colors.

When this problem is solved correctly the colors should match the screenshots.

See the solution to part c.

(*c*) In the previous subpart the colors might have been placed in the `Uni_Light_Simple` or `Uni_Misc` structures because they were already there. In this subproblem define a new structure and use that for a new buffer. The new buffer will need to be declared in `World`, initialized and later destroyed in `World::setup_and_run`. In `World::render` the new uniform will need to be bound to the `pipe_hw03` descriptor set (that's what `ds` stands for) when `pipe_hw03` is initialized, and explicitly copied `to_dev`, preferably only when the colors change (see the `serial` member of `fan_info`). Most of the steps needed for your new buffer will match those for `uni_light_simple` and `uni_misc`.

To solve this problem a structure will be prepared that has two arrays of color, `front` and `back`. The different colors (blade 0 upper front, blade 0 lower front, etc) will be written into the arrays. Color index 0 will use the first element (for front and back), etc.

First, declare a structure to hold the colors. Because data in this structure will be sent to the GPU it is important the structure members are *plain old data* (POD), meaning they don't have member functions and they aren't pointers. The structure used in this solution is:

```
constexpr int ncolors = 10;
struct HW03_Colors {
  pColor front[ncolors], back[ncolors];
};
```

Next, we need to declare a `VBufferV` object to hold this structure. This will be put in the `World` class (near the end), which holds the other parts of our scene:

```
  enum Pipeline_Variant { PV_Plain, PV_HW03_P1, PV_HW03_P2, PV_SIZE };
  int pipeline_variant;

  VBufferV<HW03_Colors> uni_hw03_colors;
```

Notice that to declare a `VBufferV` we need to use the type of the data, `HW03_Colors`, in the template parameter. The name of the object is `uni_hw03_colors`.

Next, we need to initialize `uni_hw03_colors`. Initialization is done once, and we'll do it in `World::setup_and_run`, where other buffers are initialized. When initializing a buffer we need to specify what the buffer will be used for. In this case, our usage is as a uniform buffer. Here is the initialization with some context:

```
  fan_info.serial = 0;

  uni_hw03_colors.init(vh.qs,vk::BufferUsageFlagBits::eUniformBuffer);
  bset_p2_serial = 0;

  opt_pause = false;
  scene_setup_1();
```

The first argument, `vh.qs`, is something like a context and provides various information needed to complete the initialization.

Vulkan pipelines don't automatically have access to uniform variables. In order for a pipeline's shaders to access a uniform we must *bind* the uniform to the pipeline. With the course library a buffer is bound to a pipeline by calling one of the `ds_` member functions, in this case using `ds_uniform_use(LABEL,BUFFER)`. Here `LABEL` is a string that is used in the shader code to specify the *location* of the uniform variable in the *descriptor set*. `BUFFER` is a Vulkan buffer handle (type `vk::Buffer`). The Homework 3 pipeline, `pipe_hw03`, is initialized as follows (in `World::render`):

```
  if ( !pipe_hw03 )
    pipe_hw03
      .init( vh.qs )
```

```
    .ds_uniform_use( "BIND_LIGHT_SIMPLE", uni_light_simple )
    .ds_uniform_use( "BIND_MISC", uni_misc )
    .ds_uniform_use( "BIND_HW03", uni_hw03_colors ) // SOLUTION -- Prob 1c
    .shader_inputs_info_set<pCoor,pNorm,int>()      // SOLUTION -- Prob 2
    .shader_code_set
    ("hw03-shdr-sol.cc", "vs_main();", nullptr, "fs_main();")
    .topology_set( vk::PrimitiveTopology::eTriangleList )
    .create();
```

Notice that the second argument to **ds_uniform_use** is **uni_hw03_colors**, a **VBufferV** object, not a Vulkan buffer handle. The **VBufferV** object has a cast operator-overload that returns the handle, so **uni_hw03_colors** can be used wherever a **vk::Buffer** type is expected. (The **shader_inputs_info_set** change will be discussed later.)

Object **uni_hw03_colors** needs to be written with the proper colors. That is done further down in the render routine, just below the convenience variables (which weren't used, because the fully spelled out versions are more readable):

```
pColor bi_fl [[maybe_unused]] = f.blade_i_front_lower;
pColor bi_bu [[maybe_unused]] = f.blade_i_back_upper;
pColor bi_bl [[maybe_unused]] = f.blade_i_back_lower;

if ( pipeline_variant != PV_Plain )
  {
    uni_hw03_colors->front[0] = f.blade_0_front_upper;
    uni_hw03_colors->back[0] = f.blade_0_back_upper;
    uni_hw03_colors->front[1] = f.blade_0_front_lower;
    uni_hw03_colors->back[1] = f.blade_0_back_lower;
    uni_hw03_colors->front[2] = f.blade_i_front_upper;
    uni_hw03_colors->back[2] = f.blade_i_back_upper;
    uni_hw03_colors->front[3] = f.blade_i_front_lower;
    uni_hw03_colors->back[3] = f.blade_i_back_lower;

    uni_hw03_colors.to_dev();
  }
```

Notice that the code is using **uni_hw03_colors** as though it were a pointer to the structure. It's not really a pointer, it's just that the **->** operator has been overloaded to provide convenient access to an object of the type specified in the template parameter used to declare the **VBufferV<T>** object (here **T** is the template parameter). After writing the colors, member **to_dev** is called. This copies the CPU version of the structure to the GPU (the device).

The code above that writes **uni_hw03_colors** and calls **to_dev** is guarded by an **if** statement. The **if** avoids writing **uni_hw03_colors** when we are using the plain shader (which doesn't use the uniform). A good thing to add to the **if** condition would be a check of whether the colors have changed (in the **fan_info** structure). If they hadn't changed since **uni_hw03_colors** was last written there would be no reason to write them again. The code above though wastefully updates the colors every frame.

At this point a uniform buffer holding the colors has been prepared and sent to the GPU. Next, we need to stream color indices rather than colors into the rendering pipeline. Originally the pipeline was set up to expect three vertex attribute, a **pCoor**, **pNorm**, and **pColor**, set by the **VPipeline shader_inputs_info_set** member function:

```
if ( !pipe_hw03 )
  pipe_hw03
    .init( vh.qs )
    .ds_uniform_use( "BIND_LIGHT_SIMPLE", uni_light_simple )
    .ds_uniform_use( "BIND_MISC", uni_misc )
    .shader_inputs_info_set<pCoor,pNorm,pColor>()
```

```
      .shader_code_set
      ("hw03-shdr.cc", "vs_main();", nullptr, "fs_main();")
      .topology_set( vk::PrimitiveTopology::eTriangleList )
      .create();
```

As seen further above, the **pColor** type has been replaced by an **int**. The unmodified code already has a declaration ready for the integer attribute:

```
#ifdef LOC_IN_COLOR
layout (location = LOC_IN_COLOR) in vec4 in_color;
#endif
#ifdef LOC_IN_INT1
layout (location = LOC_IN_INT1) in int in_color_idx;
#endif
```

So the vertex shader will use variable **in_color_idx** for the vertex index. Returning to the CPU code, we need to replace colors with color indices. Here, we are using 0 for blade 0 upper (front and back), 1 for blade 0 lower (front and back), etc. (The bare numbers are okay here (barely) but in real life it would be better to define some symbols with meaningful names, such as enumeration constants.) The code updating the buffer set needs to use these numbers:

```
            if ( j )
              {
                pNorm n = cross(p1_last,p3,p2_last);
                bset_hw03_p1 << p1_last << p3 << p2_last;
                bset_hw03_p1 << p2_last << p3 << p4;
                bset_hw03_p1 << n << n << n   << n << n << n;

                if ( !i ) bset_hw03_p1 << 0 << 0 << 0 << 1 << 1 << 1;
                else      bset_hw03_p1 << 2 << 2 << 2 << 3 << 3 << 3;
```

Next, the shader code needs to be modified to use these changes. There are two changes: first, we need to have the shader code compute two colors, front and back; second we need to use the colors from the uniform. Using the uniform colors is easy. We start by declaring the uniform object. When declaring a uniform be careful to use the same location label that we specified when binding it to the pipeline:

```
const int ncolors = 10;
layout ( binding = BIND_HW03 ) uniform I_can_forget_this_name_no_problem
{
  vec4 front[ncolors], back[ncolors];
} uc;
```

Notice that the uniform itself is named **uc** (for uniform color). Next, get the colors from **uc.front** and **uc.back** in the vertex shader routine, **vs_main**:

```
  vec3 vec_vl = uni_light.position.xyz - vertex_e.xyz;
  float dist_to_light = length( vec_vl );
  float phase = abs( dot( normal_e, vec_vl/dist_to_light ) );

  color = uni_light.color * uc.front[in_color_idx] * phase / dist_to_light;
  // color = uni_light.color * in_color * phase / dist_to_light; // Before
  color_back = uni_light.color * uc.back[in_color_idx] * phase / dist_to_light;
```

The code using the pre-solution color attribute, **in_color**, is commented out. Notice that **in_color_idx** is used to index both the front and back colors.

Finally, we need to carry both the front and back lighted color through the rendering pipeline and modify the fragment shader to use the appropriate color. Add `color_back` to the shader interfaces:

```
// Vertex Shader Output
layout (location = 0) out Data_VF
{
  vec4 color;
  vec4 color_back;
};
```

Further below, the fragment shader input:

```
// Fragment Shader Input
layout (location = 0) in Data_VF
{
  vec4 color;
  vec4 color_back;
};
```

The change to the fragment shader is simple, since it was already coded to use a separate front and back color, but used the same color for both:

```
void fs_main()
{
  frag_color = gl_FrontFacing ? color : color_back;
}
```

It might seem wasteful to send both the front and back colors down the pipeline when only one of them will be used. The problem is that the vertex shader can't tell which side of the primitive is facing the eye because it only has access to one vertex. If the normal always pointed out of the front face then the vertex shader could substitute the correct side. But historically (meaning compatibility profile OpenGL) did not require that a normal point out of the front face, or that it even be used at all.

**Problem 2:** As mentioned in the introductory material, the code in `PV_Plain` and `PV_HW03_P1` is wasteful because it re-computes the triangles for all the turbines, even though each turbine is identical (except for position and orientation). In this problem add code to the `PV_HW03_P2` case that writes triangles to `bset_hw03_p2` for just one turbine. In fact, only write these when the serial in `fan_info` changes. Make sure not to reset the buffer set only when it needs to be re-populated. Next, record a draw for each transformation matrix. The code should look something like this:

```
for ( auto& m: fans_xforms )
  {
    pMatrix global_from_local = ..;
    pipe_hw03.ds_set( transform * global_from_local );
    pipe_hw03.record_draw(cb, bset_hw03_p2);
  }
```

The `pipe_hw03.ds_set` line is setting the `eye_from_object` matrix used by the pipeline. A draw is then recorded using the one-turbine buffer set. The loop iterates for each pipeline. In contrast, the code for the plain and problem 1 variants record one draw for all the turbines. They must spend more time preparing the buffer set, and it takes longer to send the buffer set from the CPU to the GPU. When this problem is solved correctly the buffer set is much smaller. Though it is used many times, it is only sent from the CPU to the GPU once (or each time it changes). There is a downside to this method: there is the overhead of setting up the draw, including changing the transform. That will be fixed in later assignments (using instances and storage arrays)

If this is solved correctly then the appearance using the problem 1 and 2 variants should be the same.

First, populate the buffer set for this problem, `bset_hw03_p2`, with the with the vertices for just one turbine. That turbine will be left in its local coordinate space. That is the global-from-local transformation, `m` from `fans_xforms`, won't be applied, nor will the animation angle `theta_0`. For reference, here is part of the code for `PV_HW03_P1`:

```
// Code for Problem 1 (not this problem)
bset_hw03_p1.reset(pipe_hw03);

for ( auto& m: fans_xforms )
  for ( int i=0; i<f.n_blades; i++ ) {
      float theta = theta_0 + i * delta_theta;
      pCoor p1_last, p2_last;
      for ( int j=0; j<=n_slices; j++ ) {
          float eta = theta + j * delta_eta;
          pVect v = ax * cosf(eta) + ay * sinf(eta);
          pCoor pa = f.pos_axis_top + delta_vz * j;
          pCoor p3 = m * ( pa + r1 * v );
          pCoor p4 = m * ( pa + f.outer_radius * v );
```

For this problem the `m` loop will be omitted, `theta` will be computed without `theta_0`, and (of course) `m` will not be used to transform coordinates to global space:

```
if ( bset_p2_serial != fan_info.serial ) {
    bset_p2_serial = fan_info.serial;
    bset_hw03_p2.reset(pipe_hw03);

    for ( int i=0; i<f.n_blades; i++ ) {
        float theta = i * delta_theta;
        pCoor p1_last, p2_last;
        for ( int j=0; j<=n_slices; j++ ) {
```

```
float eta = theta + j * delta_eta;
pVect v = ax * cosf(eta) + ay * sinf(eta);
pCoor pa = f.pos_axis_top + delta_vz * j;
pCoor p3 = pa + r1 * v;
pCoor p4 = pa + f.outer_radius * v;
```

The code above lacks the `m` loop, but does include a new `if` statement (the first line in the fragment above). That `if` condition insures that the buffer set is only updated if it does not contain the latest data. As described in the Assignment Background section, variable `fan_info.serial` is incremented whenever the fan changes (for example, after pressing 1 or 2). In order to determine when `bset_hw03_p2` is *stale* (outdated) a new variable has been added, `World::bset_p2_serial`. It is initialized to zero and then, as can be seen in the fragment above, is updated whenever `bset_hw03_p2` is updated.

Another important thing to notice about the code above is that `bset_hw03_p2` is only reset and sent `to_dev` when the buffer needs is updated. (Once it is reset, the data is lost, so if it were reset every frame nothing would appear after the first frame.) Calling `to_dev` each frame would defeat the purpose of only updating it when it changes. That is, we'd be sending data from the CPU to the GPU that the GPU already has.

Finally, the transformation implementing the turbine rotation needs to be applied. The unmodified code already applies the global-from-local transform:

```
// Code without the solution to Problem 2.
for ( auto& m: fans_xforms ) {
    pMatrix global_from_local = m; // Need to modify this for Problem 2
    pipe_hw03.ds_set( transform * global_from_local );
    pipe_hw03.record_draw(cb, bset_hw03_p2);
  }
```

Variable `theta_0` specifies the amount of rotation (the animation rotation) and changes each frame. Since we don't want to resend `bset_hw03_p2` every frame `theta_0` can't be used to compute the coordinates in `bset_hw03_p2`. Instead, a rotation matrix will be computed and that will be used to update the set of transformation matrices used by the shaders. The shaders use the following uniforms to hold the transformations:

```
layout ( binding = BIND_TRANSFORM ) uniform Uni_Transform
{
  mat4 eye_from_object, clip_from_eye, clip_from_object;
  mat4 object_from_eye, eye_from_clip;
} ut;
```

Here they are part of the file `hw03-shdr.cc`, but they could have been included from `transform.h`. These matrices are updated by the `VTransform` class (`transform` object) that is part of the course library. Typically `transform` holds a eye-from-global and clip-from-eye transformation. A global-from-local transform can be specified for a particular pipeline by calling the `pipe.ds_set( transform * global_from_local )` member function.

The unmodified Problem 2 code already has a loop calling this function using `m` as the global-from-local transform. But `m` does not include the animation realized by `theta_0`. So to include that effect we need to compute a rotation matrix that rotates `theta_0` radians around the `az` axis, and then use it to update the global-from-local transform:

```
pMatrix_Rotation rot(az,theta_0);
for ( auto& m: fans_xforms ) {
    pipe_hw03.ds_set( transform * m * rot );
    pipe_hw03.record_draw(cb, bset_hw03_p2);
  }
```

Matrix `rot` rotates `theta_0` around `az`, which is the $z$ axis in the turbine local coordinate space (the space used to construct the turbine). It is important to apply `rot` before `m`. That is had we called `.ds_set( transform * rot * m );` then the rotation would be done in the global space and would be all wrong.

Ideally, with the solution above the Problem 2 pipeline should run much faster. It does not (as of this writing) probably due to avoidable inefficiencies in updating the descriptor set.

```
/// LSU EE 4702-1 (Fall 2022), GPU Programming
//
 /// Homework 3 -- SOLUTION
//
//   Modify this file AND hw03.cc.
//   Assignment: https://www.ece.lsu.edu/koppel/gpup/2022/hw03.pdf
//   Solution:   https://www.ece.lsu.edu/koppel/gpup/2022/hw03_sol.pdf

// Specify version of OpenGL Shading Language.
//
#version 460

#extension GL_GOOGLE_include_directive : enable

layout ( binding = BIND_MISC ) uniform Uni_Misc
{
  ivec4 uni_misc;
};

bool opt_tryout1 = bool(uni_misc.x);
bool opt_tryout2 = bool(uni_misc.y);


// Declare variables for communication between vertex shader
// and fragment shader.
//
#ifdef _VERTEX_SHADER_

 /// Vertex Shader Inputs
//
layout (location = LOC_IN_POS) in vec4 in_vertex_o;
layout (location = LOC_IN_NORMAL) in vec3 in_normal_o;
#ifdef LOC_IN_COLOR
layout (location = LOC_IN_COLOR) in vec4 in_color;
#endif
#ifdef LOC_IN_INT1
layout (location = LOC_IN_INT1) in int in_color_idx;
#endif
//
// The CPU must provide these inputs.


 /// SOLUTION -- Problem 1
const int ncolors = 10;
layout ( binding = BIND_HW03 ) uniform I_can_forget_this_name_no_problem
{
  vec4 front[ncolors], back[ncolors];
} uc;


 /// Vertex Shader Uniforms
//
layout ( binding = BIND_LIGHT_SIMPLE ) uniform Uni_Light
```

```
{
  vec4 position;
  vec4 color;
} uni_light;


layout ( binding = BIND_TRANSFORM ) uniform Uni_Transform
{
  // Transformation Matrices
  mat4 eye_from_object, clip_from_eye, clip_from_object;
  mat4 object_from_eye, eye_from_clip;
} ut;


 /// Vertex Shader Output
//
layout (location = 0) out Data_VF
{
  vec4 color;
  /// SOLUTION -- Problem 1
  vec4 color_back;
};

 /// The Vertex Shader
void
vs_main()
{
  // Transform the vertex object-space coordinate to clip space.
  //
  gl_Position = ut.clip_from_object * in_vertex_o;
  //
  // gl_Position is predefined (by OGSL) and must be written with
  // the clip-space coordinates of the vertex.

  // Compute eye-space coordinates for vertex and normal.
  //
  vec4 vertex_e = ut.eye_from_object * in_vertex_o;
  vec3 normal_e = normalize( mat3(ut.eye_from_object) * in_normal_o );

  // Compute lighted color.
  //
  vec3 vec_vl = uni_light.position.xyz - vertex_e.xyz;
  float dist_to_light = length( vec_vl );
  float phase = abs( dot( normal_e, vec_vl/dist_to_light ) );

  /// SOLUTION -- Problem 1
  color = uni_light.color * uc.front[in_color_idx] * phase / dist_to_light;

  /// SOLUTION -- Problem 1
  color_back = uni_light.color * uc.back[in_color_idx] * phase / dist_to_light;
}
#endif

#ifdef _FRAGMENT_SHADER_
```

```
layout (location = 0) in Data_VF
{
  vec4 color;
  /// SOLUTION -- Problem 1
  vec4 color_back;
};

// Fragment Shader Output
//
layout (location = 0) out vec4 frag_color;
//
// This will get written to frame buffer if tests pass, such as the
// depth (z-buffer) test.

void
fs_main()
{
  // All this fragment shader does is substitute red for the color if
  // the user is seeing the back of the triangle.
  //
  /// SOLUTION -- Problem 1
  frag_color = gl_FrontFacing ? color : color_back;
  //
  // On important thing fragment shaders do--but not this one--is
  // apply textures.
}
#endif
```

```
/// LSU EE 4702-1 (Fall 2022), GPU Programming
//
 /// Homework 3 --- SOLUTION
//
//   Modify this file AND hw03-shdr.cc.
//   Assignment: https://www.ece.lsu.edu/koppel/gpup/2022/hw03.pdf
//   Solution:   https://www.ece.lsu.edu/koppel/gpup/2022/hw03_sol.pdf



// Include files provided for this course.
//
#define MAIN_INCLUDE
#include <vhelper.h>

#include <gp/coord.h>
#include <gp/pstring.h>
#include <gp/misc.h>
#include <gp/colors.h>

#include <vutil-texture.h>
#include <vutil-pipeline.h>
#include "shapes.h"

struct Uni_Light_Simple {
  vec4 position;
  vec4 color;
};

struct Uni_Misc {
  ivec4 opt_tryout;
};


 /// SOLUTION -- Problem 1
constexpr int ncolors = 10;
struct HW03_Colors {
  pColor front[ncolors], back[ncolors];
};


struct Fan_Info {
  pCoor pos_axis_top;
  pCoor pos_axis_bottom;
  pVect axis_to_cyl;
  float outer_radius;
  int n_blades;

  pColor blade_0_front_upper, blade_0_front_lower;
  pColor blade_0_back_upper, blade_0_back_lower;
  pColor blade_i_front_upper, blade_i_front_lower;
  pColor blade_i_back_upper, blade_i_back_lower;
```

```
  int serial;
};


class World {
public:
  World(pVulkan_Helper &vh)
    :vh(vh),ff_state(vh.qs),frame_timer(vh.frame_timer),shapes(ff_state),
     transform(vh.qs){}
  void setup_and_run();
  void render(vk::CommandBuffer& cb);
  void keyboard_handle();

  // Class providing utilities, such as showing text.
  //
  pVulkan_Helper& vh;
  VFixed_Function_State_Manager ff_state;

  // Class for showing frame timing.
  //
  pFrame_Timer& frame_timer;

  Shapes shapes;

  // Class for easy keyboard control of variables.
  //
  pVariable_Control variable_control;

  pCoor light_location;
  float opt_light_intensity;
  enum { MI_Eye, MI_Light, MI_Ball, MI_Ball_V, MI_COUNT } opt_move_item;

  pCoor eye_location, eye_initial_location;
  pVect eye_direction;
  pNorm eye_initial_direction;

  VTransform transform;
  VBufferV<Uni_Light_Simple> uni_light_simple;
  VBufferV<Uni_Misc> uni_misc;
  VPipeline pipe_plain;
  VVertex_Buffer_Set bset_plain;

  bool opt_tryout1, opt_tryout2;  // For ad-hoc experiments.

  /// Homework 3

  Fan_Info fan_info;
  vector<pMatrix> fans_xforms;
  void fan_setup(bool rand);
  void scene_setup_1();
  void scene_setup_2();
  double world_time, time_last_update;
  bool opt_pause;
```

```
  VPipeline pipe_hw03;
  VVertex_Buffer_Set bset_hw03_p1;
  VVertex_Buffer_Set bset_hw03_p2;
  enum Pipeline_Variant { PV_Plain, PV_HW03_P1, PV_HW03_P2, PV_SIZE };
  int pipeline_variant;

  /// SOLUTION -- Problem 1c
  VBufferV<HW03_Colors> uni_hw03_colors;
  int bset_p2_serial;

};

const char* const pipeline_variant_str[] =
  { "PLAIN", "HW03-P1", "HW03-P2" };



void
World::scene_setup_1()
{
  fan_setup(true);
}

void
World::scene_setup_2()
{
  fan_setup(false);
}

float
prand(float min, float max)
{
  double r = random() * (1.0 / RAND_MAX );
  return min + r * ( max - min );
}

void
World::fan_setup(bool rand)
{
  world_time = 0;
  time_last_update = time_wall_fp();
  fan_info.serial++;

  auto prand = [=](float min, float max)
  { return rand ? ::prand(min,max) : min + (max-min)/2; };

  int n_fans = prand(7,14);

  float arr_r = 10;
  pCoor pos_ref = eye_initial_location
    + ( 20 + arr_r ) * eye_initial_direction - pVect(0,4,0);
  pNorm ax = cross( pVect(0,1,0), eye_initial_direction );
  pNorm ay(0,1,prand(0,.4));
  pNorm az = cross(ax,ay);
```

```
  float cyl_ht = min( 2.0, M_PI * arr_r / n_fans );
  float delta_theta = 2 * M_PI / n_fans;

  fan_info.pos_axis_bottom = pCoor(0,0,0);
  fan_info.pos_axis_top = fan_info.pos_axis_bottom + pVect(0,cyl_ht,0);
  fan_info.axis_to_cyl = pVect(cyl_ht * .3,0,0);
  fan_info.outer_radius = cyl_ht * .7;
  fan_info.n_blades = 12;

  if ( rand && false )
    {
    }
  else
    {
      fan_info.blade_0_front_upper = color_lsu_spirit_purple;
      fan_info.blade_0_front_lower = color_lsu_business_purple;
      fan_info.blade_0_back_upper = color_lsu_spirit_gold;
      fan_info.blade_0_back_lower = color_gold;
      fan_info.blade_i_front_upper = color_salmon;
      fan_info.blade_i_front_lower = color_salmon;
      fan_info.blade_i_back_upper = color_aquamarine;
      fan_info.blade_i_back_lower = color_aquamarine;
    }

  fans_xforms.clear();

  for ( int i=0; i<n_fans; i++ )
    {
      float theta = i * delta_theta;
      pCoor pos_axis_top =
        pos_ref  + az * arr_r * cosf(theta) + ax * arr_r * sin(theta);
      pCoor pos_axis_bottom = pos_axis_top - ay * cyl_ht;
      pMatrix_Translate trans_to_global(pos_axis_bottom);
      pMatrix_Cols rot_to_global(ax,ay,az);
      fans_xforms << trans_to_global * rot_to_global;
    }
}

void
World::setup_and_run()
{
  // Setup Vulkan context, etc.
  vh.init();
  vh.display_cb_set([&](){});
  vh.cbs_cmd_record.push_back( [&](vk::CommandBuffer& cb){ render(cb); });

  pipeline_variant = 0;
  opt_tryout1 = opt_tryout2 = false;
  eye_initial_location = pCoor( 3, .5, 8.6) ;
  eye_location = eye_initial_location;
  eye_initial_direction = pVect(0,0,-1);
  eye_direction = eye_initial_direction;
```

```
  opt_light_intensity = 22;
  light_location = pCoor( -6.6, 8.0, 2.5 );
  variable_control.insert(opt_light_intensity,"Light Intensity");

  opt_move_item = MI_Eye;

  uni_misc.init(vh.qs,vk::BufferUsageFlagBits::eUniformBuffer);
  uni_misc->opt_tryout = ivec4(opt_tryout1,opt_tryout2,0,0);

  uni_light_simple.init(vh.qs,vk::BufferUsageFlagBits::eUniformBuffer);

  fan_info.serial = 0;

  /// SOLUTION -- Problem 1c
  uni_hw03_colors.init(vh.qs,vk::BufferUsageFlagBits::eUniformBuffer);
  bset_p2_serial = 0;

  opt_pause = false;
  scene_setup_1();

  // Start the graphics. The function below does not return until the
  // user exits by closing the window.
  //
  vh.message_loop_spin();
  //
  // At this point the user exited and so it's time to
  // clean up.

  uni_light_simple.destroy();
  uni_misc.destroy();
  pipe_plain.destroy();
  pipe_hw03.destroy();
  bset_plain.destroy();
  bset_hw03_p1.destroy();
  bset_hw03_p2.destroy();
  shapes.destroy();
  transform.destroy();

  /// SOLUTION -- Problem 1c
  uni_hw03_colors.destroy();

  vh.finish();
}

void
World::render(vk::CommandBuffer& cb)
{
  // This routine called whenever window needs to be updated.

  // Get any waiting keyboard commands.
  //
  keyboard_handle();

  /// Frame Buffer Informational Messages
```

```
  //
  //  Print messages using utility functions provided for this course.
  //
  vh.fbprintf("%s\n",frame_timer.frame_rate_text_get());
  const double time_now = time_wall_fp();
  const bool blink_visible = int64_t(time_now*2) & 1;
# define BLINK(txt,pad) ( blink_visible ? txt : pad )
  vh.fbprintf
    ("Compiled: %s\n",
#ifdef __OPTIMIZE__
     "WITH OPTIMIZATION"
#else
     BLINK("WITHOUT OPTIMIZATION","")
#endif
     );

  vh.fbprintf
    ("Eye location: [%5.1f, %5.1f, %5.1f]  "
     "Eye direction: [%+.2f, %+.2f, %+.2f]\n",
     eye_location.x, eye_location.y, eye_location.z,
     eye_direction.x, eye_direction.y, eye_direction.z);

  vh.fbprintf
    ("Light location: [%5.1f, %5.1f, %5.1f]\n",
     light_location.x, light_location.y, light_location.z);

  vh.fbprintf
    ("Pipeline Variant: %s ('v')  Tryout 1: %s  ('y')  Tryout 2: %s  ('Y')  \n",
     pipeline_variant_str[pipeline_variant],
     opt_tryout1 ? BLINK("ON ","   ") : "OFF",
     opt_tryout2 ? BLINK("ON ","   ") : "OFF");


  pVariable_Control_Elt* const cvar = variable_control.current;
  vh.fbprintf("VAR %s = %.5f  (TAB or '`' to change, +/- to adjust)\n",
                      cvar->name,cvar->get_val());


  // ----------------------------------------------------------------------
  ///
  /// Specification of Transformation Matrices
  ///

  pMatrix eye_from_global =
    pMatrix_Rotation(eye_direction,pVect(0,0,-1))
    * pMatrix_Translate(-eye_location);

  transform.eye_from_global_set( eye_from_global );

  /// Setup Projection Transformation:  Eye Space -> Clip Space
  //
  const int win_width = vh.s_extent.width;
  const int win_height = vh.s_extent.height;
  const float aspect = float(win_width) / win_height;
```

```
const float n_dist = 0.01;
const float xr = .8 * n_dist;

// Frustum: left, right, bottom, top, near, far
transform.clip_from_eye_set
  ( pMatrix_Frustum( -xr, xr,                // left, right
                     -xr/aspect, xr/aspect,  // bottom, top
                     n_dist, 5000            // near, far
                     ) );

/// Lighting
//
uni_light_simple->color = color_white * opt_light_intensity;
uni_light_simple->position = transform.eye_from_global * light_location;
uni_light_simple.to_dev();

uni_misc.to_dev();

//
// ----------------------------------------------------------------------

if ( !pipe_plain )
  pipe_plain
    .init( vh.qs )
    .ds_uniform_use( "BIND_LIGHT_SIMPLE", uni_light_simple )
    .ds_uniform_use( "BIND_MISC", uni_misc )
    .shader_inputs_info_set<pCoor,pNorm,pColor>()
    .shader_code_set
    ("demo-03-shdr.cc", "vs_main();", nullptr, "fs_main();")
    .topology_set( vk::PrimitiveTopology::eTriangleList )
    .create();

if ( !pipe_hw03 )
  pipe_hw03
    .init( vh.qs )
    .ds_uniform_use( "BIND_LIGHT_SIMPLE", uni_light_simple )
    .ds_uniform_use( "BIND_MISC", uni_misc )
    .ds_uniform_use( "BIND_HW03", uni_hw03_colors ) // SOLUTION -- Prob 1c
    .shader_inputs_info_set<pCoor,pNorm,int>()      // SOLUTION -- Prob 2
    .shader_code_set
    ("hw03-shdr-sol.cc", "vs_main();", nullptr, "fs_main();")
    .topology_set( vk::PrimitiveTopology::eTriangleList )
    .create();

/// SOLUTION ABOVE


// Reset buffers in this buffer set.
//
bset_plain.reset(pipe_plain);

// Object-space coordinates of a triangle.
//
pCoor p0 = { 0, 0,  0 };
```

```
pCoor p1 = { 9, 6, -9 };
pCoor p2 = { 0, 5, -5 };

// Find triangle normal using cross product function (in coord.h).
//
pNorm tri_norm = cross( p0, p1, p2 );
pColor color_tri( .2, .900, .1 ); // Red, Green, Blue; Values in [0,1]

// Insert vertex coordinates into buffer set (bset_plain).
//
bset_plain << p0 << p1 << p2;

// Insert vertex colors and normals.
//
bset_plain << color_tri << color_red << color_blue;
bset_plain << tri_norm << tri_norm << tri_norm;
//
// All three vertices here have the same normal.

// Add a square consisting of a red and green triangle.
//
bset_plain << pCoor(-2,0,-2) << pCoor(-4,2,-2) << pCoor(-4,0,-2);
bset_plain << color_red << color_red << color_red;
bset_plain << pCoor(-2,0,-2) << pCoor(-2,2,-2) << pCoor(-4,2,-2);
bset_plain << color_green << color_green << color_green;
pNorm snorm = cross( pCoor(-4,0,-2), pCoor(-4,2,-2), pCoor(-2,0,-2) );
bset_plain << snorm << snorm << snorm << snorm << snorm << snorm;

// Update theta_0, the angle by which the turbines should be at.
//
if ( !opt_pause ) world_time += time_now - time_last_update;
time_last_update = time_now;
const float omega = 1;
const float theta_0 = world_time * omega;

const auto& f = fan_info;

// Values used in constructing the turbines.
//
int n_slices = 20;  // Each blade is rendered as n_slices pairs of triangles.
float delta_theta = 2 * M_PI / f.n_blades;
float delta_eta = delta_theta / n_slices;
pVect delta_vz = pVect(f.pos_axis_top,f.pos_axis_bottom)/n_slices;
pNorm ax(f.axis_to_cyl);
pNorm az(f.pos_axis_bottom,f.pos_axis_top);
pVect ay = cross(az,ax);
float r1 = ax.magnitude;

// Abbreviations, for your solving convenience.
pColor b0_fu [[maybe_unused]] = f.blade_0_front_upper;
pColor b0_fl [[maybe_unused]] = f.blade_0_front_lower;
pColor b0_bu [[maybe_unused]] = f.blade_0_back_upper;
pColor b0_bl [[maybe_unused]] = f.blade_0_back_lower;
pColor bi_fu [[maybe_unused]] = f.blade_i_front_upper;
```

```
pColor bi_fl [[maybe_unused]] = f.blade_i_front_lower;
pColor bi_bu [[maybe_unused]] = f.blade_i_back_upper;
pColor bi_bl [[maybe_unused]] = f.blade_i_back_lower;

  /// SOLUTION -- Problem 1c
if ( pipeline_variant != PV_Plain )
  {
    uni_hw03_colors->front[0] = f.blade_0_front_upper;
    uni_hw03_colors->back[0] = f.blade_0_back_upper;
    uni_hw03_colors->front[1] = f.blade_0_front_lower;
    uni_hw03_colors->back[1] = f.blade_0_back_lower;
    uni_hw03_colors->front[2] = f.blade_i_front_upper;
    uni_hw03_colors->back[2] = f.blade_i_back_upper;
    uni_hw03_colors->front[3] = f.blade_i_front_lower;
    uni_hw03_colors->back[3] = f.blade_i_back_lower;

    uni_hw03_colors.to_dev();

  }

switch ( pipeline_variant ) {

case PV_Plain:

  for ( auto& m: fans_xforms )
    for ( int i=0; i<f.n_blades; i++ )
      {
        float theta = theta_0 + i * delta_theta;
        pCoor p1_last, p2_last;
        for ( int j=0; j<=n_slices; j++ )
          {
            float eta = theta + j * delta_eta;
            pVect v = ax * cosf(eta) + ay * sinf(eta);
            pCoor pa = f.pos_axis_top + delta_vz * j;
            pCoor p3 = m * ( pa + r1 * v );
            pCoor p4 = m * ( pa + f.outer_radius * v );
            if ( j )
              {
                pNorm n = cross(v,pVect(p3,p1_last));
                bset_plain << p1_last << p3 << p2_last;
                bset_plain << p2_last << p3 << p4;
                bset_plain << n << n << n   << n << n << n;

                /// SOLUTION -- Problem 1a
                //
                if ( !i )
                  bset_plain
                    << f.blade_0_front_upper << f.blade_0_front_upper
                    << f.blade_0_front_upper
                    << f.blade_0_front_lower << f.blade_0_front_lower
                    << f.blade_0_front_lower;
                else
                  bset_plain
                    << f.blade_i_front_upper << f.blade_i_front_upper
```

```
                    << f.blade_i_front_upper
                    << f.blade_i_front_lower << f.blade_i_front_lower
                    << f.blade_i_front_lower;
                }
              p1_last = p3;
              p2_last = p4;
            }
        }
    break;

  case PV_HW03_P1:

    bset_hw03_p1.reset(pipe_hw03);

    for ( auto& m: fans_xforms )
      for ( int i=0; i<f.n_blades; i++ )
        {
          float theta = theta_0 + i * delta_theta;
          pCoor p1_last, p2_last;
          for ( int j=0; j<=n_slices; j++ )
            {
              float eta = theta + j * delta_eta;
              pVect v = ax * cosf(eta) + ay * sinf(eta);
              pCoor pa = f.pos_axis_top + delta_vz * j;
              pCoor p3 = m * ( pa + r1 * v );
              pCoor p4 = m * ( pa + f.outer_radius * v );
              if ( j )
                {
                  pNorm n = cross(p1_last,p3,p2_last);
                  bset_hw03_p1 << p1_last << p3 << p2_last;
                  bset_hw03_p1 << p2_last << p3 << p4;
                  bset_hw03_p1 << n << n << n   << n << n << n;

                  /// SOLUTION -- Problem 1b and 1c
                  //
                  if ( !i ) bset_hw03_p1 << 0 << 0 << 0 << 1 << 1 << 1;
                  else      bset_hw03_p1 << 2 << 2 << 2 << 3 << 3 << 3;
                  //
                  // Insert color indices rather than colors.

                  //  bset_hw03_p1 << color_salmon << color_salmon;
                  //  bset_hw03_p1 << color_salmon << color_salmon;
                  //  bset_hw03_p1 << color_salmon << color_salmon;
                }
              p1_last = p3;
              p2_last = p4;
            }
        }
    bset_hw03_p1.to_dev();
    pipe_hw03.ds_set( transform );
    pipe_hw03.record_draw(cb, bset_hw03_p1);

    break;
```

```
  case PV_HW03_P2:
    {
      /// SOLUTION -- Problem 2

      // Update the buffer set *only* if the fan_info has changed.
      //
      if ( bset_p2_serial != fan_info.serial )
        {
          bset_p2_serial = fan_info.serial;

          // Reset the buffer set *only* when we update it.
          //
          bset_hw03_p2.reset(pipe_hw03);

          /// SOLUTION -- Problem 2
          //
          // Update the buffer set with the local coordinates for a
          // turbine.
          //
          // Differences with the PV_HW03_P1 and PV_Plain cases above:
          //
          //  - This code writes bset_hw03_p2.
          //  - The vertices for just one turbine are written.
          //  - The local-to-global transform, m, is not applied.
          //  - The rotation (animation) position, theta_0, is not applied.
          //
          for ( int i=0; i<f.n_blades; i++ )
            {
              float theta = i * delta_theta;
              pCoor p1_last, p2_last;
              for ( int j=0; j<=n_slices; j++ )
                {
                  float eta = theta + j * delta_eta;
                  pVect v = ax * cosf(eta) + ay * sinf(eta);
                  pCoor pa = f.pos_axis_top + delta_vz * j;
                  pCoor p3 = pa + r1 * v;
                  pCoor p4 = pa + f.outer_radius * v;
                  if ( j )
                    {
                      pNorm n = cross(p1_last,p3,p2_last);
                      bset_hw03_p2 << p1_last << p3 << p2_last;
                      bset_hw03_p2 << p2_last << p3 << p4;
                      bset_hw03_p2 << n << n << n   << n << n << n;
                      if ( i ) bset_hw03_p2 << 2 << 2 << 2 << 3 << 3 << 3;
                      else bset_hw03_p2 << 0 << 0 << 0 << 1 << 1 << 1;
                    }
                  p1_last = p3;
                  p2_last = p4;
                }
            }

          bset_hw03_p2.to_dev();
          //
          // Note that this code only executes when fan_info changes.
```

```
        }

      /// SOLUTION -- Problem 2
      //
      // First, compute the rotation that applies to all turbines.
      //
      pMatrix_Rotation rot(az,theta_0);
      //
      // See the use of theta_0 in the PV_HW03_P1 and PV_Plain cases above.)

      for ( auto& m: fans_xforms )
        {
          // Apply the rotation before the global-from-local transform (m).
          //
          pipe_hw03.ds_set( transform * m * rot );

          pipe_hw03.record_draw(cb, bset_hw03_p2);
        }
    }
    break;

  default: assert( false );
  }

  bset_plain.to_dev();
  pipe_plain.ds_set( transform );
  pipe_plain.record_draw(cb, bset_plain);

  // Render Marker for Light Source
  //
  shapes.record_tetrahedron(cb,transform,light_location,0.2);

}



void
World::keyboard_handle()
{
  const int key = vh.keyboard_key_get();
  if ( !key ) return;
  pVect adjustment(0,0,0);
  pVect user_rot_axis(0,0,0);
  const bool kb_mod_s = vh.keyboard_shift;
  const bool kb_mod_c = vh.keyboard_control;
  const float move_amt = kb_mod_s ? 2.0 : kb_mod_c ? 0.08 : 0.4;

  switch ( key ) {
  case FB_KEY_LEFT: adjustment.x = -move_amt; break;
  case FB_KEY_RIGHT: adjustment.x = move_amt; break;
  case FB_KEY_PAGE_UP: adjustment.y = move_amt; break;
  case FB_KEY_PAGE_DOWN: adjustment.y = -move_amt; break;
  case FB_KEY_DOWN: adjustment.z = move_amt; break;
  case FB_KEY_UP: adjustment.z = -move_amt; break;
```

```
  case FB_KEY_DELETE: user_rot_axis.y = 1; break;
  case FB_KEY_INSERT: user_rot_axis.y =  -1; break;
  case FB_KEY_HOME: user_rot_axis.x = 1; break;
  case FB_KEY_END: user_rot_axis.x = -1; break;
  case '1': scene_setup_1(); break;
  case '2': scene_setup_2(); break;
  case 'b': case 'B': opt_move_item = MI_Ball; break;
  case 'e': case 'E': opt_move_item = MI_Eye; break;
  case 'l': case 'L': opt_move_item = MI_Light; break;
  case 'p': case 'P': opt_pause = !opt_pause; break;
  case 'v': case 'V':
    if ( ++pipeline_variant >= PV_SIZE ) pipeline_variant = 0;
    break;
  case 'y': opt_tryout1 = !opt_tryout1;
    uni_misc->opt_tryout.x = int(opt_tryout1);
    break;
  case 'Y': opt_tryout2 = !opt_tryout2;
    uni_misc->opt_tryout.y = int(opt_tryout2);
    break;
  case FB_KEY_TAB:
    if ( !kb_mod_s ) { variable_control.switch_var_right(); break; }
  case 96: variable_control.switch_var_left(); break;
  case '-':case '_': variable_control.adjust_lower(); break;
  case '+':case '=': variable_control.adjust_higher(); break;
  default: printf("Unknown key, %d\n",key); break;
  }

  // Update eye_direction based on keyboard command.
  //
  if ( user_rot_axis.x || user_rot_axis.y )
    {
      pMatrix_Rotation rotall(eye_direction,pVect(0,0,-1));
      user_rot_axis *= invert(rotall);
      eye_direction *= pMatrix_Rotation(user_rot_axis, M_PI * 0.03);
    }

  // Update eye_location based on keyboard command.
  //
  if ( adjustment.x || adjustment.y || adjustment.z )
    {
      const double angle =
        fabs(eye_direction.y) > 0.99
        ? 0 : atan2(eye_direction.x,-eye_direction.z);
      pMatrix_Rotation rotall(pVect(0,1,0),-angle);
      adjustment *= rotall;

      switch ( opt_move_item ){
      case MI_Light: light_location += adjustment; break;
      case MI_Eye: eye_location += adjustment; break;
      //  case MI_Ball: object_location += adjustment; break;
      default: break;
      }
    }
}
```

```
int
main(int argv, char **argc)
{
  pVulkan_Helper pvulkan_helper(argv,argc);
  World world(pvulkan_helper);

  world.setup_and_run();

  return 0;
}
```

```
/// LSU EE 4702-1 (Fall 2022), GPU Programming
//
 /// Homework 4 -- SOLUTION
//
//   Modify this file AND hw04.cc.
//   Assignment: https://www.ece.lsu.edu/koppel/gpup/2022/hw04.pdf


// Specify version of OpenGL Shading Language.
//
#version 460

#extension GL_GOOGLE_include_directive : enable

////////////////////////////////////////////////////////////////////////////////
///
 /// Uniform Variable Declarations
//

layout ( binding = BIND_MISC ) uniform Uni_Misc
{
  ivec4 uni_misc;
};

// Use these for debugging, experimentation, etc.
bool opt_tryout1 = bool(uni_misc.x);
bool opt_tryout2 = bool(uni_misc.y);

layout ( binding = BIND_TRANSFORM ) uniform Uni_Transform
{
  // Transformation Matrices
  mat4 eye_from_object, clip_from_eye, clip_from_object;
  mat4 object_from_eye, eye_from_clip;
} ut;

layout ( binding = BIND_LIGHT_SIMPLE ) uniform Uni_Light
{
  vec4 position;
  vec4 color;
} uni_light;

const int ncolors = 10;
layout ( binding = BIND_HW04 ) uniform Uni_Colors
{
  vec4 front[ncolors], back[ncolors];
} uc;

#ifdef _VERTEX_SHADER_
////////////////////////////////////////////////////////////////////////////////
///
 /// Vertex Shader Code
//
```

```
 // Vertex Shader Inputs
//
layout (location = LOC_IN_POS) in vec4 in_vertex_o;
#ifdef LOC_IN_NORMAL
layout (location = LOC_IN_NORMAL) in vec3 in_normal_o;
#endif
#ifdef LOC_IN_INT1
layout (location = LOC_IN_INT1) in int in_color_idx;
#endif

 // Vertex Shader Outputs
//
layout (location = 0) out Data_VG
{
  vec4 vertex_c;

  /// SOLUTION
  //
  //  Send send object- and eye-space vertex coordinates and color
  //  index to geometry shader.
  //
  vec4 vertex_o, vertex_e;
  int c_idx;
};

void
vs_main_hw04()
{
  // The Vertex Shader

  /// SOLUTION
  //
  // Remove the code computing the lighted color. We can't compute the
  // lighted color in the vertex shader because we don't have a normal
  // and it can't be computed here because we only have one
  // vertex. Instead, compute the clip- and eye-space vertex
  // coordinate and send them to the geometry shader.


  c_idx = int(in_vertex_o.w);
  vertex_o = vec4(in_vertex_o.xyz,1);
  vertex_c = ut.clip_from_object * vertex_o;
  vertex_e = ut.eye_from_object * vertex_o;
}
#endif

#ifdef _GEOMETRY_SHADER_
////////////////////////////////////////////////////////////////////////////////
///
 /// Geometry Shader Code
///

// Geometry Shader Inputs
//
```

```
layout (location = 0) in Data_VG
{
  vec4 vertex_c;
  /// SOLUTION
  vec4 vertex_o;
  vec4 vertex_e;
  int c_idx;
} In[];

// Geometry Shader Input Primitive
//
layout ( triangles ) in;


// Geometry Shader Outputs
//
layout (location = 0) out Data_GF
{
  vec4 color;
  vec4 color_back;
};

// Geometry Shader Output Primitive
//
layout ( triangle_strip, max_vertices = 3 ) out;

void
gs_main_hw04()
{
  // The Geometry Shader
  //
  // Pre-defined output: gl_Position (vec4, read when EmitVertex called).

  /// SOLUTION
  //
  //  Compute the triangle normal using the three vertex coordinates.
  //
  vec3 normal_e =
    normalize( cross( In[1].vertex_e.xyz - In[0].vertex_e.xyz,
                      In[2].vertex_e.xyz - In[0].vertex_e.xyz ) );

  // Use the color of the first vertex for the triangle.
  //
  int c_idx = In[0].c_idx;
  //
  // The color In[1].c_idx and In[2].c_idx will be used for the adjacent
  // triangles.

  for ( int i=0; i<3; i++ )
    {
      gl_Position = In[i].vertex_c;

      /// SOLUTION
      vec4 vertex_e = In[i].vertex_e;
```

```
      // Compute lighted color.
      //
      vec3 vec_vl = uni_light.position.xyz - vertex_e.xyz;
      float dist_to_light = length( vec_vl );
      float phase = abs( dot( normal_e, vec_vl ) / dist_to_light );

      vec4 illum = uni_light.color * max( 0.025, phase / dist_to_light );
      color = illum * uc.front[c_idx];
      color_back = illum * uc.back[c_idx];
      EmitVertex();
    }
  EndPrimitive();
}

#endif


#ifdef _FRAGMENT_SHADER_
////////////////////////////////////////////////////////////////////////////////
///
 /// Fragment Shader Code
///

// Fragment Shader Inputs
//
layout (location = 0) in Data_GF
{
  vec4 color;
  vec4 color_back;
};

// Fragment Shader Output
//
layout (location = 0) out vec4 frag_color;
//
// This will get written to frame buffer if tests pass, such as the
// depth (z-buffer) test.

void
fs_main_hw04()
{
  // The Fragment Shader

  frag_color = gl_FrontFacing ? color : color_back;
}
#endif
```

```
/// LSU EE 4702-1 (Fall 2022), GPU Programming
//
 /// Homework 4 -- SOLUTION
//
//   Most of the code is based on Homework 3 and Midterm Exam Problem 1b
//
//   Handout: https://www.ece.lsu.edu/koppel/gpup/2022/hw04.pdf


// Include files provided for this course.
//
#define MAIN_INCLUDE
#include <vhelper.h>

#include <gp/coord.h>
#include <gp/pstring.h>
#include <gp/misc.h>
#include <gp/colors.h>

#include <vutil-texture.h>
#include <vutil-pipeline.h>
#include "shapes.h"

struct Uni_Light_Simple {
  vec4 position;
  vec4 color;
};

struct Uni_Misc {
  ivec4 opt_tryout;
};


constexpr int ncolors = 10;
struct MT_Colors {
  pColor front[ncolors], back[ncolors];
};


struct Fan_Info {
  pCoor p1, p2, p3, p4;
  int n_blades;

  pCoor pos_axis_top;
  pCoor pos_axis_bottom;
  pVect axis_to_cyl;
  float outer_radius;

  pNorm az;

  pColor blade_0_front_upper, blade_0_front_lower;
  pColor blade_0_back_upper, blade_0_back_lower;
  pColor blade_i_front_upper, blade_i_front_lower;
```

```
  pColor blade_i_back_upper, blade_i_back_lower;

  int serial;
};


class World {
public:
  World(pVulkan_Helper &vh)
    :vh(vh),ff_state(vh.qs),frame_timer(vh.frame_timer),shapes(ff_state),
     transform(vh.qs){}
  void setup_and_run();
  void render(vk::CommandBuffer& cb);
  void keyboard_handle();

  // Class providing utilities, such as showing text.
  //
  pVulkan_Helper& vh;
  VFixed_Function_State_Manager ff_state;

  // Class for showing frame timing.
  //
  pFrame_Timer& frame_timer;

  Shapes shapes;

  // Class for easy keyboard control of variables.
  //
  pVariable_Control variable_control;

  pCoor light_location;
  float opt_light_intensity;
  enum { MI_Eye, MI_Light, MI_Ball, MI_Ball_V, MI_COUNT } opt_move_item;

  pCoor eye_location, eye_initial_location;
  pVect eye_direction;
  pNorm eye_initial_direction;

  VTransform transform;
  VBufferV<Uni_Light_Simple> uni_light_simple;
  VBufferV<Uni_Misc> uni_misc;
  VPipeline pipe_plain;
  VVertex_Buffer_Set bset_plain;

  bool opt_tryout1, opt_tryout2;  // For ad-hoc experiments.

  /// Homework 4

  Fan_Info fan_info;
  vector<pMatrix> fans_xforms;
  void fan_setup(bool rand);
  void scene_setup_1();
  void scene_setup_2();
  void scene_setup_3();
```

```
  double world_time, time_last_update;
  bool opt_pause;

  VPipeline pipe_gear_base;
  VPipeline pipe_gear_hw04;
  VVertex_Buffer_Set bset_mta;
  VVertex_Buffer_Set bset_gear_list;
  VVertex_Buffer_Set bset_gear_strip;

  VBufferV<MT_Colors> uni_colors;
  int bset_p2_serial;

  bool opt_tri_strip;
  bool opt_tri_shader;

  int opt_n_blades;
  int colors_serial;

};




void
World::scene_setup_1()
{
  fan_setup(true);
}

void
World::scene_setup_2()
{
  fan_setup(false);
}

void
World::scene_setup_3()
{
  // Used for the assignment screenshots.
  vh.clearValues[0].color =
    vk::ClearColorValue( array<float, 4>( { { 0.4, .4, .4, 1.0 } } ) );
  eye_location = pCoor(1.0,0.1,2.1);
  eye_direction = pVect(.37,-.19,-.91);
  opt_light_intensity = 14.51;
  fan_setup(false);
}

float
prand(float min, float max)
{
  double r = random() * (1.0 / RAND_MAX );
  return min + r * ( max - min );
}
```

```
void
World::fan_setup(bool rand)
{
  world_time = 0;
  time_last_update = time_wall_fp();
  fan_info.serial++;

  auto prand = [=](float min, float max)
  { return rand ? ::prand(min,max) : min + (max-min)/2; };

  int n_fans = rand ? prand(7,14) : 4;

  float arr_r = 3.5;
  pCoor pos_ref = eye_initial_location
    + ( 20 + 10 ) * eye_initial_direction + pVect(0,0,0);
  pNorm ax = cross( pVect(0,1,0), eye_initial_direction );
  pNorm ay(0,1,prand(0,.4));
  pNorm az = cross(ax,ay);

  float cyl_ht = rand ? min( 2.0, M_PI * arr_r / n_fans ) : 4;
  float delta_theta = 2 * M_PI / n_fans;

  fan_info.p4 = fan_info.pos_axis_bottom = pCoor(0,0,0);
  fan_info.az = pNorm(0,1,0);
  fan_info.p1 = fan_info.pos_axis_top =
    fan_info.pos_axis_bottom + fan_info.az * cyl_ht;
  float inner_r = arr_r;
  fan_info.axis_to_cyl = pVect( inner_r,0,0);
  fan_info.p2 = fan_info.p1 + fan_info.axis_to_cyl;
  fan_info.outer_radius = inner_r * 1.3;
  fan_info.p3 = fan_info.p1 + 1.3 * fan_info.axis_to_cyl;
  fan_info.n_blades = opt_n_blades;

  if ( rand && false )
    {
    }
  else
    {
      fan_info.blade_0_front_upper = color_lsu_spirit_purple;
      fan_info.blade_0_front_lower = color_lsu_business_purple;
      fan_info.blade_0_back_upper = color_lsu_spirit_gold;
      fan_info.blade_0_back_lower = color_gold;
      fan_info.blade_i_front_upper = color_aquamarine;
      fan_info.blade_i_front_lower = color_aquamarine;
      fan_info.blade_i_back_upper = color_salmon;
      fan_info.blade_i_back_lower = color_salmon;
    }

  fans_xforms.clear();

  for ( int i=0; i<n_fans; i++ )
    {
      float theta = M_PI + i * delta_theta;
      pCoor pos_axis_top =
```

```
        pos_ref  + az * 10 * cosf(theta) + ax * 10 * sin(theta);
      pCoor pos_axis_bottom = pos_axis_top - ay * cyl_ht;
      pMatrix_Translate trans_to_global(pos_axis_bottom);
      pMatrix_Cols rot_to_global(ax,ay,az);
      fans_xforms << trans_to_global * rot_to_global;
    }
}


void
World::setup_and_run()
{
  // Setup Vulkan context, etc.
  vh.init();
  vh.display_cb_set([&](){});
  vh.cbs_cmd_record.push_back( [&](vk::CommandBuffer& cb){ render(cb); });

  vh.clearValues[0].color =
    vk::ClearColorValue( array<float, 4>( { { 0.4, 0.4, 0.4, 1.0 } } ) );

  opt_tryout1 = opt_tryout2 = false;
  eye_initial_location = pCoor( 3, .5, 8.6) ;
  eye_location = eye_initial_location;
  eye_initial_direction = pVect(0,0,-1);
  eye_direction = eye_initial_direction;

  opt_n_blades = 16;
  variable_control.insert(opt_n_blades,"Num Blades",1,3);

  opt_light_intensity = 13.1;
  light_location = pCoor( -5, 3.2, -2.7 );
  variable_control.insert(opt_light_intensity,"Light Intensity");

  opt_move_item = MI_Eye;

  uni_misc.init(vh.qs,vk::BufferUsageFlagBits::eUniformBuffer);
  uni_misc->opt_tryout = ivec4(opt_tryout1,opt_tryout2,0,0);

  uni_light_simple.init(vh.qs,vk::BufferUsageFlagBits::eUniformBuffer);

  fan_info.serial = 0;

  uni_colors.init(vh.qs,vk::BufferUsageFlagBits::eUniformBuffer);
  bset_p2_serial = 0;

  opt_pause = false;

  opt_tri_strip = false;
  opt_tri_shader = false;
  colors_serial = -1;

  scene_setup_2();

  // Start the graphics. The function below does not return until the
```

```
  // user exits by closing the window.
  //
  vh.message_loop_spin();
  //
  // At this point the user exited and so it's time to
  // clean up.

  uni_light_simple.destroy();
  uni_misc.destroy();
  pipe_plain.destroy();
  pipe_gear_base.destroy();
  pipe_gear_hw04.destroy();
  bset_plain.destroy();
  bset_mta.destroy();
  bset_gear_list.destroy();
  bset_gear_strip.destroy();
  shapes.destroy();
  transform.destroy();

  uni_colors.destroy();

  vh.finish();
}

void
World::render(vk::CommandBuffer& cb)
{
  // This routine called whenever window needs to be updated.

  // Get any waiting keyboard commands.
  //
  keyboard_handle();

  /// Frame Buffer Informational Messages
  //
  //  Print messages using utility functions provided for this course.
  //
  vh.fbprintf("%s\n",frame_timer.frame_rate_text_get());
  const double time_now = time_wall_fp();
  const bool blink_visible = int64_t(time_now*2) & 1;
# define BLINK(txt,pad) ( blink_visible ? txt : pad )
  vh.fbprintf
    ("Compiled: %s\n",
#ifdef __OPTIMIZE__
     "WITH OPTIMIZATION"
#else
     BLINK("WITHOUT OPTIMIZATION","")
#endif
     );

  vh.fbprintf
    ("Eye location: [%5.1f, %5.1f, %5.1f]  "
     "Eye direction: [%+.2f, %+.2f, %+.2f]\n",
     eye_location.x, eye_location.y, eye_location.z,
```

```
   eye_direction.x, eye_direction.y, eye_direction.z);

vh.fbprintf
  ("Light location: [%5.1f, %5.1f, %5.1f]\n",
   light_location.x, light_location.y, light_location.z);

vh.fbprintf
  ("Topology: %s ('v')  Shader: %s ('V')  "
   "Tryout 1: %s  ('y')  Tryout 2: %s  ('Y')  \n",
   opt_tri_strip ? "TRI STRIP" : "TRI LIST ",
   opt_tri_shader ? "HW04 " : "BASE",
   opt_tryout1 ? BLINK("ON ","   ") : "OFF",
   opt_tryout2 ? BLINK("ON ","   ") : "OFF");


pVariable_Control_Elt* const cvar = variable_control.current;
vh.fbprintf("VAR %s = %.5f  (TAB or '`' to change, +/- to adjust)\n",
                    cvar->name,cvar->get_val());


// ------------------------------------------------------------------------
///
/// Specification of Transformation Matrices
///

pMatrix eye_from_global =
  pMatrix_Rotation(eye_direction,pVect(0,0,-1))
  * pMatrix_Translate(-eye_location);

transform.eye_from_global_set( eye_from_global );

/// Setup Projection Transformation: Eye Space -> Clip Space
//
const int win_width = vh.s_extent.width;
const int win_height = vh.s_extent.height;
const float aspect = float(win_width) / win_height;
const float n_dist = 0.01;
const float xr = .8 * n_dist;

// Frustum: left, right, bottom, top, near, far
transform.clip_from_eye_set
  ( pMatrix_Frustum( -xr, xr,                // left, right
                     -xr/aspect, xr/aspect,  // bottom, top
                     n_dist, 5000            // near, far
                     ) );

/// Lighting
//
uni_light_simple->color = color_white * opt_light_intensity;
uni_light_simple->position = transform.eye_from_global * light_location;
uni_light_simple.to_dev();

uni_misc.to_dev();
```

```
//
// ----------------------------------------------------------------------

if ( !pipe_plain )
  pipe_plain
    .init( vh.qs )
    .ds_uniform_use( "BIND_LIGHT_SIMPLE", uni_light_simple )
    .ds_uniform_use( "BIND_MISC", uni_misc )
    .shader_inputs_info_set<pCoor,pNorm,pColor>()
    .shader_code_set
    ("demo-03-shdr.cc", "vs_main();", nullptr, "fs_main();")
    .topology_set( vk::PrimitiveTopology::eTriangleList )
    .create();

const auto prim_want = opt_tri_strip
  ? vk::PrimitiveTopology::eTriangleStrip
  : vk::PrimitiveTopology::eTriangleList;

if ( pipe_gear_base && pipe_gear_base.p_in_asm_ci.topology != prim_want )
  pipe_gear_base.destroy();

if ( !pipe_gear_base )
  pipe_gear_base
    .init( vh.qs )
    .ds_uniform_use( "BIND_LIGHT_SIMPLE", uni_light_simple )
    .ds_uniform_use( "BIND_MISC", uni_misc )
    .ds_uniform_use( "BIND_HW04", uni_colors )
    .shader_inputs_info_set<pCoor,pNorm,int>()
    .shader_code_set
    ("gear-base-shdr.cc",
     "vs_main_list();", "gs_main_list();", "fs_main_list();")
    .topology_set( prim_want )
    .create();

if ( pipe_gear_hw04 && pipe_gear_hw04.p_in_asm_ci.topology != prim_want )
  pipe_gear_hw04.destroy();


/// Homework 4 -- Problem 2
//
// Modify the pipeline initialization below so that unneeded data
// is not used for vertex shader inputs.

if ( !pipe_gear_hw04 )
  pipe_gear_hw04
    .init( vh.qs )
    .ds_uniform_use( "BIND_LIGHT_SIMPLE", uni_light_simple )
    .ds_uniform_use( "BIND_MISC", uni_misc )
    .ds_uniform_use( "BIND_HW04", uni_colors )
    /// SOLUTION
    // Remove pNorm and int from shader_inputs.
    .shader_inputs_info_set<pCoor>()
    .shader_code_set
    ("hw04-shdr-sol.cc",
```

```
     "vs_main_hw04();", "gs_main_hw04();", "fs_main_hw04();")
    .topology_set( prim_want )
    .create();


// Reset buffers in this buffer set.
//
bset_plain.reset(pipe_plain);

// Prepare a triangle.
//
pCoor p0 = { 9, 0,  -10 };
pCoor p1 = { 18, 6, -4 };
pCoor p2 = { 9, 5, -9 };
pNorm tri_norm = cross( p0, p1, p2 );
bset_plain << p0 << p1 << p2;
bset_plain << color_green << color_red << color_blue;
bset_plain << tri_norm << tri_norm << tri_norm;

// Add a square consisting of a red and green triangle.
//
bset_plain << pCoor(-2,0,-4) << pCoor(-4,2,-4) << pCoor(-4,0,-4);
bset_plain << color_red << color_red << color_red;
bset_plain << pCoor(-2,0,-4) << pCoor(-2,2,-4) << pCoor(-4,2,-4);
bset_plain << color_green << color_green << color_green;
pNorm snorm = cross( pCoor(-4,0,-4), pCoor(-4,2,-4), pCoor(-2,0,-4) );
bset_plain << snorm << snorm << snorm << snorm << snorm << snorm;

bset_plain.to_dev();
pipe_plain.ds_set( transform );
pipe_plain.record_draw(cb, bset_plain);

// Update theta_0, the angle by which the gear should be at.
//
if ( !opt_pause ) world_time += time_now - time_last_update;
time_last_update = time_now;
const float omega = 1;
const float theta_0 = world_time * omega;

if ( opt_n_blades != fan_info.n_blades )
  {
    fan_info.n_blades = opt_n_blades;
    fan_info.serial++;
  }

const auto& f = fan_info;

if ( colors_serial != f.serial )
  {
    colors_serial = f.serial;
    uni_colors->front[0] = f.blade_0_front_upper;
    uni_colors->back[0] = f.blade_0_back_upper;
    uni_colors->front[1] = f.blade_0_front_lower;
    uni_colors->back[1] = f.blade_0_back_lower;
```

```
    uni_colors->front[2] = f.blade_i_front_upper;
    uni_colors->back[2] = f.blade_i_back_upper;
    uni_colors->front[3] = f.blade_i_front_lower;
    uni_colors->back[3] = f.blade_i_back_lower;
    uni_colors.to_dev();
  }

VPipeline& pipe = opt_tri_shader ? pipe_gear_hw04 : pipe_gear_base;

// Would be nice to set stale to true only when bset needs an update.
bool gear_list_stale = true;
bool gear_strip_stale = true;

if ( !opt_tri_strip && gear_list_stale )
  {
    gear_list_stale = false;
    bset_gear_list.reset(pipe);

    pCoor p1(f.p1),  p2(f.p2),  p3(f.p3),  p4(f.p4);
    float n_pieces = f.n_blades;

    pVect vx_inner(p1,p2), vx_outer(p1,p3), vz(p4,p1);
    pNorm az(vz);
    pVect vy_inner = cross( az, vx_inner );
    pVect vy_outer = cross( az, vx_outer );

    float delta_theta = 2 * M_PI / n_pieces;
    float delta_theta_h = delta_theta / 2;

    /// Homework 4 -- Do not modify this code.

    pVect v_prev = vx_inner;

    for ( int i=1; i<=2*n_pieces; i++ )
      {
        float theta = i * delta_theta_h;
        pVect v_curr =
          i & 1
          ? vx_outer * cosf(theta) + vy_outer * sinf(theta)
          : vx_inner * cosf(theta) + vy_inner * sinf(theta);

        pCoor pp_bot = p4 + v_prev;
        pCoor pc_bot = p4 + v_curr;
        pCoor pp_top = p1 + v_prev;
        pCoor pc_top = p1 + v_curr;

        // Emit Coordinates
        bset_gear_list << pc_top << pp_top << pp_bot; // Upper Triangle
        bset_gear_list << pc_bot << pc_top << pp_bot; // Lower Triangle

        // Emit Normals
        pNorm np = cross( vz, v_curr - v_prev );
        bset_gear_list << np << np << np << np << np << np;
```

```
        // Emit Colors
        if ( i < 3 ) bset_gear_list << 0 << 0 << 0 << 1 << 1 << 1;
        else         bset_gear_list << 2 << 2 << 2 << 3 << 3 << 3;

        v_prev = v_curr;
      }

    bset_gear_list.to_dev();
  }

if ( opt_tri_strip && gear_strip_stale )
  {
    gear_strip_stale = false;
    bset_gear_strip.reset(pipe);

    pCoor p1(f.p1),  p2(f.p2),  p3(f.p3),  p4(f.p4);
    float n_pieces = f.n_blades;

    pVect vx_inner(p1,p2), vx_outer(p1,p3), vz(p4,p1);
    pNorm az(vz);
    pVect vy_inner = cross( az, vx_inner );
    pVect vy_outer = cross( az, vx_outer );

    float delta_theta = 2 * M_PI / n_pieces;
    float delta_theta_h = delta_theta / 2;

    /// Homework 4 -- Problem 1
    //
    //  Modify the code below so that the vertices in
    //  bset_gear_strip describe the gear for a triangle
    //  strip topology.

    pVect v_prev = vx_inner;

    for ( int i=0; i<=2*n_pieces; i++ )
      {
        float theta = i * delta_theta_h;
        pVect v_curr =
          i & 1
          ? vx_outer * cosf(theta) + vy_outer * sinf(theta)
          : vx_inner * cosf(theta) + vy_inner * sinf(theta);

        pCoor pc_top = p1 + v_curr;
        pCoor pc_bot = p4 + v_curr;

        /// SOLUTION

        // Emit Colors
        if ( bset_gear_strip.int1.selected )
          {
            if ( i < 2 ) bset_gear_strip << 0 << 1;
            else         bset_gear_strip << 2 << 3;
          }
        else
```

```
            {
              /// SOLUTION - Pack color index into w component.
              if ( i < 2 ) { pc_top.w = 0; pc_bot.w = 1; }
              else         { pc_top.w = 2; pc_bot.w = 3; }
            }

          // Emit Coordinates
          bset_gear_strip << pc_top << pc_bot;

          if ( bset_gear_strip.normal.selected )
            {
              // Emit Normals, If Needed
              pNorm np = cross( vz, v_curr - v_prev );
              bset_gear_strip << np << np;
            }
          v_prev = v_curr;
        }

      bset_gear_strip.to_dev();
    }

  pMatrix_Rotation rot(f.az,theta_0);

  // Apply the rotation before the global-from-local transform (fans_xforms[0])
  //
  pipe.ds_set( transform * fans_xforms[0] * rot );
  pipe.record_draw(cb, opt_tri_strip ? bset_gear_strip : bset_gear_list );

  // Render Marker for Light Source
  //
  shapes.record_tetrahedron(cb,transform,light_location,0.2);
}



void
World::keyboard_handle()
{
  const int key = vh.keyboard_key_get();
  if ( !key ) return;
  pVect adjustment(0,0,0);
  pVect user_rot_axis(0,0,0);
  const bool kb_mod_s = vh.keyboard_shift;
  const bool kb_mod_c = vh.keyboard_control;
  const float move_amt = kb_mod_s ? 2.0 : kb_mod_c ? 0.08 : 0.4;

  switch ( key ) {
  case FB_KEY_LEFT: adjustment.x = -move_amt; break;
  case FB_KEY_RIGHT: adjustment.x = move_amt; break;
  case FB_KEY_PAGE_UP: adjustment.y = move_amt; break;
  case FB_KEY_PAGE_DOWN: adjustment.y = -move_amt; break;
  case FB_KEY_DOWN: adjustment.z = move_amt; break;
  case FB_KEY_UP: adjustment.z = -move_amt; break;
  case FB_KEY_DELETE: user_rot_axis.y = 1; break;
```

```
  case FB_KEY_INSERT: user_rot_axis.y =  -1; break;
  case FB_KEY_HOME: user_rot_axis.x = 1; break;
  case FB_KEY_END: user_rot_axis.x = -1; break;
  case '1': scene_setup_1(); break;
  case '2': scene_setup_2(); break;
  case '3': scene_setup_3(); break;
  case 'b': case 'B': opt_move_item = MI_Ball; break;
  case 'e': case 'E': opt_move_item = MI_Eye; break;
  case 'l': case 'L': opt_move_item = MI_Light; break;
  case 'p': case 'P': opt_pause = !opt_pause; break;
  case 'v': opt_tri_strip = !opt_tri_strip; break;
  case 'V': opt_tri_shader = !opt_tri_shader; break;
  case 'y': opt_tryout1 = !opt_tryout1;
    uni_misc->opt_tryout.x = int(opt_tryout1);
    break;
  case 'Y': opt_tryout2 = !opt_tryout2;
    uni_misc->opt_tryout.y = int(opt_tryout2);
    break;
  case FB_KEY_TAB:
    if ( !kb_mod_s ) { variable_control.switch_var_right(); break; }
  case 96: variable_control.switch_var_left(); break;
  case '-':case '_': variable_control.adjust_lower(); break;
  case '+':case '=': variable_control.adjust_higher(); break;
  default: printf("Unknown key, %d\n",key); break;
  }

  // Update eye_direction based on keyboard command.
  //
  if ( user_rot_axis.x || user_rot_axis.y )
    {
      pMatrix_Rotation rotall(eye_direction,pVect(0,0,-1));
      user_rot_axis *= invert(rotall);
      eye_direction *= pMatrix_Rotation(user_rot_axis, M_PI * 0.03);
    }

  // Update eye_location based on keyboard command.
  //
  if ( adjustment.x || adjustment.y || adjustment.z )
    {
      const double angle =
        fabs(eye_direction.y) > 0.99
        ? 0 : atan2(eye_direction.x,-eye_direction.z);
      pMatrix_Rotation rotall(pVect(0,1,0),-angle);
      adjustment *= rotall;

      switch ( opt_move_item ){
      case MI_Light: light_location += adjustment; break;
      case MI_Eye: eye_location += adjustment; break;
      //  case MI_Ball: object_location += adjustment; break;
      default: break;
      }
    }
}
```

```
int
main(int argv, char **argc)
{
  pVulkan_Helper pvulkan_helper(argv,argc);
  World world(pvulkan_helper);

  world.setup_and_run();

  return 0;
}
```

```
/// LSU EE 4702-1 (Fall 2022), GPU Programming
//

 /// Homework 5 --- SOLUTION
//
//   Modify this file AND hw05.cc.
//   Assignment: https://www.ece.lsu.edu/koppel/gpup/2022/hw05.pdf


// Specify version of OpenGL Shading Language.
//
#version 460

#extension GL_GOOGLE_include_directive : enable

layout ( binding = BIND_MISC ) uniform Uni_Misc
{
  ivec4 uni_misc;
  vec4 opt_tryoutf4;

  /// SOLUTION -- Problems 1 and 2.

  // Problem 1, provide radius of hole.
  //
  float radius_inner;
  float p1, p2, p3; // Pad so that number of floats is a multiple 4.
  //
  // If the number of consecutive floats is not a multiple of 4 then
  // the values of mat4 and vec4 (and others) will not be read
  // correctly. This is because in OGSL variables of type vec4 and
  // mat4 must start at an offset that is a multiple of 16.
  // Placement of the padding variables is necessary on the CPU
  // code but optional on the GPU code.

  // Problem 2, provide rotation matrix.
  //
  mat4 rot;
};

bool opt_tryout1 = bool(uni_misc.x);
bool opt_tryout2 = bool(uni_misc.y);
float opt_tryoutf = opt_tryoutf4.x;

const int ncolors = 10;
layout ( binding = BIND_HW05 ) uniform I_can_forget_this_name_no_problem
{
  vec4 front[ncolors], back[ncolors];
} uc;


 /// Uniforms
//
layout ( binding = BIND_LIGHT_SIMPLE ) uniform Uni_Light
```

```
{
  vec4 position;
  vec4 color;
} uni_light;

layout ( binding = BIND_TRANSFORM ) uniform Uni_Transform
{
  // Transformation Matrices
  mat4 eye_from_object, clip_from_eye, clip_from_object;
  mat4 object_from_eye, eye_from_clip;
} ut;

#ifdef _VERTEX_SHADER_

 /// Vertex Shader Inputs -- For all Problems
//
layout (location = LOC_IN_POS) in vec4 in_vertex_o;
layout (location = LOC_IN_NORMAL) in vec3 in_normal_o;
layout (location = LOC_IN_INT1) in int in_color_idx;
//
// The CPU must provide these inputs.

#endif


#ifdef HW05_PROB1
////////////////////////////////////////////////////////////////////////////////
//
 /// Homework 5 Problem 1
//
//   Put solution to Problem 1 here.
//   (The section for Problem 2 is further below.)


#ifdef _VERTEX_SHADER_

 /// Vertex Shader Output
//
layout (location = 0) out Data_VG
{
  vec4 vertex_c;
  vec3 vertex_e, normal_e;
  int color_idx;

  /// SOLUTION -- Problem 1
  //
  vec3 inner_e;  // Eye-space coordinates of point on circle forming hole.
  vec4 inner_c;  // Clip-space coordinates of point on circle forming hole.

  /// SOLUTION -- Problem 2
  //
  int vtx_idx;   // Vertex index. (Needed to retrieve rotation.)
};
```

```
 /// The Vertex Shader
void
vs_main_prob1()
{
  // Transform the vertex object-space coordinate to clip space.
  //
  vertex_c = ut.clip_from_object * in_vertex_o;

  // Compute eye-space coordinates for vertex and normal.
  //
  vertex_e = vec3( ut.eye_from_object * in_vertex_o );
  normal_e = normalize( mat3(ut.eye_from_object) * in_normal_o );
  color_idx = in_color_idx;

  /// SOLUTION -- Problem 1 (Until end of procedure).
  //

  /// Compute eye-space coordinate of gear center.
  //
  // Object-space coordinate of gear center.
  //
  vec4 ctr_o = vec4(0,0,in_vertex_o.z,1);
  //
  // This is the same as the local-space coordinate.
  //
  // Transform object space to eye space.
  //
  vec4 ctr_e = ut.eye_from_object * vec4(0,0,in_vertex_o.z,1);

  /// Compute coordinates of point on the hole corresponding to this vtx.
  //
  // Compute a unit vector from gear center to this vertex.
  //
  vec3 v = normalize( vertex_e - ctr_e.xyz );
  //
  // Use unit vector to compute coordinate of vertex on hole circle.
  //
  inner_e = ctr_e.xyz + v * radius_inner;
  inner_c = ut.clip_from_eye * vec4(inner_e,1);

  // Pass along vertex index. Will be used in the geometry shader.
  //
  vtx_idx = gl_VertexIndex;
}

#endif

#ifdef _GEOMETRY_SHADER_
layout (location = 0) in Data_VG
{
  vec4 vertex_c;
  vec3 vertex_e, normal_e;
  int color_idx;
```

```
  /// SOLUTION -- Problem 1
  //
  vec3 inner_e;  // Eye-space coordinate of point on circle forming hole.
  vec4 inner_c;  // Clip-space coordinate of point on circle forming hole.

  /// SOLUTION -- Problem 2
  //
  int vtx_idx;   // Vertex index. (Needed to retrieve rotation.)
} In[];

layout (location = 0) out Data_GF
{
  vec3 vertex_e;
  vec3 normal_e;
  flat int color_idx;
};

layout ( triangles ) in;

 /// SOLUTION -- Problem 1
//
//  Increase max vertices from 3 to 9.
//
layout ( triangle_strip, max_vertices = 9 ) out;

void
gs_main_prob1()
{
  /// Geometry Shader
  //

  for ( int i=0; i<3; i++ )
    {
      gl_Position = In[i].vertex_c;
      vertex_e = In[i].vertex_e;
      normal_e = In[i].normal_e;
      color_idx = In[i].color_idx;
      EmitVertex();
    }
  EndPrimitive();

  /// SOLUTION -- Problem 1 (Until end of this procedure.)

  /// Determine which two vertices to use to draw the sides.
  //
  //  There are three input vertices, two will be on one side of the
  //  triangle strip (or gear), one will be on the other. If the three
  //  input vertices, In[0], In[1], and In[2], were in the same order
  //  as the vertices sent to the GPU, then vertices In[0] and In[2]
  //  would be on the same side of the strip. But for odd-numbered
  //  triangles In[1] is the vertex after In[2]. This is so that
  //  triangle In[0], In[1], In[2] is always facing up.

  int idx_skip = In[1].vtx_idx - In[0].vtx_idx;
```

```
  // Compute eye-space normal of gear sides.
  //
  vec3 normal_side_o = vec3(0,0,1);
  //
  vec3 normal_side_e = mat3(ut.eye_from_object) * normal_side_o;

  // Emit coordinates of triangles forming sides.
  //
  for ( int i=0; i<3; i++ )
    if ( i != idx_skip )
      {
        gl_Position = In[i].vertex_c;
        vertex_e = In[i].vertex_e;
        color_idx = 6;
        normal_e = normal_side_e;
        EmitVertex();
      }

  for ( int i=0; i<3; i++ )
    if ( i != idx_skip )
      {
        gl_Position = In[i].inner_c;
        vertex_e = In[i].inner_e.xyz;
        color_idx = 6;
        normal_e = normal_side_e;
        EmitVertex();
      }

  EndPrimitive();
}

#endif

#endif

#ifdef HW05_PROB2
////////////////////////////////////////////////////////////////////////
//
 /// Homework 5 Problem 2
//
//   Put solution to Problem 2 here.
//   (The section for Problem 1 is above.)


layout ( binding = BIND_XF ) buffer Gears { mat4 gears_xform[]; };

#ifdef _VERTEX_SHADER_

 /// Vertex Shader Output
//
layout (location = 0) out Data_VG
{
  vec4 vertex_c;
```

```
  vec3 vertex_e, normal_e;
  int color_idx;
};

void
vs_main_prob2()
{
  int inst = gl_InstanceIndex;

  /// SOLUTION -- Problem 2
  //

  // Compute transformation from local space to global space.
  //
  mat4 g_from_l = gears_xform[inst] * rot;
  //
  // Array gears_xform gives the position of the gear and rot gives
  // the rotation. Matrix rot is updated every frame, but gears_xform
  // is only updated when the number of gears change.

  // Transform input vertex from local space to global space.
  //
  vec4 vertex_g = g_from_l * in_vertex_o;

  // Transform the vertex object space to clip space.
  //
  vertex_c = ut.clip_from_object * vertex_g;

  // Compute eye-space coordinates for vertex and normal.
  //
  vertex_e = vec3( ut.eye_from_object * vertex_g );
  normal_e =
    normalize( mat3(ut.eye_from_object) * mat3(g_from_l) * in_normal_o );
  color_idx = in_color_idx;
}

#endif

#ifdef _GEOMETRY_SHADER_
layout (location = 0) in Data_VG
{
  vec4 vertex_c;
  vec3 vertex_e, normal_e;
  int color_idx;
} In[];

layout (location = 0) out Data_GF
{
  vec3 vertex_e;
  vec3 normal_e;
  flat int color_idx;
};

layout ( triangles ) in;
```

```
layout ( triangle_strip, max_vertices = 3 ) out;

void
gs_main_prob2()
{
  /// Geometry Shader
  //

  for ( int i=0; i<3; i++ )
    {
      gl_Position = In[i].vertex_c;
      vertex_e = In[i].vertex_e;
      normal_e = In[i].normal_e;
      color_idx = In[i].color_idx;
      EmitVertex();
    }
  EndPrimitive();
}

#endif

#endif



////////////////////////////////////////////////////////////////////////////
//
 /// Homework 5: No need to modify the fragment shader.
//   The same fragment shader should work with all problems.

#ifdef _FRAGMENT_SHADER_

layout (location = 0) in Data_GF
{
  vec3 vertex_e;
  vec3 normal_e;
  flat int color_idx;
};

// Fragment Shader Output
//
layout (location = 0) out vec4 frag_color;
//
// This will get written to frame buffer if tests pass, such as the
// depth (z-buffer) test.

void
fs_main()
{
  // Compute lighted color.
  //
  vec3 vec_vl = uni_light.position.xyz - vertex_e;
  float dist_to_light = length( vec_vl );
  bool lit_side = dot( normal_e, vec_vl ) > 0 == dot( normal_e, -vertex_e ) > 0;
```

```
  float phase = lit_side ? abs( dot( normal_e, vec_vl ) / dist_to_light ) : 0;
  vec4 illum = uni_light.color * max( 0.015, phase / dist_to_light );
  vec4 color = gl_FrontFacing ? uc.front[color_idx] : uc.back[color_idx];

  frag_color = illum * color;
}
#endif
```

```
/// LSU EE 4702-1 (Fall 2022), GPU Programming
//
 /// Homework 5 --- SOLUTION
//
//   Modify this file AND hw05-shdr.cc.
//   Handout: https://www.ece.lsu.edu/koppel/gpup/2022/hw05.pdf


// Include files provided for this course.
//
#define MAIN_INCLUDE
#include <vhelper.h>

#include <gp/coord.h>
#include <gp/pstring.h>
#include <gp/misc.h>
#include <gp/colors.h>

#include <vutil-texture.h>
#include <vutil-pipeline.h>
#include "shapes.h"

struct Uni_Light_Simple {
  vec4 position;
  vec4 color;
};

struct Uni_Misc {
  ivec4 opt_tryout;
  vec4 opt_tryoutf4;

  /// SOLUTION -- Problems 1 and 2.

  // Problem 1, provide radius of hole.
  //
  float radius_inner;
  float p1, p2, p3; // Pad so that number of floats is a multiple 4.
  //
  // This padding is necessary on the CPU so that the starting address
  // of mat4 (below) is a multiple of 16 (16 is the size of 4 floats).

  // Problem 2, provide rotation matrix.
  //
  mat4 rot;
};


constexpr int ncolors = 10;
struct MT_Colors {
  pColor front[ncolors], back[ncolors];
};

struct Gear_Info {
```

```
  float radius_inner;
  float width;
  int n_teeth;
  float land_bot_frac;
  float land_top_frac;

  int serial;
};


class World {
public:
  World(pVulkan_Helper &vh)
    :vh(vh),ff_state(vh.qs),frame_timer(vh.frame_timer),shapes(ff_state),
     transform(vh.qs){}
  void setup_and_run();
  void render(vk::CommandBuffer& cb);
  void keyboard_handle();

  // Class providing utilities, such as showing text.
  //
  pVulkan_Helper& vh;
  VFixed_Function_State_Manager ff_state;

  // Class for showing frame timing.
  //
  pFrame_Timer& frame_timer;

  Shapes shapes;

  // Class for easy keyboard control of variables.
  //
  pVariable_Control variable_control;

  pCoor light_location;
  float opt_light_intensity;
  enum { MI_Eye, MI_Light, MI_Ball, MI_Ball_V, MI_COUNT } opt_move_item;

  pCoor eye_location, eye_initial_location;
  pCoor object_location;
  pVect object_direction;
  pMatrix object_orientation;
  pVect eye_direction;
  pNorm eye_initial_direction;

  VTransform transform;
  VBufferV<Uni_Light_Simple> uni_light_simple;
  VBufferV<Uni_Misc> uni_misc;
  VPipeline pipe_plain;
  VVertex_Buffer_Set bset_plain;

  bool opt_tryout1, opt_tryout2;  // For ad-hoc experiments.
  float opt_tryoutf;
```

```
  /// Homework 5

  Gear_Info gear_info;

  VBufferVV<mat4> buf_gears_xforms;
  void gear_setup(bool rand);
  void scene_setup_1();
  void scene_setup_2();
  void scene_setup_3();
  double world_time, time_last_update;
  bool opt_pause;

  VPipeline pipe_gear_base;
  VPipeline pipe_gear_prob1;
  VPipeline pipe_gear_prob2;
  VVertex_Buffer_Set bset_gear_strip;
  int bset_size_tooth;

  VBufferV<MT_Colors> uni_colors;

  int opt_shader;
  enum HW_Shader
    { HW_Shader_Base, HW_Shader_Prob1, HW_Shader_Prob2, HW_Shader_SIZE };

  int opt_n_teeth;
  int opt_n_gears;
  int colors_serial;

};

static const char* const hw_shader_str[] = { "BASE", "HW05-P1", "HW05-P2" };



void
World::scene_setup_1()
{
}

void
World::scene_setup_2()
{
}

void
World::scene_setup_3()
{
  // Used for the assignment screenshots.
  vh.clearValues[0].color =
    vk::ClearColorValue( array<float, 4>( { { 0.4, .4, .4, 1.0 } } ) );
  eye_location = pCoor(9.4,1.3,16.6);
  eye_direction = pVect(0,-.28,-.96);
  opt_light_intensity = 14.51;
```

```
}

float
prand(float min, float max)
{
  double r = random() * (1.0 / RAND_MAX );
  return min + r * ( max - min );
}

void
World::gear_setup(bool rand)
{
  world_time = 0;
  time_last_update = time_wall_fp();

  gear_info.serial++;
  gear_info.n_teeth = opt_n_teeth;
  gear_info.radius_inner = 4;
  gear_info.width = 1.5;
  gear_info.land_bot_frac = 0.3;
  gear_info.land_top_frac = 0.1;

  /// SOLUTION -- Problem 1
  //
  //  Compute the hole diameter and write it to the uniform.
  //
  uni_misc->radius_inner = gear_info.radius_inner * 0.5;


  float pitch_theta = 2 * M_PI / gear_info.n_teeth;
  float face_frac =
    ( 1 - gear_info.land_bot_frac - gear_info.land_top_frac ) / 2;
  float width_theta_face = face_frac * pitch_theta;
  float width_it = 1; // Starting point for iterative solution.
  float tan_dtf = tanf(width_theta_face);
  for ( int n_iter = 0; n_iter < 20; n_iter++ )
    {
      float t1 = cosf(width_it) + width_it * sinf(width_it);
      float tanth = ( sinf(width_it) - width_it * cosf(width_it) ) / t1;
      float err = tan_dtf - tanth;
      if ( fabs(err) < 1e-6 ) break;
      // Divide err by derivative of tanth w.r.t. width_it.
      width_it += err * powf( t1 / width_it, 2 );
    }
  float r_outer = gear_info.radius_inner * sqrt( 1 + width_it * width_it );

  int n_gears = opt_n_gears;
  float delta_theta = 2 * M_PI / ( ( n_gears + 1 ) & 0xfffffe );
  float gears_rad =
    1.001 * ( r_outer + gear_info.radius_inner ) / ( 2 * sinf(delta_theta/2) );

  buf_gears_xforms.clear();

  pMatrix m_flip_flip( pVect(1,0,0), pVect(0,-1,0), pVect(0,0,-1) );
```

```
  pMatrix_Rotation m_rot_tooth( pVect(0,0,1), pitch_theta/2 );
  pMatrix m_flip =
    pMatrix_Translate(0,0,-gear_info.width) * m_rot_tooth * m_flip_flip;

  for ( int i=0; i<n_gears; i++ )
    {
      float theta = M_PI + i * delta_theta;
      pMatrix_Rotation rot( pVect(0,0,1), theta );
      pMatrix_Translate trans_to_global( rot * pVect(gears_rad,0,0) );
      pMatrix m = object_orientation * trans_to_global * rot;
      if ( i & 1 ) m = m * m_flip;
      buf_gears_xforms << m;
    }

  buf_gears_xforms.to_dev();
}


void
World::setup_and_run()
{
  // Setup Vulkan context, etc.
  vh.init();
  vh.display_cb_set([&](){});
  vh.cbs_cmd_record.push_back( [&](vk::CommandBuffer& cb){ render(cb); });

  vh.clearValues[0].color =
    vk::ClearColorValue( array<float, 4>( { { 0.3, 0.3, 0.3, 1.0 } } ) );

  opt_tryout1 = opt_tryout2 = false;
  eye_initial_location = pCoor(9.4,1.3,16.6);

  eye_location = eye_initial_location;
  eye_initial_direction = pVect(0,-.28,-.96);
  eye_direction = eye_initial_direction;

  {
    pCoor pos_ref(12.2,-2.3,-1);
    // Names are in gear local space. Values are global space.
    pNorm ax(1,0,0);
    pNorm ay(0,0,-1); // [sic]
    pNorm az = cross(ax,ay);
    object_orientation = pMatrix_Translate(pos_ref) * pMatrix_Cols(ax,ay,az);
  }

  opt_tryoutf = 1;
  variable_control.insert(opt_tryoutf,"opt_tryoutf",1.2);
  opt_n_gears = 6;
  opt_n_teeth = 16;
  gear_info.n_teeth = 0;
  variable_control.insert(opt_n_gears,"Num Gears",1,2);
  variable_control.insert(opt_n_teeth,"Num Teeth",1,3);

  opt_light_intensity = 22;
```

```
  light_location = pCoor( 12.6, 10.8,25.3);
  variable_control.insert(opt_light_intensity,"Light Intensity");

  opt_move_item = MI_Eye;

  uni_misc.init(vh.qs,vk::BufferUsageFlagBits::eUniformBuffer);
  uni_misc->opt_tryout = ivec4(opt_tryout1,opt_tryout2,0,0);
  uni_misc->opt_tryoutf4.x = opt_tryoutf;

  uni_light_simple.init(vh.qs,vk::BufferUsageFlagBits::eUniformBuffer);

  gear_info.serial = 0;

  uni_colors.init(vh.qs,vk::BufferUsageFlagBits::eUniformBuffer);

  buf_gears_xforms.init(vh.qs,vk::BufferUsageFlagBits::eStorageBuffer);

  opt_pause = false;

  opt_shader = 0;
  colors_serial = -1;

  scene_setup_2();

  // Start the graphics. The function below does not return until the
  // user exits by closing the window.
  //
  vh.message_loop_spin();
  //
  // At this point the user exited and so it's time to
  // clean up.

  uni_light_simple.destroy();
  uni_misc.destroy();
  pipe_plain.destroy();
  pipe_gear_base.destroy();
  pipe_gear_prob1.destroy();
  pipe_gear_prob2.destroy();
  bset_plain.destroy();
  bset_gear_strip.destroy();
  shapes.destroy();
  transform.destroy();

  uni_colors.destroy();
  buf_gears_xforms.destroy();

  vh.finish();
}

void
World::render(vk::CommandBuffer& cb)
{
  // This routine called whenever window needs to be updated.
```

```
  // Get any waiting keyboard commands.
  //
  keyboard_handle();

  /// Frame Buffer Informational Messages
  //
  //  Print messages using utility functions provided for this course.
  //
  vh.fbprintf("%s\n",frame_timer.frame_rate_text_get());
  const double time_now = time_wall_fp();
  const bool blink_visible = int64_t(time_now*2) & 1;
# define BLINK(txt,pad) ( blink_visible ? txt : pad )
  vh.fbprintf
    ("Compiled: %s\n",
#ifdef __OPTIMIZE__
     "WITH OPTIMIZATION"
#else
     BLINK("WITHOUT OPTIMIZATION","")
#endif
     );

  vh.fbprintf
    ("Eye location: [%5.1f, %5.1f, %5.1f]  "
     "Eye direction: [%+.2f, %+.2f, %+.2f]\n",
     eye_location.x, eye_location.y, eye_location.z,
     eye_direction.x, eye_direction.y, eye_direction.z);

  vh.fbprintf
    ("Light location: [%5.1f, %5.1f, %5.1f]\n",
     light_location.x, light_location.y, light_location.z);

  vh.fbprintf
    ("Object location: [%5.1f, %5.1f, %5.1f]  "
     "Object direction: [%+.2f, %+.2f, %+.2f]\n",
     object_orientation.rc_get(0,3),
     object_orientation.rc_get(1,3),
     object_orientation.rc_get(2,3),
     0.0,0.0,0.0);

  vh.fbprintf
    ("Shader: %s ('v')  "
     "Tryout 1: %s  ('y')  Tryout 2: %s  ('Y')  \n",
     hw_shader_str[opt_shader],
     opt_tryout1 ? BLINK("ON ","   ") : "OFF",
     opt_tryout2 ? BLINK("ON ","   ") : "OFF");


  pVariable_Control_Elt* const cvar = variable_control.current;
  vh.fbprintf("VAR %s = %.5f  (TAB or '`' to change, +/- to adjust)\n",
                    cvar->name,cvar->get_val());


  // ----------------------------------------------------------------------
  ///
```

```
/// Specification of Transformation Matrices
///

pMatrix eye_from_global =
  pMatrix_Rotation(eye_direction,pVect(0,0,-1))
  * pMatrix_Translate(-eye_location);

transform.eye_from_global_set( eye_from_global );

/// Setup Projection Transformation: Eye Space -> Clip Space
//
const int win_width = vh.s_extent.width;
const int win_height = vh.s_extent.height;
const float aspect = float(win_width) / win_height;
const float n_dist = 0.01;
const float xr = .8 * n_dist;

// Frustum: left, right, bottom, top, near, far
transform.clip_from_eye_set
  ( pMatrix_Frustum( -xr, xr,                // left, right
                     -xr/aspect, xr/aspect,  // bottom, top
                     n_dist, 5000            // near, far
                     ) );

/// Lighting
//
uni_light_simple->color = color_white * opt_light_intensity;
uni_light_simple->position = transform.eye_from_global * light_location;
uni_light_simple.to_dev();
uni_misc->opt_tryoutf4.x = opt_tryoutf;

//
// ----------------------------------------------------------------------------

if ( !pipe_plain )
  pipe_plain
    .init( vh.qs )
    .ds_uniform_use( "BIND_LIGHT_SIMPLE", uni_light_simple )
    .ds_uniform_use( "BIND_MISC", uni_misc )
    .shader_inputs_info_set<pCoor,pNorm,pColor>()
    .shader_code_set
    ("demo-03-shdr.cc", "vs_main();", nullptr, "fs_main();")
    .topology_set( vk::PrimitiveTopology::eTriangleList )
    .create();

if ( !pipe_gear_base )
  pipe_gear_base
    .init( vh.qs )
    .ds_uniform_use( "BIND_LIGHT_SIMPLE", uni_light_simple )
    .ds_uniform_use( "BIND_MISC", uni_misc )
    .ds_uniform_use( "BIND_HW05", uni_colors )
    .shader_inputs_info_set<pCoor,pNorm,int>()
    .shader_code_set
    ("gear-base-shdr.cc",
```

```
      "vs_main_list();", "gs_main_list();", "fs_main_list();")
    .topology_set( vk::PrimitiveTopology::eTriangleStrip )
    .create();

if ( !pipe_gear_prob1 )
  pipe_gear_prob1
    .init( vh.qs )
    .ds_uniform_use( "BIND_LIGHT_SIMPLE", uni_light_simple )
    .ds_uniform_use( "BIND_MISC", uni_misc )
    .ds_uniform_use( "BIND_HW05", uni_colors )
    .shader_inputs_info_set<pCoor,pNorm,int>()
    .shader_code_set
    ("hw05-shdr-sol.cc",
     "vs_main_prob1();", "gs_main_prob1();", "fs_main();",
     "#define HW05_PROB1\n")
    .topology_set( vk::PrimitiveTopology::eTriangleStrip )
    .create();

if ( !pipe_gear_prob2 )
  pipe_gear_prob2
    .init( vh.qs )
    .ds_uniform_use( "BIND_LIGHT_SIMPLE", uni_light_simple )
    .ds_uniform_use( "BIND_MISC", uni_misc )
    .ds_uniform_use( "BIND_HW05", uni_colors )
    .ds_storage_follow( "BIND_XF", buf_gears_xforms )
    .shader_inputs_info_set<pCoor,pNorm,int>()
    .shader_code_set
    ("hw05-shdr-sol.cc",
     "vs_main_prob2();", "gs_main_prob2();", "fs_main();",
     "#define HW05_PROB2\n")
    .topology_set( vk::PrimitiveTopology::eTriangleStrip )
    .create();


// Reset buffers in this buffer set.
//
bset_plain.reset(pipe_plain);

// Prepare a triangle.
//
pCoor p0 = { 9, 0,  -10 };
pCoor p1 = { 18, 6, -4 };
pCoor p2 = { 9, 5, -9 };
pNorm tri_norm = cross( p0, p1, p2 );
bset_plain << p0 << p1 << p2;
bset_plain << color_green << color_red << color_blue;
bset_plain << tri_norm << tri_norm << tri_norm;

// Add a square consisting of a red and green triangle.
//
bset_plain << pCoor(-2,0,-4) << pCoor(-4,2,-4) << pCoor(-4,0,-4);
bset_plain << color_red << color_red << color_red;
bset_plain << pCoor(-2,0,-4) << pCoor(-2,2,-4) << pCoor(-4,2,-4);
bset_plain << color_green << color_green << color_green;
```

```
pNorm snorm = cross( pCoor(-4,0,-4), pCoor(-4,2,-4), pCoor(-2,0,-4) );
bset_plain << snorm << snorm << snorm << snorm << snorm << snorm;

bset_plain.to_dev();
pipe_plain.ds_set( transform );
pipe_plain.record_draw(cb, bset_plain);

// Update theta_0, the angle by which the gear should be at.
//
if ( !opt_pause ) world_time += time_now - time_last_update;
time_last_update = time_now;
const float omega = 0.5;
const float theta_0 = world_time * omega;

const bool gear_stale =
  opt_n_teeth != gear_info.n_teeth || opt_n_gears != buf_gears_xforms.size();

if ( gear_stale )
  {
    gear_info.n_teeth = opt_n_teeth;
    gear_setup(false);
    gear_info.serial++;
  }

const bool gear_bset_stale = gear_stale;

const auto& f = gear_info;

if ( colors_serial != f.serial )
  {
    colors_serial = f.serial;
    uni_colors->front[0] = color_lsu_spirit_purple;
    uni_colors->back[0] = color_lsu_business_purple;
    uni_colors->front[1] = color_lsu_spirit_gold;
    uni_colors->back[1] = color_gold;
    uni_colors->front[2] = color_aquamarine;
    uni_colors->back[2] = color_salmon;
    uni_colors->front[3] = color_aquamarine;
    uni_colors->back[3] = color_salmon;
    uni_colors->front[4] = color_gray;
    uni_colors->back[4] = color_red;
    uni_colors->front[5] = color_light_gray;
    uni_colors->back[5] = color_orange_red;
    uni_colors->front[6] = color_light_steel_blue;
    uni_colors->back[6] = color_light_steel_blue;
    uni_colors.to_dev();
  }

if ( gear_bset_stale )
  {
    /// Homework 5: Do not modify the code preparing the buffer set (below).

    bset_gear_strip.reset(pipe_gear_base); // Will work for all pipes.
```

```
float n_teeth = f.n_teeth;
float pitch_theta = 2 * M_PI / n_teeth;
float th_resolution_target = 2 * M_PI / 300;

float face_frac = ( 1 - f.land_bot_frac - f.land_top_frac ) / 2;

float width_theta_face = face_frac * pitch_theta;
float width_theta_top = f.land_top_frac * pitch_theta;
float width_theta_bot = f.land_bot_frac * pitch_theta;

float off_theta_top = width_theta_face;
float off_land_bot = pitch_theta * ( 2*face_frac + f.land_top_frac );

int n_slices_top =
  max(1, int(0.5 + width_theta_top / th_resolution_target));
float delta_theta_top = width_theta_top / n_slices_top;

int n_slices_bot =
  max(1, int(0.5 + width_theta_bot / th_resolution_target));
float delta_theta_bot = width_theta_bot / n_slices_bot;

const int n_slices_invo = max(5, int(pitch_theta/th_resolution_target) );

pVect vz(0,0,-f.width);
float r_inner = f.radius_inner;
pCoor p1(0,0,0);
pCoor p4 = p1 + vz;

// width_it: difference, WIDTH, in Involute curve parameter, T.
float width_it = 1; // Starting point for iterative solution.
float tan_dtf = tanf(width_theta_face);
for ( int n_iter = 0; n_iter < 20; n_iter++ )
  {
    float t1 = cosf(width_it) + width_it * sinf(width_it);
    float tanth = ( sinf(width_it) - width_it * cosf(width_it) ) / t1;
    float err = tan_dtf - tanth;
    if ( fabs(err) < 1e-6 ) break;
    // Divide err by derivative of tanth w.r.t. width_it.
    width_it += err * powf( t1 / width_it, 2 );
  }
float delta_it_invo = width_it / n_slices_invo;
float r_outer = r_inner * sqrt( 1 + width_it * width_it );
bset_size_tooth = -1;

/// Homework 5: Do not modify the code preparing the buffer set.

for ( int i=0; i<n_teeth; i++ )
  {
    float theta = i * pitch_theta;

    // Emit vertices for one face of a tooth.
    for ( int j=0; j<=n_slices_invo; j++ )
      {
        float d_theta_a = j * delta_it_invo;
```

```
    float theta_invo = theta + d_theta_a;
    pVect v_invo = r_inner *
      pVect( cosf(theta_invo) + d_theta_a * sinf(theta_invo),
             sinf(theta_invo) - d_theta_a * cosf(theta_invo), 0 );
    pVect d_invo( cosf(theta_invo), sinf(theta_invo), 0);
    pNorm np = cross( vz, d_invo );

    // Emit Colors, Coordinates, and Normals
    if ( i < 1 ) bset_gear_strip << 0 << 1;
    else         bset_gear_strip << 2 << 3;
    bset_gear_strip << p1 + v_invo << p4 + v_invo;
    bset_gear_strip << np << np;
  }

// Emit vertices for the top land (outer cylinder).
for ( int j=0; j<=n_slices_top; j++ )
  {
    float theta_top = theta + off_theta_top + j * delta_theta_top;
    pVect n_land( cosf(theta_top), sinf(theta_top), 0 );
    pVect v_land = r_outer * n_land;

    // Emit Colors, Coordinates, and Normals
    if ( i < 1 ) bset_gear_strip << 4 << 5;
    else         bset_gear_strip << 2 << 3;
    bset_gear_strip << p1 + v_land << p4 + v_land;
    bset_gear_strip << n_land << n_land;
  }

// Emit vertices for other face of a tooth.
for ( int j=n_slices_invo; j>=0; j-- )
  {
    float d_theta_a = -j * delta_it_invo;
    float theta_invo = theta + off_land_bot + d_theta_a;
    pVect v_invo = r_inner *
      pVect( cosf(theta_invo) + d_theta_a * sinf(theta_invo),
             sinf(theta_invo) - d_theta_a * cosf(theta_invo), 0 );
    pVect d_invo( cosf(theta_invo), sinf(theta_invo), 0);
    pNorm np = cross(vz,d_invo);

    // Emit Colors, Coordinates, and Normals
    if ( i < 1 ) bset_gear_strip << 0 << 1;
    else         bset_gear_strip << 2 << 3;
    bset_gear_strip << p1 + v_invo << p4 + v_invo;
    bset_gear_strip << np << np;
  }

// Emit vertices for the bottom land (inner cylinder).
for ( int j=0; j<=n_slices_bot; j++ )
  {
    float theta_bot = theta + off_land_bot + j * delta_theta_bot;
    pVect n_land( cosf(theta_bot), sinf(theta_bot), 0 );
    pVect v_land = r_inner * n_land;

    // Emit Coordinates and Normals
```

```
            if ( i < 1 ) bset_gear_strip << 4 << 5;
            else         bset_gear_strip << 2 << 3;
            bset_gear_strip << p1 + v_land << p4 + v_land;
            bset_gear_strip << n_land << n_land;
          }

        if ( i == 0 ) bset_size_tooth = bset_gear_strip.size();
      }

    bset_gear_strip.to_dev();

    /// Homework 5: Do not modify the code preparing the buffer set (above).
  }

pMatrix_Rotation rot(pVect(0,0,1),theta_0);

switch ( opt_shader ) {
case HW_Shader_Base:
  for ( auto m: buf_gears_xforms )
    {
      pipe_gear_base.ds_set( transform * m * rot );
      pipe_gear_base.record_draw(cb, bset_gear_strip );
    }
  break;

case HW_Shader_Prob1:
  for ( auto m: buf_gears_xforms )
    {
      pipe_gear_prob1.ds_set( transform * m * rot );
      pipe_gear_prob1.record_draw(cb, bset_gear_strip );
    }
  break;

case HW_Shader_Prob2:

  /// SOLUTION -- Problem 2
  //
  //  Put the rotation matrix into the uniform.
  //
  uni_misc->rot = rot;

  // Set the transform only to the current global space.
  // The shader will retrieve buf_gears_xform[] and rat and use
  // them to compute the transformation.
  //
  pipe_gear_prob2.ds_set( transform );

  // Do an instanced draw, with one instance per transform in
  // buf_gears_xforms.
  //
  pipe_gear_prob2.record_draw_instanced
    ( cb, bset_gear_strip, buf_gears_xforms.size() );

  break;
```

```
    // Original Code Below
    pipe_gear_prob2.ds_set( transform * buf_gears_xforms[0] * rot );
    pipe_gear_prob2.record_draw_instanced( cb, bset_gear_strip, 1 );

    break;

  default:
    assert( false );

  };

  uni_misc.to_dev();

  // Render Marker for Light Source
  //
  shapes.record_tetrahedron(cb,transform,light_location,0.2);
}



void
World::keyboard_handle()
{
  const int key = vh.keyboard_key_get();
  if ( !key ) return;
  pVect adjustment(0,0,0);
  pVect user_rot_axis(0,0,0);
  const bool kb_mod_s = vh.keyboard_shift;
  const bool kb_mod_c = vh.keyboard_control;
  const float move_amt = kb_mod_s ? 2.0 : kb_mod_c ? 0.08 : 0.4;

  switch ( key ) {
  case FB_KEY_LEFT: adjustment.x = -move_amt; break;
  case FB_KEY_RIGHT: adjustment.x = move_amt; break;
  case FB_KEY_PAGE_UP: adjustment.y = move_amt; break;
  case FB_KEY_PAGE_DOWN: adjustment.y = -move_amt; break;
  case FB_KEY_DOWN: adjustment.z = move_amt; break;
  case FB_KEY_UP: adjustment.z = -move_amt; break;
  case FB_KEY_DELETE: user_rot_axis.y = 1; break;
  case FB_KEY_INSERT: user_rot_axis.y =  -1; break;
  case FB_KEY_HOME: user_rot_axis.x = 1; break;
  case FB_KEY_END: user_rot_axis.x = -1; break;
  case '1': scene_setup_1(); break;
  case '2': scene_setup_2(); break;
  case '3': scene_setup_3(); break;
  case 'b': case 'B': opt_move_item = MI_Ball; break;
  case 'e': case 'E': opt_move_item = MI_Eye; break;
  case 'l': case 'L': opt_move_item = MI_Light; break;
  case 'p': case 'P': opt_pause = !opt_pause; break;
  case 'v': case 'V':
    opt_shader++; if ( opt_shader == HW_Shader_SIZE ) opt_shader = 0;
    break;
  case 'y': opt_tryout1 = !opt_tryout1;
```

```
    uni_misc->opt_tryout.x = int(opt_tryout1);
    break;
  case 'Y': opt_tryout2 = !opt_tryout2;
    uni_misc->opt_tryout.y = int(opt_tryout2);
    break;
  case FB_KEY_TAB:
    if ( !kb_mod_s ) { variable_control.switch_var_right(); break; }
  case 96: variable_control.switch_var_left(); break;
  case '-':case '_': variable_control.adjust_lower(); break;
  case '+':case '=': variable_control.adjust_higher(); break;
  default: printf("Unknown key, %d\n",key); break;
  }

  // Update eye_direction based on keyboard command.
  //
  if ( user_rot_axis.x || user_rot_axis.y )
    {
      if ( opt_move_item == MI_Ball )
        {
          pMatrix_Rotation rot(user_rot_axis, M_PI * 0.03);
          pMMultiply3x3(object_orientation, rot, object_orientation);
          gear_setup(false);
        }
      else
        {
          pMatrix_Rotation rotall(eye_direction,pVect(0,0,-1));
          user_rot_axis *= invert(rotall);
          eye_direction *= pMatrix_Rotation(user_rot_axis, M_PI * 0.03);
        }
    }

  // Update eye_location based on keyboard command.
  //
  if ( adjustment.x || adjustment.y || adjustment.z )
    {
      const double angle =
        fabs(eye_direction.y) > 0.99
        ? 0 : atan2(eye_direction.x,-eye_direction.z);
      pMatrix_Rotation rotall(pVect(0,1,0),-angle);
      adjustment *= rotall;

      switch ( opt_move_item ){
      case MI_Light: light_location += adjustment; break;
      case MI_Eye: eye_location += adjustment; break;
      case MI_Ball:
        object_orientation = pMatrix_Translate(adjustment) * object_orientation;
        gear_setup(false);
        break;
      default: break;
      }
    }
}
```

```
int
main(int argv, char **argc)
{
  pVulkan_Helper pvulkan_helper(argv,argc);
  World world(pvulkan_helper);

  world.setup_and_run();

  return 0;
}
```

# 22 Fall 2021 Solutions

```
/// LSU EE 4702-1 (Fall 2021), GPU Programming
//
 /// Homework 1 -- SOLUTION
 //
 //  Based on demo-05.

/// What Code Does

// Shows a sphere and a triangle.


///  Keyboard Commands
 //
 /// Object (Eye, Light, Ball) Location or Push
 //   Arrows (←,→,↑,↓) Page Up, Page Down
 //        Move object or push ball, depending on mode.
 //        Shift + KEY: motion is 5x faster.
 //        Ctrl + KEY : motion is 5x slower.
 //   'e': Move eye.
 //   'l': Move light.
 //   'b': Move sphere.
 //
 /// Eye Direction and GUI Options
 //
 //   'Home', 'End', 'Delete', 'Insert'
 //         Turn the eye direction. Home should rotate eye direction
 //         up, End should rotate eye down, Delete should rotate eye
 //         left, Insert should rotate eye right. The eye direction
 //         vector is displayed in the upper left.
 //  'C-='  (Ctrl =) Increase green text size.
 //  'C--'  (Ctrl -) Decrease green text size.
 //  'F12'  Write screenshot to PNG file.

 /// Simulation Options
 //  (Also see variables below.)
 //
 //  'F12'  Write screenshot to file.

 /// Variables
 //   Selected program variables can be modified using the keyboard.
 //   Use "Tab" to cycle through the variable to be modified, the
 //   name of the variable is displayed next to "VAR" on the bottom
 //   line of green text.

 //  'Tab' Cycle to next variable.
 //  '`'   Cycle to previous variable.
 //  '+'   Increase variable value.
 //  '-'   Decrease variable value.
 //
 //  VAR Light Intensity - The light intensity.
```

```
// Include files provided for this course.
//
#define MAIN_INCLUDE
#include <vhelper.h>

#include <vstroke.h>
#include <gp/coord.h>
#include <gp/pstring.h>
#include <gp/misc.h>
#include <gp/colors.h>

#include <vutil-texture.h>
#include <vutil-pipeline.h>
#include "shapes.h"


class World {
public:
  World(pVulkan_Helper &vh)
    :vh(vh),ff_state(vh.qs),frame_timer(vh.frame_timer),shapes(ff_state),
     transform(vh.qs){}
  void setup_and_run();
  void render(vk::CommandBuffer& cb);
  void cb_keyboard();

  // Class providing utilities, such as showing text.
  //
  pVulkan_Helper& vh;
  VFixed_Function_State_Manager ff_state;

  // Class for showing frame timing.
  //
  pFrame_Timer& frame_timer;

  Shapes shapes;

  // Class for easy keyboard control of variables.
  //
  pVariable_Control variable_control;

  bool opt_show_lines; // Set normal to triangle normal.

  pCoor light_location;
  float opt_light_intensity;
  enum { MI_Eye, MI_Light, MI_Ball, MI_Ball_V, MI_COUNT } opt_move_item;

  bool opt_tryout1, opt_tryout2, opt_tryout3;
  float opt_tryoutf;

  int slices; // Level of detail for sphere.
  pCoor sphere_location;
  float sphere_radius;

  pCoor eye_location;
```

```
  pVect eye_direction;

  bool global_transform_stale;

  VTransform transform;
  VBufferV<Uni_Lighting> uni_light, *puni_light;
  VPipeline pipe_lonely;
  VPipeline pipe_sphere;
  VVertex_Buffer_Set bset_lonely, bset_sphere;

  /// For Homework 1 (2021)
  //
  bool opt_recompute;
  int opt_n_objects;
  int slices_seen;
  //
  // Homework 1: Feel free to declare new members.

};

void
World::setup_and_run()
{
  // Set up Vulkan.
  //
  vh.init();

  // Set per-frame callback.
  //
  vh.display_cb_set([&](){});
  //
  // In this case it is being set to an empty lambda function.

  // Set render command recording callback.
  //
  vh.cbs_cmd_record.push_back( [&](vk::CommandBuffer& cb){ render(cb); });

  eye_location = pCoor(1,0.5,3);
  eye_location = pCoor(1,.5,10.2);
  eye_direction = pVect(0,0,-1);

  opt_light_intensity = 4.3;
  light_location = pCoor(6.2,0.0,3.7);

  opt_tryout1 = opt_tryout2 = opt_tryout3 = false;
  opt_tryoutf = 0.8;
  variable_control.insert(opt_tryoutf,"tryoutf");

  sphere_location = pCoor(-1,0,-0.1);
  sphere_radius = 2;

  slices = 20;
  variable_control.insert(slices,"Slices in Sphere");
```

```
  variable_control.insert(opt_light_intensity,"Light Intensity");
  variable_control.insert(sphere_radius,"sphere_radius");

  opt_move_item = MI_Eye;

  opt_show_lines = false;

  uni_light.init(vh.dev_phys,vh.dev,vk::BufferUsageFlagBits::eUniformBuffer);
  pColor black(0,0,0,0);
  uni_light->cgl_LightModel.ambient = pColor(.4,.4,.4,1);
  uni_light->cgl_LightSource[0].position = light_location;
  uni_light->cgl_LightSource[0].diffuse = pColor(1,1,1,1);
  uni_light->cgl_LightSource[0].ambient = black;
  uni_light->cgl_LightSource[0].specular = black;
  uni_light->cgl_LightSource[0].constantAttenuation = .5;
  uni_light->cgl_LightSource[0].linearAttenuation = 1.0;
  uni_light->cgl_LightSource[0].quadraticAttenuation = 0;

  puni_light = &uni_light;

  opt_recompute = true;
  opt_n_objects = 3;
  slices_seen = -1;

  variable_control.insert(opt_n_objects,"opt_n_objects",1,1);

  // Start the graphics. The function below does not return until the
  // user exits by closing the window.
  //
  vh.message_loop_spin();
  //
  // At this point the user exited and so it's time to
  // clean up.

  uni_light.destroy();
  pipe_lonely.destroy();
  pipe_sphere.destroy();
  bset_lonely.destroy();
  bset_sphere.destroy();
  shapes.destroy();

  vh.finish();
}

void
World::render(vk::CommandBuffer& cb)
{
  // This routine called whenever window needs to be updated.

  // Get any waiting keyboard commands.
  //
  cb_keyboard();

  /// Frame Buffer Informational Messages
```

```
  //
  //  Print messages using utility functions provided for this course.
  //
  vh.fbprintf("%s\n",frame_timer.frame_rate_text_get());

  const double time_now = time_wall_fp();
  const bool blink_visible = int64_t(time_now*3) & 1;
# define BLINK(txt,pad) ( blink_visible ? txt : pad )

  vh.fbprintf
    ("Compiled: %s\n",
#ifdef __OPTIMIZE__
     "WITH OPTIMIZATION"
#else
     BLINK("WITHOUT OPTIMIZATION","")
#endif
     );

  vh.fbprintf
    ("Eye location: [%5.1f, %5.1f, %5.1f]  "
     "Eye direction: [%+.2f, %+.2f, %+.2f]\n",
     eye_location.x, eye_location.y, eye_location.z,
     eye_direction.x, eye_direction.y, eye_direction.z);

  vh.fbprintf
    ("Light location: [%5.1f, %5.1f, %5.1f]  Render Using: ('n') %s\n",
     light_location.x, light_location.y, light_location.z,
     opt_show_lines ? "LINES" : "TRIANGLES" );

  vh.fbprintf
    ("Recompute: %s ('r')  "
     "Tryout 1: %s  ('y')  Tryout 2: %s  ('Y')  Tryout 3: %s ('Z')\n",
     opt_recompute ? " ON" : "OFF",
     opt_tryout1 ? BLINK("ON ","   ") : "OFF",
     opt_tryout2 ? BLINK("ON ","   ") : "OFF",
     opt_tryout3 ? BLINK("ON ","   ") : "OFF"
     );

  pVariable_Control_Elt* const cvar = variable_control.current;
  vh.fbprintf("VAR %s = %.5f  (TAB or '`' to change, +/- to adjust)\n",
                     cvar->name,cvar->get_val());


  // ----------------------------------------------------------------------
  ///
  /// Specification of Transformation Matrices
  ///

  /// Set Transformation Matrix for:  Global Space -> Eye Space
  //
  transform.eye_from_global_set
    ( pMatrix_Rotation(eye_direction,pVect(0,0,-1))
      * pMatrix_Translate(-eye_location) );
  //
```

```
// This matrix applies to all rendering pipelines.

/// Set Transformation Matrix for: Eye Space -> Clip Space
//
const int win_width = vh.get_width();
const int win_height = vh.get_height();
const float aspect = float(win_width) / win_height;
//
// Frustum: left, right, bottom, top, near, far
transform.clip_from_eye_set
  ( pMatrix_Frustum(-.8,.8,-.8/aspect,.8/aspect,1,5000) );
//
// This matrix also applies to all rendering pipelines.

/// Set Location and Intensity of Light.
//
uni_light->cgl_LightSource[0].diffuse =
  { opt_light_intensity, opt_light_intensity, opt_light_intensity, 1 };
uni_light->cgl_LightSource[0].position =
  transform.eye_from_global * light_location;
//
// Note that the light location is being specified in eye space.
//
// Send updated light information to device.
//
uni_light.to_dev();

//
// ---------------------------------------------------------------------------


///
/// Paint Single Triangle.
///

if ( !pipe_lonely )
  pipe_lonely
    .init( vh.qs )
    .use_uni_light( uni_light )
    .color_uniform_back_set( color_red )
    .lighting_on()
    .topology_set( vk::PrimitiveTopology::eTriangleList )
    .create();

bset_lonely.reset(pipe_lonely);
transform.use_for(pipe_lonely);

/// Specify normal for triangle.
//
// Use cross product function (in coord.h) to find normal.
//

// Object-Space Coordinates
//
```

```
pCoor p0 = { 0, 0,  0 };
pCoor p1 = { 9, 6, -4 };
pCoor p2 = { 0, 5, -3 };

vector<pCoor> tri_vtx = { p0, p1, p2 };

pNorm tri_norm = cross( p0, p1, p2 );
pColor color_tri( .470, .553, .965 ); // Red, Green, Blue

bset_lonely << color_green << tri_norm << p0;
bset_lonely << color_lsu_spirit_gold << tri_norm << p1;
bset_lonely << color_lsu_spirit_purple << tri_norm << p2;


/// Homework 1, Problem 1: Draw additional triangles closer to the light.
//  The number of triangles to draw is in opt_n_objects.

// Problem 1 Hint:
//   Compute v0l, a vector from triangle vertex 0 to light.
//   Use v0l (and other information) to find the coordinate
//   of one of the vertex of each additional triangle.
//
pVect v0l(p0,light_location);

/// SOLUTION -- Problem 1
//
for ( int i=0; i<opt_n_objects; i++ )
  {
    vector<pCoor> nvtx;
    for ( auto& p: tri_vtx )
      {
        pVect vpl(p,light_location);
        pCoor cpl = p + (i+1.0)/(opt_n_objects+1)* vpl;
        bset_lonely << color_light_sea_green << cpl;
        nvtx.push_back(cpl);
      }
    pNorm n = cross(nvtx[0],nvtx[1],nvtx[2]);
    if ( opt_tryout1 )
      bset_lonely << n << n << n;
    else
      bset_lonely << tri_norm << tri_norm << tri_norm;
  }


bset_lonely.to_dev();
pipe_lonely.record_draw(cb, bset_lonely);


///
/// Paint a Sphere
///

auto prim_want = opt_show_lines
  ? vk::PrimitiveTopology::eLineStrip
```

```
    : vk::PrimitiveTopology::eTriangleList;

  if ( pipe_sphere.p_in_asm_ci.topology != prim_want )
    pipe_sphere.destroy();

  if ( !pipe_sphere )
    pipe_sphere
      .init( vh.qs )
      .use_uni_light( uni_light )
      .lighting_on()
      .topology_set( prim_want )
      .create();

  // Construct color objects using hex RGB codes. See coord.h and colors.h.
  //
  const pColor lsu_spirit_purple(0x580da6);
  const pColor lsu_spirit_gold(0xf9b237);

  pColor color = lsu_spirit_gold;

  const float delta_eta = M_PI / slices;

  /// SOLUTION -- Problem 3
  //
  const bool sphere_stale = slices != slices_seen || opt_recompute;

  if ( sphere_stale )
    {

  bset_sphere.reset(pipe_sphere);

  // Outer (eta) Loop: Iterate over longitude (north-to-south).
  // Inner (theta) Loop: Iterate over latitude (east-to-west)
  //
  for ( int slice = 0; slice < slices - 1; slice++ )
    {
      const float eta0 = slice * delta_eta;
      const float eta1 = eta0 + delta_eta;
      const float y0 = cosf(eta0),        y1 = cosf(eta1);
      const float slice_r0 = sinf(eta0),  slice_r1 = sinf(eta1);
      const float delta_theta = delta_eta * slice_r1;

      for ( float theta0 = 0; theta0 < 2 * M_PI; theta0 += delta_theta )
        {
          const float theta1 = theta0 + delta_theta;

          /// Triangle 1

          // Vertex 1
          bset_sphere
            << color
            << pNorm( slice_r1 * cosf(theta0), y1, slice_r1 * sinf(theta0) )
            << pCoor( slice_r1 * cosf(theta0), y1, slice_r1 * sinf(theta0) )
```

```
            // Vertex 2
            << color
            << pNorm( slice_r0 * cosf(theta0), y0, slice_r0 * sinf(theta0) )
            << pCoor( slice_r0 * cosf(theta0), y0, slice_r0 * sinf(theta0) )

            // Vertex 3
            << color
            << pNorm( slice_r1 * cosf(theta1), y1, slice_r1 * sinf(theta1) )
            << pCoor( slice_r1 * cosf(theta1), y1, slice_r1 * sinf(theta1) )

            /// Triangle 2

            // Vertex 3
            << color
            << pNorm( slice_r1 * cosf(theta1), y1, slice_r1 * sinf(theta1) )
            << pCoor( slice_r1 * cosf(theta1), y1, slice_r1 * sinf(theta1) )

            // Vertex 2
            << color
            << pNorm( slice_r0 * cosf(theta0), y0, slice_r0 * sinf(theta0) )
            << pCoor( slice_r0 * cosf(theta0), y0, slice_r0 * sinf(theta0) )

            // Vertex 4
            << color
            << pNorm( slice_r0 * cosf(theta1), y0, slice_r0 * sinf(theta1) )
            << pCoor( slice_r0 * cosf(theta1), y0, slice_r0 * sinf(theta1) );
        }
    }

slices_seen = slices;
bset_sphere.to_dev();

}


// In sphere's coordinate space the sphere center is at the origin
// and its radius is one. Therefore we need to specify a local to
// global (global from local) transformation matrix so that sphere
// is at the value of coordinate sphere_location and its radius is
// sphere_radius.
//
pMatrix glo_from_loc =
  pMatrix_Translate( sphere_location ) * pMatrix_Scale(sphere_radius);
transform.global_from_local_set_for( glo_from_loc, pipe_sphere );
//
// This makes a copy of the global_from_local transformation matrix
// each time it is called. Because it is a copy it does not affect
// other pipeline draw operations.

pipe_sphere.record_draw(cb, bset_sphere);

/// Homework 1, Problem 2
//
//  The code below is not a solution to problem 2, but it might help
```

```
  //  you get started.

  if ( opt_tryout3 )
    {
      pMatrix_Translate move_up( 0, 2*sphere_radius*opt_tryoutf,0 );
      // One of the values of opt_tryout1 is correct. Which is it?
      pMatrix glo_from_loc_2 =
        opt_tryout1 ? move_up * glo_from_loc : glo_from_loc * move_up;
      transform.global_from_local_set_for( glo_from_loc_2, pipe_sphere );
      pipe_sphere.record_draw(cb, bset_sphere);
    }

  /// SOLUTION -- Problem 2
  //
  pNorm stol(sphere_location,light_location);
  float s0_dist = 0;
  float radius_last = sphere_radius;

  for ( int i=0; i<opt_n_objects; i++ )
    {
      float radius_here = radius_last * 0.5;
      float dist_to_ctr = s0_dist + radius_last + radius_here;
      pCoor sloc2 = sphere_location + dist_to_ctr * stol;

      transform.global_from_local_set_for
        ( pMatrix_Translate( sloc2 ) * pMatrix_Scale(radius_here),
          pipe_sphere );

      pipe_sphere.record_draw(cb, bset_sphere);

      s0_dist = dist_to_ctr;
      radius_last = radius_here;

    }

  // Render Marker for Light Source
  //
  shapes.record_tetrahedron(cb,transform,light_location,0.2);

}



void
World::cb_keyboard()
{
  const int key = vh.keyboard_key_get();
  if ( !key ) return;
  pVect adjustment(0,0,0);
  pVect user_rot_axis(0,0,0);
  const bool kb_mod_s = vh.keyboard_shift;
  const bool kb_mod_c = vh.keyboard_control;
  const float move_amt = kb_mod_s ? 2.0 : kb_mod_c ? 0.08 : 0.4;
```

```
switch ( key ) {
case FB_KEY_LEFT: adjustment.x = -move_amt; break;
case FB_KEY_RIGHT: adjustment.x = move_amt; break;
case FB_KEY_PAGE_UP: adjustment.y = move_amt; break;
case FB_KEY_PAGE_DOWN: adjustment.y = -move_amt; break;
case FB_KEY_DOWN: adjustment.z = move_amt; break;
case FB_KEY_UP: adjustment.z = -move_amt; break;
case FB_KEY_DELETE: user_rot_axis.y = 1; break;
case FB_KEY_INSERT: user_rot_axis.y =  -1; break;
case FB_KEY_HOME:
  if ( opt_move_item == MI_Light ) adjustment.x = 1; else user_rot_axis.x = 1;
  break;
case FB_KEY_END: user_rot_axis.x = -1; break;
case 'b': case 'B': opt_move_item = MI_Ball; break;
case 'e': case 'E': opt_move_item = MI_Eye; break;
case 'l': case 'L': opt_move_item = MI_Light; break;
case 'n': opt_show_lines = !opt_show_lines; break;

case 'r': case 'R': opt_recompute = !opt_recompute; break;

case 'y': opt_tryout1 = !opt_tryout1; break;
case 'Y': opt_tryout2 = !opt_tryout2; break;
case 'Z': opt_tryout3 = !opt_tryout3; break;

case FB_KEY_TAB:
  if ( !kb_mod_s ) { variable_control.switch_var_right(); break; }
case 96: variable_control.switch_var_left(); break;
case '-':case '_': variable_control.adjust_lower(); break;
case '+':case '=': variable_control.adjust_higher(); break;
default: printf("Unknown key, %d\n",key); break;
}

// Update eye_direction based on keyboard command.
//
if ( user_rot_axis.x || user_rot_axis.y )
  {
    pMatrix_Rotation rotall(eye_direction,pVect(0,0,-1));
    user_rot_axis *= invert(rotall);
    eye_direction *= pMatrix_Rotation(user_rot_axis, M_PI * 0.03);
  }

// Update eye_location based on keyboard command.
//
if ( adjustment.x || adjustment.y || adjustment.z )
  {
    if ( opt_move_item != MI_Ball && opt_move_item != MI_Ball_V )
      global_transform_stale = true;
    const double angle =
      fabs(eye_direction.y) > 0.99
      ? 0 : atan2(eye_direction.x,-eye_direction.z);
    pMatrix_Rotation rotall(pVect(0,1,0),-angle);
    adjustment *= rotall;
```

```
      switch ( opt_move_item ){
      case MI_Light: light_location += adjustment; break;
      case MI_Eye: eye_location += adjustment; break;
      case MI_Ball: sphere_location += adjustment; break;
      default: break;
      }
    }
}


int
main(int argv, char **argc)
{
  pVulkan_Helper pvulkan_helper(argv,argc);
  World world(pvulkan_helper);

  world.setup_and_run();

  return 0;
}
```

```
/// LSU EE 4702-1 (Fall 2021), GPU Programming
//

 /// Homework 2 -- SOLUTION


// Specify version of OpenGL Shading Language.
//
#version 460

#extension GL_GOOGLE_include_directive : enable
#include <light.h>
#include <transform.h>
#include "links-shdr-common.h"
#include "shader-common-generic-lighting.h"



layout ( binding = BIND_SPHERE_POS ) buffer spr { vec4 sphere_pos_rad[]; };
layout ( binding = BIND_SPHERE_ROT ) buffer sr { mat4 sphere_rot[]; };
layout ( binding = BIND_SPHERE_COLOR ) buffer sc { vec4 sphere_color[]; };

layout ( binding = BIND_UNI_COMMON ) uniform Common_Uniform
{
  Shdr_Uni_Common com;
};

bool opt_tryout1 = bool(com.tryout.x);
bool opt_tryout2 = bool(com.tryout.y);
float opt_tryoutf = com.tryoutf.x;
float opt_hw02_curl = com.tryoutf.y;



#ifdef _VERTEX_SHADER_

layout ( location = LOC_IN_POS ) in vec4 in_vertex_o;

#ifdef LOC_IN_TCOOR
layout ( location = LOC_IN_TCOOR ) in vec2 in_tex_coor;
#endif

// Interface block for vertex shader output / geometry shader input.
//
layout ( location = 0 ) out Data_to_GS
{
  vec3 normal_e;
  vec4 vertex_e;
  vec2 tex_coor;
  vec4 color;
  vec4 vertex_c;

  // Any changes here must also be made to the fragment shader input.
```

```
  /// SOLUTION -- Problem 1
  //
  //  Send along the sphere radius.
  //
  float sphere_radius;
  //
  //  The sphere radius will be used to compute the distance between
  //  the shrunken triangles and the original ones.

};

void
vs_main_instances_sphere()
{
  vec4 pos_rad = sphere_pos_rad[gl_InstanceIndex];
  float rad = pos_rad.w;
  mat4 rot = sphere_rot[gl_InstanceIndex];
  vec4 normr = rot * in_vertex_o;
  vec3 normal_o = normr.xyz;
  vec4 vertex_o = vec4( pos_rad.xyz + rad * normal_o, 1 );

  vertex_c = gl_ModelViewProjectionMatrix * vertex_o;
  vertex_e = gl_ModelViewMatrix * vertex_o;
  normal_e = normalize(gl_NormalMatrix * normal_o );
  tex_coor = in_tex_coor;
  color = sphere_color[gl_InstanceIndex];

  /// SOLUTION
  //
  //  Pass the radius along to the geometry shader.
  sphere_radius = rad;
}
#endif


#ifdef _GEOMETRY_SHADER_

layout ( location = 0 ) in Data_to_GS
{
  vec3 normal_e;
  vec4 vertex_e;
  vec2 tex_coor;
  vec4 color;
  vec4 vertex_c;

  // SOLUTION
  float sphere_radius;
} In[];

layout ( location = 0 ) out Data_to_FS
{
  vec3 normal_e;
  vec4 vertex_e;
```

```
  vec2 tex_coor;
  flat vec4 color;

  // SOLUTION
  flat int use_tex;

};

// Type of primitive at geometry shader input.
//
layout ( triangles ) in;

// Type of primitives emitted geometry shader output.
//


// layout ( triangle_strip, max_vertices = 3 ) out;

 /// SOLUTION
 //
const int slices = 7;
const int max_v = 6 + 2 * slices + 2;
layout ( triangle_strip, max_vertices = max_v ) out;


void
gs_main_shrink()
{
  const bool opt_normal_sphere = bool(com.tryout.z);
  const bool unworn = false;

  if ( unworn )
    {
      // This code renders an ordinary triangle, and includes the
      // sphere-normal/triangle normal effect.
      for ( int i=0; i<3; i++ )
        {
          normal_e = In[opt_normal_sphere?i:0].normal_e;
          vertex_e = In[i].vertex_e;
          color = In[i].color;
          use_tex = 1; // SOLUTION.
          gl_Position = In[i].vertex_c;
          tex_coor = In[i].tex_coor;
          EmitVertex();
        }
      EndPrimitive();
      return;
    }

  /// SOLUTION -- Problem 1

  // Compute the coordinates of the center of the triangle.
  vec3 ctr_e = vec3(0), cnorm_e = vec3(0);
  for ( int i=0; i<3; i++ )
```

```
    {
      ctr_e += In[i].vertex_e.xyz;
      cnorm_e += In[i].normal_e;
    }
  ctr_e *= 0.333333333333f;
  cnorm_e *= 0.333333333333f;

  // Compute a distance between the original triangle and the shrunken
  // one.
  //
  float ht = In[0].sphere_radius * 0.01;
  //
  // If this distance were zero then the shrunken triangle would be in
  // the same plane as the original triangle. It is possible that the
  // original triangle would cover the shrunken one, at least in some
  // places.

  /// The mix OGSL Library Function
  //
  //  mix( a, b, f )  ->  a * (1-f) + b * f

  // Emit Shrunken Triangle
  //
  for ( int i=0; i<3; i++ )
    {
      // Interpolate normal value so that it's appropriate for this
      // position on the sphere.
      //
      normal_e = mix( In[i].normal_e, cnorm_e, opt_hw02_curl );

      // Interpolate vertex coordinate.
      //
      vertex_e.xyz =
        mix( In[i].vertex_e.xyz, ctr_e, opt_hw02_curl ) + normal_e * ht;
      vertex_e.w = 1;

      // Use eye-space coordinate to compute clip-space coordinate.
      //
      gl_Position = gl_ProjectionMatrix * vertex_e;
      //
      // Note that interpolating clip-space coordinates is wrong
      // because distances between them are distorted by the
      // perspective transformation.

      // The texture coordinate should not be interpolated, because we
      // want the texture image to shrink along with the triangle.
      // There is no point interpolating the color because all
      // vertices are the same color. (See the vertex shader.)
      //
      tex_coor = In[i].tex_coor;
      color = In[i].color;

      // Signal the fragment shader to apply the texture.
      use_tex = 1;
```

```
      EmitVertex();
    }
  EndPrimitive();

  // Emit Untextured Original Triangle
  //
  for ( int i=0; i<3; i++ )
    {
      vertex_e = In[i].vertex_e;
      normal_e = In[opt_normal_sphere?i:0].normal_e;

      // Darken the input color.
      //
      color.rgb = 0.5 * In[i].color.rgb;
      color.a = In[i].color.a; // Just in case a is set.

      // Signal the fragment shader to not apply the texture.
      //
      use_tex = 0;

      gl_Position = gl_ProjectionMatrix * vertex_e;
      EmitVertex();
    }
  EndPrimitive();
}


void
gs_main_curl()
{
  const bool opt_normal_sphere = bool(com.tryout.z);

  const bool unworn = false;

  if ( unworn )
    {
      // This code renders an ordinary triangle, and includes the
      // sphere-normal/triangle normal effect.
      for ( int i=0; i<3; i++ )
        {
          normal_e = In[opt_normal_sphere?i:0].normal_e;
          vertex_e = In[i].vertex_e;
          color = In[i].color;
          gl_Position = In[i].vertex_c;
          tex_coor = In[i].tex_coor;
          EmitVertex();
        }
      EndPrimitive();
      return;
    }

  // The start of a solution to Homework 2 Problem 2 appears below.
```

```
// The index of the vertex that will peel up.
int v1_idx = 0;

const int v0_idx[2] = { ( v1_idx + 1 ) % 3, ( v1_idx + 2 ) % 3 };
vec4 p1_e = In[v1_idx].vertex_e;  // Vertex peeling up.
vec3 n1_e = In[v1_idx].normal_e;  // Its normal.
vec2 tc1 = In[v1_idx].tex_coor;

const float p_frac = opt_hw02_curl;
const float f0 = 1 - p_frac;

/// SOLUTION - Problem 2
//
// Emit the uncurled part of the triangle.
//
// The loop below emits four triangles (including one with a zero
// area). The four triangles together form the original triangle.
// The first two are the part of the original triangle that has not
// yet curled up. The last two (one of which has zero area) are the
// part of the original triangle that is underneath the curled up
// part.
//
float fi[] = { 0, f0, 1 };
for ( int i=0; i<3; i++ )
  {
    const float f = fi[i];
    for ( int j=0; j<2; j++ )
      {
        const int idx = v0_idx[j];
        normal_e = opt_normal_sphere
          ? mix( In[idx].normal_e, n1_e, f ) : In[0].normal_e;
        vertex_e = mix( In[idx].vertex_e, p1_e, f );
        use_tex = int( i < 1 );
        color = i < 1 ? In[0].color : In[0].color * vec4(0.5,0.5,0.5,1);
        gl_Position = gl_ProjectionMatrix * vertex_e;
        tex_coor = mix(In[idx].tex_coor, tc1, f);
        EmitVertex();
      }
  }
EndPrimitive();

//
// Curled part of the triangle.
//

// Number of times that peeled part wraps around cylinder.
const float n_rev = 0.5;

const float two_pi = 2 * 3.14159265;

const float delta_theta = n_rev * p_frac * two_pi / slices;
const float delta_f = p_frac / slices;

// The two vertices that are opposite the peeling vertex.
```

```
vec3 p00_e = In[v0_idx[0]].vertex_e.xyz;
vec3 p01_e = In[v0_idx[1]].vertex_e.xyz;

// Compute local axes for cylinder.
//
vec3 ax = normalize(p01_e - p00_e);
vec3 v001 = p1_e.xyz - p00_e;
vec3 az = normalize( cross(ax,v001) );
vec3 ay = cross(az,ax);
//
// Cylinder axis is parallel to ax.
// Axes ay and az are used to draw a circle on the cylinder surface.

// Choose the radius of the cylinder.
const float d = dot(v001,ay);
const float r = d / ( n_rev * two_pi );

// A point on the cylinder axis.
vec3 ctr_e = p1_e.xyz - p_frac * d * ay + r * az;

// Iterate over parts of the peeled triangle. The peeling starts at i=0.
for ( int i=0; i<=slices; i++ )
  {
    const float theta = 1.5f * 3.1415927f + delta_theta * i;
    const float f = f0 + delta_f * i;
    vec3 nc = ay * cos(theta) + az * sin(theta);
    vec3 vc = r * nc;

    // Iterate over the two legs of the triangle.
    for ( int j=0; j<2; j++ )
      {
        const int idx = v0_idx[j];
        vec3 p_e = mix( In[idx].vertex_e.xyz, p1_e.xyz, f );
        vec3 v_pc = ctr_e - p_e;
        // Compute the coordinate of this edge (leg) of the triangle
        // wrapped on the cylinder.
        vertex_e.xyz = ctr_e + vc - dot(v_pc,ax) * ax;
        vertex_e.w = 1;

        /// SOLUTION - Problem 2
        //
        // Write the remaining vertex attributes and emit the vertex.

        // Use the cylinder normal for the surface normal.
        normal_e = nc;

        use_tex = 1;
        tex_coor = mix( In[idx].tex_coor, tc1, f );
        color = In[0].color;
        gl_Position = gl_ProjectionMatrix * vertex_e;
        EmitVertex();
      }
  }
```

```
  /// SOLUTION - Problem 2
  EndPrimitive();
}

#endif


#ifdef _FRAGMENT_SHADER_

#ifdef BIND_TEXUNIT
layout ( binding = BIND_TEXUNIT ) uniform sampler2D tex_unit_0;
#endif

layout ( location = 0 ) in Data_to_FS
{
  vec3 normal_e;
  vec4 vertex_e;
  vec2 tex_coor;
  flat vec4 color;

  // SOLUTION
  flat int use_tex;
};

layout ( location = 0 ) out vec4 frag_color;

void
fs_main_common()
{
  // Perform lighting, fetch and blend texture, then emit fragment.
  //

  // Get filtered texel.
  //
#ifdef BIND_TEXUNIT

  /// SOLUTION
  //
  //  Apply the texture only if use_tex == 1
  //
  vec4 texel = gl_FrontFacing && use_tex == 1
    ? texture(tex_unit_0,tex_coor) : vec4(1,1,1,1);
#else
  vec4 texel = vec4(1,1,1,1);
#endif

  vec4 color2 = gl_FrontFacing ? color : vec4(0.5,0,0,1);

  // Multiply filtered texel color with lighted color of fragment.
  //
  frag_color = texel * generic_lighting(vertex_e, color2, normal_e);
}
```

```
#endif
```

```
/// LSU EE 4702-1 (Fall 2021), GPU Programming
//

 /// SOLUTION -- Homework 3


// Specify version of OpenGL Shading Language.
//
#version 460

#extension GL_GOOGLE_include_directive : enable
#include <light.h>
#include <transform.h>
#include "links-shdr-common.h"
#include "shader-common-generic-lighting.h"



layout ( binding = BIND_SPHERE_POS ) buffer spr { vec4 sphere_pos_rad[]; };
layout ( binding = BIND_SPHERE_ROT ) buffer sr { mat4 sphere_rot[]; };
layout ( binding = BIND_SPHERE_COLOR ) buffer sc { vec4 sphere_color[]; };

layout ( binding = BIND_UNI_COMMON ) uniform Common_Uniform
{
  Shdr_Uni_Common com;
};

bool opt_tryout1 = bool(com.tryout.x);
bool opt_tryout2 = bool(com.tryout.y);
float opt_tryoutf = com.tryoutf.x;
float opt_hw03_curl = com.tryoutf.y;



#ifdef _VERTEX_SHADER_

layout ( location = LOC_IN_POS ) in vec4 in_vertex_o;

#ifdef LOC_IN_TCOOR
layout ( location = LOC_IN_TCOOR ) in vec2 in_tex_coor;
#endif

// Interface block for vertex shader output / geometry shader input.
//
layout ( location = 0 ) out Data_to_GS
{
  vec3 normal_e;
  vec4 vertex_e;
  vec2 tex_coor;
  vec4 color;
  vec4 vertex_c;
  float sphere_radius;
```

```
  // Any changes here must also be made to the fragment shader input.

};

void
vs_main_instances_sphere()
{
  vec4 pos_rad = sphere_pos_rad[gl_InstanceIndex];
  float rad = pos_rad.w;
  mat4 rot = sphere_rot[gl_InstanceIndex];
  vec4 normr = rot * in_vertex_o;
  vec3 normal_o = normr.xyz;
  vec4 vertex_o = vec4( pos_rad.xyz + rad * normal_o, 1 );

  vertex_c = gl_ModelViewProjectionMatrix * vertex_o;
  vertex_e = gl_ModelViewMatrix * vertex_o;
  normal_e = normalize(gl_NormalMatrix * normal_o );
  tex_coor = in_tex_coor;
  color = sphere_color[gl_InstanceIndex];
  sphere_radius = rad;
}
#endif


#ifdef _GEOMETRY_SHADER_

layout ( location = 0 ) in Data_to_GS
{
  vec3 normal_e;
  vec4 vertex_e;
  vec2 tex_coor;
  vec4 color;
  vec4 vertex_c;
  float sphere_radius;
} In[];

layout ( location = 0 ) out Data_to_FS
{
  vec3 normal_e;
  vec4 vertex_e;
  vec2 tex_coor;
  flat vec4 color;
  flat int use_tex;

};

// Type of primitive at geometry shader input.
//
layout ( triangles ) in;

// Type of primitives emitted geometry shader output.
//
```

```
const int slices = 5;
const int max_v = 6 + ( 2 + 4 ) * (slices+1) + 2;
layout ( triangle_strip, max_vertices = max_v ) out;


void
gs_main_shrink()
{
  const bool opt_normal_sphere = bool(com.tryout.z);

  // Compute the coordinates of the center of the triangle.
  vec3 ctr_e = vec3(0), cnorm_e = vec3(0);
  for ( int i=0; i<3; i++ )
    {
      ctr_e += In[i].vertex_e.xyz;
      cnorm_e += In[i].normal_e;
    }
  ctr_e *= 0.333333333333f;
  cnorm_e *= 0.333333333333f;

  /// SOLUTION -- Problem 1, Constant Volume Constraint
  //

  // Compute a distance between the original triangle and the shrunken
  // one.
  //
  float ht0 = In[0].sphere_radius * 0.1;
  //
  // This distance, ht0, will be used to compute the constant volume.
  // That volume is computed using height ht0 and the triangle at its
  // original size.
  //
  // Let
  //   l0 denote the length of a side of the base triangle.
  //   l1 denote the length of a side of the shrunken triangle.
  //   y  ∋ [0,1] denote the normalized distance from the base triangle
  //        to the shrunken triangle (top of the frustum). Here
  //        y=0 corresponds to a point on the base, y=1 corresponds
  //        to a point on the top.
  //   h  denote the (unormalized) height of the frustum.
  //   c  denote the value of opt_hw03_curl.
  //
  // As a simplification, the volume of a frustum with a square
  // base will be computed.
  //
  // ∫ y=0, y=1   ( (1-y)l0 + yl1 )^2 h dy
  //
  //  To simplify let l1 = (1-c)l0 and l0 = 1
  // y=0, y=1   ( (1-y) + y(1-c) )^2 h dy
  // y=0, y=1   ( 1-y + y - yc )^2 h dy
  // y=0, y=1   ( 1 - yc )^2 h dy
  // y=0, y=1   ( 1 + y^2c^2 -2yc ) h dy
  // y=0, y=1   ( y + y^3c^2/3 -y^2c ) h
  // ( 1 + c^2/3 - c ) h
```

```
//
// Next solve for h in (1+c^2/3-c) h == l0^2 h0 :
// h = h0 / ( 1 + c^2/3  -c )

float ht = ht0 / ( 1 + opt_hw03_curl * ( opt_hw03_curl / 3 - 1 ) );



/// The mix OGSL Library Function
//
//  mix( a, b, f )  ->  a * (1-f) + b * f

vec3 vtx_shr_e[3];

// Emit Shrunken Triangle
//
for ( int i=0; i<3; i++ )
  {
    // Interpolate normal value so that it's appropriate for this
    // position on the sphere.
    //
    normal_e = mix( In[i].normal_e, cnorm_e, opt_hw03_curl );

    // Interpolate vertex coordinate.
    //
    vtx_shr_e[i] = vertex_e.xyz =
      mix( In[i].vertex_e.xyz, ctr_e, opt_hw03_curl ) + normal_e * ht;
    vertex_e.w = 1;

    // Use eye-space coordinate to compute clip-space coordinate.
    //
    gl_Position = gl_ProjectionMatrix * vertex_e;
    //
    // Note that interpolating clip-space coordinates is wrong
    // because distances between them are distorted by the
    // perspective transformation.

    // The texture coordinate should not be interpolated, because we
    // want the texture image to shrink along with the triangle.
    // There is no point interpolating the color because all
    // vertices are the same color. (See the vertex shader.)
    //
    tex_coor = In[i].tex_coor;
    color = In[i].color;

    // Signal the fragment shader to apply the texture.
    use_tex = 1;

    EmitVertex();
  }
EndPrimitive();

/// SOLUTION -- Emit the Sides
//
```

```
  vec4 gr_color = vec4( 0.5 * In[0].color.rgb, 1 );

  for ( int i=0; i<3; i++ )
    {
      int i1 = (i+1) % 3;
      vec3 vo_0 = In[i].vertex_e.xyz;
      vec3 vo_1 = In[i1].vertex_e.xyz;
      vec3 vs_0 = vtx_shr_e[i];
      vec3 vs_1 = vtx_shr_e[i1];
      vec3 n_e = normalize(cross( vs_0 - vo_0, vs_1 - vs_0 ));

      vec3 vstrip[] = { vs_0, vo_0, vs_1, vo_1 };
      int tstrip[] =  { i,    i,    i1,   i1   };

      for ( int j=0; j<4; j++ )
        {
          vertex_e = vec4( vstrip[j], 1 );
          normal_e = n_e;
          color = gr_color;
          use_tex = 1;
          tex_coor = In[tstrip[j]].tex_coor;
          gl_Position = gl_ProjectionMatrix * vertex_e;
          EmitVertex();
        }
      EndPrimitive();
    }
}


void
gs_main_curl()
{
  const bool opt_normal_sphere = bool(com.tryout.z);

  const bool unworn = false;

  if ( unworn )
    {
      // This code renders an ordinary triangle, and includes the
      // sphere-normal/triangle normal effect.
      for ( int i=0; i<3; i++ )
        {
          normal_e = In[opt_normal_sphere?i:0].normal_e;
          vertex_e = In[i].vertex_e;
          color = In[i].color;
          gl_Position = In[i].vertex_c;
          tex_coor = In[i].tex_coor;
          EmitVertex();
        }
      EndPrimitive();
      return;
    }

  // The start of a solution to Homework 2 Problem 2 appears below.
```

```
// The index of the vertex that will peel up.
int v1_idx = 0;

const int v0_idx[2] = { ( v1_idx + 1 ) % 3, ( v1_idx + 2 ) % 3 };
vec4 p1_e = In[v1_idx].vertex_e;  // Vertex peeling up.
vec3 n1_e = In[v1_idx].normal_e;  // Its normal.
vec2 tc1 = In[v1_idx].tex_coor;

const float p_frac = opt_hw03_curl;
const float f0 = 1 - p_frac;

// Emit the uncurled part of the triangle.
//
// The loop below emits four triangles (including one with a zero
// area). The four triangles together form the original triangle.
// The first two are the part of the original triangle that has not
// yet curled up. The last two (one of which has zero area) are the
// part of the original triangle that is underneath the curled up
// part.
//
float fi[] = { 0, f0, 1 };
for ( int i=0; i<3; i++ )
  {
    const float f = fi[i];
    for ( int j=0; j<2; j++ )
      {
        const int idx = v0_idx[j];
        normal_e = mix( In[idx].normal_e, n1_e, f );
        vertex_e = mix( In[idx].vertex_e, p1_e, f );
        use_tex = i < 1 ? 1 : 0;
        color = i < 1 ? In[0].color : In[0].color * vec4(0.5,0.5,0.5,1);
        gl_Position = gl_ProjectionMatrix * vertex_e;
        tex_coor = mix(In[idx].tex_coor, tc1, f);
        EmitVertex();
      }
  }
EndPrimitive();

//
// Curled the part part of the triangle.
//

// Number of times that peeled part wraps around cylinder.
const float n_rev = 0.5;

const float two_pi = 2 * 3.14159265;

const float delta_theta = n_rev * p_frac * two_pi / slices;
const float delta_f = p_frac / slices;

// The two vertices that are opposite the peeling vertex.
vec3 p00_e = In[v0_idx[0]].vertex_e.xyz;
vec3 p01_e = In[v0_idx[1]].vertex_e.xyz;
```

```
// Compute local axes for cylinder.
//
vec3 ax = normalize(p01_e - p00_e);
vec3 v001 = p1_e.xyz - p00_e;
vec3 az = normalize( cross(ax,v001) );
vec3 ay = cross(az,ax);
//
// Cylinder axis is parallel to ax.
// Axes ay and az are used to draw a circle on the cylinder surface.

// Choose the radius of the cylinder.
const float d = dot(v001,ay);
const float r = d / ( n_rev * two_pi );

// A point on the cylinder axis.
vec3 ctr_e = p1_e.xyz - p_frac * d * ay + r * az;

vec3 peel_v[2][slices+1];
vec3 peel_n[slices+1];

// Iterate over parts of the peeled triangle. The peeling starts at i=0.
for ( int i=0; i<=slices; i++ )
  {
    const float theta = 1.5f * 3.1415927f + delta_theta * i;
    const float f = f0 + delta_f * i;
    vec3 nc = ay * cos(theta) + az * sin(theta);
    vec3 vc = r * nc;
    peel_n[i] = nc;

    // Iterate over the two legs of the triangle.
    for ( int j=0; j<2; j++ )
      {
        const int idx = v0_idx[j];
        vec3 p_e = mix( In[idx].vertex_e.xyz, p1_e.xyz, f );
        vec3 v_pc = ctr_e - p_e;
        // Compute the coordinate of this edge (leg) of the triangle
        // wrapped on the cylinder.
        vertex_e.xyz = ctr_e + vc - dot(v_pc,ax) * ax;
        vertex_e.w = 1;
        peel_v[j][i] = vertex_e.xyz;

        // Write the remaining vertex attributes and emit the vertex.

        // Use the cylinder normal for the surface normal.
        normal_e = nc;

        use_tex = 1;
        tex_coor = mix( In[idx].tex_coor, tc1, f );
        color = In[0].color;
        gl_Position = gl_ProjectionMatrix * vertex_e;
        EmitVertex();
      }
  }
```

```
  EndPrimitive();

  /// SOLUTION -- To Unassigned Problem 2

  // Iterate over parts of the peeled triangle. The peeling starts at i=0.

  float th = In[0].sphere_radius * .01f;

  for ( int j=0; j<2; j++ )
    {
      for ( int i=0; i<=slices; i++ )
        {
          int i1 = i < slices ? i+1 : slices-1;
          vec3 v0 = peel_v[j][i];
          vec3 n0 = peel_n[i];
          vec3 v0l = v0 + n0 * th;
          vec3 v1 = peel_v[j][i1];
          vec3 n1 = peel_n[i];
          vec3 v1l = v1 + n1 * th;

          vec3 sn = normalize(cross( v1l - v0l, v0 - v0l ));
          if ( i == slices ) sn = -sn;

          vertex_e = vec4(j==1?v0:v0l,1);
          normal_e = sn;
          use_tex = 0;
          color = vec4(0.2,0.2,0.2,1);
          gl_Position = gl_ProjectionMatrix * vertex_e;
          EmitVertex();

          vertex_e = vec4(j==1?v0l:v0,1);
          normal_e = sn;
          use_tex = 0;
          color = vec4(0.2,0.2,0.2,1);
          gl_Position = gl_ProjectionMatrix * vertex_e;
          EmitVertex();
        }
      EndPrimitive();
    }
}

#endif


#ifdef _FRAGMENT_SHADER_

#ifdef BIND_TEXUNIT
layout ( binding = BIND_TEXUNIT ) uniform sampler2D tex_unit_0;
#endif

layout ( location = 0 ) in Data_to_FS
{
  vec3 normal_e;
  vec4 vertex_e;
```

```
  vec2 tex_coor;
  flat vec4 color;
  flat int use_tex;
};

layout ( location = 0 ) out vec4 frag_color;

void
fs_main_common()
{
  // Perform lighting, fetch and blend texture, then emit fragment.
  //

  // Get filtered texel.
  //
#ifdef BIND_TEXUNIT

  //  Apply the texture only if use_tex == 1
  //
  vec4 texel = gl_FrontFacing && use_tex == 1
    ? texture(tex_unit_0,tex_coor) : vec4(1,1,1,1);
#else
  vec4 texel = vec4(1,1,1,1);
#endif

  vec4 color2 = true || gl_FrontFacing ? color : vec4(0.5,0,0,1);

  // Multiply filtered texel color with lighted color of fragment.
  //
  frag_color = texel * generic_lighting(vertex_e, color2, normal_e);
}

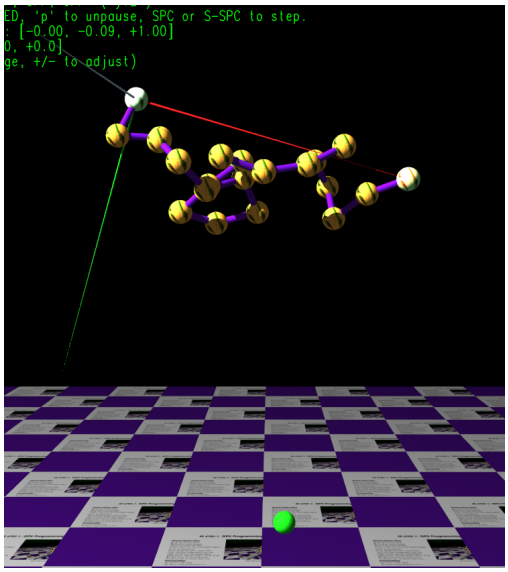

#endif
```

# 23 Fall 2020 Solutions

# LSU EE 4702-1 Homework 1 Solution Due: 23 September 2020

The solution has been checked into the repo as file `hw01-sol.cc`. A colorized version is at `https://www.ece.lsu.edu/koppel/gpup/2020/hw01-sol.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 string of balls compressed between two points, the first and last balls are fixed in place. The balls quickly expand and eventually will form an arc-like shape. The screenshot to the right, taken when the simulation was paused, shows the balls uncompressing, with pointy markers showing local axes, and a green ball showing the approximate place where the lowermost part of the string will be once the balls come to rest.




## User Interface

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 tail 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 `Shift` and an arrow key will move by a larger distance (than if `Shift` were not pressed) and pressing `Ctrl` and an arrow key will move by a smaller distance. The eye can be aimed up and down by pressing `Home` and `End`, and the eye can be rotated by pressing `Insert` and `Delete`.

Pressing `p` will pause and un-pause the simulation. While the simulation is paused time can be advanced by one frame by pressing the space bar, and by one time step by pressing `Shift` and the space bar.

Press digits `1` through `4` to initialize different scenes, the program starts with scene 1. In the unmodified code scenes 1, 2, and 3 are identical: they start with the balls arranged in a line, and separated by springs at less than the relaxed distance. Once this assignment is solved scenes 1, 2, and 3 will behave differently.

Pressing `h` (head) will grab or release one end (to be precise, the ball at one end) and pressing `t` (tail) will grab or release the other end. (Actually, those keys toggle between the `OC_Locked` and `OC_Free` constraint of their respective balls.)

Press `Ctrl=` to increase the size of the green text and `Ctrl-` to decrease the size.

The `+` and `-` keys can be used to change the value of certain variables. These variables specify things such as the light intensity, spring constant, 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` cycles through the different variables.

## Code Generation and Debug Support

The compiler generates two versions of the code, `hw01` and `hw01-debug`. Use `hw01` to measure

performance, but use `hw01-debug` for debugging. The `hw01-debug` version is compiled with optimization turned off and with OpenGL error checking turned on. You are strongly encouraged to run `hw01-debug` under the GNU debugger, `gdb`. See the material under "Running and Debugging the Assignment" on the course procedures page.

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. These variables are intended for debugging and trying things out.

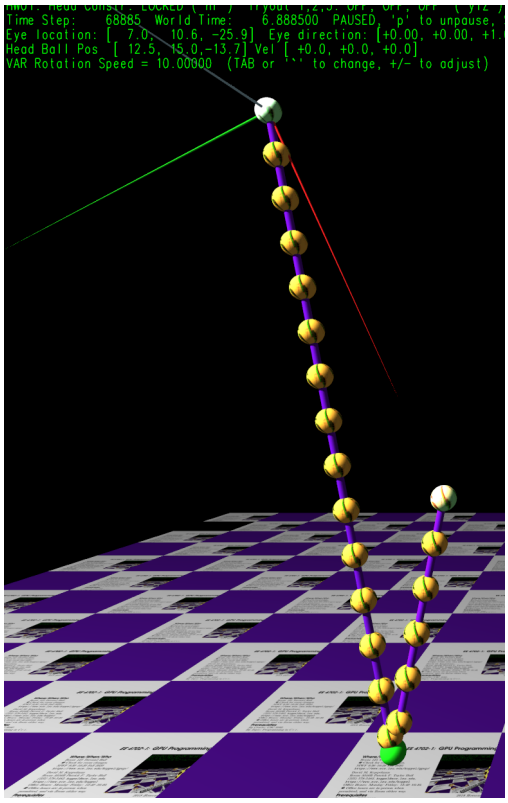
**Problem 1:** Scene 1 starts with the balls arranged in a line and the first and last balls locked in place. The distance between adjacent balls is set to less than the spring's relaxed distance and so the balls are quickly pushed away from each other. They will eventually come to rest in an arc. The goal of this problem is to modify the setup routine so that the initial position of the balls are chosen such that the balls are closer to equilibrium when the scene starts.

In the ideal solution to this problem the balls are arranged to form a *catenary*, the name given to the curve made by an ideal cable with two ends fixed in space. A catenary is close in shape to a parabola.

This problem will be solved with a much cruder approximation: two straight lines. The code in `ball_setup_hw01` chooses a first and last position for the balls, `first_pos` and `last_pos`, denote these $P_f$ and $P_l$. Let $n$ denote the value of `chain_length`, the number of balls, and let $l_r$ denote the value of `distance_relaxed`. The setup routine computes `nadir_pos`, denote that $P_n$.

Call $L = (n-1)l_r$ the (relaxed) length of the chain. Assume for a moment that the chain were an un-stretchable string, with the ends still fixed at $P_f$ and $P_l$. Take a pencil and pull the string in some direction until it is taut. (That is, until the string forms a straight line from $P_f$ to the pencil tip and $P_l$ to the pencil tip.) Draw a dot. Pull in some other direction and draw a dot. Repeat $\infty$ times. Perhaps you've guessed that shape drawn this way is an ellipse. Points $P_f$ and $P_l$ are the foci. Since an ellipse is defined on a plane, we need a normal. The normal used is parallel to the $xz$ plain. The point $P_n$ is on this ellipse, and it is chosen so that it is the point on the ellipse with the minimum $y$ value.

To help understanding these concepts the location of $P_f$, $P_l$, $P_n$, and the axes in the ellipse's local space can be visualized. To do so pause the simulation (using `p`) in scene 1, 2, or 3. (Switch scenes using `1`, `2`, etc.) The ellipse local $x$, $y$, and $z$ axes are shown by red, green, and blue needles. Positions $P_f$, $P_l$, and $P_f$ are shown by white, red, and green featureless spheres (which may be hidden by balls). See the screenshot to the right and the code below `hw01.markers_show` in `ball_setup_hw01` to see which colors are assigned.




$P_n$ is chosen such that $\|\overrightarrow{P_f P_n}\| + \|\overrightarrow{P_n P_l}\| = L$. This means that the first several balls can be arranged on the line from $P_f$ to $P_n$ and the remaining balls can be arranged on the line from $P_n$ to $P_l$, while keeping the distance between adjacent balls to $l_r$ with possibly one exception. The exception occurs when no ball is exactly at $P_n$. In the exception case consider the two balls closest to $P_n$. The distance from the first of these balls, to $P_n$ to the second of these balls will be $l_r$, but

the distance between these two balls will be less than
$l_r$.

($a$) Modify the code in `ball_setup_hw01` so that the balls are arranged in a straight line starting from `first_pos` reaching exactly or close to `nadir_pos`, and ending at `last_pos`. The distance between adjacent balls must be `distance_relaxed` except for the exception condition described above.

No ball should be placed below `nadir_pos`. The first ball must be at `first_pos` and the last ball must be at `last_pos`. See the screenshot above.

If this is solved correctly the two balls near `last_pos` should twitch (because they are closer than their relaxed distance). Initially the two lowest balls should be near the green marker (the location of `nadir_pos`) and the white and red spherical markers should not be visible.

To help debug your solution pause the simulation using `p`, then start the scene, `1`. Compare your placement of the balls to the marker balls.

Consider the loop that computes positions in the unmodified code:

```
pNorm dfn(first_pos,nadir_pos), dln(last_pos,nadir_pos);
for ( int i=0; i<chain_length; i++ )
  {
    Ball* const ball = &balls[i];
    pCoor pos = first_pos + i * first_last_dist / ( chain_length-1 ) * ax;
```

The balls need to be arranged on a line either starting at `first_pos` and toward `nadir_pos` or on a line from `nadir_pos` to `last_pos`. The balls on each line will be spaced exactly `distance_relaxed` apart.

A simple way to solve the problem is to compute the value of `i` for the first ball on the line from `nadir_pos` to `last_pos`. That's called `i_dir_change` in the solution (below). It's computed by dividing the distance from `first_pos` to `nadir_pos` by `distance_relaxed` and rounding up.

Then `ball->position` is set based on its distance from `first_pos` if `i < i_dir_change`, otherwise based on its distance from `last_pos`.

```
pNorm dfn(first_pos,nadir_pos), dln(last_pos,nadir_pos);
const int i_dir_change = 0.9999f + dfn.magnitude / distance_relaxed;
for ( int i=0; i<chain_length; i++ )
  {
    Ball* const ball = &balls[i];
    ball->position = i < i_dir_change
      ? first_pos +                       i   * distance_relaxed * dfn
      : last_pos  + ( chain_length - 1 - i ) * distance_relaxed * dln;
```

Grading note: many solutions were more complicated than they needed to be for two reasons. First, some solutions computed the coordinates of the balls on the line toward `last_pos` based on their distance from `nadir_pos`. That made the code a bit messier. What made the code messier still was the attempt to find this distance within the loop, in some cases using a second iterator for the balls toward `last_pos`.

($b$) In this problem that twitching when a scene is started will be fixed. The solution is not ideal from a physical model point-of-view, but it makes a decent question. Modify the code so that each spring can have its own relaxed distance. Do so by adding a relaxed distance member to `Ball` and using it appropriately. The code in `ball_setup_hw01` must set this new member to `distance_relaxed` for all balls except for one. That one ball is the one that's closer to its neighbor. Set it to the appropriate distance if `opt_special_dist_relaxed` is true, otherwise set it to `distance_relaxed`. Modify the code in `time_step_cpu` so that the per-ball `distance_relaxed`

values will be used. Part of this problem is deciding whether the value of `distance_relaxed` set for a ball applies to its connection to its predecessor in the `balls` array or to its connection with its successor.

Use scene 2 for this problem. In scene 2 (and 3) `opt_special_dist_relaxed` is true.

If this is solved directly scene 2 should start without a sudden twitch. Instead the balls will gently come to rest in the correct place.

In the solution below a new member, `Ball::distance_relaxed`, was added that holds the relaxed distance of the spring connecting a ball to its predecessor. (That means `ball->distance_relaxed` for the first ball is meaningless.) For the ball closest to `nadir_pos` on the line towards `last_pos` `ball->distance_relaxed` is set to the distance to the previous ball, otherwise it is set to the global `distance_relaxed`:

```
ball->distance_relaxed =
  opt_special_dist_relaxed && i == i_dir_change
  ? pNorm(balls[i-1].position,ball->position).magnitude
  : distance_relaxed;
```

The time step routine needs to be modified so that it uses the correct `Ball::distance_relaxed` when computing `spring_stretch`. The inner loop in the time step routine finds the force between `ball` and `neighbor_ball`. Since `Ball::distance_relaxed` is the spring to the predecessor (lower numbered) ball, the code should use `Ball::distance_relaxed` from the higher-numbered ball. That is done by the code below, which has been shortened for clarity:

```
for ( int i=0; i<chain_length; i++ ) {
    Ball* const ball = &balls[i];
    ball->force = ball->mass * gravity_accel;
    for ( int direction: { -1, +1 } ) {
        const int n_idx = i + direction;  // Compute neighbor index.
        if ( n_idx < 0 or n_idx >= chain_length ) continue;
        Ball* const neighbor_ball = &balls[n_idx];
        pNorm ball_to_neighbor( ball->position, neighbor_ball->position );
        const float distance_between_balls = ball_to_neighbor.magnitude;

        const int idx_gt = max(i,n_idx);
        const float spring_stretch =
          distance_between_balls
          - balls[idx_gt].distance_relaxed * fabs(direction);
```

One common problem was not realizing that a single distance relaxed member of `Ball` would have to refer to either the ball's predecessor or successor. In those incorrect solutions the time step routine itself would determine what other ball `ball->distance_relaxed` was supposed to connect to. That was wrong because it makes `ball->distance_relaxed` something that could only be used for one particular setup.

(*c*) Modify the code in `ball_setup_hw01` so that when `opt_spin` is true (which is the case in scene 3) the balls are given an initial velocity so that they rotate rigidly around the line defined by `first_pos` and `last_pos`. To do this set the ball velocity. Note that balls closer to the line will have a lower velocity.

If this is solved correctly the balls will swing around the axis while generally keeping their shape.

To solve this find the distance from a ball to the line defined by `first_pos` and `last_pos` and multiply that by `az`, which is the normal to the plane defined by `first_pos`, `last_pos`, and `nadir_pos`. In the solution this was done in a separate loop for reasons of clarity. The velocity could have been set in the main `i` loop.

```
pNorm spin_dir = cross(first_pos,nadir_pos,last_pos);
if ( opt_spin ) for ( auto& b: balls ) {
      pVect to_b0(b.position, first_pos);
      const float dist = dot( to_b0, ay );
      b.velocity = dist * 10 * spin_dir;
    }
```

```
/// LSU EE 4702-1 (Fall 2020), GPU Programming
//
 /// Homework 1 -- SOLUTION
 //

 /// Instructions
 //
 //  Read the assignment: https://www.ece.lsu.edu/koppel/gpup/2020/hw01.pdf
 //
 //  Also, feel free to add members to structure HW01_Stuff.
 //
 //  Only this file will be collected.


/// Purpose
//
//   Demonstrate simulation of string modeled as point masses and springs


/// What Code Does

// Simulates a string of beads over a platform. The string is modeled
// as point masses connected by springs with a long relaxed
// length. The platform consists of tiles, some are purple-tinted
// mirrors (showing a reflection of the ball), the others show the
// course syllabus.


///  Keyboard Commands
 //
 /// Object (Eye, Light, Ball) Location or Push
 //   Arrows, Page Up, Page Down
 //        Move object or push ball, depending on mode.
 //        With shift key pressed, motion is 5x faster.
 //   'e': Move eye.
 //   'l': Move light.
 //   'b': Move head (first) ball. (Change position but not velocity.)
 //   'B': Push head ball. (Add velocity.)
 //
 /// Eye Direction
 //   Home, End, Delete, Insert
 //   Turn the eye direction.
 //   Home should rotate eye direction up, End should rotate eye
 //   down, Delete should rotate eye left, Insert should rotate eye
 //   right.  The eye direction vector is displayed in the upper left.

 /// Simulation Options
 //  (Also see variables below.)
 //
 //  '1'    Set up scene 1.
 //  '2'    Set up scene 2.
 //  'p'    Pause simulation. (Press again to resume.)
```

```
//  ' '    (Space bar.) Advance simulation by 1/30 second.
//  'S- '  (Shift-space bar.) Advance simulation by one time step.
//  'h'    Freeze position of first (head) ball. (Press again to release.)
//  't'    Freeze position of last (tail) ball. (Press again to release.)
//  's'    Stop balls.
//  'g'    Turn gravity on and off.
//  'F12'  Write screenshot to file.

/// Variables
//   Selected program variables can be modified using the keyboard.
//   Use "Tab" to cycle through the variable to be modified, the
//   name of the variable is displayed next to "VAR" on the bottom
//   line of green text.

//  'Tab' Cycle to next variable.
//  '`'   Cycle to previous variable.
//  '+'   Increase variable value.
//  '-'   Decrease variable value.
//
//  VAR Spring Constant - Set spring constant.
//  VAR Air Resistance - Set air resistance.
//  VAR Light Intensity - The light intensity.
//  VAR Gravity - Gravitational acceleration. (Turn on/off using 'g'.)


#define GL_GLEXT_PROTOTYPES

#include <GL/gl.h>
#include <GL/freeglut.h>

#include <gp/util.h>
#include <gp/coord.h>
#include <gp/shader.h>
#include <gp/pstring.h>
#include <gp/misc.h>
#include <gp/gl-buffer.h>
#include <gp/texture-util.h>

#include "shapes.h"


///
/// Main Data Structures
///
//
// class World: All data about scene.


class World;

struct HW01_Stuff {

  /// Homework 1 -- Feel free to add new members to this structure.
  // This is a member of world, named hw01. E.g., hw01.markers_show;
```

```
  HW01_Stuff():opt_rot_speed(10),markers_show(false){}

  float opt_rot_speed;


  // For debugging.
  bool markers_show;  // If true, show vectors and positions assigned below.
  pVect mark_vec_red, mark_vec_green, mark_vec_blue;
  pCoor mark_pos_white, mark_pos_red, mark_pos_green;

};

enum Object_Constraint
{
  OC_Free,
  OC_Locked,  // Position is never changed.
  OC_Ring_Animated,
  OC_Ring_Free
};


// Object Holding Ball State
//
class Ball {
public:
  pCoor position;
  pVect velocity;

  float mass;
  float radius;
  Object_Constraint constraint;

  /// SOLUTION -- 2020 Homework 1, Problem 1, Part b
  //
  float distance_relaxed;  // Spring from this ball to next ball in array.

  pVect force;
  bool contact;                 // Can be used for special effects.

  void push(pVect amt);
  void translate(pVect amt);
  void stop();
  void freeze();
};

#include "hw01-graphics.cc"


void
World::init()
{
  srand48(4702);
```

```
  opt_tryout1 = opt_tryout2 = opt_tryout3 = false;

  chain_length = 21;
  balls.resize(chain_length);

  variable_control.insert(hw01.opt_rot_speed,"Rotation Speed");

  distance_relaxed = 15.0 / chain_length;
  opt_spring_constant = 1000;
  variable_control.insert(opt_spring_constant,"Spring Constant");

  opt_gravity_accel = 9.8;
  opt_gravity = true;
  gravity_accel = pVect(0,-opt_gravity_accel,0);
  variable_control.insert(opt_gravity_accel,"Gravity");

  opt_air_resistance = 0.01;
  //  variable_control.insert(opt_air_resistance,"Air Resistance");

  world_time = 0;
  time_step_count = 0;
  last_frame_wall_time = time_wall_fp();
  frame_timer.work_unit_set("Steps / s");

  init_graphics();

  curr_setup = 1; ball_setup_1();
}

///
/// Physical Simulation Code
///

 /// Initialize Simulation
//
void
World::ball_setup_1()
{
  ball_setup_hw01(1);
}
void
World::ball_setup_2()
{
  ball_setup_hw01(2);
}
void
World::ball_setup_3()
{
  ball_setup_hw01(3);
}


void
```

```
World::ball_setup_hw01(int option)
{
  /// Homework 1:
  //  Most of the solution goes in this file.

  const bool opt_special_dist_relaxed = option > 1;
  const bool opt_spin = option > 2;

  // The position of the first ball is the same each time.
  //
  pCoor first_pos(12.5, chain_length * distance_relaxed, -13.7);

  // This randomly chosen unit vector points from first to last pos.
  //
  pNorm ax( -drand48(), -drand48()/3, -drand48()/7 );

  // Distance between first and last ball.
  //
  const float first_last_dist =
    ( 0.2 + drand48()*0.6 ) * distance_relaxed * ( chain_length - 1 );

  // Length of chain between first and last ball.
  //
  const float chain_distance = distance_relaxed * ( chain_length - 1 );

  // The position of the last ball.
  //
  pCoor last_pos = first_pos + first_last_dist * ax;

  // Compute a local z axis, az, that is parallel to the global x/z axis.
  //
  pNorm az = cross( ax, pVect(0,1,0) );

  // Compute a local y axis.
  //
  pVect ay = cross( az, ax );
  //
  // This is a tilted version of the global y axis.

  // Construct an ellipse defined as follows:
  //
  //  Imagine the chain were inflexible string and the ends
  //  were attached to first_pos and last_pos.
  //
  //  Stretch the string tight using an imaginary pencil tip. The
  //  ellipse is formed by the possible locations of the pencil top/
  //
  //  Note: the ellipse is in the plane with normal az.
  //
  // In geometric terminology:
  //
  //  The foci are at first_pos and last_pos.
  //  The distance between the foci is first_last_dist.
  //
```

```
// Eccentricity of ellipse.
//
const float ec = first_last_dist / chain_distance;

// Halfway point between the two balls. This will be the origin
// of the local space in which the ellipse is drawn.
//
pCoor mid_pos = first_pos + 0.5 * first_last_dist * ax;

// Compute handy values.
//
const float ays = ay.y * ay.y,  axs = ax.y * ax.y,  ecs = ec * ec;

// Compute local x coordinate of point on ellipse that has the
// minimum value of y in the global coordinate space. This assumes
// that ax.y < 0;
//
const float lx_min =
  -chain_distance * 0.5 * ax.y / sqrt( axs + ays * ( 1 - ecs ) );

// Compute local y coordinate of point on ellipse corresponding to lx_min.
//
const float ly_min =
  -sqrt( ( chain_distance * chain_distance / 4 - lx_min * lx_min )
         * ( 1 - ecs ) );

// Compute coordinates of the point on the ellipse with the minimum local y.
//
pCoor nadir_pos = mid_pos + lx_min * ax + ly_min * ay;


// Verify that distance from first, to nadir, to last is correct.
pNorm dfn(first_pos,nadir_pos);
pNorm dln(last_pos,nadir_pos);
assert( fabs( dfn.magnitude + dln.magnitude - chain_distance ) < 0.0001 );

// Set markers that will be displayed when the simulation is paused.
// These may be useful for debugging. Feel free to change them.
//
hw01.markers_show = true;
hw01.mark_vec_red = ax;     // Red needle points along ax.
hw01.mark_vec_green = -ay;
hw01.mark_vec_blue = az;
hw01.mark_pos_white = first_pos;
hw01.mark_pos_red = last_pos;
hw01.mark_pos_green = nadir_pos;


/// SOLUTION -- 2020 Homework 1, Problem 1, Part a
//
//  Compute index of last ball in the line starting at first_pos.
//
const int i_dir_change = 0.9999f + dfn.magnitude / distance_relaxed;
```

```
  for ( int i=0; i<chain_length; i++ )
    {
      Ball* const ball = &balls[i];

      /// SOLUTION -- 2020 Homework 1, Problem 1, Part a
      //
      //  Compute ball position based either on position relative to
      //  first_pos or on position relative to last_pos.
      //
      ball->position = i < i_dir_change
        ? first_pos +                      i   * distance_relaxed * dfn
        : last_pos  + ( chain_length - 1 - i ) * distance_relaxed * dln;

      ball->velocity = pVect(0,0,0);
      ball->radius = 0.2;
      ball->mass = 4/3.0 * M_PI * pow(ball->radius,3);
      ball->constraint = i > 0 && i < chain_length - 1 ? OC_Free : OC_Locked;
      ball->contact = false;

      /// SOLUTION -- 2020 Homework 1, Problem 1, Part b
      //
      //  Ball::distance_relaxed is the relaxed distance of the spring
      //  connecting ball i to ball i-1.
      //
      //  Set distance_relaxed to the distance between this and
      //  previous ball if opt_special_dist_relaxed and if the i-1
      //  ball is on the line from first_pos and this ball is on the
      //  line to last_pos.
      //
      ball->distance_relaxed =
        opt_special_dist_relaxed && i == i_dir_change
        ? pNorm(balls[i-1].position,ball->position).magnitude
        : distance_relaxed;
    }

  /// SOLUTION -- 2020 Homework 1, Problem 1, Part c
  //
  //  Compute the spin direction. (This will be the same as az AoTW.)
  //
  pNorm spin_dir = cross(first_pos,nadir_pos,last_pos);
  //
  if ( opt_spin )
    for ( auto& b: balls )
      {
        //  Set speed of each ball to 10 times distance between ball
        //  and nearest point on line through first_pos and last_pos.

        pVect to_b0(b.position, first_pos);

        // Compute distance from this ball to the closest point on
        // first_pos/last_pos line.
        //
        const float dist = dot( to_b0, ay );
        //
```

```
        // This only works when the ball is on the ax/ay plane.

        // Make velocity proportional to distance.
        //
        b.velocity = dist * 10 * spin_dir;
      }
}



void
World::ball_setup_4()
{
  /// Arrange and size balls to form a pendulum.

  // Desired position of first ball.
  //
  pCoor first_ball_pos(13.4,17.8,-9.2);

  // Desired distance between adjacent balls.
  //
  pVect ball_separation(distance_relaxed, 0, 0);  // Points in +x direction.

  for ( int i=0; i<chain_length; i++ )
    {
      Ball* const ball = &balls[i];
      ball->position = first_ball_pos + i * ball_separation;

      ball->velocity = pVect(0,0,0);
      ball->radius = ( i == 0 ? 0.6 : 0.3 );
      ball->mass = 4/3.0 * M_PI * pow(ball->radius,3);
      ball->contact = false;
      ball->constraint = OC_Free;

      /// SOLUTION -- 2020 Homework 1, Problem 1, Part b
      //
      //  Need to initialize distance_relaxed wherever balls are inited.
      //
      ball->distance_relaxed = distance_relaxed;
    }

}

void
World::ball_setup_5()
{
}


 /// Advance Simulation State by delta_t Seconds
//
void
World::time_step_cpu(double delta_t)
{
```

```
  time_step_count++;

  //
  /// Compute total force acting on each ball
  //
  for ( int i=0; i<chain_length; i++ )
    {
      Ball* const ball = &balls[i];

      ball->force = pVect(0,0,0);

      // Gravitational Force
      //
      ball->force += ball->mass * gravity_accel;

      // Spring Force from Neighbor Balls
      //
      for ( int direction: { -1, +1 } )
        {
          const int n_idx = i + direction;  // Compute neighbor index.

          // Skip this neighbor if neighbor doesn't exit.
          //
          if ( n_idx < 0 ) continue;
          if ( n_idx >= chain_length ) continue;

          Ball* const neighbor_ball = &balls[n_idx];

          // Construct a normalized (Unit) Vector from ball to neighbor.
          //
          pNorm ball_to_neighbor( ball->position, neighbor_ball->position );

          // Get distance between balls using pNorm member magnitude.
          //
          const float distance_between_balls = ball_to_neighbor.magnitude;

          /// SOLUTION -- 2020 Homework 1, Problem 1, Part b
          //
          //  Find index of the higher-numbered ball. It is the one with
          //  the distance_relaxed value to use.
          //
          const int idx_gt = max(i,n_idx);

          // Compute by how much the spring is stretched (positive value)
          // or compressed (negative value).
          //
          /// SOLUTION -- 2020 Homework 1, Problem 1, Part b
          //
          //  Get the relaxed distance from the ball rather than using
          //  World::distance_relaxed.
          //
          const float spring_stretch =
            distance_between_balls
            - balls[idx_gt].distance_relaxed * fabs(direction);
```

```
        // Compute the speed of ball towards neighbor_ball.
        //
        pVect delta_v = neighbor_ball->velocity - ball->velocity;
        float delta_s = dot( delta_v, ball_to_neighbor );


        // Determine whether spring is gaining energy (whether its length
        // is getting further from its relaxed length).
        //
        const bool gaining_e = ( delta_s > 0.0 ) == ( spring_stretch > 0 );

        // Use a smaller spring constant when spring is loosing energy,
        // a quick and dirty way of simulating energy loss due to spring
        // friction.
        //
        const float spring_constant =
          gaining_e ? opt_spring_constant : opt_spring_constant * 0.7;

        ball->force += spring_constant * spring_stretch * ball_to_neighbor;
      }
  }

// Impact on velocity due to air resistance.
const double fs = pow(1+opt_air_resistance,-delta_t);

/// Update Velocity and Position of Each Ball
//
for ( int i=0; i<chain_length; i++ )
  {
    Ball* const ball = &balls[i];

    // Select appropriate velocity/position update based upon
    // ball's constraint.

    switch ( ball->constraint ) {

    case OC_Free:
      //
      // No constraints, use the force as computed above ..
      // .. and check for and resolve platform collision.

      // Update Velocity and Position
      //
      ball->velocity += ( ball->force / ball->mass ) * delta_t;
      ball->velocity *= fs;  // Air resistance.
      ball->position += ball->velocity * delta_t;

      // Possible Collision with Platform
      //
      // Skip if collision impossible.
      //
      if ( !platform_collision_possible(ball->position) ) break;
      if ( ball->position.y >= ball->radius ) break;
```

```
        // Reflect y (vertical) component of velocity, with a reduction
        // due to energy lost in the collision.
        //
        if ( ball->velocity.y < 0 ) ball->velocity.y = - 0.9 * ball->velocity.y;
        break;

      case OC_Locked:
        //
        // Ball does not move. Forces are ignored and position does not change.

        ball->velocity = pVect(0,0,0);
        break;

      default: assert( false );
      }
    }
}


bool
World::platform_collision_possible(pCoor pos)
{
  // Assuming no motion in x or z axes.
  //
  return pos.x >= platform_xmin && pos.x <= platform_xmax
    && pos.z >= platform_zmin && pos.z <= platform_zmax;
}

 /// External Modifications to State
//
//   These allow the user to play with state while simulation
//   running.

// Move the ball.
//
void Ball::translate(pVect amt) {position += amt;}

// Add velocity to the ball.
//
void Ball::push(pVect amt) {velocity += amt;}

// Set the velocity to zero.
//
void Ball::stop() {velocity = pVect(0,0,0); }

// Set the velocity and rotation (not yet supported) to zero.
//
void Ball::freeze() {velocity = pVect(0,0,0); }



void World::balls_translate(pVect amt,int b){balls[b].translate(amt);}
void World::balls_push(pVect amt,int b){balls[b].push(amt);}
```

```
void World::balls_translate(pVect amt)
{ for(int i=0;i<chain_length;i++)balls[i].translate(amt);}
void World::balls_push(pVect amt)
{ for(int i=0;i<chain_length;i++)balls[i].push(amt);}
void World::balls_stop()
{
  for ( auto& b: balls ) b.stop();
}
void World::balls_freeze(){balls_stop();}


void
World::frame_callback()
{
  // This routine called whenever window needs to be updated.

  const double time_now = time_wall_fp();

  if ( !opt_pause || opt_single_frame || opt_single_time_step )
    {
      /// Advance simulation state.

      // Amount of time since the user saw the last frame.
      //
      const double wall_delta_t = time_now - last_frame_wall_time;

      const double time_step_duration = 0.0001;

      // Compute amount by which to advance simulation state for this frame.
      //
      const double duration =
        opt_single_time_step ? time_step_duration :
        opt_single_frame ? 1/30.0 :
        wall_delta_t;

      const double world_time_target = world_time + duration;

      while ( world_time < world_time_target )
        {
          time_step_cpu(time_step_duration);
          world_time += time_step_duration;
        }

      // Reset these, just in case they were set.
      //
      opt_single_frame = opt_single_time_step = false;
    }

  last_frame_wall_time = time_now;
  render();
}

int
main(int argv, char **argc)
```

```
{
  pOpenGL_Helper popengl_helper(argv,argc);
  World world(popengl_helper);

  popengl_helper.rate_set(30);
  popengl_helper.display_cb_set(world.frame_callback_w,&world);
}
```

# 24 Fall 2018 Solutions

```
/// # LSU EE 4702-1 (Fall 2018), GPU Programming
//
 /// ## Homework 1 -- SOLUTION
 //
 //  Search for SOLUTION in this file to find changed parts.

 /// ## Instructions
 //
 //  Read the assignment: https://www.ece.lsu.edu/koppel/gpup/2018/hw01.pdf
 //
 //  Search for "Problem" in this file to find the places to modify.
 //
 //  Only this file will be collected.


/// # Purpose
//
//   Demonstrate simulation of string modeled as point masses and springs


/// # What Code Does

// Simulates a string of beads over a platform. The string is modeled
// as point masses connected by springs with a long relaxed
// length. The platform consists of tiles, some are purple-tinted
// mirrors (showing a reflection of the ball), the others show the
// course syllabus.


///  # Keyboard Commands
 //
 /// ### Object (Eye, Light, Ball) Location or Push
 //   Arrows, Page Up, Page Down
 //        Move object or push ball, depending on mode.
 //        With shift key pressed, motion is 5x faster.
 //   'e': Move eye.
 //   'l': Move light.
 //   'b': Move head (first) ball. (Change position but not velocity.)
 //   'B': Push head ball. (Add velocity.)
 //
 /// ### Eye Direction
 //   Home, End, Delete, Insert
 //   Turn the eye direction.
 //   Home should rotate eye direction up, End should rotate eye
 //   down, Delete should rotate eye left, Insert should rotate eye
 //   right.  The eye direction vector is displayed in the upper left.

 /// ### Simulation Options
 //  (Also see variables below.)
 //
 //  '1'    Set up scene 1.
 //  '2'    Set up scene 2.
```

```
//  'p'    Pause simulation. (Press again to resume.)
//  ' '    (Space bar.) Advance simulation by 1/30 second.
//  'S- '  (Shift-space bar.) Advance simulation by one time step.
//  'h'    Freeze position of first (head) ball. (Press again to release.)
//  't'    Freeze position of last (tail) ball. (Press again to release.)
//  's'    Stop balls.
//  'g'    Turn gravity on and off.
//  'F12'  Write screenshot to file.

/// Variables
//   Selected program variables can be modified using the keyboard.
//   Use "Tab" to cycle through the variable to be modified, the
//   name of the variable is displayed next to "VAR" on the bottom
//   line of green text.

//  'Tab' Cycle to next variable.
//  '`'   Cycle to previous variable.
//  '+'   Increase variable value.
//  '-'   Decrease variable value.
//
//  VAR Spring Constant - Set spring constant.
//  VAR Air Resistance - Set air resistance.
//  VAR Light Intensity - The light intensity.
//  VAR Gravity - Gravitational acceleration. (Turn on/off using 'g'.)


#define GL_GLEXT_PROTOTYPES

#include <GL/gl.h>
#include <GL/freeglut.h>

#include <gp/util.h>
#include <gp/coord.h>
#include <gp/shader.h>
#include <gp/pstring.h>
#include <gp/misc.h>
#include <gp/gl-buffer.h>
#include <gp/texture-util.h>

#include "shapes.h"


///
/// Main Data Structures
///
//
// class World: All data about scene.


class World;


// Object Holding Ball State
//
```

```
class Ball {
public:
  pCoor position;
  pVect velocity;

  float mass;
  float radius;

  bool contact;                 // Can be used for special effects.

  void push(pVect amt);
  void translate(pVect amt);
  void stop();
  void freeze();
};

#include "hw01-graphics.cc"


void
World::init()
{
  chain_length = 40;
  balls = new Ball[chain_length];

  distance_relaxed = 15.0 / chain_length;
  opt_spring_constant = 1000;
  variable_control.insert(opt_spring_constant,"Spring Constant");

  opt_gravity_accel = 9.8;
  opt_gravity = true;
  gravity_accel = pVect(0,-opt_gravity_accel,0);
  variable_control.insert(opt_gravity_accel,"Gravity");

  opt_time_step_hw01 = false;

  opt_air_resistance = 0.001;
  variable_control.insert(opt_air_resistance,"Air Resistance");

  world_time = 0;
  time_step_count = 0;
  last_frame_wall_time = time_wall_fp();
  frame_timer.work_unit_set("Steps / s");

  init_graphics();

  // Initialize Scene 3/4 positions so that they can be used in
  // Scene 5 without error when Scene 5 is invoked before 3 and 4.
  hw01_center = pCoor(13.4,17.8,-9.2);
  hw01_dir = pVect(0,-10,0);
  hw01_r = 3;

  curr_setup = 1; ball_setup_1();
}
```

```
///
/// Physical Simulation Code
///

 /// Initialize Simulation
//
void
World::ball_setup_1()
{
  /// Arrange balls almost vertically.

  // Desired position of bottom ball.
  //
  pCoor bottom_pos(12.5,distance_relaxed,-13.7);

  // Desired distance between adjacent balls.
  //
  pVect ball_separation(0.02, distance_relaxed, 0.01);  // Points up.

  for ( int i=0; i<chain_length; i++ )
    {
      Ball* const ball = &balls[chain_length-i-1];

      ball->position = bottom_pos + i * ball_separation;

      ball->velocity = pVect(0,0,0);
      ball->radius = 0.3 * distance_relaxed;
      ball->mass = 4/3.0 * M_PI * pow(ball->radius,3);
      ball->contact = false;
    }

  opt_head_lock = false;
  opt_tail_lock = false;
}

void
World::ball_setup_2()
{
  /// Arrange and size balls to form a pendulum.

  // Desired position of first ball.
  //
  pCoor first_ball_pos(13.4,17.8,-9.2);

  // Desired distance between adjacent balls.
  //
  pVect ball_separation(distance_relaxed, 0, 0);  // Points in +x direction.

  for ( int i=0; i<chain_length; i++ )
    {
      Ball* const ball = &balls[i];
      ball->position = first_ball_pos + i * ball_separation;
```

```
      ball->velocity = pVect(0,0,0);
      ball->radius = ( i == chain_length - 1 ? 0.6 : 0.3 ) * distance_relaxed;
      ball->mass = 4/3.0 * M_PI * pow(ball->radius,3);
      ball->contact = false;
    }

  opt_head_lock = true;
}

void
World::ball_setup_3()
{
  hw01_center = pCoor(13.4,17.8,-9.2);
  hw01_dir = pVect(0,-10,0);
  hw01_r = 3;
  hw01(hw01_center, hw01_dir,hw01_r);
}

void
World::ball_setup_4()
{
  hw01_center = pCoor(13.4,17.8,-9.2);
  hw01_dir = pVect(drand48(),drand48()*4-10,drand48());
  hw01_r = 1 + drand48() * 3;
  hw01(hw01_center,hw01_dir,hw01_r);
}

void
World::ball_setup_5()
{
  hw01_alt(hw01_center,hw01_dir,hw01_r);
}

void
World::hw01(pCoor center, pVect dir, float r)
{
  /// SOLUTION -- Problem 2 -- Good Solution

  // The following solution is reasonable and would receive full
  // credit. See hw01_alt below for an alternative solution which
  // performs less computation by replacing the cosf/sinf with a
  // transformation matrix.

  // Compute angle between neighboring balls based on 8 balls per revolution.
  //
  const float delta_theta = 2 * M_PI / 8;
  //
  // Eight is an arbitrary choice. It looks good.

  //
  // Compute local-space axes used to draw the spiral.
  //

  // First, find one vector orthogonal to dir.
```

```
  //
  pNorm ax = dir.x ? pVect( -dir.y, dir.x, 0 ) : pVect( 0, -dir.z, dir.y );

  // Then compute one orthogonal to dir and ax.
  //
  pNorm ay = cross(dir,ax);
  // pNorm az(dir); // This is commented out because we don't need az directly.
  //
  // In the local-space coordinates a circle is drawn on the xy plane.

  // Scale the axes so that we get a spiral of the correct radius
  // and a cylinder of the correct length.
  //
  pVect vx = ax * r;
  pVect vy = ay * r;
  pVect vz = dir / chain_length;  // Or: az * dir.mag() / chain_length.

  for ( int i=0; i<chain_length; i++ )
    {
      Ball* const ball = &balls[i];

      const float theta = i * delta_theta;
      ball->position =
        center + cosf(theta) * vx + sinf(theta) * vy + i * vz;
      //
      // Note: cosf(theta) * vx converts the x component of the
      // local coordinate, cosf(theta), to the global coordinate space.

      ball->velocity = pVect(0,0,0);
      ball->radius = 0.1;
      ball->mass = 4/3.0 * M_PI * pow(ball->radius,3);
      ball->contact = false;
    }

  // Set the spring distance_relaxed to the distance between adjacent
  // balls.
  //
  distance_relaxed = pNorm(balls[0].position,balls[1].position).magnitude;

  // Here is how distance_relaxed can be computed geometrically:
  //
  const float dr =
    pow( dot(vz,vz) /* the length of vz, squared */ +
         r * r * ( pow(1-cosf(delta_theta),2) + pow(sinf(delta_theta),2) ),
         0.5);
  // Make sure that dr is very close to distance_relaxed.
  const float err = fabs( distance_relaxed - dr );
  assert( err < 0.00001 );

  opt_head_lock = true;
  opt_pause = true;
}

void
```

```
World::hw01_alt(pCoor center, pVect dir, float r)
{
  /// SOLUTION -- Problem 2 - Alternative solution.

  // This solution uses a transformation matrix to avoid computing
  // cosf and sinf each iteration. The transformation matrix moves pos
  // (the current ball position) to the next position on the spiral.
  //
  // Here are the transformation performed by the matrix, m_next_point:
  //
  //  - Translate pos from its global position to the local position
  //    in which center (see call arguments above) is at the origin.
  //    See matrix to_local.
  //
  //  - Rotate the coordinate space so that dir points in the +z direction.
  //    See rot_local.
  //
  //    At this point the spiral rotates around the z axis and moves
  //    down the z axis.
  //
  //  - Rotate the coordinate space by delta_theta. This rotates
  //    the current pos to the next spiral pos, but doesn't change z.
  //    See rot_local.
  //
  //  - More the position down the z axis. See m_down.
  //
  //  - Rotate the coordinate space so that what was the z axis now
  //    points in direction dir. See rot_to_global.
  //
  //  - Translate pos from its local position back to the global
  //    position. See to_global.
  //

  // Compute angle between neighboring balls based on 8 balls per revolution.
  //
  const float delta_theta = 2 * M_PI / 8;
  //
  // Eight is an arbitrary choice.

  // Compute local-space axes used to draw the spiral.
  //
  pNorm az(dir);
  pNorm ax = dir.x ? pVect( -dir.y, dir.x, 0 ) : pVect( 0, -dir.z, dir.y );
  // No need for pNorm because az and ay are orthonormal:
  //   each is a unit vector and they are orthogonal to each other.
  pVect ay = cross(az,ax);

  //
  // The following two transformations are used to go from global
  // space to local space.
  //
  // Translate so that center (see call argument) is at the origin.
  //
  pMatrix_Translate to_local( -center );
```

```
//
// Construct a matrix that will rotate from global space to local
// space by placing the local axes in the rows.
//
pMatrix_Rows rot_to_local(ax,ay,az); // Construct a matrix using args as rows.

//
// The following two transforms move a point in local space along
// the spiral.
//
// Rotate, around the z axis, from one spiral position to the next.
//
pMatrix_Rotation rot_local( pVect(0,0,1), delta_theta );
//
// Translate down along z axis.
//
const float dir_len = az.magnitude;
const float delta_az = dir_len / chain_length;
pMatrix_Translate m_down(0,0,delta_az);

//
// The following two transformations are used to go from local
// space back to global space.
//
// Construct a matrix that rotates from local space back to global space
// by placing the local axes in the matrix columns.
//
pMatrix rot_to_global = transpose(rot_to_local);
//
// Translate back to center point.
//
pMatrix_Translate to_global( center );

pCoor pos0_local(r,0,0);  // The first point on the spiral in local coords.

// The first point on the spiral in global coords.
//
pCoor pos = to_global * rot_to_global * pos0_local;

// Compute matrix that will move a global coordinate to the next
// spiral position.
//
pMatrix m_next_point =
  to_global * rot_to_global * m_down * rot_local * rot_to_local * to_local;

for ( int i=0; i<chain_length; i++ )
  {
    Ball* const ball = &balls[i];
    ball->position = pos;

    // Use transformation to move pos to next position on spiral.
    //
    pos = m_next_point * pos;
    //
```

```
      // Note that we aren't computing sin/cos functions.

      ball->velocity = pVect(0,0,0);
      ball->radius = i & 1 ? 0.1 : 0.15; // So that it looks different hw01.
      ball->mass = 4/3.0 * M_PI * pow(ball->radius,3);
      ball->contact = false;
    }

  distance_relaxed = pNorm(balls[0].position,balls[1].position).magnitude;

  opt_head_lock = true;
  opt_pause = true;
}


 /// Advance Simulation State by delta_t Seconds
//
void
World::time_step_cpu_hw01(double delta_t)
{
  /// Homework 1 Problem 1 Solution Goes In This Routine

  time_step_count++;

  //
  /// Compute force and update velocity of each ball.
  //
  for ( int i=0; i<chain_length; i++ )
    {
      Ball* const ball = &balls[i];

      // Skip locked balls.
      //
      if ( opt_head_lock && i == 0 || opt_tail_lock && i == chain_length - 1 )
        {
          ball->velocity = pVect(0,0,0);
          continue;
        }

      pVect force(0,0,0);

      // Gravitational Force
      //
      force += ball->mass * gravity_accel;

      // Spring Force from Neighbor Balls
      //
      /// SOLUTION -- Problem 1 -- Also check distance-2 neighbors.
      for ( int direction: { -2, -1, +1, +2 } )
        {
          const int n_idx = i + direction;  // Compute neighbor index.

          // Skip this neighbor if neighbor doesn't exit.
          //
```

```
      if ( n_idx < 0 ) continue;

      /// SOLUTION -- Problem 1 -- Take into account +2.
      if ( n_idx >= chain_length ) continue;

      Ball* const neighbor_ball = &balls[n_idx];

      // Construct a normalized (Unit) Vector from ball to neighbor.
      //
      pNorm ball_to_neighbor( ball->position, neighbor_ball->position );

      // Get distance between balls using pNorm member magnitude.
      //
      const float distance_between_balls = ball_to_neighbor.magnitude;

      // Compute by how much the spring is stretched (positive value)
      // or compressed (negative value).
      //
      /// SOLUTION -- Problem 1 -- Scale relaxed length by neighbor distance.
      const float spring_stretch =
        distance_between_balls - distance_relaxed * abs(direction);

      // Compute the speed of ball towards neighbor_ball.
      //
      pVect delta_v = neighbor_ball->velocity - ball->velocity;
      float delta_s = dot( delta_v, ball_to_neighbor );


      // Determine whether spring is gaining energy (whether its length
      // is getting further from its relaxed length).
      //
      const bool gaining_e = ( delta_s > 0.0 ) == ( spring_stretch > 0 );

      // Use a smaller spring constant when spring is loosing energy,
      // a quick and dirty way of simulating energy loss due to spring
      // friction.
      //
      const float spring_constant =
        gaining_e ? opt_spring_constant : opt_spring_constant * 0.7;

      force += spring_constant * spring_stretch * ball_to_neighbor;
    }

  // Update Velocity
  //
  // This code assumes that force on ball is constant over time
  // step. This is clearly wrong when balls are moving with
  // respect to each other because the springs are changing
  // length. This inaccuracy will make the simulation unstable
  // when spring constant is large for the time step.
  //
  ball->velocity += ( force / ball->mass ) * delta_t;

  // Air Resistance
```

```
      //
      const double fs = pow(1+opt_air_resistance,-delta_t);
      ball->velocity *= fs;
    }

  ///
  /// Update Position of Each Ball
  ///

  for ( int i=0; i<chain_length; i++ )
    {
      Ball* const ball = &balls[i];

      // Update Position
      //
      // Assume that velocity is constant.
      //
      ball->position += ball->velocity * delta_t;

      // Possible Collision with Platform
      //

      // Skip if collision impossible.
      //
      if ( !platform_collision_possible(ball->position) ) continue;
      if ( ball->position.y >= 0 ) continue;

      // Reflect y (vertical) component of velocity, with a reduction
      // due to energy lost in the collision.
      //
      if ( ball->velocity.y < 0 )
        ball->velocity.y = - 0.9 * ball->velocity.y;
    }
}

 /// Advance Simulation State by delta_t Seconds
//
void
World::time_step_cpu_orig(double delta_t)
{
  time_step_count++;

  //
  /// Compute force and update velocity of each ball.
  //
  for ( int i=0; i<chain_length; i++ )
    {
      Ball* const ball = &balls[i];

      // Skip locked balls.
      //
      if ( opt_head_lock && i == 0 || opt_tail_lock && i == chain_length - 1 )
        {
          ball->velocity = pVect(0,0,0);
```

```
      continue;
    }

  pVect force(0,0,0);

  // Gravitational Force
  //
  force += ball->mass * gravity_accel;

  // Spring Force from Neighbor Balls
  //
  for ( int direction: { -1, +1 } )
    {
      const int n_idx = i + direction;  // Compute neighbor index.

      if ( n_idx < 0 ) continue;
      if ( n_idx == chain_length ) continue;

      Ball* const neighbor_ball = &balls[n_idx];

      // Construct a normalized (Unit) Vector from ball to neighbor.
      //
      pNorm ball_to_neighbor( ball->position, neighbor_ball->position );

      // Get distance between balls using pNorm member magnitude.
      //
      const float distance_between_balls = ball_to_neighbor.magnitude;

      // Compute by how much the spring is stretched (positive value)
      // or compressed (negative value).
      //
      const float spring_stretch =
        distance_between_balls - distance_relaxed;

      // Compute the speed of ball towards neighbor_ball.
      //
      pVect delta_v = neighbor_ball->velocity - ball->velocity;
      float delta_s = dot( delta_v, ball_to_neighbor );


      // Determine whether spring is gaining energy (whether its length
      // is getting further from its relaxed length).
      //
      const bool gaining_e = ( delta_s > 0.0 ) == ( spring_stretch > 0 );

      // Use a smaller spring constant when spring is loosing energy,
      // a quick and dirty way of simulating energy loss due to spring
      // friction.
      //
      const float spring_constant =
        gaining_e ? opt_spring_constant : opt_spring_constant * 0.7;

      force += spring_constant * spring_stretch * ball_to_neighbor;
    }
```

```
      // Update Velocity
      //
      // This code assumes that force on ball is constant over time
      // step. This is clearly wrong when balls are moving with
      // respect to each other because the springs are changing
      // length. This inaccuracy will make the simulation unstable
      // when spring constant is large for the time step.
      //
      ball->velocity += ( force / ball->mass ) * delta_t;

      // Air Resistance
      //
      const double fs = pow(1+opt_air_resistance,-delta_t);
      ball->velocity *= fs;
    }

  ///
  /// Update Position of Each Ball
  ///

  for ( int i=0; i<chain_length; i++ )
    {
      Ball* const ball = &balls[i];

      // Update Position
      //
      // Assume that velocity is constant.
      //
      ball->position += ball->velocity * delta_t;

      // Possible Collision with Platform
      //

      // Skip if collision impossible.
      //
      if ( !platform_collision_possible(ball->position) ) continue;
      if ( ball->position.y >= 0 ) continue;

      // Snap ball position to surface.
      //
      ball->position.y = 0;

      // Reflect y (vertical) component of velocity, with a reduction
      // due to energy lost in the collision.
      //
      if ( ball->velocity.y < 0 )
        ball->velocity.y = - 0.9 * ball->velocity.y;
    }
}

bool
World::platform_collision_possible(pCoor pos)
{
```

```
  // Assuming no motion in x or z axes.
  //
  return pos.x >= platform_xmin && pos.x <= platform_xmax
    && pos.z >= platform_zmin && pos.z <= platform_zmax;
}

 /// External Modifications to State
//
//   These allow the user to play with state while simulation
//   running.

// Move the ball.
//
void Ball::translate(pVect amt) {position += amt;}

// Add velocity to the ball.
//
void Ball::push(pVect amt) {velocity += amt;}

// Set the velocity to zero.
//
void Ball::stop() {velocity = pVect(0,0,0); }

// Set the velocity and rotation (not yet supported) to zero.
//
void Ball::freeze() {velocity = pVect(0,0,0); }



void World::balls_translate(pVect amt,int b){balls[b].translate(amt);}
void World::balls_push(pVect amt,int b){balls[b].push(amt);}
void World::balls_translate(pVect amt)
{ for(int i=0;i<chain_length;i++)balls[i].translate(amt);}
void World::balls_push(pVect amt)
{ for(int i=0;i<chain_length;i++)balls[i].push(amt);}
void World::balls_stop()
{ for(int i=0;i<chain_length;i++)balls[i].stop();}
void World::balls_freeze(){balls_stop();}


void
World::frame_callback()
{
  // This routine called whenever window needs to be updated.

  const double time_now = time_wall_fp();

  if ( !opt_pause || opt_single_frame || opt_single_time_step )
    {
      /// Advance simulation state.

      // Amount of time since the user saw the last frame.
      //
```

```
      const double wall_delta_t = time_now - last_frame_wall_time;

      const double time_step_duration = 0.0001;

      // Compute amount by which to advance simulation state for this frame.
      //
      const double duration =
        opt_single_time_step ? time_step_duration :
        opt_single_frame ? 1/30.0 :
        wall_delta_t;

      const double world_time_target = world_time + duration;

      while ( world_time < world_time_target )
        {
          if ( opt_time_step_hw01 )
            time_step_cpu_hw01(time_step_duration);
          else
            time_step_cpu_orig(time_step_duration);
          world_time += time_step_duration;
        }

      // Reset these, just in case they were set.
      //
      opt_single_frame = opt_single_time_step = false;
    }

  last_frame_wall_time = time_now;
  render();
}

int
main(int argv, char **argc)
{
  pOpenGL_Helper popengl_helper(argv,argc);
  World world(popengl_helper);

  popengl_helper.rate_set(30);
  popengl_helper.display_cb_set(world.frame_callback_w,&world);
}
```

```
/// LSU EE 4702-1 (Fall 2018), GPU Programming
//
 /// Homework 2 -- SOLUTION
 //

 /// Instructions
 //
 //  Read the assignment: https://www.ece.lsu.edu/koppel/gpup/2018/hw02.pdf
 //
 //  Search for "Problem" in this file to find the places to modify.
 //
 //  Only this file will be collected.


/// Purpose
//
//   Demonstrate simulation of string modeled as point masses and springs


/// What Code Does

// Simulates a string of beads over a platform. The string is modeled
// as point masses connected by springs with a long relaxed
// length. The platform consists of tiles, some are purple-tinted
// mirrors (showing a reflection of the ball), the others show the
// course syllabus.


///  Keyboard Commands
 //
 /// Object (Eye, Light, Ball) Location or Push
 //   Arrows, Page Up, Page Down
 //        Move object or push ball, depending on mode.
 //        With shift key pressed, motion is 5x faster.
 //   'e': Move eye.
 //   'l': Move light.
 //   'b': Move head (first) ball. (Change position but not velocity.)
 //   'B': Push head ball. (Add velocity.)
 //
 /// Eye Direction
 //   Home, End, Delete, Insert
 //   Turn the eye direction.
 //   Home should rotate eye direction up, End should rotate eye
 //   down, Delete should rotate eye left, Insert should rotate eye
 //   right.  The eye direction vector is displayed in the upper left.

 /// Simulation Options
 //  (Also see variables below.)
 //
 //  '1'    Set up scene 1.
 //  '2'    Set up scene 2.
 //  'p'    Pause simulation. (Press again to resume.)
 //  ' '    (Space bar.) Advance simulation by 1/30 second.
 //  'S- '  (Shift-space bar.) Advance simulation by one time step.
 //  'h'    Freeze position of first (head) ball. (Press again to release.)
 //  't'    Freeze position of last (tail) ball. (Press again to release.)
 //  's'    Stop balls.
 //  'g'    Turn gravity on and off.
 //  'F12'  Write screenshot to file.

 /// Variables
 //   Selected program variables can be modified using the keyboard.
```

```
//   Use "Tab" to cycle through the variable to be modified, the
//   name of the variable is displayed next to "VAR" on the bottom
//   line of green text.

//  'Tab' Cycle to next variable.
//  '`'   Cycle to previous variable.
//  '+'   Increase variable value.
//  '-'   Decrease variable value.
//
//  VAR Spring Constant - Set spring constant.
//  VAR Air Resistance - Set air resistance.
//  VAR Light Intensity - The light intensity.
//  VAR Gravity - Gravitational acceleration. (Turn on/off using 'g'.)


#define GL_GLEXT_PROTOTYPES

#include <GL/gl.h>
#include <GL/freeglut.h>

#include <gp/util.h>
#include <gp/coord.h>
#include <gp/shader.h>
#include <gp/pstring.h>
#include <gp/misc.h>
#include <gp/gl-buffer.h>
#include <gp/texture-util.h>

#include "shapes.h"


///
/// Main Data Structures
///
//
// class World: All data about scene.


class World;


// Object Holding Ball State
//
class Ball {
public:
  pCoor position;
  pVect velocity;

  float mass;
  float radius;

  bool contact;                 // Can be used for special effects.

  void push(pVect amt);
  void translate(pVect amt);
  void stop();
  void freeze();
};

#include "hw02-graphics.cc"


void
```

```
World::init()
{
  chain_length = 20;
  balls = new Ball[chain_length];

  opt_n_segs = 10;
  variable_control.insert(opt_n_segs,"Number of louver segments.");

  distance_relaxed = 30.0 / chain_length;
  opt_spring_constant = 1000;
  variable_control.insert(opt_spring_constant,"Spring Constant");

  opt_gravity_accel = 9.8;
  opt_gravity = true;
  gravity_accel = pVect(0,-opt_gravity_accel,0);
  variable_control.insert(opt_gravity_accel,"Gravity");

  opt_air_resistance = 0.001;
  variable_control.insert(opt_air_resistance,"Air Resistance");

  world_time = 0;
  time_step_count = 0;
  last_frame_wall_time = time_wall_fp();
  frame_timer.work_unit_set("Steps / s");

  init_graphics();

  curr_setup = 1; ball_setup_1();
}

void
World::hw02_render(bool shadows)
{
  /// Homework 2: All solutions in this routine.

  glDisable(GL_COLOR_MATERIAL); // Don't worry about re-enabling it.
  glMaterialfv( GL_FRONT, GL_AMBIENT_AND_DIFFUSE, color_salmon );
  glMaterialfv( GL_BACK, GL_AMBIENT_AND_DIFFUSE, color_spring_green );

  if ( opt_fill == Fill_None ) return;

  if ( opt_fill == Fill_Wye )
    {
      if ( shadows ) return;
      glMaterialfv
        ( GL_FRONT_AND_BACK, GL_AMBIENT_AND_DIFFUSE, color_lsu_spirit_gold );
      for ( int i=2; i<chain_length; i++ )
        {
          vector<pCoor> pa =
            { balls[i-2].position, balls[i-1].position, balls[i].position };
          pCoor ctr = (pa[0]+pa[1]+pa[2])/3;

          glBegin(GL_LINES);
          for ( pCoor& c: pa )
            {
              glVertex3fv(c);
              glVertex3fv(ctr);
            }
          glEnd();
        }
      return;
    }

  pColor fcolors[] = { color_salmon, color_tomato };
  pColor bcolors[] = { color_spring_green, color_chartreuse };
```

```
/// Homework 2: SOLUTION

glMaterialfv( GL_FRONT, GL_AMBIENT_AND_DIFFUSE, color_salmon );
glMaterialfv( GL_BACK, GL_AMBIENT_AND_DIFFUSE, color_spring_green );

for ( int i=2; i<chain_length; i++ )
  {
    // Put ball structures and coordinates into convenient variables.
    //
    Ball* const b0 = &balls[i-2];
    Ball* const b1 = &balls[i-1];
    Ball* const b2 = &balls[i-0];
    pCoor p0 = b0->position;
    pCoor p1 = b1->position;
    pCoor p2 = b2->position;

    // Compute location of triangle center.
    //
    pCoor ctr = (p0+p1+p2)/3;

    // Compute vectors from triangle center to each ball location ..
    // .. and put in array va.
    //
    pCoor pa[3] = {p0,p1,p2};
    vector<pVect> va;
    for ( auto& p: pa ) va.emplace_back( ctr, p );

    pNorm nz = cross(p0,p1,p2);
    pVect vz = 0.1 * nz;

    const float delta_a = 0.8 / opt_n_segs;

    // Coordinate of previous position in triangular spiral.
    //
    pCoor pprev(0,0,0);

    for ( int j=0; j<opt_n_segs; j++ )
      {
        // Distance from triangle center.
        //
        const float a = j * delta_a;

        // The vector to use for the current iteration.
        //
        pVect v = va[j%3];

        // Compute point on bend of spiral, and its normal.
        //
        pCoor p = ctr + a * v;
        pNorm n = cross(pVect(pprev,p),vz);

        if ( j )
          {
            if ( shadows )
              {

                // Render just one segment if shadow volumes are visible.
                //
                if ( opt_shadow_volumes_visible
                     && ( i != 3 || j != opt_n_segs - 1 ) )
                  {
                    pprev = p;
                    continue;
                  }
```

```
                // Otherwise, render the shadow volume.
                //
                const bool facing_light =
                  dot(n,pVect(p,light_location)) > 0;
                glBegin(GL_TRIANGLE_STRIP);

                // Iterate around vertices of segment. The first and
                // last vertices are the same.
                //
                for ( pCoor c: { pprev+vz, pprev-vz, p-vz, p+vz, pprev+vz } )
                  {
                    pCoor c2 = c + 1000 * pNorm(light_location,c);
                    glVertex3fv(facing_light ? c : c2);
                    glVertex3fv(facing_light ? c2 : c);
                  }
                glEnd();
              }
            else
              {
                // Set colors to use for this position.
                //
                glMaterialfv(GL_FRONT, GL_AMBIENT_AND_DIFFUSE, fcolors[j&1] );
                glMaterialfv(GL_BACK, GL_AMBIENT_AND_DIFFUSE, bcolors[j&1] );

                glBegin(GL_TRIANGLE_STRIP);
                glNormal3fv(n);
                glVertex3fv(pprev + vz);
                glVertex3fv(pprev - vz);
                glVertex3fv(p + vz);
                glVertex3fv(p - vz);
                glEnd();
              }
          }
        pprev = p;
      }
  }
}


///
/// Physical Simulation Code
///

 /// Initialize Simulation
//
void
World::ball_setup_1()
{
  /// Arrange balls almost vertically.

  pNorm dir(0.1,1,0.2);

  pCoor bottom_pos(12.5,distance_relaxed,-13.7);

  pNorm ax = dir.x ? pVect(-dir.y,dir.x,0) : pVect(0,-dir.z,dir.y);
  pVect leg_x = distance_relaxed * powf(0.75,0.5) * ax;
  pVect leg_y = distance_relaxed * 0.5f * dir;

  for ( int i=0; i<chain_length; i++ )
    {
      Ball* const ball = &balls[chain_length-i-1];
```

```
      ball->position = bottom_pos + i * leg_y + ( i & 1 ) * leg_x;

      ball->velocity = pVect(0,0,0);
      ball->radius = 0.1 * distance_relaxed;
      ball->mass = 4/3.0 * M_PI * pow(ball->radius,3);
      ball->contact = false;
    }

  opt_head_lock = true;
  opt_tail_lock = false;
}

void
World::ball_setup_2()
{
  /// Arrange and size balls to form a pendulum.

  // Desired position of first ball.
  //
  pCoor first_ball_pos(13.4,17.8,-9.2);

  // Desired distance between adjacent balls.
  //
  pVect ball_separation(distance_relaxed, 0, 0);  // Points in +x direction.

  for ( int i=0; i<chain_length; i++ )
    {
      Ball* const ball = &balls[i];
      ball->position = first_ball_pos + i * ball_separation;

      ball->velocity = pVect(0,0,0);
      ball->radius = ( i == chain_length - 1 ? 0.6 : 0.3 ) * distance_relaxed;
      ball->mass = 4/3.0 * M_PI * pow(ball->radius,3);
      ball->contact = false;
    }

  opt_head_lock = true;
}

void
World::ball_setup_3()
{
}

void
World::ball_setup_4()
{
}

void
World::ball_setup_5()
{
}


 /// Advance Simulation State by delta_t Seconds
//
void
World::time_step_cpu(double delta_t)
{
  time_step_count++;

  //
  /// Compute force and update velocity of each ball.
```

```
//
for ( int i=0; i<chain_length; i++ )
  {
    Ball* const ball = &balls[i];

    // Skip locked balls.
    //
    if ( opt_head_lock && i == 0 || opt_tail_lock && i == chain_length - 1 )
      {
        ball->velocity = pVect(0,0,0);
        continue;
      }

    pVect force(0,0,0);

    // Gravitational Force
    //
    force += ball->mass * gravity_accel;

    // Spring Force from Neighbor Balls
    //
    for ( int direction: { -2, -1, +1, +2 } )
      {
        const int n_idx = i + direction;  // Compute neighbor index.

        // Skip this neighbor if neighbor doesn't exit.
        //
        if ( n_idx < 0 ) continue;

        if ( n_idx >= chain_length ) continue;

        Ball* const neighbor_ball = &balls[n_idx];

        // Construct a normalized (Unit) Vector from ball to neighbor.
        //
        pNorm ball_to_neighbor( ball->position, neighbor_ball->position );

        // Get distance between balls using pNorm member magnitude.
        //
        const float distance_between_balls = ball_to_neighbor.magnitude;

        // Compute by how much the spring is stretched (positive value)
        // or compressed (negative value).
        //
        const float spring_stretch =
          distance_between_balls - distance_relaxed;

        // Compute the speed of ball towards neighbor_ball.
        //
        pVect delta_v = neighbor_ball->velocity - ball->velocity;
        float delta_s = dot( delta_v, ball_to_neighbor );


        // Determine whether spring is gaining energy (whether its length
        // is getting further from its relaxed length).
        //
        const bool gaining_e = ( delta_s > 0.0 ) == ( spring_stretch > 0 );

        // Use a smaller spring constant when spring is loosing energy,
        // a quick and dirty way of simulating energy loss due to spring
        // friction.
        //
        const float spring_constant =
          gaining_e ? opt_spring_constant : opt_spring_constant * 0.7;
```

```
          force += spring_constant * spring_stretch * ball_to_neighbor;
        }

      // Update Velocity
      //
      // This code assumes that force on ball is constant over time
      // step. This is clearly wrong when balls are moving with
      // respect to each other because the springs are changing
      // length. This inaccuracy will make the simulation unstable
      // when spring constant is large for the time step.
      //
      ball->velocity += ( force / ball->mass ) * delta_t;

      // Air Resistance
      //
      const double fs = pow(1+opt_air_resistance,-delta_t);
      ball->velocity *= fs;
    }

  ///
  /// Update Position of Each Ball
  ///

  for ( int i=0; i<chain_length; i++ )
    {
      Ball* const ball = &balls[i];

      // Update Position
      //
      // Assume that velocity is constant.
      //
      ball->position += ball->velocity * delta_t;

      // Possible Collision with Platform
      //

      // Skip if collision impossible.
      //
      if ( !platform_collision_possible(ball->position) ) continue;
      if ( ball->position.y >= 0 ) continue;

      // Reflect y (vertical) component of velocity, with a reduction
      // due to energy lost in the collision.
      //
      if ( ball->velocity.y < 0 )
        ball->velocity.y = - 0.9 * ball->velocity.y;
    }
}

bool
World::platform_collision_possible(pCoor pos)
{
  // Assuming no motion in x or z axes.
  //
  return pos.x >= platform_xmin && pos.x <= platform_xmax
    && pos.z >= platform_zmin && pos.z <= platform_zmax;
}

 /// External Modifications to State
//
//   These allow the user to play with state while simulation
//   running.

// Move the ball.
//
```

```
void Ball::translate(pVect amt) {position += amt;}

// Add velocity to the ball.
//
void Ball::push(pVect amt) {velocity += amt;}

// Set the velocity to zero.
//
void Ball::stop() {velocity = pVect(0,0,0); }

// Set the velocity and rotation (not yet supported) to zero.
//
void Ball::freeze() {velocity = pVect(0,0,0); }



void World::balls_translate(pVect amt,int b){balls[b].translate(amt);}
void World::balls_push(pVect amt,int b){balls[b].push(amt);}
void World::balls_translate(pVect amt)
{ for(int i=0;i<chain_length;i++)balls[i].translate(amt);}
void World::balls_push(pVect amt)
{ for(int i=0;i<chain_length;i++)balls[i].push(amt);}
void World::balls_stop()
{ for(int i=0;i<chain_length;i++)balls[i].stop();}
void World::balls_freeze(){balls_stop();}


void
World::frame_callback()
{
  // This routine called whenever window needs to be updated.

  const double time_now = time_wall_fp();

  if ( !opt_pause || opt_single_frame || opt_single_time_step )
    {
      /// Advance simulation state.

      // Amount of time since the user saw the last frame.
      //
      const double wall_delta_t = time_now - last_frame_wall_time;

      const double time_step_duration = 0.0001;

      // Compute amount by which to advance simulation state for this frame.
      //
      const double duration =
        opt_single_time_step ? time_step_duration :
        opt_single_frame ? 1/30.0 :
        wall_delta_t;

      const double world_time_target = world_time + duration;

      while ( world_time < world_time_target )
        {
          time_step_cpu(time_step_duration);
          world_time += time_step_duration;
        }

      // Reset these, just in case they were set.
      //
      opt_single_frame = opt_single_time_step = false;
    }
```

```
  last_frame_wall_time = time_now;
  render();
}

int
main(int argv, char **argc)
{
  pOpenGL_Helper popengl_helper(argv,argc);
  World world(popengl_helper);
# ifdef __OPTIMIZE__
  glDisable(GL_DEBUG_OUTPUT);
# else
  glEnable(GL_DEBUG_OUTPUT);
# endif
  glDebugMessageControl(GL_DONT_CARE,GL_DONT_CARE,
                        GL_DEBUG_SEVERITY_NOTIFICATION,0,NULL,false);

  popengl_helper.rate_set(30);
  popengl_helper.display_cb_set(world.frame_callback_w,&world);
}
```

```
/// LSU EE 4702-1 (Fall 2018), GPU Programming
//

 /// Homework 3 -- SOLUTION
 //
 //  Search for SOLUTION in this file to find solution code.

 /// Instructions
 //
 //  Read the assignment: https://www.ece.lsu.edu/koppel/gpup/2018/hw03.pdf
 //
 //  Modify code and declarations throughout this file.
 //
 //  This file and hw03.cc will be collected.


// Specify version of OpenGL Shading Language.
//
#version 450 compatibility

vec4 generic_lighting
(vec4 vertex_e, vec4 color, vec3 normal_e, bool front_facing);


uniform sampler2D tex_unit_0;


// Use these variables to debug your code. Press 'y' to toggle
// tryout.x and 'Y' to toggle debug_bool.y (between true and false).
//
layout ( location = 3 ) uniform bvec2 tryout;
layout ( location = 4 ) uniform float tryoutf;


 /// SOLUTION - Problems 2 and 3
layout ( location = 5 ) uniform vec3 spiral_normal;
layout ( location = 6 ) uniform float tex_ht;


///
/// Shader Input and Output Variables
///


// Declare variables for communication between vertex shader
// and fragment shader.
//
#ifdef _VERTEX_SHADER_

out Data
{
  vec3 normal_e;
  vec4 vertex_e;
  vec2 tex_coord;
};

#endif

#ifdef _FRAGMENT_SHADER_
in Data
{
 /// SOLUTION - Problem 1a
 //  Use flat interpolation so that one normal is used for the entire
```

```
 //  primitive.
 flat vec3 normal_e;

 vec4 vertex_e;
 vec2 tex_coord;
};

#endif


///
/// Shaders
///

#ifdef _VERTEX_SHADER_
void
vs_main_hw03()
{
  /// Homework 3:  Can put solution here, and other places.

  // Perform basic vertex shading operations.

  // Transform vertex to clip space.
  gl_Position = gl_ModelViewProjectionMatrix * gl_Vertex;

  // Compute eye-space vertex coordinate and normal.
  // These are outputs of the vertex shader and inputs to the frag shader.
  //
  vertex_e = gl_ModelViewMatrix * gl_Vertex;
  normal_e = normalize(gl_NormalMatrix * gl_Normal);

  // Copy color to output unmodified. Lighting calculations will
  // be performed in the fragment shader.
  //
  /// SOLUTION -- Problem 1b. Don't bother sending colors down the pipe.
  //  gl_BackColor = gl_FrontColor = gl_Color;

  // Copy texture coordinate to output (no need to modify it).
  tex_coord = gl_MultiTexCoord0.xy;
}

void
vs_main_plain()
{
  /// HOMEWORK 3: DO NOT PLACE SOLUTION HERE

  // Perform basic vertex shading operations.

  // Transform vertex to clip space.
  gl_Position = gl_ModelViewProjectionMatrix * gl_Vertex;

  // Compute eye-space vertex coordinate and normal.
  // These are outputs of the vertex shader and inputs to the frag shader.
  //
  vertex_e = gl_ModelViewMatrix * gl_Vertex;
  normal_e = normalize(gl_NormalMatrix * gl_Normal);

  // Copy color to output unmodified. Lighting calculations will
  // be performed in the fragment shader.
  //
  gl_BackColor = gl_FrontColor = gl_Color;

  // Copy texture coordinate to output (no need to modify it).
  tex_coord = gl_MultiTexCoord0.xy;
```

```
}

#endif


#ifdef _FRAGMENT_SHADER_
void
fs_main_hw03()
{
  /// Homework 3:  Can put solution here, and other places.

  /// SOLUTION -- Problem 2b
  //
  //  The value of tex_coord.x indicates distance along the triangular
  //  spiral. The value at the beginning of the spiral is zero and it
  //  increases based on the length of the spiral segments. This value
  //  is passed unmodified to the texture library call below (see the
  //  line starting "vec4 texel". On the host (CPU) code the texture
  //  object for the image has been set to wrap along the x (called s
  //  in OpenGL) dimension, and so a tex_coord.x value of 1.1 is
  //  equivalent to a value of 0.1.
  //
  //  The value of tex_coord.y varies from 0 to 1, a 0 indicates that
  //  the fragment is on the back edge of the segment, 0.5 indicates
  //  the middle, and so on. If tex_coord.y were used in the call to
  //  texture, below, then a segment, rather than showing about two
  //  lines of text, would show all lines (the entire vertical length)
  //  of a page. The actual value used in the call to texture, tc.y,
  //  is computed below. The computation uses line_num, which is the
  //  integer portion of tex_coord.x. The idea is that if tex_coord.x
  //  is, say, 0.1, then we should be near the top of the image, lets
  //  call that the first line. If tex_coord.x is 1.1 then we have
  //  gone off the right edge (since tex_coord.x > 1 ) and so we
  //  advance down in the y direction to the second line. Variable
  //  line_num is the number of lines down. Basically, we are adding
  //  the integer part of tex_coord.x to tex_coord.y, then scaling it
  //  by tex_ht. (Recall that tex_ht is the fraction of a page covered
  //  by one "line". If tex_ht = 0.1, that means that the text applied
  //  to a segment is 1/10 the height of the texture image.)

  float line_num = floor(tex_coord.x);
  vec2 tc = vec2( tex_coord.x, ( line_num + tex_coord.y ) * tex_ht );

  // Get filtered texel.
  //
  /// SOLUTION -- Problem 2
  //
  //  Use texture coordinate computed above in which y is advanced by
  //  the number of lines.
  //
  vec4 texel = texture(tex_unit_0,tc);

  vec3 nne = normalize(normal_e);

  // Homework 3: This was a placeholder value. It has been changed
  // to something based on fragment's position within primitive.
  //
  /// SOLUTION - Compute edge distance from texture y coordinate.
  //
  //  Use tex_coord.y unmodified.
  //
  float edge_dist = tex_coord.y;
  //
  // Range [0,1].  0.5, center of segment; 0, back edge; 1, front edge.
```

```
  // Homework 3: This was a placeholder value. It has been commented
  // out since spiral_normal has been declared as a uniform.
  //
  /// SOLUTION - Problem 3
  //  Use spiral normal value sent as a uniform.
  //  vec3 spiral_normal = vec3(1,0,0);

  // Homework 3: Blend incoming normal with triangular spiral's normal.
  vec3 bnorm =
    tryout.x
    ? nne
    : edge_dist > 0.9
    ? normalize(mix(nne,spiral_normal,2*(edge_dist-0.9)))
    : edge_dist < 0.1
    ? normalize(mix(-spiral_normal,nne,0.8+2*edge_dist))
    : nne;

  /// SOLUTION -- Problem 1b
  //  Use material property uniforms instead of color value.
  //
  vec4 color = gl_FrontFacing
    ? gl_FrontMaterial.diffuse : gl_BackMaterial.diffuse;

  // Compute lighted color of fragment.
  //
  /// SOLUTION - Problem 1 and 3
  //
  //  Use color and bnorm computed above.
  //
  vec4 lighted_color =
    generic_lighting( vertex_e, color, bnorm, gl_FrontFacing );

  // Combine filtered texel color with lighted color of fragment.
  //
  gl_FragColor = texel * lighted_color;

  // Copy fragment depth unmodified.
  //
  gl_FragDepth = gl_FragCoord.z;
}

void
fs_main_plain()
{
  /// HOMEWORK 3: DO NOT PLACE SOLUTION HERE

  // Get filtered texel.
  //
  vec4 texel = texture(tex_unit_0,tex_coord);

  // Compute lighted color of fragment.
  //
  vec4 lighted_color =
    generic_lighting( vertex_e, gl_Color, normalize(normal_e), gl_FrontFacing );

  // Combine filtered texel color with lighted color of fragment.
  //
  gl_FragColor = texel * lighted_color;

  // Copy fragment depth unmodified.
  //
  gl_FragDepth = gl_FragCoord.z;

  /// HOMEWORK 3: DO NOT PLACE SOLUTION HERE
}
```

```
#endif


vec4
generic_lighting(vec4 vertex_e, vec4 color, vec3 normal_e, bool front_facing)
{
  // Return lighted color of vertex_e.
  //
  vec4 light_pos = gl_LightSource[0].position;
  vec3 v_vtx_light = light_pos.xyz - vertex_e.xyz;
  float dist = length(v_vtx_light);
  float d_n_vl = dot(normalize(normal_e), v_vtx_light) / dist;
  float phase_light = max(0, front_facing ? d_n_vl : -d_n_vl );

  vec3 ambient_light = gl_LightSource[0].ambient.rgb;
  vec3 diffuse_light = gl_LightSource[0].diffuse.rgb;
  float distsq = dist * dist;
  float atten_inv =
    0 * gl_LightSource[0].constantAttenuation +
    gl_LightSource[0].linearAttenuation * dist +
    gl_LightSource[0].quadraticAttenuation * distsq;
  vec4 lighted_color;
  lighted_color.rgb =
    tryoutf * color.rgb * gl_LightModel.ambient.rgb
    + color.rgb
    * ( tryoutf * ambient_light + phase_light * diffuse_light ) / atten_inv;
  lighted_color.a = color.a;
  return lighted_color;
}
```

```
/// LSU EE 4702-1 (Fall 2018), GPU Programming
//
 /// Homework 3 -- SOLUTION
 //
 //  Search for SOLUTION in this file to find solution code.

 /// Instructions
 //
 //  Read the assignment: https://www.ece.lsu.edu/koppel/gpup/2018/hw03.pdf
 //
 //  Modify the code in routine hw03_render.
 //
 //  This file and hw03-shdr.cc will be collected.


/// Purpose
//
//   Demonstrate simulation of string modeled as point masses and springs


/// What Code Does

// Simulates a string of beads over a platform. The string is modeled
// as point masses connected by springs with a long relaxed
// length. The platform consists of tiles, some are purple-tinted
// mirrors (showing a reflection of the ball), the others show the
// course syllabus.


///  Keyboard Commands
 //
 /// Object (Eye, Light, Ball) Location or Push
 //   Arrows, Page Up, Page Down
 //        Move object or push ball, depending on mode.
 //        With shift key pressed, motion is 5x faster.
 //   'e': Move eye.
 //   'l': Move light.
 //   'b': Move head (first) ball. (Change position but not velocity.)
 //   'B': Push head ball. (Add velocity.)
 //
 /// Eye Direction
 //   Home, End, Delete, Insert
 //   Turn the eye direction.
 //   Home should rotate eye direction up, End should rotate eye
 //   down, Delete should rotate eye left, Insert should rotate eye
 //   right.  The eye direction vector is displayed in the upper left.

 /// Simulation Options
 //  (Also see variables below.)
 //
 //  '1'    Set up scene 1.
 //  '2'    Set up scene 2.
 //  'p'    Pause simulation. (Press again to resume.)
 //  ' '    (Space bar.) Advance simulation by 1/30 second.
 //  'S- '  (Shift-space bar.) Advance simulation by one time step.
 //  'h'    Freeze position of first (head) ball. (Press again to release.)
 //  't'    Freeze position of last (tail) ball. (Press again to release.)
 //  's'    Stop balls.
 //  'g'    Turn gravity on and off.
 //  'F12'  Write screenshot to file.

 /// Variables
```

```
//   Selected program variables can be modified using the keyboard.
//   Use "Tab" to cycle through the variable to be modified, the
//   name of the variable is displayed next to "VAR" on the bottom
//   line of green text.

//  'Tab' Cycle to next variable.
//  '`'   Cycle to previous variable.
//  '+'   Increase variable value.
//  '-'   Decrease variable value.
//
//  VAR Spring Constant - Set spring constant.
//  VAR Air Resistance - Set air resistance.
//  VAR Light Intensity - The light intensity.
//  VAR Gravity - Gravitational acceleration. (Turn on/off using 'g'.)


#define GL_GLEXT_PROTOTYPES

#include <GL/gl.h>
#include <GL/freeglut.h>

#include <gp/util.h>
#include <gp/coord.h>
#include <gp/shader.h>
#include <gp/pstring.h>
#include <gp/misc.h>
#include <gp/gl-buffer.h>
#include <gp/texture-util.h>

#include "shapes.h"


///
/// Main Data Structures
///
//
// class World: All data about scene.


class World;


// Object Holding Ball State
//
class Ball {
public:
  pCoor position;
  pVect velocity;

  float mass;
  float radius;

  bool contact;                 // Can be used for special effects.

  void push(pVect amt);
  void translate(pVect amt);
  void stop();
  void freeze();
};

#include "hw03-graphics.cc"


void
```

```
World::init()
{
  chain_length = 7;
  balls = new Ball[chain_length];

  variable_control.insert_linear(opt_tryoutf,"Tryout F",0.1);

  opt_n_segs = 17;
  variable_control.insert(opt_n_segs,"Number of louver segments.");

  distance_relaxed = 30.0 / chain_length;
  opt_spring_constant = 1000;
  variable_control.insert(opt_spring_constant,"Spring Constant");

  opt_gravity_accel = 9.8;
  opt_gravity = true;
  gravity_accel = pVect(0,-opt_gravity_accel,0);
  variable_control.insert(opt_gravity_accel,"Gravity");

  opt_air_resistance = 0.001;
  variable_control.insert(opt_air_resistance,"Air Resistance");

  world_time = 0;
  time_step_count = 0;
  last_frame_wall_time = time_wall_fp();
  frame_timer.work_unit_set("Steps / s");

  init_graphics(true);

  curr_setup = 1; ball_setup_1();
}

void
World::hw03_render(bool shadows)
{
  // Homework 3 -- Problem 2, and maybe other problems.

  switch ( opt_shader ){
  case SO_Fixed: sp_fixed->use(); break;
  case SO_Plain: sp_plain->use(); break;
  case SO_HW03: sp_hw03->use(); break;
  default: assert( false );
  }

  pColor color_inside = 0.5 * color_salmon;
  pColor color_outside = 0.5 * color_chartreuse;

  if ( opt_shader != SO_Fixed )
    {
      glUniform2i(3, opt_tryout_1, opt_tryout_2);
      glUniform1f(4, opt_tryoutf);
    }

  if ( opt_shader == SO_Plain )
    {
      glColor3fv( color_outside );
    }
  else
    {
      glDisable(GL_COLOR_MATERIAL); // Don't worry about re-enabling it.
      glMaterialfv( GL_FRONT, GL_AMBIENT_AND_DIFFUSE, color_inside );
      glMaterialfv( GL_BACK, GL_AMBIENT_AND_DIFFUSE, color_outside );
    }

  glEnable(GL_TEXTURE_2D);
```

```
glActiveTexture(GL_TEXTURE0);
glBindTexture(GL_TEXTURE_2D,texid_spiral_image);
glTexParameterf(GL_TEXTURE_2D,GL_TEXTURE_MAG_FILTER,GL_LINEAR);
glTexEnvi(GL_TEXTURE_ENV,GL_TEXTURE_ENV_MODE,GL_MODULATE);
glTexParameterf(GL_TEXTURE_2D,GL_TEXTURE_WRAP_S,GL_REPEAT);
glTexParameterf(GL_TEXTURE_2D,GL_TEXTURE_WRAP_T,GL_REPEAT);

for ( int i=2; i<chain_length; i++ )
  {
    // Put ball structures and coordinates into convenient variables.
    //
    Ball* const b0 = &balls[i-2];
    Ball* const b1 = &balls[i-1];
    Ball* const b2 = &balls[i-0];
    pCoor p0 = b0->position;
    pCoor p1 = b1->position;
    pCoor p2 = b2->position;

    // Compute location of triangle center.
    //
    pCoor ctr = (p0+p1+p2)/3;

    // Compute vectors from triangle center to each ball location ..
    // .. and put in array va.
    //
    pCoor pa[3] = {p0,p1,p2};
    vector<pVect> va;
    for ( auto& p: pa ) va.emplace_back( ctr, p );

    pNorm nz = cross(p0,p1,p2);
    const float vz_len = 0.1;
    pVect vz = vz_len * nz;

    const float delta_a = 0.8 / opt_n_segs;

    // Coordinate of previous position in triangular spiral.
    //
    pCoor pprev(0,0,0);

    /// SOLUTION -- Problem 2
    //
    //  Choose scaling factors for texture image.
    //
    const float tex_scale = 0.2;
    const float tex_ht = 2 * vz_len * tex_scale;
    //
    //  Start the position within the texture image at the
    //  upper-left corner. For Problem 2a the y component is
    //  computed further below. For Problem 2b the y component is
    //  computed in the fragment shader.
    //
    float tex_x = 0;

    /// SOLUTION -- Problem 2 and 3
    //
    if ( opt_shader == SO_HW03 )
      {
        // Send the spiral normal needed for Problem 3.
        //
        glUniform3fv(5, 1, nz);

        // Send the texture height used in Problem 2.
        //
        glUniform1f(6, tex_ht);
      }
```

```
if ( opt_shader != SO_HW03 )
  {
    /// Small Strips Rendering
    //
    // Render spiral using many triangle strips ..
    // .. one triangle strip for each louver.
    //

    for ( int j=0; j<opt_n_segs; j++ )
      {
        // Distance from triangle center.
        //
        const float a = j * delta_a;

        // The vector to use for the current iteration.
        //
        pVect v = va[j%3];

        // Compute point on bend of spiral, and its normal.
        //
        pCoor p = ctr + a * v;
        pNorm n = cross(pVect(pprev,p),vz);

        if ( j )
          {
            /// SOLUTION -- Problem 2a
            //
            //  Compute the length of the segment. This determines
            //  how wide a piece of the texture image is used.
            //
            const float llen = pVect(p,pprev).mag();

            /// SOLUTION -- Problem 2a
            //
            //  Set the y coordinate based on the x coordinate.
            //
            const float tex_y = tex_ht * floor(tex_x);

            glBegin(GL_TRIANGLE_STRIP);
            glNormal3fv(n);

            /// SOLUTION -- Problem 2a -- provide texture coordinate.
            glTexCoord2f( tex_x, tex_y );
            glVertex3fv(pprev + vz);

            /// SOLUTION -- Problem 2a -- provide texture coordinate.
            glTexCoord2f( tex_x, tex_y + tex_ht );
            glVertex3fv(pprev - vz);

            /// SOLUTION -- Problem 2a
            //
            //  Advance texture x component based on length of
            //  segment.
            //
            tex_x += llen * tex_scale;

            /// SOLUTION -- Problem 2a -- provide texture coordinate.
            glTexCoord2f( tex_x, tex_y );
            glVertex3fv(p + vz);

            /// SOLUTION -- Problem 2a -- provide texture coordinate.
            glTexCoord2f( tex_x, tex_y + tex_ht );
            glVertex3fv(p - vz);
            glEnd();
```

```
                }
              pprev = p;
            }
        }
      else
        {
          /// Large Strip Rendering
          //
          // Render entire spiral using one triangle strip.
          //

          glBegin(GL_TRIANGLE_STRIP);

          for ( int j=0; j<opt_n_segs; j++ )
            {
              // Distance from triangle center.
              //
              const float a = j * delta_a;

              // The vector to use for the current iteration.
              //
              pVect v = va[j%3];

              // Compute point on bend of spiral, and its normal.
              //
              pCoor p = ctr + a * v;
              pNorm n = cross(pVect(pprev,p),vz);

              glNormal3fv(n);

              /// SOLUTION -- Problem 2b
              //
              //  The texture x component is computed in the same way
              //  as for Problem 2a: it is incremented by the width of
              //  the segment.
              //
              //  However, the texture y component is either 0 or 1.
              //  The fragment shader will compute the correct value
              //  based on the value of the x component.


              /// SOLUTION -- Problem 2b -- provide texture coordinate.
              glTexCoord2f(tex_x,0);
              glVertex3fv(p + vz);

              /// SOLUTION -- Problem 2b -- provide texture coordinate.
              glTexCoord2f(tex_x,1);
              glVertex3fv(p - vz);

              /// SOLUTION -- Problem 2b
              //
              //  Advance texture x component based on length of segment.
              //
              const float llen = pVect(p,pprev).mag();
              tex_x += llen * tex_scale;

              pprev = p;
            }

          glEnd();
        }
    }

    sp_fixed->use();
```

```
}


///
/// Physical Simulation Code
///

 /// Initialize Simulation
//
void
World::ball_setup_1()
{
  /// Arrange balls almost vertically.

  pNorm dir(0.1,1,0.2);

  pCoor bottom_pos(12.5,distance_relaxed,-13.7);

  pNorm ax = dir.x ? pVect(-dir.y,dir.x,0) : pVect(0,-dir.z,dir.y);
  pVect leg_x = distance_relaxed * powf(0.75,0.5) * ax;
  pVect leg_y = distance_relaxed * 0.5f * dir;

  for ( int i=0; i<chain_length; i++ )
    {
      Ball* const ball = &balls[chain_length-i-1];

      ball->position = bottom_pos + i * leg_y + ( i & 1 ) * leg_x;

      ball->velocity = pVect(0,0,0);
      ball->radius = 0.1 * distance_relaxed;
      ball->mass = 4/3.0 * M_PI * pow(ball->radius,3);
      ball->contact = false;
    }

  opt_head_lock = true;
  opt_tail_lock = false;
}

void
World::ball_setup_2()
{
  /// Arrange and size balls to form a pendulum.

  // Desired position of first ball.
  //
  pCoor first_ball_pos(13.4,17.8,-9.2);

  // Desired distance between adjacent balls.
  //
  pVect ball_separation(distance_relaxed, 0, 0);  // Points in +x direction.

  for ( int i=0; i<chain_length; i++ )
    {
      Ball* const ball = &balls[i];
      ball->position = first_ball_pos + i * ball_separation;

      ball->velocity = pVect(0,0,0);
      ball->radius = ( i == chain_length - 1 ? 0.6 : 0.3 ) * distance_relaxed;
      ball->mass = 4/3.0 * M_PI * pow(ball->radius,3);
      ball->contact = false;
    }

  opt_head_lock = true;
}
```

```
void
World::ball_setup_3()
{
}

void
World::ball_setup_4()
{
}

void
World::ball_setup_5()
{
}


 /// Advance Simulation State by delta_t Seconds
//
void
World::time_step_cpu(double delta_t)
{
  time_step_count++;

  //
  /// Compute force and update velocity of each ball.
  //
  for ( int i=0; i<chain_length; i++ )
    {
      Ball* const ball = &balls[i];

      // Skip locked balls.
      //
      if ( opt_head_lock && i == 0 || opt_tail_lock && i == chain_length - 1 )
        {
          ball->velocity = pVect(0,0,0);
          continue;
        }

      pVect force(0,0,0);

      // Gravitational Force
      //
      force += ball->mass * gravity_accel;

      // Spring Force from Neighbor Balls
      //
      for ( int direction: { -2, -1, +1, +2 } )
        {
          const int n_idx = i + direction;  // Compute neighbor index.

          // Skip this neighbor if neighbor doesn't exit.
          //
          if ( n_idx < 0 ) continue;

          if ( n_idx >= chain_length ) continue;

          Ball* const neighbor_ball = &balls[n_idx];

          // Construct a normalized (Unit) Vector from ball to neighbor.
          //
          pNorm ball_to_neighbor( ball->position, neighbor_ball->position );

          // Get distance between balls using pNorm member magnitude.
```

```
          //
          const float distance_between_balls = ball_to_neighbor.magnitude;

          // Compute by how much the spring is stretched (positive value)
          // or compressed (negative value).
          //
          const float spring_stretch =
            distance_between_balls - distance_relaxed;

          // Compute the speed of ball towards neighbor_ball.
          //
          pVect delta_v = neighbor_ball->velocity - ball->velocity;
          float delta_s = dot( delta_v, ball_to_neighbor );


          // Determine whether spring is gaining energy (whether its length
          // is getting further from its relaxed length).
          //
          const bool gaining_e = ( delta_s > 0.0 ) == ( spring_stretch > 0 );

          // Use a smaller spring constant when spring is loosing energy,
          // a quick and dirty way of simulating energy loss due to spring
          // friction.
          //
          const float spring_constant =
            gaining_e ? opt_spring_constant : opt_spring_constant * 0.7;

          force += spring_constant * spring_stretch * ball_to_neighbor;
        }

      // Update Velocity
      //
      // This code assumes that force on ball is constant over time
      // step. This is clearly wrong when balls are moving with
      // respect to each other because the springs are changing
      // length. This inaccuracy will make the simulation unstable
      // when spring constant is large for the time step.
      //
      ball->velocity += ( force / ball->mass ) * delta_t;

      // Air Resistance
      //
      const double fs = pow(1+opt_air_resistance,-delta_t);
      ball->velocity *= fs;
    }

  ///
  /// Update Position of Each Ball
  ///

  for ( int i=0; i<chain_length; i++ )
    {
      Ball* const ball = &balls[i];

      // Update Position
      //
      // Assume that velocity is constant.
      //
      ball->position += ball->velocity * delta_t;

      // Possible Collision with Platform
      //

      // Skip if collision impossible.
      //
```

```
      if ( !platform_collision_possible(ball->position) ) continue;
      if ( ball->position.y >= 0 ) continue;

      // Reflect y (vertical) component of velocity, with a reduction
      // due to energy lost in the collision.
      //
      if ( ball->velocity.y < 0 )
        ball->velocity.y = - 0.9 * ball->velocity.y;
    }
}

bool
World::platform_collision_possible(pCoor pos)
{
  // Assuming no motion in x or z axes.
  //
  return pos.x >= platform_xmin && pos.x <= platform_xmax
    && pos.z >= platform_zmin && pos.z <= platform_zmax;
}

 /// External Modifications to State
//
//   These allow the user to play with state while simulation
//   running.

// Move the ball.
//
void Ball::translate(pVect amt) {position += amt;}

// Add velocity to the ball.
//
void Ball::push(pVect amt) {velocity += amt;}

// Set the velocity to zero.
//
void Ball::stop() {velocity = pVect(0,0,0); }

// Set the velocity and rotation (not yet supported) to zero.
//
void Ball::freeze() {velocity = pVect(0,0,0); }



void World::balls_translate(pVect amt,int b){balls[b].translate(amt);}
void World::balls_push(pVect amt,int b){balls[b].push(amt);}
void World::balls_translate(pVect amt)
{ for(int i=0;i<chain_length;i++)balls[i].translate(amt);}
void World::balls_push(pVect amt)
{ for(int i=0;i<chain_length;i++)balls[i].push(amt);}
void World::balls_stop()
{ for(int i=0;i<chain_length;i++)balls[i].stop();}
void World::balls_freeze(){balls_stop();}


void
World::frame_callback()
{
  // This routine called whenever window needs to be updated.

  const double time_now = time_wall_fp();

  if ( !opt_pause || opt_single_frame || opt_single_time_step )
    {
```

```
      /// Advance simulation state.

      // Amount of time since the user saw the last frame.
      //
      const double wall_delta_t = time_now - last_frame_wall_time;

      const double time_step_duration = 0.0001;

      // Compute amount by which to advance simulation state for this frame.
      //
      const double duration =
        opt_single_time_step ? time_step_duration :
        opt_single_frame ? 1/30.0 :
        wall_delta_t;

      const double world_time_target = world_time + duration;

      while ( world_time < world_time_target )
        {
          time_step_cpu(time_step_duration);
          world_time += time_step_duration;
        }

      // Reset these, just in case they were set.
      //
      opt_single_frame = opt_single_time_step = false;
    }

  last_frame_wall_time = time_now;
  render();
}

int
main(int argv, char **argc)
{
  pOpenGL_Helper popengl_helper(argv,argc);
  World world(popengl_helper);
# ifdef __OPTIMIZE__
  glDisable(GL_DEBUG_OUTPUT);
# else
  glEnable(GL_DEBUG_OUTPUT);
# endif
  glDebugMessageControl(GL_DONT_CARE,GL_DONT_CARE,
                        GL_DEBUG_SEVERITY_NOTIFICATION,0,NULL,false);

  popengl_helper.rate_set(30);
  popengl_helper.display_cb_set(world.frame_callback_w,&world);
}
```

```
/// LSU EE 4702-1 (Fall 2018), GPU Programming
//
 ///  Homework 4 -- SOLUTION
//
 ///  Put solution in this file, most or all code in routine |fs_main|.
 //
 //   Assignment: https://www.ece.lsu.edu/koppel/gpup/2018/hw04.pdf

 //   Note: Code in this file is based on the code from
 //         links-shdr-sphere-true.cc.


// Specify version of OpenGL Shading Language.
//
#version 460 compatibility


layout ( binding = 1 ) buffer sr { mat4 sphere_rot[]; };
layout ( binding = 2 ) buffer spr { vec4 sphere_pos_rad[]; };
layout ( binding = 3 ) buffer sc { vec4 sphere_color[]; };

// Use this variable to debug your code. Press 'y' to toggle tryout.x
// and 'Y' to toggle debug_bool.y (between true and false).
//
layout ( location = 3 ) uniform bvec4 tryout;
layout ( location = 2 ) uniform int lighting_options;
layout ( location = 4 ) uniform float tryoutf;
layout ( location = 5 ) uniform bool mirrored;
layout ( location = 6 ) uniform mat4 trans_proj;
layout ( location = 7 ) uniform int opt_n_holes_eqt;
layout ( location = 8 ) uniform int opt_holes;

const int OHO_None = 0, OHO_Holes = 1, OHO_Lenses = 2;


#ifdef _VERTEX_SHADER_

// Interface block for vertex shader output / geometry shader input.
//
out Data_to_GS{ int vertex_id; };

void
vs_main()
{
  vertex_id = gl_VertexID;
}

#endif


#ifdef _GEOMETRY_SHADER_

in Data_to_GS { int vertex_id; } In[1];

out Data_to_FS
{
  flat int vertex_id;
  vec3 vertex_o;
};

// Type of primitive at geometry shader input.
//
layout ( points ) in;
```

```
// Type of primitives emitted geometry shader output.
//
layout ( triangle_strip, max_vertices = 4 ) out;

void
gs_main()
{
  vertex_id = In[0].vertex_id;
  vec4 pos_rad = sphere_pos_rad[vertex_id];
  vec3 ctr_o = pos_rad.xyz;
  float r = pos_rad.w;

  if ( mirrored ) ctr_o.y = -ctr_o.y;

  // Eye location in object space.
  vec3 e_o = vec3(gl_ModelViewMatrixInverse * vec4(0,0,0,1));

  // Vectors from eye to sphere center.
  vec3 ec_o = ctr_o - e_o;  // Eye to Center (of sphere).
  float ec_len = length(ec_o);
  vec3 ne = ec_o / ec_len;

  // Vectors orthogonal to ec_o.
  //
  vec3 atr_o = abs(ec_o);
  int min_idx = atr_o.x < atr_o.y ? ( atr_o.x < atr_o.z ? 0 : 2 )
    : ( atr_o.y < atr_o.z ? 1 : 2 );
  vec3 nx_raw = min_idx == 0 ? vec3(0,-ec_o.z,ec_o.y)
    : min_idx == 1 ? vec3(-ec_o.z,0,ec_o.x) : vec3(-ec_o.y,ec_o.x,0);
  vec3 nx = normalize(nx_raw);
  vec3 ny = cross(ne,nx);

  // Compute center of the limb, lmb_o, the circle formed by the most
  // distant visible parts of the sphere surface.
  float sin_theta = r / ec_len;
  float a = r * sin_theta;
  vec3 lmb_o = ctr_o - a * ne;

  // Compute axes to draw the limb.
  float b = r * sqrt( 1 - sin_theta * sin_theta );
  vec3 ax = b * nx;
  vec3 ay = b * ny;

  mat4 mvp = trans_proj * gl_ModelViewMatrix;

  // Emit a bounding square for the limb.
  for ( int i = -1; i < 2; i += 2 )
    for ( int j = -1; j < 2; j += 2 )
      {
        vertex_o = lmb_o + ax * i + ay * j;
        gl_Position = mvp * vec4(vertex_o,1);
        EmitVertex();
      }
  EndPrimitive();
}

#endif


#ifdef _FRAGMENT_SHADER_

uniform sampler2D tex_unit_0;

in Data_to_FS
{
```

```
  flat int vertex_id;
  vec3 vertex_o;
};

vec4 generic_lighting(vec4 vertex_e, vec4 color, vec3 normal_e);


void
fs_main()
{
  /// Put Homework 4 solution in this file.

  vec4 pos_rad = sphere_pos_rad[vertex_id];

  // Center of sphere in original object-space coördinates (oo) and possibly
  // reflected (o).
  vec3 ctr_oo = pos_rad.xyz;
  vec3 ctr_o = mirrored ? ctr_oo * vec3(1,-1,1) : ctr_oo;

  float rsq = pos_rad.w * pos_rad.w;

  // Eye location in object-space coördinates.
  vec3 e_o = gl_ModelViewMatrixInverse[3].xyz;

  // Prepare to compute intersection of ray from eye to through fragment with
  // sphere. That intersection is the point on the sphere corresponding
  // to this fragment.
  //
  vec3 ef = vertex_o - e_o;   // Eye to fragment.
  vec3 ce = e_o-ctr_o;
  float qfa = dot(ef,ef);
  float qfb = 2 * dot(ef,ce);
  float qfc = dot(ce,ce) - rsq;
  float discr = qfb*qfb - 4 * qfa * qfc;

  // If outside the limb, return.
  if ( discr < 0 ) discard;

  const float pi = 3.14159265359;
  const float two_pi = 2 * pi;

  const bool holes = opt_holes == OHO_Holes;    // If true sphere has holes
  const bool lenses = opt_holes == OHO_Lenses;
  const bool solid = !holes && !lenses;         // If true show complete sphere.


  /// SOLUTION  -- Organizing Front Surface / Back Surface Computation
  //
  //  Since this fragment can be on the front or the back of the sphere ..
  //  .. put code for finding the sphere coördinates in a loop.
  //
  //  First compute the coördinates of the front surface of the sphere ..
  //  .. if that point is visible (e.g., not in a hole), show it, ..
  //  .. otherwise find the coördinates of the back surface ..
  //  .. if that point is visible (e.g., not in a hole), show it, ..
  //  .. otherwise discard the fragment (don't use any coördinates).
  //
  //  To avoid code duplication ..
  //  .. put the sphere-coördinates computation in a loop.
  //

  vec4 sur_o, sur_oo;  // Global surface coördinates (front or back) of sphere.
  vec3 normal_oo, sur_l;
  bool front;          // If true, coördinates are of front of sphere.
  float theta, eta;    // Local angular coördinates of sphere.
```

```
/// SOLUTION
//
//  Use loop to iterate over two possibilities ..
//  .. front of sphere (dir=-1) and back of sphere (dir=1).
//
for ( float dir = -1;  dir <= 1;  dir+=2 )
  {
    front = dir == -1;

    // Finish computing coordinate of point for this fragment.
    //
    float t = ( -qfb + dir * sqrt( discr ) ) / ( 2 * qfa );

    // Compute true sphere surface coordinate.
    //
    sur_o = vec4(e_o + t * ef, 1);

    // Compute possibly reflected sphere surface coordinate.
    //
    sur_oo = mirrored ? sur_o * vec4(1,-1,1,1) : sur_o;
    normal_oo = normalize(sur_oo.xyz - ctr_oo);

    /// SOLUTION
    //
    //  Reverse normal if this is the back surface of the sphere.
    //
    if ( !front ) normal_oo = -normal_oo;

    // Use sphere-local coordinates to compute texture coordinates.
    //
    sur_l = normalize(mat3(sphere_rot[vertex_id]) * normal_oo);
    theta = atan(sur_l.x,sur_l.z);
    eta = acos(sur_l.y);

    /// SOLUTION
    //
    //  If this is just a normal sphere, we are finished.
    //
    if ( solid ) break;

    const float hole_frac = 0.8;
    const float radians_per_hole_eq = two_pi / opt_n_holes_eqt;
    const float hole_radius = 0.5 * hole_frac * radians_per_hole_eq;

    //
    /// SOLUTION -- Find coördinates of nearest hole.
    //
    //
    // Round surface eta coördinate to that of nearest hole.
    //
    float eta_hole = round( eta / radians_per_hole_eq ) * radians_per_hole_eq;
    //
    // Find distance around sphere at this latitude (value of eta).
    //
    float r = sin(eta_hole);
    //
    // Find number of holes that will fit around sphere at this latitude.
    //
    const float n_holes = floor( two_pi * r / radians_per_hole_eq );
    //
    // Don't try to find a hole if there are none this close to the pole.
    //
    if ( n_holes < 1 )
      {
```

```
          if ( holes ) break;
          if ( dir == 1 ) discard;
        }
      //
      // Round surface theta coördinate to that of nearest hole.
      //
      const float radians_per_hole = two_pi / n_holes;
      float theta_hole = round( theta / radians_per_hole ) * radians_per_hole;
      //
      // Compute local coördinates of center of hole.
      //
      vec3 hole_dir_l =
        vec3( r * sin(theta_hole), cos(eta_hole), r * cos(theta_hole) );

      /// SOLUTION -- Determine distance to hole and take appropriate action.
      //
      // Find distance from center of hole to sphere surface point ..
      //
      float dist = distance(hole_dir_l,sur_l);
      //
      // .. and check whether surface point is in hole.
      //
      bool found_hole = dist < hole_radius;
      //
      // If this part of the sphere is visible, break.
      //
      if ( lenses && found_hole || holes && !found_hole ) break;
      //
      // If we are already at the back of the sphere, don't show anything.
      //
      if ( !front ) discard;
    }

  vec3 normal_ee = normalize(gl_NormalMatrix * normal_oo);
  vec4 sur_ee = gl_ModelViewMatrix * sur_oo;

  // Compute clip-space depth. Take care so that compiler avoids full
  // matrix / vector multiplication.
  vec4 sur_e = gl_ModelViewMatrix * sur_o;
  vec4 sur_c = trans_proj * vec4(0,0,2*sur_e.z,1);
  gl_FragDepth = sur_c.z / sur_c.w;

  vec2 tcoord = vec2( ( 1.5 * pi + theta ) / two_pi, eta / pi );

  // Get filtered texel.
  //
  vec4 texel = front ? texture(tex_unit_0,tcoord) : vec4(1);

  vec4 color = sphere_color[vertex_id];
  vec4 color2 = front ? color : vec4(0.3,0,0,1);

  // Multiply filtered texel color with lighted color of fragment.
  //
  gl_FragColor = texel * generic_lighting(sur_ee, color2, normal_ee);
}


///
/// Routine used by Either Vertex or Fragment Shader
///

vec4
generic_lighting(vec4 vertex_e, vec4 color, vec3 normal_e)
{
```

```
  // Return lighted color of vertex_e.
  //

  vec3 nspc_color = color.rgb * gl_LightModel.ambient.rgb;
  vec3 spec_color = vec3(0);

  for ( int i=0; i<2; i++ )
    {
      if ( ( lighting_options & ( 1 << i ) ) == 0 ) continue;
      vec4 light_pos = gl_LightSource[i].position;
      vec3 v_vtx_light = light_pos.xyz - vertex_e.xyz;
      float dist = length(v_vtx_light);
      float dist_vl_inv = 1.0 / dist;
      vec3 v_vtx_l_n = v_vtx_light * dist_vl_inv;

      float d_n_vl = dot(normal_e, v_vtx_l_n);
      float phase_light = max(0,gl_FrontFacing != mirrored ? d_n_vl : -d_n_vl );

      vec3 ambient_light = gl_LightSource[i].ambient.rgb;
      vec3 diffuse_light = gl_LightSource[i].diffuse.rgb;
      float distsq = dist * dist;
      float atten_inv =
        gl_LightSource[i].constantAttenuation +
        gl_LightSource[i].linearAttenuation * dist +
        gl_LightSource[i].quadraticAttenuation * distsq;
      vec3 lighted_color =
        color.rgb
        * ( ambient_light + phase_light * diffuse_light ) / atten_inv;
      nspc_color += lighted_color;

      vec3 h = normalize( v_vtx_l_n - normalize(vertex_e.xyz) );

      spec_color +=
        pow(max(0.0,dot(normal_e,h)),16)
        * color.rgb
        * gl_LightSource[i].specular.rgb / atten_inv;
    }

  return vec4(nspc_color+spec_color,1);
}


#endif
```

# 25 Fall 2017 Solutions

```
/// LSU EE 4702-1 (Fall 2017), GPU Programming
//
 /// Homework 1 -- SOLUTION
 //
 //  For solution search for SOLUTION in this file.

 /// Instructions
 //
 //  Read the assignment: http://www.ece.lsu.edu/koppel/gpup/2017/hw01.pdf


/// Purpose
//
//   Demonstrate simulation of point masses connected by springs.


/// What Code Does

// Simulates balls connected by springs over a platform. Balls and
// springs can be initialized in different arrangements (called
// scenes). Currently scene 1 is a simple string of beads. The
// platform consists of tiles, some are purple-tinted mirrors (showing
// a reflection of the ball), the others show the course syllabus.



///  Keyboard Commands
 //
 /// Object (Eye, Light, Ball) Location or Push
 //   Arrows, 'Page Up', 'Page Down'
 //        Move object or push ball, depending on mode.
 //        With shift key pressed, motion is 5x faster.
 //   'e': Move eye.
 //   'l': Move light.
 //   'b': Move head (first) ball. (Change position but not velocity.)
 //   'B': Push head ball. (Add velocity.)
 //
 /// Eye Direction, View, and Screenshot Options
 //
 //  'Home', 'End', 'Delete', 'Insert'
 //         Change the eye direction.
 //         Home rotates eye direction up, End rotates eye
 //         down, Delete rotates eye to the left, Insert rotates eye
 //         to the right.
 //         The eye direction vector is displayed in the upper left.
 //
 //  'Ctrl' '+'  or  'Ctrl' '=',  and  'Ctrl' '-'  or  'Ctrl' '_',
 //         Increase and decrease green text size.
 //  'F12'  Write screenshot to file.

 /// Simulation Options
 //  (Also see variables below.)
 //
 //  'w'    Twirl balls around axis formed by head and tail. (Prob 2 soln).
 //  '1'    Set up scene 1.
 //  '2'    Set up scene 2.
 //  '3'    Set up scene 3.
 //  'p'    Pause simulation. (Press again to resume.)
 //  ' '    (Space bar.) Advance simulation by 1/30 second.
 //  'S- '  (Shift-space bar.) Advance simulation by one time step.
 //  'h'    Freeze position of first (head) ball. (Press again to release.)
 //  't'    Freeze position of last (tail) ball. (Press again to release.)
```

```
//  's'    Stop balls.
//  'g'    Turn gravity on and off.
//  'y'    Toggle value of opt_tryout1. Intended for experiments and debugging.
//  'Y'    Toggle value of opt_tryout2. Intended for experiments and debugging.

/// Variables
//   Selected program variables can be modified using the keyboard.
//   Use "Tab" to cycle through the variable to be modified, the
//   name of the variable is displayed next to "VAR" on the bottom
//   line of green text.

//  'Tab' Cycle to next variable.
//  '`'   Cycle to previous variable.
//  '+'   Increase variable value.
//  '-'   Decrease variable value.
//
//  VAR Spring Constant - Set spring constant.
//  VAR Time Step Duration - Set physics time step.
//  VAR Air Resistance - Set air resistance.
//  VAR Light Intensity - The light intensity.
//  VAR Gravity - Gravitational acceleration. (Turn on/off using 'g'.)


#define GL_GLEXT_PROTOTYPES
#define GLX_GLXEXT_PROTOTYPES

#include <GL/gl.h>
#include <GL/glext.h>
#include <GL/glx.h>
#include <GL/glxext.h>
#include <GL/glu.h>
#include <GL/freeglut.h>

#include <gp/util.h>
#include <gp/glextfuncs.h>
#include <gp/coord.h>
#include <gp/shader.h>
#include <gp/pstring.h>
#include <gp/misc.h>
#include <gp/gl-buffer.h>
#include <gp/texture-util.h>
#include <gp/colors.h>

#include "util-containers.h"
#include "shapes.h"


///
/// Main Data Structures
///
//
// class World: All data about scene.


class World;


// Object Holding Ball State
//
class Ball {
public:
  Ball():velocity(pVect(0,0,0)),locked(false),persistant(false),
         color(color_lsu_spirit_gold),contact(false){};
  pCoor position;
```

```
  pVect velocity;

  float mass;
  float mass_min; // Mass below which simulation is unstable.
  float radius;

  bool locked;
  bool persistant;  // Don't delete it when resetting scenes.

  pVect force;
  pColor color;
  bool contact;                 // When true, ball rendered in gray.
  float spring_constant_sum;    // Used to compute minimum mass.

  void push(pVect amt);
  void translate(pVect amt);
  void stop();
  void freeze();
};

class Link {
public:
  Link(Ball *b1, Ball *b2):ball1(b1),ball2(b2),
     distance_relaxed(pDistance(b1->position,b2->position)),
     color(color_lsu_spirit_purple){}
  Link(Ball *b1, Ball *b2, pColor colorp):ball1(b1),ball2(b2),
     distance_relaxed(pDistance(b1->position,b2->position)),
     color(colorp){}
  Ball* const ball1;
  Ball* const ball2;
  float distance_relaxed;
  pColor color;
};

// Declare containers and iterators for Balls and Links.
// (See util_container.h.)
//
typedef pVectorI<Link> Links;
typedef pVectorI<Ball> Balls;
typedef pVector<pCoor> pCoors;
typedef pVector<pVect> pVects;

#include "hw01-graphics.cc"


void
World::init()
{
  chain_length = 14;

  opt_height = 1;
  variable_control.insert_linear(opt_height,"Volcano height (opt_height)",0.04);

  opt_e = 0.3;
  variable_control.insert(opt_e,"Volcano exponent (opt_e)");

  opt_layers = 8;
  variable_control.insert(opt_layers,"Volcano num layers (opt_layers)",1,2);

  opt_time_step_duration = 0.0003;
  //  variable_control.insert(opt_time_step_duration,"Time Step Duration");

  distance_relaxed = 15.0 / chain_length;
```

```
  opt_spring_constant = 15000;
  variable_control.insert(opt_spring_constant,"Spring Constant");

  opt_gravity_accel = 9.8;
  opt_gravity = true;
  gravity_accel = pVect(0,-opt_gravity_accel,0);
  variable_control.insert(opt_gravity_accel,"Gravity");

  opt_air_resistance = 0.04;
  //  variable_control.insert(opt_air_resistance,"Air Resistance");

  world_time = 0;
  time_step_count = 0;
  last_frame_wall_time = time_wall_fp();
  frame_timer.work_unit_set("Steps / s");

  init_graphics();

  pCoor marker_pos(13.4,15.8,-9.2);
  balls += marker_red = make_marker(marker_pos,color_red);
  marker_pos.y += 2 * marker_red->radius;
  balls += marker_blue = make_marker(marker_pos,color_blue);
  marker_pos.y += 2 * marker_red->radius;
  balls += marker_khaki = make_marker(marker_pos,color_khaki);

  ball_setup_1();
  lock_update();
}

Ball*
World::make_marker(pCoor position, pColor color)
{
  Ball* const ball = new Ball;
  ball->position = position;
  ball->locked = true;
  ball->velocity = pVect(0,0,0);
  ball->radius = 0.2;
  ball->mass = 0;
  ball->contact = false;
  ball->color = color;
  ball->persistant = true;
  return ball;
}

void
World::lock_update()
{
  // This routine called when options like opt_head_lock might have
  // changed.

  // Update locked status.
  //
  if ( head_ball ) head_ball->locked = opt_head_lock;
  if ( tail_ball ) tail_ball->locked = opt_tail_lock;

  // Re-compute minimum mass needed for stability.
  //
  for ( Ball *ball: balls ) ball->spring_constant_sum = 0;
  const double dtis = pow( opt_time_step_duration, 2 );
  for ( Link *link: links )
    {
      Ball* const b1 = link->ball1;
      Ball* const b2 = link->ball2;
      b1->spring_constant_sum += opt_spring_constant;
      b2->spring_constant_sum += opt_spring_constant;
```

```
    }
  for ( Ball *ball: balls )
    ball->mass_min = ball->spring_constant_sum * dtis;
}

void
World::render_p0()
{
  /// Use this code for minor experiments, or leave it unchanged.

  // Make sure that this scene set the thing_1 positions.
  if ( !thing_1_apex ) return;

  pCoor apex = thing_1_apex->position;
  pCoor base = thing_1_base->position;
  vector<pCoor> ring;
  for ( Ball *b: thing_1_ring ) ring.push_back(b->position);

  glColorMaterial(GL_FRONT,GL_AMBIENT_AND_DIFFUSE);
  glMaterialfv(GL_BACK,GL_AMBIENT_AND_DIFFUSE,color_gray);

  glBegin(GL_TRIANGLES);

  for ( pCoor circ: ring )
    {
      pNorm norm = cross(apex,circ,base);

      glColor3fv(color_red);

      glNormal3fv(norm);
      glVertex3fv(apex);
      glVertex3fv(base);
      glVertex3fv(circ);

    }
  glEnd();
}

void
World::render_p1()
{
  /// Put Problem 1 solution in this routine.

  // Make sure that this scene set the thing_1 positions.
  if ( !thing_1_apex ) return;

  pCoor apex = thing_1_apex->position;
  pCoor base = thing_1_base->position;
  vector<pCoor> ring;
  for ( Ball *b: thing_1_ring ) ring.push_back(b->position);

  glColorMaterial(GL_FRONT,GL_AMBIENT_AND_DIFFUSE);
  glMaterialfv(GL_BACK,GL_AMBIENT_AND_DIFFUSE,color_gray);

  /// SOLUTION:
  //  Remove glBegin.

  for ( int i=0; i<3; i++ )
    {
      pCoor circ = ring[i];

      pNorm norm = cross(apex,circ,base);

      // Compute the center of the triangle ..
      //
```

```
      pVect ab(apex,base);
      pVect ac(apex,circ);
      pCoor mid = apex + ( ab + ac ) / 3;

      // .. and drop a marker there.
      //
      switch ( i ) {
        case 0: marker_red->position = mid; break;
        case 0: marker_blue->position = mid; break;
        case 0: marker_khaki->position = mid; break;
      }


      glColor3fv(color_red);

      /// SOLUTION
      //
      //  Compute vertices of triangle hole.

      pNorm ma(mid,apex);
      pNorm mb(mid,base);
      pNorm mc(mid,circ);
      pCoor a2 = mid + 0.5 * ma.magnitude * ma;
      pCoor b2 = mid + 0.5 * mb.magnitude * mb;
      pCoor c2 = mid + 0.5 * mc.magnitude * mc;

      // Draw triangle strip.

      glBegin(GL_TRIANGLE_STRIP);

      glNormal3fv(norm);
      glVertex3fv(a2);
      glVertex3fv(apex);
      glVertex3fv(b2);
      glVertex3fv(base);
      glVertex3fv(c2);
      glVertex3fv(circ);
      glVertex3fv(a2);
      glVertex3fv(apex);

      glEnd();

    }
}

void
World::render_p2()
{
  // Make sure that this scene set the thing_1 positions.
  if ( !thing_1_apex ) return;

  pCoor apex = thing_1_apex->position;
  pCoor base = thing_1_base->position;
  vector<pCoor> ring;
  for ( Ball *b: thing_1_ring ) ring.push_back(b->position);

  glColorMaterial(GL_FRONT,GL_AMBIENT_AND_DIFFUSE);
  glMaterialfv(GL_BACK,GL_AMBIENT_AND_DIFFUSE,color_gray);

  for ( int i=0; i<3; i++ )
    {
      pCoor circ = ring[i];

      pNorm norm = cross(apex,circ,base);
```

```
      // Compute the center of the triangle ..
      //
      pVect ab(apex,base);
      pVect ac(apex,circ);
      pCoor mid = apex + ( ab + ac ) / 3;

      // .. and drop a marker there.
      //
      switch ( i ) {
      case 0: marker_red->position = mid; break;
      case 1: marker_blue->position = mid; break;
      case 2: marker_khaki->position = mid; break;
      }

      glColor3fv(color_red);

      /// SOLUTION

      // Compute vectors from vertices to triangle center (mid).
      //
      pVect am(apex,mid);
      pVect bm(base,mid);
      pVect cm(circ,mid);

      float t_stop = 0.9;
      float delta_t = t_stop / opt_layers;

      // Use an array to conveniently obtain adjacent vertices of triangle
      // using loop iterator, j.
      //
      pCoor coors[4] = {apex,base,circ,apex};

      // Iterate over sides of volcano.
      //
      for ( int j=0; j<3; j++ )
        {
          pCoor p0 = coors[j];
          pCoor q0 = coors[j+1];
          pVect pm(p0,mid);
          pVect qm(q0,mid);

          // Use a triangle strip for the path up the volcano.

          glBegin(GL_TRIANGLE_STRIP);
          glColor3fv(color_red);

          for ( int k = 0;  k <= opt_layers;  k++ )
            {
              float t = k * delta_t;
              pVect up = t * opt_height * norm;
              float frac = pow(t,opt_e);

              // Compute points on lines going up volcano.
              //
              pCoor p = p0 + up + frac * pm;
              pCoor q = q0 + up + frac * qm;

              // Compute derivative of parametric line function,
              // and evaluate it at each point to get vectors along surface.
              //
              pVect dupdt = opt_height * norm;
              float dfracdt = t ? opt_e * pow(t,opt_e-1) : 0;
              pVect dpdt = dupdt + dfracdt * pm;
              pVect dqdt = dupdt + dfracdt * qm;
```

```
              // Take cross product to find normal.
              //
              pNorm n = cross(dpdt,dqdt);

              glNormal3fv(n);
              glVertex3fv(p);
              glVertex3fv(q);
            }

          glEnd();
        }
    }
}

void
World::objects_erase()
{
  thing_1_apex = NULL;
  thing_1_ring.clear();
  thing_1_pseudo.clear();
  Balls save;
  for ( auto b: balls ) if ( b->persistant ) save += b; else delete b;
  balls = move(save);
  links.erase();
  chain_starts.clear();
}

///
/// Physical Simulation Code
///

 /// Initialize Simulation
//

void
World::ball_setup_1()
{
  // Arrange and size balls to form a cable suspended between two fixed points.

  pCoor first_pos(7.2,7.8,-20.2);

  pVect head_to_tail(25,0,3.33333);
  const int len = 5;
  pVect delta_pos = head_to_tail / len;

  // Remove objects from the simulated objects lists, balls and links.
  // The delete operator is used on objects in the lists.
  //
  objects_erase();

  Balls persist = move(balls);
  pCoor marker_pos = first_pos + head_to_tail + pVect(0,1,0);
  for ( Ball *marker : persist )
    { marker->position = marker_pos; marker_pos.y += 2 * marker_red->radius; }

  auto new_ball = [&](pCoor pos)
    {
      // Construct a new ball and add it to the simulated objects list (balls).
      //
      Ball* const ball = balls += new Ball;

      // Initialize position and other information.
      //
      ball->position = pos;
```

```
      ball->locked = false;
      ball->velocity = pVect(0,0,0);
      ball->radius = 0.3;
      ball->mass = 4/3.0 * M_PI * pow(ball->radius,3);
      ball->contact = false;
      return ball;
    };

  for ( int i=0; i<=len; i++ )
    {
      // Construct a new ball and add it to the simulated objects list (balls).
      //
      Ball* const ball = new_ball(first_pos + i * delta_pos);

      // If it's not the first ball link it to the previous ball.
      if ( i > 0 ) links += new Link( ball, balls[i-1] );
    }

  // The balls pointed to by head_ball and tail_ball can be manipulated
  // using the user interface (by pressing 'h' or 't', for example).
  // Set these variables.
  //
  head_ball = balls[0];
  tail_ball = balls[balls-1];

  const int obj_idx = max(1,len - 1 - len / 5);
  Ball* const b0 = balls[obj_idx];
  Ball* const b1 = balls[obj_idx+1];
  pCoor pc = b1->position - 0.2 * delta_pos;
  thing_1_apex = b0;
  thing_1_base = b1;
  thing_1_base->color = color_black;
  pVect ayraw(0,1,0);
  pNorm ax = delta_pos;
  pNorm az = cross(ax,ayraw);
  pNorm ay = cross(az,ax);
  const double rrad = 4;
  const int n = 3;
  Balls ring;
  for ( int i=0; i<n; i++ )
    {
      const double theta = i * 2 * M_PI / n;
      Ball* const ball =
        new_ball( pc + rrad * cos(theta) * ay + rrad * sin(theta) * az );
      ball->color = color_lime_green;
      ring += ball;
      links += new Link( b0, ball );
      links += new Link( b1, ball );
    }
  thing_1_ring = ring;
  for ( int i=0; i<n; i++ )
    links += new Link( ring[i], ring[(i+1)%n] );

  opt_head_lock = true;    // Head ball will be frozen in space.
  opt_tail_lock = true;    // Tail ball too.

  balls += persist;
}


void
World::ball_setup_2()
{
  // Arrange and size balls to form a pendulum.
```

```
  pCoor first_pos(5.1,17.8,-13.1);
  pNorm delta_dir = pVect(1,-1,0);
  pVect delta_pos = distance_relaxed * delta_dir;

  // Remove objects from the simulated objects lists, balls and links.
  // The delete operator is used on objects in the lists.
  //
  objects_erase();
  Balls persist = move(balls);

  int i;
  for ( i=0; i<chain_length/2; i++ )
    {
      // Construct a new ball and add it to the simulated objects list (balls).
      //
      Ball* const ball = balls += new Ball;

      // Initialize position and other information.
      //
      ball->position = first_pos + i * delta_pos;

      // If it's not the first ball link it to the previous ball.
      if ( i > 0 ) links += new Link( ball, balls[i-1] );
    }

  int sides = 4;
  pNorm hx = delta_pos.x
    ? pVect(-delta_pos.y,delta_pos.x,0) : pVect(0, -delta_pos.z,delta_pos.y);
  pNorm hy = cross(delta_pos,hx);
  Ball* const l1 = balls.back();
  for ( int i=0; i<sides; i++ )
    {
      const double angle = i * 2 * M_PI / sides;
      Ball* const ball = balls += new Ball;
      chain_starts += ball;
      ball->position =
        l1->position + delta_pos
        + 2 * distance_relaxed * cos(angle) * hx
        + 2 * distance_relaxed * sin(angle) * hy;
      links += new Link( ball, l1 );
      if ( i > 0 )
        links += new Link( ball, chain_starts[i-1] );
      if ( i > 1 && i == sides - 1 )
        links += new Link( ball, chain_starts[0] );
      if ( sides > 3 && i > sides/2 )
        links += new Link( ball, chain_starts[i-sides/2] );
    }

  for ( int i=1; i<chain_length/2; i++ )
    for ( int j=0; j<sides; j++ )
      {
      // Construct a new ball and add it to the simulated objects list (balls).
      //
      Ball* const ball = balls += new Ball;

      // Initialize position and other information.
      //
      ball->position = chain_starts[j]->position + i * delta_pos;

      // If it's not the first ball link it to the previous ball.
      links += new Link( ball, balls[balls-sides-1] );
    }

  for ( Ball* ball : balls )
    {
```

```
      ball->locked = false;
      ball->velocity = pVect(0,0,0);
      ball->radius = 0.3;
      ball->mass = 4/3.0 * M_PI * pow(ball->radius,3);
      ball->contact = false;
    }

  // The balls pointed to by head_ball and tail_ball can be manipulated
  // using the user interface (by pressing 'h' or 't', for example).
  // Set these variables.
  //
  head_ball = balls[0];
  tail_ball = balls[balls-1];

  opt_head_lock = true;    // Head ball will be frozen in space.
  opt_tail_lock = false;   // Tail ball can move freely.

  balls += persist;

}


void
World::ball_setup_3()
{
}

void
World::ball_setup_4()
{
}


void
World::ball_setup_5()
{
}


 /// Advance Simulation State by delta_t Seconds
//
void
World::time_step_cpu(double delta_t)
{
  time_step_count++;

  for ( Ball *ball: balls )
    ball->force = ball->mass * gravity_accel;

  for ( Link *link: links )
    {
      // Spring Force from Neighbor Balls
      //
      Ball* const ball1 = link->ball1;
      Ball* const ball2 = link->ball2;

      // Construct a normalized (Unit) Vector from ball to neighbor.
      //
      pNorm ball_to_neighbor(ball1->position,ball2->position);

      const float distance_between_balls = ball_to_neighbor.magnitude;

      // Compute the speed of ball towards neighbor_ball.
      //
```

```
    pVect delta_v = ball2->velocity - ball1->velocity;
    float delta_s = dot( delta_v, ball_to_neighbor );

    // Compute by how much the spring is stretched (positive value)
    // or compressed (negative value).
    //
    const float spring_stretch =
      distance_between_balls - link->distance_relaxed;

    // Determine whether spring is gaining energy (whether its length
    // is getting further from its relaxed length).
    //
    const bool gaining_e = ( delta_s > 0.0 ) == ( spring_stretch > 0 );

    // Use a smaller spring constant when spring is loosing energy,
    // a quick and dirty way of simulating energy loss due to spring
    // friction.
    //
    const float spring_constant =
      gaining_e ? opt_spring_constant : opt_spring_constant * 0.7;

    ball1->force += spring_constant * spring_stretch * ball_to_neighbor;
    ball2->force -= spring_constant * spring_stretch * ball_to_neighbor;
  }

///
/// Update Position of Each Ball
///

for ( Ball *ball: balls )
  {
    if ( ball->locked )
      {
        ball->velocity = pVect(0,0,0);
        continue;
      }

    // Update Velocity
    //
    // This code assumes that force on ball is constant over time
    // step. This is clearly wrong when balls are moving with
    // respect to each other because the springs are changing
    // length. This inaccuracy will make the simulation unstable
    // when spring constant is large for the time step.
    //
    float mass = std::max(ball->mass, ball->mass_min );
    ball->velocity += ( ball->force / mass ) * delta_t;

    // Air Resistance
    //
    const double fs = pow(1+opt_air_resistance,-delta_t);
    ball->velocity *= fs;

    // Update Position
    //
    // Assume that velocity is constant.
    //
    ball->position += ball->velocity * delta_t;

    // Possible Collision with Platform
    //

    // Skip if collision impossible.
    //
    if ( !platform_collision_possible(ball->position) ) continue;
```

```
      if ( ball->position.y >= 0 ) continue;

      // Snap ball position to surface.
      //
      ball->position.y = 0;

      // Reflect y (vertical) component of velocity, with a reduction
      // due to energy lost in the collision.
      //
      if ( ball->velocity.y < 0 )
        ball->velocity.y = - 0.9 * ball->velocity.y;
    }
}

bool
World::platform_collision_possible(pCoor pos)
{
  // Assuming no motion in x or z axes.
  //
  return pos.x >= platform_xmin && pos.x <= platform_xmax
    && pos.z >= platform_zmin && pos.z <= platform_zmax;
}

 /// External Modifications to State
//
//   These allow the user to play with state while simulation
//   running.

// Move the ball.
//
void Ball::translate(pVect amt) {position += amt;}

// Add velocity to the ball.
//
void Ball::push(pVect amt) {velocity += amt;}

// Set the velocity to zero.
//
void Ball::stop() {velocity = pVect(0,0,0); }

// Set the velocity and rotation (not yet supported) to zero.
//
void Ball::freeze() {velocity = pVect(0,0,0); }



void World::balls_translate(pVect amt,int b){head_ball->translate(amt);}
void World::balls_push(pVect amt,int b){head_ball->push(amt);}
void World::balls_translate(pVect amt)
{ for ( Ball *ball: balls ) ball->translate(amt); }
void World::balls_push(pVect amt)
{ for ( Ball *ball: balls ) ball->push(amt); }
void World::balls_stop()
{ for ( Ball *ball: balls ) ball->stop(); }
void World::balls_freeze(){balls_stop();}


void
World::frame_callback()
{
  // This routine called whenever window needs to be updated.

  const double time_now = time_wall_fp();
```

```
  if ( !opt_pause || opt_single_frame || opt_single_time_step )
    {
      /// Advance simulation state.

      // Amount of time since the user saw the last frame.
      //
      const double wall_delta_t = time_now - last_frame_wall_time;

      // Compute amount by which to advance simulation state for this frame.
      //
      const double duration =
        opt_single_time_step ? opt_time_step_duration :
        opt_single_frame ? 1/30.0 :
        wall_delta_t;

      const double world_time_target = world_time + duration;

      while ( world_time < world_time_target )
        {
          time_step_cpu(opt_time_step_duration);
          world_time += opt_time_step_duration;
        }

      // Reset these, just in case they were set.
      //
      opt_single_frame = opt_single_time_step = false;
    }

  last_frame_wall_time = time_now;
  render();
}

int
main(int argv, char **argc)
{
  pOpenGL_Helper popengl_helper(argv,argc);
  World world(popengl_helper);

  popengl_helper.rate_set(30);
  popengl_helper.display_cb_set(world.frame_callback_w,&world);
}
```

```
/// LSU EE 4702-1 (Fall 2017), GPU Programming
//
 /// Homework 2 -- SOLUTION
 //


 /// Instructions
 //
 //  Read the assignment: http://www.ece.lsu.edu/koppel/gpup/2017/hw02.pdf

/// What Code Does

// Simulates balls connected by springs over a platform. Balls and
// springs can be initialized in different arrangements (called
// scenes). Currently scene 1 is a simple string of beads, and scenes
// 2, 3, and 4 are trusses. The platform consists of tiles, some are
// purple-tinted mirrors (showing a reflection of the ball), the
// others show the course syllabus.



///  Keyboard Commands
 //
 /// Object (Eye, Light, Ball) Location or Push
 //   Arrows, Page Up, Page Down
 //        Move object or push ball, depending on mode.
 //        With shift key pressed, motion is 5x faster.
 //        With control key pressed, motion is 5x slower.
 //   'e': Move eye.
 //   'l': Move light.
 //   'b': Move head (first) ball. (Change position but not velocity.)
 //   'B': Push head ball. (Add velocity.)
 //
 /// Eye Direction
 //   Home, End, Delete, Insert
 //   Turn the eye direction.
 //   Home should rotate eye direction up, End should rotate eye
 //   down, Delete should rotate eye left, Insert should rotate eye
 //   right.  The eye direction vector is displayed in the upper left.

 /// Simulation Options
 //  (Also see variables below.)
 //
 //  'w'    Twirl balls around axis formed by head and tail. (Prob 2 soln).
 //  '1'    Set up scene 1.
 //  '2'    Set up scene 2.
 //  '3'    Set up scene 3.
 //  '4'    Set up scene 4.
 //  'p'    Pause simulation. (Press again to resume.)
 //  ' '    (Space bar.) Advance simulation by 1/30 second.
 //  'S- '  (Shift-space bar.) Advance simulation by one time step.
 //  'h'    Freeze position of first (head) ball. (Press again to release.)
 //  't'    Freeze position of last (tail) ball. (Press again to release.)
 //  's'    Stop balls.
 //  'g'    Turn gravity on and off.
 //  'y'    Toggle value of opt_tryout1. Intended for experiments and debugging.
 //  'Y'    Toggle value of opt_tryout2. Intended for experiments and debugging.
 //  'C-='  (Ctrl =) Increase text size.
 //  'C--'  (Ctrl -) Decrease text size.
 //  'F12'  Write screenshot to file.

 /// Variables
```

```
//   Selected program variables can be modified using the keyboard.
//   Use "Tab" to cycle through the variable to be modified, the
//   name of the variable is displayed next to "VAR" on the bottom
//   line of green text.

//  'Tab' Cycle to next variable.
//  '`'   Cycle to previous variable.
//  '+'   Increase variable value.
//  '-'   Decrease variable value.
//
//  VAR Spring Constant - Set spring constant.


#define GL_GLEXT_PROTOTYPES
#define GLX_GLXEXT_PROTOTYPES

#include <GL/gl.h>
#include <GL/glext.h>
#include <GL/glx.h>
#include <GL/glxext.h>
#include <GL/glu.h>
#include <GL/freeglut.h>
#include <cuda_runtime.h>

#include <gp/util.h>
#include <gp/glextfuncs.h>
#include <gp/coord.h>
#include <gp/shader.h>
#include <gp/pstring.h>
#include <gp/misc.h>
#include <gp/gl-buffer.h>
#include <gp/texture-util.h>
#include <gp/colors.h>

#include <gp/util-containers.h>
#include <gp/coord-containers.h>
#include <gp/cuda-gpuinfo.h>
#include "shapes.h"
#include "links.cuh"

///
/// Main Data Structures
///
//
// class World: All data about scene.


class World;

enum GPU_Physics_Method { GP_cpu, GP_cuda, GP_ENUM_SIZE };
const char* const gpu_physics_method_str[] = { "CPU", "CUDA" };

// Object Holding Ball State
//
class Ball {
public:
  Ball():velocity(pVect(0,0,0)),omega(pVect(0,0,0)),
         density(1.00746),
         fdt_to_do(0),
         locked(false),
         color(color_lsu_spirit_gold),contact(false)
         {
           orientation_set(pVect(0,1,0),0);
         }
```

```
  pCoor position;
  pVect velocity;
  pQuat orientation;
  pMatrix3x3p omatrix;
  pVect omega;                  // Spin rate and axis.

  int idx; // Position in balls_pos_rad;

  float mass;
  float mass_min; // Mass below which simulation is unstable.
  float radius;
  float radius_sq, radius_inv;  // Pre-computed based on radius.
  float density;
  float mass_inv;
  float fdt_to_do; // Radius / moment of inertia.

  bool locked;

  pVect force;
  pVect torque;
  pColor color;
  bool contact;                 // When true, ball rendered in gray.
  float spring_constant_sum;    // Used to compute minimum mass.

  void orientation_set(pNorm dir, float angle)
    { orientation_set(pQuat(dir,angle)); }
  void orientation_set(pQuat ori)
    {
      orientation = ori;
      omatrix = pMatrix3x3_Rotation(orientation);
    }
  void constants_update();
  pVect point_rot_vel(pNorm tact_dir);
  pVect point_rot_vel(pVect tact_dir);
  void apply_tan_force(pNorm tact_dir, pNorm force_dir, double force_mag);
  void apply_tan_force(pNorm tact_dir, pVect force);
  void push(pVect amt);
  void translate(pVect amt);
  void stop();
  void freeze();
};


void
Ball::constants_update()
{
  assert( radius );
  radius_sq = radius * radius;
  mass = 4.0 / 3 * M_PI * radius_sq * radius * density;
  radius_inv = 1 / radius;
  mass_inv = 1 / mass;
  // FYI, notice simplifications:
  //   const float mo_inertia = 0.4 * mass * radius_sq;
  //   fdt_to_do = radius / mo_inertia;
  fdt_to_do = radius_inv * 2.5 * mass_inv;
}

pVect
Ball::point_rot_vel(pNorm direction)
{
  /// Return velocity of point on surface of ball.
  //
```

```
  return radius * cross( omega, direction );
}

pVect
Ball::point_rot_vel(pVect rel_pos)
{
  /// Return velocity of point relative to center.
  //
  return cross( omega, rel_pos );
}

void
Ball::apply_tan_force(pNorm tact_dir, pVect force)
{
  torque += cross(tact_dir, force);
}

void
Ball::apply_tan_force(pNorm tact_dir, pNorm force_dir, double force_mag)
{
  apply_tan_force(tact_dir, force_mag * force_dir);
}

class Link {
public:
  Link(Ball *b1, Ball *b2):ball1(b1),ball2(b2),
     snapped(false),
     natural_color(color_lsu_spirit_purple),color(color_lsu_spirit_purple)
  { init(); }
  Link(Ball *b1, Ball *b2, pColor colorp):ball1(b1),ball2(b2),
     snapped(false),
     natural_color(colorp),color(colorp)
  { init(); }
  Link(Link *link, pVect cb1p, pVect cb2p, float drp):
    ball1(link->ball1), ball2(link->ball2), cb1(cb1p), cb2(cb2p),
    is_surface_connection(true), is_renderable(false),
    is_simulatable(true), distance_relaxed(drp),
    snapped(false){ link->is_simulatable = false; }
  void init()
    {
      assert( ball1->radius > 0 );
      assert( ball2->radius > 0 );
      is_simulatable = true;
      is_renderable = true;
      pNorm n12(ball1->position,ball2->position);
      const float rad_sum = ball1->radius + ball2->radius;
      is_surface_connection = n12.magnitude >= rad_sum;
      if ( !is_surface_connection )
        {
          distance_relaxed = n12.magnitude;
          cb1 = cb2 = pVect(0,0,0);
          return;
        }
      distance_relaxed = n12.magnitude - rad_sum;
      cb1 = ball1->radius * mult_MTV( ball1->omatrix, n12 );
      cb2 = ball2->radius * mult_MTV( ball2->omatrix, -n12 );
    }
  int serial;
  Ball* const ball1;
  Ball* const ball2;
  int ball1_idx;
  int ball2_idx;
  pVect cb1, cb2, b1_dir, b2_dir;
```

```
  bool is_surface_connection;
  bool is_renderable, is_simulatable;
  float distance_relaxed;
  bool snapped;
  pColor natural_color;
  pColor color;

  pVect spring_force_12;
  pVect torque1,torque2;

};

// Declare containers for Balls and Links. The containers are derived
// from std::vector and use overloaded operator += for push_back. (See
// include/util-containers.h.)
//
typedef pVectorI<Link> Links;
typedef pVectorI<Ball> Balls;


enum Render_Option { RO_Normally, RO_Simple, RO_Shadow_Volumes };

class World {
public:
  World(pOpenGL_Helper &fb, int argc, char **argv):ogl_helper(fb)
  {init(argc,argv);}
  void init(int argc, char **argv);
  void init_graphics();
  static void frame_callback_w(void *moi){((World*)moi)->frame_callback();}
  void frame_callback();
  void render();
  void render_objects(Render_Option render_option);
  void objects_erase();
  void cb_keyboard();
  void modelview_update();

  pOpenGL_Helper& ogl_helper;
  pVariable_Control variable_control;
  pFrame_Timer frame_timer;
  double world_time;
  double last_frame_wall_time;
  float opt_time_step_duration;
  int time_step_count;
  float opt_gravity_accel;      // Value chosen by user.
  pVect gravity_accel;          // Set to zero when opt_gravity is false;
  bool opt_gravity;
  bool opt_head_lock, opt_tail_lock;
  bool opt_tryout1, opt_tryout2;  // For ad-hoc experiments.

  // Tiled platform for ball.
  //
  float platform_xmin, platform_xmax, platform_zmin, platform_zmax;
  float platform_pi_xwidth_inv;
  pBuffer_Object<pVect> platform_tile_coords;
  pBuffer_Object<float> platform_tex_coords;
  pVect platform_normal;
  GLuint texid_hw;
  GLuint texid_syl;
  GLuint texid_emacs;
  void platform_update();
  bool platform_collision_possible(pCoor pos);

  pCoor light_location;
```

```
float opt_light_intensity;
enum { MI_Eye, MI_Light, MI_Ball, MI_Ball_V, MI_COUNT } opt_move_item;
bool opt_pause;
bool opt_single_frame;      // Simulate for one frame.
bool opt_single_time_step;  // Simulate for one time step.

pCoor eye_location;
pVect eye_direction;
pMatrix modelview;
pMatrix transform_mirror;

pVect adj_vector;
double adj_t_prev;
double adj_t_stop;
double adj_duration;

uint opt_shader;

bool opt_mirror;
bool opt_shadows;
bool opt_shadow_volumes;      // Make shadow volumes visible.
pShader *sp_fixed;          // Fixed functionality.
pShader *sp_phong;          // Basic stuff.
pShader *sp_instances_sphere;
pShader *sp_instances_sv;

GLuint balls_pos_rad_bo;
GLuint balls_color_bo;
GLuint links_indices_bo;

pColor *balls_color;
pCoor *balls_pos_rad;
size_t balls_size;
size_t links_size;
int last_setup; // Last scene set up.
bool link_change;

void ball_setup_1(bool omit_branches = true);
void ball_setup_2();
void ball_setup_3();
void ball_setup_4();
void ball_setup_5();
void setup_at_end();
void time_step_cpu(double);
void balls_stop();
void balls_freeze();
void balls_translate(pVect amt, int idx);
void balls_translate(pVect amt);
void balls_push(pVect amt, int idx);
void balls_push(pVect amt);
void balls_twirl();
void lock_update();

Ball *make_marker(pCoor pos, pColor col);

float opt_spring_constant;
float opt_air_resistance;
float distance_relaxed;
int chain_length;
Balls balls;
Ball *head_ball, *tail_ball;
Links links;
```

```
  Sphere sphere;
  Cylinder cyl;

  Links link_new(Ball *ball1, Ball *ball2, float stiffness = 0.2);


  /// CUDA Stuff

  int opt_physics_method;
  GPU_Info gpu_info;
  cudaEvent_t frame_start_ce, frame_stop_ce;
  int opt_block_size;
  bool gpu_const_data_stale;
  bool cpu_phys_data_stale;

  void data_cpu_to_gpu_constants();
  void data_cpu_to_gpu_dynamic();
  void data_gpu_to_cpu_dynamic();
  CPU_GPU_Common c;  // c is for common.  See links.cuh.

  void render_link_start();
  void render_link_gather(Link *link);
  void render_link_render(bool shadow_volumes = false);
  void render_link_render_sv(){ render_link_render(true); };

  pShader *sp_set_2, *sp_set_2_sv;
  GLuint link_bo_vtx, link_bo_nrm, link_bo_tco;
  double world_time_link_update;

  int opt_sides, opt_segments;

  pVectorI<pCoors> lis_pvects;

  pCoors lis_pos1, lis_pos2;
  pCoors lis_v1, lis_v2, lis_b1_ydir, lis_b2_ydir, lis_tex_rad;
  GLuint *lis_bos;

  /// Homework 2:
  vector<pCoor> sphere_coords;
  GLuint sphere_bo;
  void render_p1();
  void render_p2();
  float opt_omega;
  int opt_slices, opt_spirals;
  int opt_hide_stuff;
  bool opt_normal_sphere;
};

enum { OH_Links = 1, OH_Platform = 2, OH_Sphere = 4 };

static const char* const sh_names[] =
  { "INSTANCE",
    "PROBLEM 1 (render_p1)",
    "PROBLEM 2 (render_p2)" };

void
World::init_graphics()
{
  ///
  /// Graphical Model Initialization
  ///

  balls_pos_rad = NULL;
```

```
  balls_pos_rad_bo = 0;
  balls_color = NULL;
  balls_color_bo = 0;
  balls_size = 0;
  links_size = 0;
  link_change = true;

  opt_head_lock = false;
  opt_tail_lock = false;
  opt_tryout1 = opt_tryout2 = false;

  eye_location = pCoor(24.2,11.6,-38.7);
  eye_direction = pVect(-0.42,-0.09,0.9);

  platform_xmin = -40; platform_xmax = 40;
  platform_zmin = -40; platform_zmax = 40;
  texid_syl = pBuild_Texture_File("gpup.png",false,255);
  texid_emacs = pBuild_Texture_File("mult.png", false,-1);
  texid_hw = pBuild_Texture_File("hw05-assign.png", false,255);

  opt_light_intensity = 100.2;
  light_location = pCoor(platform_xmax,platform_xmax,platform_zmin);

  variable_control.insert(opt_light_intensity,"Light Intensity");

  opt_move_item = MI_Eye;
  opt_pause = false;
  opt_single_time_step = false;
  opt_single_frame = false;

  sphere.init(35);

  platform_update();
  modelview_update();

  adj_t_stop = 0;
  adj_duration = 0.25;

  link_bo_vtx = 0;
  opt_sides = 20;
  opt_segments = 15;

  /// Homework 2
  sphere_bo = 0;
  opt_omega = 5;
  variable_control.insert(opt_omega,"Spiral Factor (opt_omega)");
  opt_slices = 20;
  variable_control.insert(opt_slices,"Slices",1,1);
  opt_spirals = 20;
  variable_control.insert(opt_spirals,"spirals",1,2);
  opt_hide_stuff = 0;
  opt_normal_sphere = false;
}


void
World::platform_update()
{
  const float tile_count = 19;
  const float ep = 1.00001;
  const float delta_x = ( platform_xmax - platform_xmin ) / tile_count * ep;
  const float zdelta = ( platform_zmax - platform_zmin ) / tile_count * ep;

  const float trmin = 0.05;
  const float trmax = 0.7;
```

```
  const float tsmin = 0;
  const float tsmax = 0.4;

  platform_normal = pVect(0,1,0);

  PStack<pVect> p_tile_coords;
  PStack<pVect> p1_tile_coords;
  PStack<float> p_tex_coords;
  bool even = true;

  for ( int i = 0; i < tile_count; i++ )
    {
      const float x0 = platform_xmin + i * delta_x;
      const float x1 = x0 + delta_x;
      const float y = 0;
      for ( float z = platform_zmin; z < platform_zmax; z += zdelta )
        {
          PStack<pVect>& t_coords = even ? p_tile_coords : p1_tile_coords;
          p_tex_coords += trmax; p_tex_coords += tsmax;
          t_coords += pVect(x0,y,z);
          p_tex_coords += trmax; p_tex_coords += tsmin;
          t_coords += pVect(x0,y,z+zdelta);
          p_tex_coords += trmin; p_tex_coords += tsmin;
          t_coords += pVect(x1,y,z+zdelta);
          p_tex_coords += trmin; p_tex_coords += tsmax;
          t_coords += pVect(x1,y,z);
          even = !even;
        }
    }

  while ( pVect* const v = p1_tile_coords.iterate() ) p_tile_coords += *v;

  platform_tile_coords.re_take(p_tile_coords);
  platform_tile_coords.to_gpu();
  platform_tex_coords.re_take(p_tex_coords);
  platform_tex_coords.to_gpu();

}

void
World::modelview_update()
{
  pMatrix_Translate center_eye(-eye_location);
  pMatrix_Rotation rotate_eye(eye_direction,pVect(0,0,-1));
  modelview = rotate_eye * center_eye;
  pMatrix reflect; reflect.set_identity(); reflect.rc(1,1) = -1;
  transform_mirror = modelview * reflect * invert(modelview);
}

void
World::render_objects(Render_Option option)
{
  const float shininess_ball = 5;
  pColor spec_color(1,1,1);
  pColor spec_light = color_white * opt_light_intensity;
  GLboolean l0, l1;
  glGetBooleanv(GL_LIGHT0,&l0);
  glGetBooleanv(GL_LIGHT1,&l1);
  int light_state = l0 | (l1<<1);
  glLightfv(GL_LIGHT0, GL_SPECULAR, spec_light);
  const double wt = world_time;

  const bool hide_links = opt_hide_stuff & OH_Links;
  const bool hide_platform = opt_hide_stuff & OH_Platform;
  const bool hide_sphere = opt_hide_stuff & OH_Sphere;
```

```
  if ( option == RO_Normally )
    {
      glLightf(GL_LIGHT0, GL_LINEAR_ATTENUATION, 1.0);
      glEnable(GL_TEXTURE_2D);
      glBindTexture(GL_TEXTURE_2D,texid_emacs);
      glMaterialf(GL_FRONT_AND_BACK,GL_SHININESS,shininess_ball);
      glMaterialfv(GL_FRONT_AND_BACK,GL_SPECULAR,spec_color);
      glLightModeli(GL_LIGHT_MODEL_COLOR_CONTROL, GL_SEPARATE_SPECULAR_COLOR);
      glEnable(GL_COLOR_SUM);
    }

  cyl.apex_radius = 1; cyl.set_color(color_lsu_spirit_purple);
  if ( option == RO_Shadow_Volumes )
    {
      if ( opt_shadow_volumes )
        {
          glEnable(GL_LIGHTING);
          glColorMask(true,true,true,true);
          glLightf(GL_LIGHT0, GL_LINEAR_ATTENUATION, 1.0);
          glEnable(GL_BLEND);
          glBlendEquation(GL_FUNC_ADD);
          glBlendFunc(GL_ONE,GL_ONE);
          glColor3f(0.8,0,0);
        }
      cyl.light_pos = light_location;
      sphere.light_pos = light_location;

      if ( !hide_sphere )
        {
          sphere.render_bunch_gather_sv(wt);
          for ( Ball *ball: balls )
            sphere.render_shadow_volume(ball->radius,ball->position);

          sp_instances_sv->use();
          glUniform3i(3, opt_tryout1, opt_tryout2, opt_normal_sphere);
          sphere.render_bunch_render_sv();
          sp_fixed->use();
        }

      if ( !hide_links )
        {
          render_link_start();
          for ( Link *link: links )
            if ( link->is_renderable ) render_link_gather(link);
          render_link_render_sv();
        }
    }
  else
    {
      if ( !hide_sphere )
        {
          sphere.opt_texture = true;

          sphere.render_bunch_gather(wt);
          for ( Ball *ball: balls )
            {
              if ( ball->contact )
                sphere.color = color_gray;
              else if ( ball->mass > 0 && ball->mass < ball->mass_min )
                sphere.color = color_red;
              else if ( ball->mass > 0 && ball->locked )
                sphere.color = color_pale_green;
              else
                sphere.color = ball->color;
```

```
            sphere.render
              (ball->radius,ball->position,
               pMatrix_Rotation(ball->orientation));
          }

        switch ( opt_shader ) {
        case 0: sp_instances_sphere->use(); break;
        case 1: case 2: sp_phong->use(); break;
        default: assert( false );
        }

        glUniform1i(2, light_state);
        glUniform3i(3, opt_tryout1, opt_tryout2, opt_normal_sphere);
        glDisable(GL_TEXTURE_2D);

        switch ( opt_shader ) {
        case 0: sphere.render_bunch_render(); break;
        case 1: render_p1(); break;
        case 2: render_p2(); break;
        default: assert( false );
        }
      }

    if ( !hide_links )
      {
        glBindTexture(GL_TEXTURE_2D,texid_hw);
        glEnable(GL_TEXTURE_2D);

        render_link_start();

        for ( Link *link: links )
          {
            if ( !link->is_renderable ) continue;
            render_link_gather(link);
            continue;

            Ball *const ball1 = link->ball1;
            Ball *const ball2 = link->ball2;
            pVect dir1 = ball1->omatrix * link->cb1;
            pCoor pos1 = ball1->position + dir1;

            pVect dir2 = ball2->omatrix * link->cb2;
            pCoor pos2 = ball2->position + dir2;

            cyl.set_color(link->color);
            cyl.render(pos1,0.3*ball1->radius, pos2-pos1);
          }

        render_link_render();
      }
  }

sp_fixed->use();

if ( hide_platform ) return;

if ( option == RO_Shadow_Volumes ) return;

sp_phong->use();
glUniform1i(2, light_state);
glUniform3i(3, opt_tryout1, opt_tryout2, opt_normal_sphere);

//
// Render Platform
//
```

```
  const int half_elements = platform_tile_coords.elements >> 3 << 2;

  glEnable(GL_TEXTURE_2D);

  // Set up attribute (vertex, normal, etc.) arrays.
  //
  glBindTexture(GL_TEXTURE_2D,texid_syl);
  platform_tile_coords.bind();
  glVertexPointer(3, GL_FLOAT, sizeof(platform_tile_coords.data[0]), 0);
  glEnableClientState(GL_VERTEX_ARRAY);
  glNormal3fv(platform_normal);
  platform_tex_coords.bind();
  glTexCoordPointer(2, GL_FLOAT,2*sizeof(float), 0);
  glEnableClientState(GL_TEXTURE_COORD_ARRAY);

  // Write lighter-colored, textured tiles.
  //
  glMaterialfv(GL_FRONT_AND_BACK,GL_SPECULAR,spec_color);
  glMaterialf(GL_FRONT_AND_BACK,GL_SHININESS,5.0);
  glColor3f(0.55,0.55,0.55);
  glDrawArrays(GL_QUADS,0,half_elements+4);

  glDisableClientState(GL_VERTEX_ARRAY);
  glDisableClientState(GL_TEXTURE_COORD_ARRAY);
  glBindBuffer(GL_ARRAY_BUFFER,0);

  sp_fixed->use();
}

void
World::render()
{
  // Get any waiting keyboard commands.
  //
  cb_keyboard();

  // Start a timer object used for tuning this code.
  //
  frame_timer.frame_start();

  /// Emit a Graphical Representation of Simulation State
  //

  // Understanding of the code below not required for introductory
  // lectures.

  // That said, much of the complexity of the code is to show
  // the ball shadow and reflection.


  const int win_width = ogl_helper.get_width();
  const int win_height = ogl_helper.get_height();
  const float aspect = float(win_width) / win_height;

  glMatrixMode(GL_MODELVIEW);
  glLoadTransposeMatrixf(modelview);

  glMatrixMode(GL_PROJECTION);
  glLoadIdentity();
  // Frustum: left, right, bottom, top, near, far
  const float xr = .08;
  glFrustum(-xr,xr,-xr/aspect,xr/aspect,.1,5000);

  glViewport(0, 0, win_width, win_height);
  pError_Check();
```

```
  glClearColor(0,0,0,0.5);
  glClearDepth(1.0);
  glClearStencil(0);
  glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT | GL_STENCIL_BUFFER_BIT );

  glEnable(GL_DEPTH_TEST);
  glDepthFunc(GL_LESS);
  glDisable(GL_BLEND);

  glLightModeli(GL_LIGHT_MODEL_TWO_SIDE,1);
  glLightfv(GL_LIGHT0, GL_POSITION, light_location);

  glLightf(GL_LIGHT0, GL_CONSTANT_ATTENUATION, 0.5);
  glLightf(GL_LIGHT0, GL_LINEAR_ATTENUATION, 1.0);
  glLightf(GL_LIGHT0, GL_QUADRATIC_ATTENUATION, 0);

  pColor ambient_color(0x555555);

  glLightModelfv(GL_LIGHT_MODEL_AMBIENT, ambient_color);
  glLightfv(GL_LIGHT0, GL_DIFFUSE, color_white * opt_light_intensity);
  glLightfv(GL_LIGHT0, GL_AMBIENT, color_black);
  glLightfv(GL_LIGHT0, GL_SPECULAR, color_white * opt_light_intensity);

  glEnable(GL_LIGHT0);
  glEnable(GL_LIGHTING);

  glEnable(GL_COLOR_MATERIAL);
  glColorMaterial(GL_FRONT_AND_BACK,GL_AMBIENT_AND_DIFFUSE);

  glShadeModel(GL_SMOOTH);

  // Common to all textures.
  //
  glActiveTexture(GL_TEXTURE0);
  glEnable(GL_TEXTURE_2D);
  glTexParameterf(GL_TEXTURE_2D,GL_TEXTURE_MIN_FILTER,
                  GL_LINEAR_MIPMAP_LINEAR);
  glTexParameterf(GL_TEXTURE_2D,GL_TEXTURE_MAG_FILTER,GL_LINEAR);
  glTexEnvi(GL_TEXTURE_ENV,GL_TEXTURE_ENV_MODE,GL_MODULATE);
  glTexParameterf(GL_TEXTURE_2D,GL_TEXTURE_WRAP_S,GL_REPEAT);
  glTexParameterf(GL_TEXTURE_2D,GL_TEXTURE_WRAP_T,GL_REPEAT);

  glEnable(GL_RESCALE_NORMAL);
  glEnable(GL_NORMALIZE);

  const double time_now = time_wall_fp();
  const bool blink_visible = int64_t(time_now*3) & 1;
# define BLINK(txt,pad) ( blink_visible ? txt : pad )

  ogl_helper.fbprintf("%s\n",frame_timer.frame_rate_text_get());

  ogl_helper.fbprintf
    ("Compiled: %s\n",
#ifdef __OPTIMIZE__
     "WITH OPTIMIZATION"
#else
     BLINK("WITHOUT OPTIMIZATION","")
#endif
     );

  ogl_helper.fbprintf
    ("Time Step: %8d  World Time: %11.6f  %s\n",
     time_step_count, world_time,
     opt_pause ? BLINK("PAUSED, 'p' to unpause, SPC or S-SPC to step.","") :
```

```
    "Press 'p' to pause."
    );

  ogl_helper.fbprintf
    ("Eye location: [%5.1f, %5.1f, %5.1f]  "
     "Eye direction: [%+.2f, %+.2f, %+.2f]\n",
     eye_location.x, eye_location.y, eye_location.z,
     eye_direction.x, eye_direction.y, eye_direction.z);

  Ball& ball = *balls[0];

  ogl_helper.fbprintf
    ("Head Ball Pos  [%5.1f,%5.1f,%5.1f] Vel [%+5.1f,%+5.1f,%+5.1f]\n",
     ball.position.x,ball.position.y,ball.position.z,
     ball.velocity.x,ball.velocity.y,ball.velocity.z );

  ogl_helper.fbprintf
    ("%4zd Balls  %4zd Links  Physics: %s ('a' to change)  \n",
     balls.size(), links.size(), gpu_physics_method_str[opt_physics_method]);

  ogl_helper.fbprintf
    ("Hide: %c%c%c ('!@#')  Effect: %c%c ('or')  Shader: %s  ('v')  "
     "Tryout 1: %s  ('y')  Tryout 2: %s  ('Y')  Normals: %s ('n')\n",
     opt_hide_stuff & OH_Sphere ? 'S' : '_',
     opt_hide_stuff & OH_Links ? 'L' : '_',
     opt_hide_stuff & OH_Platform ? 'P' : '_',
     opt_shadows ? 'S' : '_',
     opt_mirror ? 'M' : '_',
     sh_names[opt_shader],
     opt_tryout1 ? BLINK("ON ","   ") : "OFF",
     opt_tryout2 ? BLINK("ON ","   ") : "OFF",
     opt_normal_sphere ? "SPHERE" : "TRI");

  pVariable_Control_Elt* const cvar = variable_control.current;
  ogl_helper.fbprintf("VAR %s = %.5f  (TAB or '`' to change, +/- to adjust)\n",
                      cvar->name,cvar->get_val());

  const int half_elements = platform_tile_coords.elements >> 3 << 2;

  //
  // Render ball reflection.  (Will be blended with dark tiles.)
  //

  const bool hide_platform = opt_hide_stuff & OH_Platform;

  if ( !hide_platform )
    {
      if ( opt_mirror )
        {
          // Write stencil value 2 at location of dark (mirrored) tiles.
          //
          glDisable(GL_LIGHTING);
          glEnable(GL_STENCIL_TEST);
          glStencilFunc(GL_NEVER,2,2);
          glStencilOp(GL_REPLACE,GL_KEEP,GL_KEEP);
          platform_tile_coords.bind();
          glVertexPointer(3, GL_FLOAT, sizeof(platform_tile_coords.data[0]), 0);
          glEnableClientState(GL_VERTEX_ARRAY);
          glDrawArrays(GL_QUADS,half_elements+4,half_elements-4);
          glEnable(GL_LIGHTING);
          glDisableClientState(GL_VERTEX_ARRAY);
          glBindBuffer(GL_ARRAY_BUFFER,0);

          // Prepare to write only stenciled locations.
          //
```

```
          glStencilFunc(GL_EQUAL,2,2);
          glStencilOp(GL_KEEP,GL_KEEP,GL_KEEP);

          // Use a transform that reflects objects to other side of platform.
          //
          glMatrixMode(GL_PROJECTION);
          glPushMatrix();
          glMultTransposeMatrixf(transform_mirror);

          // Reflected front face should still be treated as the front face.
          //
          glFrontFace(GL_CW);

          render_objects(RO_Normally);

          glFrontFace(GL_CCW);
          glMatrixMode(GL_PROJECTION);
          glPopMatrix();
        }

      // Blend mirror tiles with possibly existing ball reflection.
      //
      glDepthFunc(GL_ALWAYS);

      glDisable(GL_STENCIL_TEST);
      glDisable(GL_TEXTURE_2D);

      glEnable(GL_LIGHTING);
      glLightModeli(GL_LIGHT_MODEL_COLOR_CONTROL, GL_SINGLE_COLOR);
      glLightfv(GL_LIGHT0, GL_SPECULAR, color_black);

      glColor3fv(color_lsu_spirit_purple);

      glEnable(GL_BLEND);
      glBlendEquation(GL_FUNC_ADD);
      glBlendFunc(GL_CONSTANT_ALPHA,GL_ONE_MINUS_CONSTANT_ALPHA); // src, dst
      glBlendColor(0,0,0,0.5);

      platform_tile_coords.bind();
      glVertexPointer
        (3, GL_FLOAT,sizeof(platform_tile_coords.data[0]), 0);
      glEnableClientState(GL_VERTEX_ARRAY);

      glDrawArrays(GL_QUADS,half_elements+4,half_elements-4);

      glDisableClientState(GL_VERTEX_ARRAY);
      glBindBuffer(GL_ARRAY_BUFFER,0);

    }

  glDepthFunc(GL_LESS);
  glDisable(GL_BLEND);

  glLightModelfv(GL_LIGHT_MODEL_AMBIENT, ambient_color);

  if ( !opt_shadows )
    {
      // Render.
      //
      render_objects(RO_Normally);
    }
  else
    {
      //
      // First pass, render using only ambient light.
```

```
    //
    glDisable(GL_LIGHT0);

    // Send balls, tiles, and platform to opengl.
    // Do occlusion test too.
    //
    render_objects(RO_Normally);

    //
    // Second pass, add on light0.
    //

    // Turn off ambient light, turn on light 0.
    //
    glLightModelfv(GL_LIGHT_MODEL_AMBIENT, color_black);
    glEnable(GL_LIGHT0);


    glClear(GL_STENCIL_BUFFER_BIT);

    // Make sure that only stencil buffer written.
    //
    glDepthMask(false);

    glColorMask(false,false,false,false);

    // Don't waste time computing lighting.
    //
    glDisable(GL_LIGHTING);
    glDisable(GL_TEXTURE_2D);

    // Set up stencil test to count shadow volume surfaces: plus 1 for
    // entering the shadow volume, minus 1 for leaving the shadow
    // volume.
    //
    glEnable(GL_STENCIL_TEST);
    // sfail, dfail, dpass
    glStencilOpSeparate(GL_FRONT,GL_KEEP,GL_KEEP,GL_INCR_WRAP);
    glStencilOpSeparate(GL_BACK,GL_KEEP,GL_KEEP,GL_DECR_WRAP);
    glStencilFuncSeparate(GL_FRONT_AND_BACK,GL_ALWAYS,1,-1); // ref, mask

    // Write stencil with shadow locations based on shadow volumes
    // cast by light0 (light_location).  Shadowed locations will
    // have a positive stencil value.
    //
    render_objects(RO_Shadow_Volumes);

    glEnable(GL_LIGHTING);
    glColorMask(true,true,true,true);
    glDepthMask(true);

    // Use stencil test to prevent writes to shadowed areas.
    //
    glStencilOp(GL_KEEP,GL_KEEP,GL_KEEP);
    glStencilFunc(GL_EQUAL,0,-1); // ref, mask

    // Allow pixels to be re-written.
    //
    glDepthFunc(GL_LEQUAL);
    glEnable(GL_BLEND);
    glBlendEquation(GL_FUNC_ADD);
    glBlendFunc(GL_ONE,GL_ONE);

    // Render.
    //
```

```
      render_objects(RO_Normally);

      glDisable(GL_BLEND);
      glDisable(GL_STENCIL_TEST);

    }

  glDepthFunc(GL_LESS);

  // Render Marker for Light Source
  //
  insert_tetrahedron(light_location,0.5);

  pError_Check();

  glColor3f(0.5,1,0.5);

  frame_timer.frame_end();

  ogl_helper.user_text_reprint();

  glutSwapBuffers();
}


void
World::cb_keyboard()
{
  if ( !ogl_helper.keyboard_key ) return;
  pVect adjustment(0,0,0);
  pVect user_rot_axis(0,0,0);
  const bool kb_mod_s = ogl_helper.keyboard_shift;
  const bool kb_mod_c = ogl_helper.keyboard_control;
  const float move_amt = kb_mod_s ? 2.0 : kb_mod_c ? 0.08 : 0.4;

  switch ( ogl_helper.keyboard_key ) {
  case FB_KEY_LEFT: adjustment.x = -move_amt; break;
  case FB_KEY_RIGHT: adjustment.x = move_amt; break;
  case FB_KEY_PAGE_UP: adjustment.y = move_amt; break;
  case FB_KEY_PAGE_DOWN: adjustment.y = -move_amt; break;
  case FB_KEY_DOWN: adjustment.z = move_amt; break;
  case FB_KEY_UP: adjustment.z = -move_amt; break;
  case FB_KEY_DELETE: user_rot_axis.y = 1; break;
  case FB_KEY_INSERT: user_rot_axis.y =  -1; break;
  case FB_KEY_HOME: user_rot_axis.x = 1; break;
  case FB_KEY_END: user_rot_axis.x = -1; break;
  case '1': ball_setup_1(); break;
  case '2': ball_setup_2(); break;
  case '3': ball_setup_3(); break;
  case '4': ball_setup_4(); break;
  case '5': ball_setup_5(); break;
  case 'a':
    opt_physics_method++;
    if ( opt_physics_method == GP_ENUM_SIZE ) opt_physics_method = GP_cpu;
    break;
  case 'b': opt_move_item = MI_Ball; break;
  case 'B': opt_move_item = MI_Ball_V; break;
  case 'e': case 'E': opt_move_item = MI_Eye; break;
  case 'g': case 'G': opt_gravity = !opt_gravity; break;
  case 'h': case 'H': opt_head_lock = !opt_head_lock; break;
  case 'k': case 'K': opt_hide_stuff = 0x7 & ( opt_hide_stuff + 1 ); break;
  case '!': opt_hide_stuff ^= OH_Sphere; break;
  case '@': opt_hide_stuff ^= OH_Links; break;
  case '#': opt_hide_stuff ^= OH_Platform; break;
  case 't': case 'T': opt_tail_lock = !opt_tail_lock; break;
```

```
  case 'l': case 'L': opt_move_item = MI_Light; break;
  case 'r': case 'R': opt_mirror = !opt_mirror; break;
  case 'o': opt_shadows = !opt_shadows; break;
  case 'O':
    opt_shadow_volumes = !opt_shadow_volumes;
    if ( opt_shadow_volumes ) opt_shadows = true;
    break;
  case 'n': case 'N': opt_normal_sphere = !opt_normal_sphere; break;
  case 'p': case 'P': opt_pause = !opt_pause; break;
  case 's': case 'S': balls_stop(); break;
  case 'v': case 'V': opt_shader++;
    if ( opt_shader >= sizeof(sh_names)/sizeof(sh_names[0]) ) opt_shader = 0;
    sphere_coords.clear();
    break;
  case 'w': case 'W': balls_twirl(); break;
  case 'y': opt_tryout1 = !opt_tryout1; break;
  case 'Y': opt_tryout2 = !opt_tryout2; break;
  case ' ':
    if ( kb_mod_s ) opt_single_time_step = true; else opt_single_frame = true;
    opt_pause = true;
    break;
  case 9: variable_control.switch_var_right(); break;
  case 96: variable_control.switch_var_left(); break; // `, until S-TAB works.
  case '-':case '_':
    variable_control.adjust_lower();
    sphere_coords.clear();
    break;
  case '+':case '=':
    variable_control.adjust_higher();
    sphere_coords.clear();
    break;
  default: printf("Unknown key, %d\n",ogl_helper.keyboard_key); break;
  }

  gravity_accel.y = opt_gravity ? -opt_gravity_accel : 0;

  lock_update();
  sphere.bunch_invalidate();

  // Update eye_direction based on keyboard command.
  //
  if ( user_rot_axis.x || user_rot_axis.y )
    {
      pMatrix_Rotation rotall(eye_direction,pVect(0,0,-1));
      user_rot_axis *= invert(rotall);
      eye_direction *= pMatrix_Rotation(user_rot_axis, M_PI * 0.03);
      modelview_update();
    }

  // Update eye_location based on keyboard command.
  //
  if ( adjustment.x || adjustment.y || adjustment.z )
    {
      const double angle =
        fabs(eye_direction.y) > 0.99
        ? 0 : atan2(eye_direction.x,-eye_direction.z);
      pMatrix_Rotation rotall(pVect(0,1,0),-angle);
      adjustment *= rotall;

      switch ( opt_move_item ){
      case MI_Ball:
        adj_vector = adjustment;
        if ( adj_t_stop == 0 )
          adj_t_prev = adj_t_stop = world_time;
        adj_t_stop += adj_duration;
```

```
        break;
      case MI_Ball_V: balls_push(adjustment,0); break;
      case MI_Light: light_location += adjustment; break;
      case MI_Eye: eye_location += adjustment; break;
      default: break;
      }
      modelview_update();
    }

}




void
World::init(int argc, char **argv)
{
  chain_length = 14;
  const bool all_vars = false;

  opt_time_step_duration = 0.0003;
  if ( all_vars )
    variable_control.insert(opt_time_step_duration,"Time Step Duration");

  distance_relaxed = 15.0 / chain_length;
  opt_spring_constant = 500;
  variable_control.insert(opt_spring_constant,"Spring Constant");

  opt_gravity_accel = 9.8;
  opt_gravity = true;
  gravity_accel = pVect(0,-opt_gravity_accel,0);
  if ( all_vars )
    variable_control.insert(opt_gravity_accel,"Gravity");

  opt_air_resistance = 0.04;
  if ( all_vars )
    variable_control.insert(opt_air_resistance,"Air Resistance");

  world_time = 0;
  time_step_count = 0;
  last_frame_wall_time = time_wall_fp();
  frame_timer.work_unit_set("Steps / s");

  world_time_link_update = 0;

  opt_mirror = true;
  opt_shadows = true;
  opt_shadow_volumes = false;
  opt_shader = 0;

  /// Init CUDA

  // Get information about GPU and its ability to run CUDA.
  //
  int device_count;
  cudaGetDeviceCount(&device_count); // Get number of GPUs.
  if ( device_count == 0 )
    {
      fprintf(stderr,"No GPU found, exiting.\n");
      exit(1);
    }

  // Print information about the available GPUs.
  //
  gpu_info_print();
```

```
// Choose GPU 0 because we don't have time to provide a way to let
// the user choose.
//
int dev = gpu_choose_index();
CE(cudaSetDevice(dev));
printf("Using GPU %d\n",dev);

gpu_info.get_gpu_info(dev);
cuda_setup(&gpu_info);

// Print information about time_step routine.
//
printf("\nCUDA Routine Resource Usage:\n");

int block_size_max = 4096;

for ( int i=0; i<gpu_info.num_kernels; i++ )
  {
    printf("For %s:\n", gpu_info.ki[i].name);
    printf("  %6zd shared, %zd const, %zd loc, %d regs; "
           "%d max threads per block.\n",
           gpu_info.ki[i].cfa.sharedSizeBytes,
           gpu_info.ki[i].cfa.constSizeBytes,
           gpu_info.ki[i].cfa.localSizeBytes,
           gpu_info.ki[i].cfa.numRegs,
           gpu_info.ki[i].cfa.maxThreadsPerBlock);
    set_min(block_size_max,gpu_info.ki[i].cfa.maxThreadsPerBlock);
  }

opt_physics_method = GP_cuda;
gpu_const_data_stale = true;
cpu_phys_data_stale = false;

CE( cudaEventCreate(&frame_start_ce) );
CE( cudaEventCreate(&frame_stop_ce) );

opt_block_size = 32;

if ( all_vars )
  variable_control.insert_power_of_2
    (opt_block_size, "Block Size", 32, block_size_max);

c.alloc_n_balls = 0;
c.alloc_n_links = 0;

init_graphics();

// Declared like a programmable shader, but used for fixed-functionality.
//
sp_fixed = new pShader();

sp_phong = new pShader
  ("links-shdr.cc",// File holding shader program.
   "vs_main(); ",       // Name of vertex shader main routine.
   "gs_main_simple();", // Name of geometry shader main routine.
   "fs_main();"         // Name of fragment shader main routine.
   );

sp_instances_sphere = new pShader
  ("links-shdr.cc",// File holding shader program.
   "vs_main_instances_sphere(); ", // Used to render many spheres at once.
   "gs_main_simple();", // Name of geometry shader main routine.
   "fs_main();"         // Name of fragment shader main routine.
   );
```

```
  sp_instances_sv = new pShader
    ("links-shdr.cc",// File holding shader program.
     "vs_main_instances_sv();" // Instances of vtx coord only. Use gl_Color.
     ,"gs_main_sv();",
     "fs_main_sv();"
     );


  PSplit exe_pieces(argv[0],'/');
  pString this_exe_name(exe_pieces.pop());

  const char* const links_shader_code_path = "links-shdr-links.cc";

  sp_set_2 = new pShader
    (links_shader_code_path,
     "vs_main_2(); ",     // Name of vertex shader main routine.
     "gs_main_2();",      // Name of geometry shader main routine.
     "fs_main();"       // Name of fragment shader main routine.
     );
  sp_set_2_sv = new pShader
    (links_shader_code_path,
     "vs_main_2_sv(); ",     // Name of vertex shader main routine.
     "gs_main_2_sv();",      // Name of geometry shader main routine.
     "fs_main_sv();"       // Name of fragment shader main routine.
     );

  ball_setup_1();
}

Ball*
World::make_marker(pCoor position, pColor color)
{
  Ball* const ball = new Ball;
  ball->position = position;
  ball->locked = true;
  ball->velocity = pVect(0,0,0);
  ball->radius = 0.2;
  ball->mass = 0;
  ball->contact = false;
  ball->color = color;
  return ball;
}

void
World::lock_update()
{
  // This routine called when options like opt_head_lock might have
  // changed.

  // Update locked status.
  //
  if ( head_ball ) head_ball->locked = opt_head_lock;
  if ( tail_ball ) tail_ball->locked = opt_tail_lock;

  // Re-compute minimum mass needed for stability.
  //
  for ( Ball *ball: balls ) ball->spring_constant_sum = 0;
  const double dtis = pow( opt_time_step_duration, 2 );
  for ( Link *link: links )
    {
      Ball* const b1 = link->ball1;
      Ball* const b2 = link->ball2;
      b1->spring_constant_sum += opt_spring_constant;
      b2->spring_constant_sum += opt_spring_constant;
```

```
    }
  for ( Ball *ball: balls )
    {
      ball->mass_min = ball->spring_constant_sum * dtis;
      ball->constants_update();
    }
  gpu_const_data_stale = true;
}

void
World::balls_twirl()
{
  if ( !head_ball || !tail_ball ) return;

  pNorm axis(head_ball->position, tail_ball->position);

  for ( Ball *ball: balls )
    {
      pVect b_to_top(ball->position,head_ball->position);
      const float dist_along_axis = dot(b_to_top,axis);
      const float lsq = b_to_top.mag_sq() - dist_along_axis * dist_along_axis;
      if ( lsq <= 1e-5 ) { ball->velocity = pVect(0,0,0); continue; }
      const float dist_to_axis = sqrt(lsq);
      pNorm rot_dir = cross(b_to_top,axis);
      ball->velocity += 2 * dist_to_axis * rot_dir;
    }
}

void
World::objects_erase()
{
  link_change = true;
  balls.erase();
  links.erase();
}

Links
World::link_new(Ball *ball1, Ball *ball2, float stiffness)
{
  Links links_rv;
  assert( ball1->radius > 0 );
  assert( ball2->radius > 0 );

  Link* const rlink = new Link(ball1,ball2);
  links_rv += rlink;

  pNorm n12(ball1->position,ball2->position);
  const float rad_sum = ball1->radius + ball2->radius;
  pMatrix3x3 b1rot = ball1->omatrix;
  pMatrix3x3 b2rot = ball2->omatrix;
  pCoor ctr = ball1->position
    + ( ball1->radius + 0.5 * ( n12.magnitude - rad_sum ) ) * n12;

  pNorm b1_y = b1rot * pVect(0,1,0);
  pNorm b1_x = b1rot * pVect(1,0,0);
  bool b1_dir_is_y = fabs(dot(b1_y,n12)) < 0.999;
  pNorm b1_dir = b1_dir_is_y ? b1_y : b1_x;

  pNorm con_x = cross(b1_dir,n12);
  pVect con_y = cross(n12,con_x);
  rlink->b1_dir = mult_MTV( b1rot, con_y );
  rlink->b2_dir = mult_MTV( b2rot, con_y );

  const float lrad = stiffness * ball1->radius;
```

```
  for ( int i=0; i<3; i++ )
    {
      const double theta = i * 2 * M_PI / 3;
      pVect convec = lrad * ( cos(theta) * con_x + sin(theta) * con_y );
      pCoor con = ctr + convec;
      pVect cb1 = mult_MTV( b1rot, con - ball1->position );
      pVect cb2 = mult_MTV( b2rot, con - ball2->position );
      links_rv += new Link(rlink, cb1, cb2, 0);
    }
  return links_rv;
}

///
/// CUDA Code
///

float4 tof4(pCoor c)
{
  float4 f4;
  f4.x = c.x;
  f4.y = c.y;
  f4.z = c.z;
  f4.w = c.w;
  return f4;
}

float4 tof4(pVect c)
{
  float4 f4;
  f4.x = c.x;
  f4.y = c.y;
  f4.z = c.z;
  f4.w = 0;
  return f4;
}

pCoor topc(float4 c)
{
  pCoor f4;
  f4.x = c.x;
  f4.y = c.y;
  f4.z = c.z;
  f4.w = c.w;
  return f4;
}

void
World::data_cpu_to_gpu_constants()
{
  c.platform_xmin = platform_xmin;
  c.platform_xmax = platform_xmax;
  c.platform_zmax = platform_zmax;
  c.platform_zmin = platform_zmin;

  c.n_balls = balls.size();
  c.n_links = links.size();
  c.opt_tryout1 = opt_tryout1;
  c.opt_tryout2 = opt_tryout2;
  c.opt_spring_constant = opt_spring_constant;
  c.opt_air_resistance = opt_air_resistance;
  c.opt_head_lock = opt_head_lock;
  c.opt_tail_lock = opt_tail_lock;
  c.gravity_accel = gravity_accel;
```

```
  data_cpu_to_gpu_common(&c);
}


void
World::data_cpu_to_gpu_dynamic()
{
  const int n_balls = balls.size();
  const int n_links = links.size();
  const bool need_alloc =
    n_balls > c.alloc_n_balls || n_links > c.alloc_n_links;
  const bool need_init = c.alloc_n_balls == 0;

#define ALLOC_MOVE_ZERO(n_balls,balls,memb,do_move,do_zero)                  \
  {                                                                          \
    const int size_elt_bytes = sizeof(c.balls.memb[0]);                       \
    const int size_bytes = n_balls * size_elt_bytes;                         \
    if ( need_alloc )                                                        \
      {                                                                      \
        if ( !need_init )                                                    \
          {                                                                  \
            CE( cudaFree( c.balls.memb ) );                                  \
            free(c.h_##balls.memb);                                          \
          }                                                                  \
        CE( cudaMalloc( &c.balls.memb, size_bytes ) );                       \
        c.h_##balls.memb = (decltype(c.balls.memb)) malloc( size_bytes );    \
      }                                                                      \
    if ( do_move )                                                           \
      {                                                                      \
        for ( int i=0; i<n_balls; i++ )                                      \
          memcpy(&c.h_##balls.memb[i],&balls[i]->memb,size_elt_bytes);       \
        CE( cudaMemcpy                                                       \
            ( c.balls.memb, c.h_##balls.memb,                                \
              size_bytes, cudaMemcpyDefault ) );                             \
      }                                                                      \
    else if ( do_zero )                                                      \
      {                                                                      \
        CE( cudaMemset( c.balls.memb, 0, size_bytes ) );                     \
      }                                                                      \
  }

#define MOVEc(num,ptr,memb) \
  ALLOC_MOVE_ZERO(num,ptr,memb,gpu_const_data_stale,0)
#define MOVE(num,ptr,memb) ALLOC_MOVE_ZERO(num,ptr,memb,1,0)
#define ALLOC(num,ptr,memb) ALLOC_MOVE_ZERO(num,ptr,memb,0,0)
#define ZERO(num,ptr,memb) ALLOC_MOVE_ZERO(num,ptr,memb,0,1)
  MOVEc(n_balls,balls,position);
  MOVEc(n_balls,balls,velocity);
  ZERO(n_balls,balls,force);

  MOVEc(n_balls,balls,orientation);
  MOVEc(n_balls,balls,omatrix);
  MOVEc(n_balls,balls,omega);
  ZERO(n_balls,balls,torque);
  MOVEc(n_balls,balls,mass);
  MOVEc(n_balls,balls,fdt_to_do);
  MOVEc(n_balls,balls,locked);

  MOVEc(n_links,links,ball1_idx);
  MOVEc(n_links,links,ball2_idx);
  MOVEc(n_links,links,cb1);
  MOVEc(n_links,links,cb2);
  MOVEc(n_links,links,distance_relaxed);
```

```
  MOVEc(n_links,links,is_simulatable);
  MOVEc(n_links,links,is_surface_connection);

#undef MOVE
#undef MOVEc
#undef ALLOC
#undef ZERO
#undef ALLOC_MOVE
  if ( need_alloc )
    {
      c.alloc_n_balls = n_balls;
      c.alloc_n_links = n_links;
      data_cpu_to_gpu_common(&c);
    }
  gpu_const_data_stale = false;
}

void
World::data_gpu_to_cpu_dynamic()
{
  const int n_balls = balls.size();

#define MOVE(n_balls,balls,memb)                                                \
  {                                                                            \
    const int size_elt_fr_bytes = sizeof(c.balls.memb[0]);                      \
    const int size_elt_to_bytes = sizeof(balls[0]->memb);                       \
    const int size_bytes = n_balls * size_elt_fr_bytes;                         \
    CE( cudaMemcpy                                                             \
        ( c.h_##balls.memb, c.balls.memb, size_bytes,                          \
          cudaMemcpyDeviceToHost ) );                                           \
    for ( int i=0; i<n_balls; i++ )                                            \
      memcpy(&balls[i]->memb,&c.h_##balls.memb[i], size_elt_to_bytes);         \
  }

  MOVE(n_balls,balls,position);
  MOVE(n_balls,balls,velocity);
  MOVE(n_balls,balls,orientation);
  MOVE(n_balls,balls,omatrix);
  MOVE(n_balls,balls,omega);

#undef MOVE

}


///
/// Physical Simulation Code
///

 /// Initialize Simulation
//

void
World::ball_setup_1(bool post_maria)
{
  // Arrange balls to form sort of tree.

  last_setup = 1;

  float ball_rad = 1;  // Warning: Re-assigned, changes behavior of nb.

  pCoor first_pos(17.2,ball_rad,-20.2);

  pVect dir_up(0.1,3,0.1);
```

```
  pVect dir_z(3*distance_relaxed,0,0);
  pVect dir_x(0,0,3*distance_relaxed);

  // Remove objects from the simulated objects lists, balls and links.
  // The delete operator is used on objects in the lists.
  //
  objects_erase();

  auto nb =
    [&] (pCoor pos) {
      Ball* const ball = new Ball;
      ball->position = pos;
      ball->locked = false;
      ball->velocity = pVect(0,0,0);
      ball->radius = ball_rad;
      ball->contact = false;
      balls += ball;
      return ball;
      };

  tail_ball = nb(first_pos);

  pCoor last_pos = tail_ball->position;

  Balls tail_balls;
  for ( int i=1; i<8; i++ )
    {
      ball_rad *= 0.8;
      Ball* const ball = nb(last_pos + (i==1?1.0:ball_rad) * dir_up);
      tail_balls += ball;
      links += link_new(balls[balls-2],ball,0.8);
      last_pos = ball->position;
    }

  head_ball = balls[balls-1];

  pNorm dir_zn(dir_z);
  pNorm dir_xn(dir_x);
  if ( !post_maria )
  for ( Ball *ball: tail_balls )
    {
      if ( ball == head_ball ) break;
      pNorm to_top(ball->position,head_ball->position);
      const float r = to_top.magnitude;
      int num_hairs = max(4,int(7*r));
      ball_rad = ball->radius / 4;
      const float delta_theta = 2 * M_PI / num_hairs;
      for ( int i=0; i<num_hairs; i++ )
        {
          const float theta = i * delta_theta;
          pCoor pos = ball->position +
            r * cos(theta) * dir_xn + r * sin(theta) * dir_zn;
          Ball* const ballr = nb(pos);
          links += link_new(ballr,ball);
        }
    }

  opt_head_lock = false;
  opt_tail_lock = true;
  setup_at_end();
}

void
World::ball_setup_2()
{
```

```
// Arrange balls to form a suspended thing.

last_setup = 2;

pCoor first_pos(17.2,14.8f,-20.2);

pVect dir_dn(0.5,first_pos.y/8,0.5);
pVect dir_z(3*distance_relaxed,0,0);
pVect dir_x(0,0,3*distance_relaxed);

// Remove objects from the simulated objects lists, balls and links.
// The delete operator is used on objects in the lists.
//
objects_erase();

float ball_rad = 0.3;  // Warning: Re-assigned, changes behavior of nb.

auto nb =
  [&] (pCoor pos) {
    Ball* const ball = new Ball;
    ball->position = pos;
    ball->locked = false;
    ball->velocity = pVect(0,0,0);
    ball->radius = ball_rad;
    ball->contact = false;
    balls += ball;
    return ball;
    };

head_ball = nb(first_pos);
for ( int j: {-1,1} )
  for ( int i: {-1,0,1} )
    {
      Ball* const ball = nb(first_pos - dir_dn + i * dir_x + j * dir_z );
      if ( i != 0 ) links += link_new(head_ball,ball);
      if ( i != -1 ) links += link_new(balls[balls-2],ball);
      if ( j == 1 && i != 0 )  links += link_new(balls[balls-4],ball);
    }

Ball* const tail_start = nb(first_pos - dir_dn);
links += link_new(balls[balls-6],tail_start);
links += link_new(balls[balls-3],tail_start);
pCoor last_pos = tail_start->position;
Balls tail_balls;
for ( int i=1; i<5; i++ )
  {
    Ball* const ball = nb(last_pos - 0.5 * dir_dn);
    tail_balls += ball;
    links += link_new(balls[balls-2],ball);
    last_pos = ball->position;
  }

tail_ball = balls[balls-1];

pNorm dir_zn(dir_z);
pNorm dir_xn(dir_x);
int num_hairs = 20;
ball_rad = 0.1;
for ( Ball *ball: tail_balls )
  {
    const float delta_theta = 2 * M_PI / num_hairs;
    for ( int i=0; i<num_hairs; i++ )
      {
        const float theta = i * delta_theta;
        const float r = ball->radius * 10;
```

```
          pCoor pos = ball->position +
            r * cos(theta) * dir_xn + r * sin(theta) * dir_zn;
          Ball* const ballr = nb(pos);
          links += link_new(ballr,ball);
        }
    }

  opt_head_lock = true;    // Head ball will be frozen in space.
  opt_tail_lock = false;   // Tail ball can move freely.
  setup_at_end();
}

void
World::ball_setup_3()
{
  last_setup = 3;
  objects_erase();

  pCoor first_pos(17.2,14.8f,-10.2);

  float ball_radius = 1;

  auto nb =
    [&] (pCoor pos) {
      Ball* const ball = new Ball;
      ball->position = pos;
      ball->locked = false;
      ball->velocity = pVect(0,0,0);
      ball->radius = ball_radius;
      ball->contact = false;
      balls += ball;
      return ball;
      };

  head_ball = nb(first_pos);
  ball_radius = ball_radius * .1;

  for ( int i=0; i<500; i++ )
    {
      pNorm dir(0.5-drand48(),0.5-drand48(),0.5-drand48());
      Ball* const ball = nb(first_pos + head_ball->radius * 3 * dir);
      links += link_new(ball,head_ball,2);
    }

  tail_ball = balls[1];

  opt_head_lock = false;    // Head ball will be frozen in space.
  opt_tail_lock = false;   // Tail ball can move freely.

  setup_at_end();
}


void
World::ball_setup_4()
{
  ball_setup_1(false);
  return;
  last_setup = 4;
  objects_erase();
  setup_at_end();
}

void
```

```
World::ball_setup_5()
{
  last_setup = 5;
  objects_erase();
  setup_at_end();
}


void
World::setup_at_end()
{
  lock_update();
  for ( uint i=0; i<balls.size(); i++ ) balls[i]->idx = i;
  for ( Link *l : links )
    { l->ball1_idx = l->ball1->idx; l->ball2_idx = l->ball2->idx; }
}


 /// Advance Simulation State by delta_t Seconds
//
void
World::time_step_cpu(double delta_t)
{
  const int n_balls = balls;
  const int n_links = links;

  // Smoothly move ball in response to user input.
  //
  if ( adj_t_stop )
    {
      const double dt = min(world_time,adj_t_stop) - adj_t_prev;
      pVect adj = dt/adj_duration * adj_vector;
      balls_translate(adj,0);
      adj_t_prev = world_time;
      if ( world_time >= adj_t_stop ) adj_t_stop = 0;
    }

#pragma omp parallel for
  for ( int bi=0; bi<n_balls; bi++ )
    {
      Ball* const ball = balls[bi];
      assert( ball->fdt_to_do );
      ball->force = ball->mass * gravity_accel;
      ball->torque = pVect(0,0,0);
    }

#pragma omp parallel for
  for ( int i=0; i<n_links; i++ )
    {
      Link* const link = links[i];
      if ( !link->is_simulatable ) continue;
      // Spring Force from Neighbor Balls
      //
      Ball* const ball1 = link->ball1;
      Ball* const ball2 = link->ball2;

      // Find position and velocity of the point where the link touches
      // the surface of ball 1 ...
      //
      pVect dir1 = ball1->omatrix * link->cb1;
      pCoor pos1 = ball1->position + dir1;
      pVect vel1 = ball1->velocity + ball1->point_rot_vel(dir1);

      // ... and ball 2.
```

```
      //
      pVect dir2 = ball2->omatrix * link->cb2;
      pCoor pos2 = ball2->position + dir2;
      pVect vel2 = ball2->velocity + ball2->point_rot_vel(dir2);

      // Construct a normalized (Unit) Vector from ball to neighbor
      // based on link connection points and ball centers.
      //
      pNorm link_dir(pos1,pos2);
      pNorm c_to_c(ball1->position,ball2->position);

      const float link_length = link_dir.magnitude;

      // Compute the speed of ball's end of link towards neighbor's end of link.
      //
      pVect delta_v = vel2 - vel1;
      float delta_s = dot( delta_v, link_dir );

      // Compute by how much the spring is stretched (positive value)
      // or compressed (negative value).
      //
      const float spring_stretch = link_length - link->distance_relaxed;

      // Determine whether spring is gaining energy (whether its length
      // is getting further from its relaxed length).
      //
      const bool gaining_e = ( delta_s > 0.0 ) == ( spring_stretch > 0 );

      // Use a smaller spring constant when spring is loosing energy,
      // a quick and dirty way of simulating energy loss due to spring
      // friction.
      //
      const float spring_constant =
        gaining_e ? opt_spring_constant : opt_spring_constant * 0.7;

      const float force_mag = spring_constant * spring_stretch;
      pVect spring_force_12 = force_mag * link_dir;

      // Apply forces affecting linear momentum.
      //
      link->spring_force_12 = spring_force_12;

      if ( ! link->is_surface_connection ) continue;

      // Apply torque.
      //
      link->torque1 = cross(pNorm(dir1), spring_force_12);
      link->torque2 = cross(pNorm(dir2), spring_force_12);
    }

  // Note: Because two links can reference the same ball this should
  // not be done in parallel.
  for ( Link *link: links )
    {
      if ( !link->is_simulatable ) continue;
      Ball* const ball1 = link->ball1;
      Ball* const ball2 = link->ball2;
      ball1->force += link->spring_force_12;
      ball2->force -= link->spring_force_12;
      ball1->torque += link->torque1;
      ball2->torque -= link->torque2;
    }

  ///
  /// Update Position of Each Ball
```

```
  ///

#pragma omp parallel for
  for ( int bi=0; bi<n_balls; bi++ )
    {
      Ball* const ball = balls[bi];
      if ( ball->locked )
        {
          ball->velocity = pVect(0,0,0);
          ball->omega = pVect(0,0,0);
          continue;
        }

      // Update Velocity
      //
      // This code assumes that force on ball is constant over time
      // step. This is clearly wrong when balls are moving with
      // respect to each other because the springs are changing
      // length. This inaccuracy will make the simulation unstable
      // when spring constant is large for the time step.
      //
      const float mass = max( ball->mass, ball->mass_min );

      pVect delta_v = ( ball->force / mass ) * delta_t;

      if ( platform_collision_possible(ball->position) && ball->position.y < 0 )
        {
          const float spring_constant_plat =
            ball->velocity.y < 0 ? 100000 : 50000;
          const float fric_coefficient = 0.1;
          const float force_up = -ball->position.y * spring_constant_plat;
          const float delta_v_up = force_up / mass * delta_t;
          const float fric_force_mag = fric_coefficient * force_up;
          pNorm surface_v(ball->velocity.x,0,ball->velocity.z);
          const float delta_v_surf = fric_force_mag / mass * delta_t;

          if ( delta_v_surf > surface_v.magnitude )
            {
              // Ignoring other forces?
              delta_v =
                pVect(-ball->velocity.x,delta_v.y,-ball->velocity.z);
            }
          else
            {
              delta_v -= delta_v_surf * surface_v;
            }
          delta_v.y += delta_v_up;
        }

      ball->velocity += delta_v;

      // Air Resistance
      //
      const double fs = pow(1+opt_air_resistance,-delta_t);
      ball->velocity *= fs;

      // Update Position
      //
      // Assume that velocity is constant.
      //
      ball->position += ball->velocity * delta_t;

      ball->omega += delta_t * ball->fdt_to_do * ball->torque;

      pNorm axis(ball->omega);
```

```
      // Update Orientation
      //
      // If ball isn't spinning fast skip expensive rotation.
      //
      if ( axis.mag_sq < 0.000001 ) continue;

      // Update ball orientation.
      //
      ball->orientation_set
        ( pQuat(axis,delta_t * axis.magnitude) * ball->orientation );
    }
}

bool
World::platform_collision_possible(pCoor pos)
{
  // Assuming no motion in x or z axes.
  //
  return pos.x >= platform_xmin && pos.x <= platform_xmax
    && pos.z >= platform_zmin && pos.z <= platform_zmax;
}

 /// External Modifications to State
//
//   These allow the user to play with state while simulation
//   running.

// Move the ball.
//
void Ball::translate(pVect amt) {position += amt;}

// Add velocity to the ball.
//
void Ball::push(pVect amt) {velocity += amt;}

// Set the velocity to zero.
//
void Ball::stop() {velocity = pVect(0,0,0); }

// Set the velocity and rotation (not yet supported) to zero.
//
void Ball::freeze() {velocity = pVect(0,0,0); }



void World::balls_translate(pVect amt,int b){head_ball->translate(amt);}
void World::balls_push(pVect amt,int b){head_ball->push(amt);}
void World::balls_translate(pVect amt)
{ for ( Ball *ball: balls ) ball->translate(amt); }
void World::balls_push(pVect amt)
{ for ( Ball *ball: balls ) ball->push(amt); }
void World::balls_stop()
{ for ( Ball *ball: balls ) ball->stop(); }
void World::balls_freeze(){balls_stop();}


void
World::render_p1()
{
  /// Problem 1

  ///
```

```
/// Code adapted from demo-7-vtx-arrays.cc
///
if ( sphere_coords.empty() )
  //
  // Note: sphere_coords is automatically emptied when variables
  // affecting its contents change.
  {
    /// Initialize array coords with sphere's coordinates.
    //
    /// Problem 1 Note:
    //
    //  There is no need to modify the code in this block.

    const double delta_eta = M_PI / opt_slices;

    for ( double eta = 0; eta < M_PI - 0.0001 - delta_eta; eta += delta_eta )
      {
        const double eta1 = eta + delta_eta;
        const float  y0 = cos(eta),        y1 = cos(eta1);
        const double slice_r0 = sin(eta),  slice_r1 = sin(eta1);
        const double delta_theta = delta_eta * slice_r1;

        // Vertex 1 and 2
        sphere_coords.push_back( pCoor( slice_r1, y1, 0 ) );
        sphere_coords.push_back( pCoor( slice_r0, y0, 0 ) );

        for ( double theta = 0; theta < 2 * M_PI; theta += delta_theta )
          {
            const double theta1 = theta + delta_theta;
            // Vertex 3 and 4.
            sphere_coords.push_back
              ( pCoor( slice_r1 * cos(theta1), y1, slice_r1 * sin(theta1) ) );
            sphere_coords.push_back
              ( pCoor( slice_r0 * cos(theta1), y0, slice_r0 * sin(theta1) ) );
          }
      }

    // Set up or update a buffer object to hold coordinates.
    //
    // Generate buffer id (name), if necessary.
    //
    if ( !sphere_bo ) glGenBuffers(1,&sphere_bo);

    // Tell GL that subsequent array pointers refer to this buffer.
    //
    glBindBuffer(GL_ARRAY_BUFFER, sphere_bo);

    // Copy data into buffer.
    //
    glBufferData
      (GL_ARRAY_BUFFER,           // Kind of buffer object.
       // Number of bytes to copy.
       sphere_coords.size()*sizeof(sphere_coords[0]),
       sphere_coords.data(),      // Pointer to data to copy.
       GL_STATIC_DRAW);           // Hint about who, when, how accessed.

    // Tell GL that subsequent array pointers refer to host storage.
    //
    glBindBuffer(GL_ARRAY_BUFFER, 0);
  }

/// SOLUTION STARTS
const int coords_size = sphere_coords.size();

/// SOLUTION
```

```
  //  Set up vertex and normal sources for sphere. This is
  //  done only once and used for all of the balls.

  // Tell GL that subsequent array pointers refer to this buffer.
  //
  glBindBuffer(GL_ARRAY_BUFFER,sphere_bo);

  // Specify array of vertices.
  //
  glVertexPointer( 3, GL_FLOAT, sizeof(pCoor), NULL);
  //               N  Type      Stride         Pointer

  glEnableClientState(GL_VERTEX_ARRAY);

  // Ditto. Note that vertices and normals read from same buffer.
  //
  glNormalPointer(GL_FLOAT,sizeof(sphere_coords[0]),NULL);
  glEnableClientState(GL_NORMAL_ARRAY);

  glMatrixMode(GL_MODELVIEW);

  for ( Ball *ball: balls )
    {
      pColor color;
      if ( ball->contact )
        color = color_gray;
      else if ( ball->mass > 0 && ball->mass < ball->mass_min )
        color = color_red;
      else if ( ball->mass > 0 && ball->locked )
        color = color_pale_green;
      else
        color = ball->color;

      float radius = ball->radius;
      pCoor center(ball->position);

      pMatrix_Rotation rot(ball->orientation);
      //
      // rot is the orientation of the ball.

      /// SOLUTION

      glColor3fv(color);

      glPushMatrix();
      glTranslatef(center.x,center.y,center.z);
      glMultTransposeMatrixf(rot);
      glScalef(radius,radius,radius);
      glDrawArrays(GL_TRIANGLE_STRIP,0,coords_size);
      glPopMatrix();
    }

  // Tell GL that subsequent array pointers refer to host storage.
  //
  glBindBuffer(GL_ARRAY_BUFFER,0);
  // Don't forget to do this, since it affects glVertexPointer and
  // related calls.

  glDisableClientState(GL_NORMAL_ARRAY);
  glDisableClientState(GL_VERTEX_ARRAY);

}

void
World::render_p2()
```

```
{
  const double omega = opt_omega;

  /// SOLUTION -- Problem 2

  const double delta_theta = 2 * M_PI / opt_spirals;

  if ( sphere_coords.empty() )
    //
    // Note: sphere_coords is automatically emptied when variables
    // affecting its contents change.
    {
      const double delta_t = 1.0 / opt_slices;

      // Vertex 1
      sphere_coords.push_back( pCoor( 0, 1, 0 ) );

      // SOLUTION
      // Note that loop starts at 1 because "north pole" vertex added above.

      for ( int i=1; i<=opt_slices; i++ )
        {
          const double t = i * delta_t;
          const double srt = pow(t,0.5);
          const double theta0 = srt * omega;
          const double theta1 = theta0 + delta_theta;
          const double eta = srt * M_PI / 2;
          // Note that r and y computed once and used multiple times.
          const double r = sin(eta);
          const double y = cos(eta);
          pCoor pt0(r * cos(theta0), y, r * sin(theta0));
          pCoor pt1(r * cos(theta1), y, r * sin(theta1));
          sphere_coords.push_back(pt1);
          sphere_coords.push_back(pt0);
        }

      // Set up or update a buffer object to hold coordinates.
      //
      // Generate buffer id (name), if necessary.
      //
      if ( !sphere_bo ) glGenBuffers(1,&sphere_bo);

      // Tell GL that subsequent array pointers refer to this buffer.
      //
      glBindBuffer(GL_ARRAY_BUFFER, sphere_bo);

      // Copy data into buffer.
      //
      glBufferData
        (GL_ARRAY_BUFFER,           // Kind of buffer object.
         // Number of bytes to copy.
         sphere_coords.size()*sizeof(sphere_coords[0]),
         sphere_coords.data(),      // Pointer to data to copy.
         GL_STATIC_DRAW);           // Hint about who, when, how accessed.

      // Tell GL that subsequent array pointers refer to host storage.
      //
      glBindBuffer(GL_ARRAY_BUFFER, 0);
    }

  /// SOLUTION -- Problem 2
  //    The code above and below has been modified, rearranged, etc. as
  //    part of the solution.

  const int coords_size = sphere_coords.size();
```

```
// Tell GL that subsequent array pointers refer to this buffer.
//
glBindBuffer(GL_ARRAY_BUFFER,sphere_bo);

// Specify array of vertices.
//
glVertexPointer( 3, GL_FLOAT, sizeof(pCoor), NULL);
//                N  Type      Stride         Pointer

glEnableClientState(GL_VERTEX_ARRAY);

// Ditto. Note that vertices and normals read from same buffer.
//
glNormalPointer(GL_FLOAT,sizeof(sphere_coords[0]),NULL);
glEnableClientState(GL_NORMAL_ARRAY);

// be sent to the GPU just once and used for all subsequent
// vertices.
//
glColor3fv(color_lsu_spirit_gold);

glMatrixMode(GL_MODELVIEW);

/// SOLUTION
//  Pre-compute rotation matrices since these are expensive to compute.
//
pMatrix_Rotation rot_xz( pVect(0,1,0), delta_theta );
pMatrix_Rotation rot_yz( pVect(1,0,0), M_PI );

///
/// Code extracting color, position, and orientation of each ball.
///
for ( Ball *ball: balls )
  {
    pColor color;
    if ( ball->contact )
      color = color_gray;
    else if ( ball->mass > 0 && ball->mass < ball->mass_min )
      color = color_red;
    else if ( ball->mass > 0 && ball->locked )
      color = color_pale_green;
    else
      color = ball->color;

    float radius = ball->radius;
    pCoor center(ball->position);

    pMatrix_Rotation rot(ball->orientation);
    //
    // rot is the orientation of the ball.

    // Save matrix just once per ball.
    glPushMatrix();
    glTranslatef(center.x,center.y,center.z);
    glScalef(radius,radius,radius);
    glMultTransposeMatrixf(rot);

    // Rather than re-compute the entire transformation matrix for
    // each spiral, the inner loop below just rotates the existing
    // transformation matrix around the (local) y axis by
    // delta_theta. The outer loop multiplies the transformation
    // matrix by a rotation around the (local) x axis.
    //
    for ( int j = 0; j < 2; j++ )
```

```
        {
          for ( int i=0; i<opt_spirals; i++ )
            {
              glColor3fv( i & 1 ? color_lsu_spirit_purple : color );
              glDrawArrays(GL_TRIANGLE_STRIP,0,coords_size);

              // SOLUTION: Rotate by delta_theta each iteration.
              glMultTransposeMatrixf(rot_xz);
              if ( opt_tryout1 ) break;
            }
          if ( opt_tryout1 ) break;
          glMultTransposeMatrixf(rot_yz);
        }


      glPopMatrix();
    }

  // Tell GL that subsequent array pointers refer to host storage.
  //
  glBindBuffer(GL_ARRAY_BUFFER,0);
  // Don't forget to do this, since it affects glVertexPointer and
  // related calls.

  glDisableClientState(GL_NORMAL_ARRAY);
  glDisableClientState(GL_VERTEX_ARRAY);
}


void
World::render_link_start()
{
  if ( world_time_link_update == world_time ) return;

  if ( lis_pvects.size() == 0 )
    {
      lis_pvects += &lis_pos1;
      lis_pvects += &lis_pos2;

      lis_pvects += &lis_v1;
      lis_pvects += &lis_v2;
      lis_pvects += &lis_b1_ydir;
      lis_pvects += &lis_b2_ydir;
      lis_pvects += &lis_tex_rad;
      const int n_pvects = lis_pvects.size();
      lis_bos = new GLuint[n_pvects];
      glGenBuffers(n_pvects,lis_bos);
    }

  // Empty containers in preparation for a new pass.
  for ( auto vl: lis_pvects ) vl->clear();
}

void
World::render_link_gather(Link *link)
{
  // Place data needed to render link (the argument to this function)
  // in arrays. Just before rendering those arrays will be placed in
  // buffer objects.


  // If we've already collected links, just return.  This happens
  // when doing multiple passes to render shadows and reflections.
  //
  if ( world_time_link_update == world_time ) return;
```

```
  Ball *const ball1 = link->ball1;
  Ball *const ball2 = link->ball2;

  // Direction of link relative to center off ball.
  //
  pVect dir1 = ball1->omatrix * link->cb1;

  // Place on surface of ball where link connects.
  //
  pCoor pos1 = ball1->position + dir1;

  // Direction and connection location for ball 2.
  //
  pVect dir2 = ball2->omatrix * link->cb2;
  pCoor pos2 = ball2->position + dir2;

  pVect p1p2(pos1,pos2);
  pNorm p1p2n(p1p2);

  // Radius of link.
  //
  const float rad = ball1->radius * 0.3;

  // Compute scale factors for texture.
  const float tex_aspect_ratio = 8.5 / 11;  // Yes, inches. Can't remember mm.
  const float page_width_o = 2.5;  // Width of texture in object-space coords.
  const float tex_t_scale = link->distance_relaxed / page_width_o;
  const float tex_angle_scale = rad * tex_aspect_ratio / page_width_o;

  pNorm dirn1(dir1);
  pNorm dirn2(dir2);

  // Vectors used to describe the cubic Hermite curve.
  //
  pVect v1 = p1p2n.magnitude * dirn1;
  pVect v2 = p1p2n.magnitude * dirn2;

  // Convert link's local x and y axes to global coordinates.
  //
  pVect b1_ydir = ball1->omatrix * link->b1_dir;
  pVect b2_ydir = ball2->omatrix * link->b2_dir;


  // Place data needed to render links in lists. Note that
  // the += operator appends the right-hand side to the container
  // on the left-hand side.
  //
  lis_pos1 += pos1;
  lis_pos2 += pos2;
  lis_v1 += v1;
  lis_v2 += v2;
  lis_b1_ydir += b1_ydir;
  lis_b2_ydir += b2_ydir;
  pVect misc(tex_t_scale,tex_angle_scale,rad);
  lis_tex_rad += misc;
}



void
World::render_link_render(bool shadow_volumes)
{
  // Perform a rendering pass to render all the links collected
  // by the routine render_link_2.
```

```
  // Check whether this is the first time we are rendering this data.
  //
  const bool first_render = world_time_link_update != world_time;
  world_time_link_update = world_time;

  if ( shadow_volumes ) sp_set_2_sv->use(); else sp_set_2->use();

  // Number of segments used to construct link.  Each segment is
  // approximately a cylinder.
  //
  const int segments = shadow_volumes ? max(2,opt_segments/4) : opt_segments;

  // Number of sides of each cylinder.
  //
  const int sides = opt_sides;

  const float delta_tee = 1.0 / segments;
  const size_t n_instances = lis_pos1.size();

  // Bind buffer objects holding link and ball info to locations
  // specified in file links-shdr-links.cc
  //
  for ( size_t i=0; i<lis_pvects.size(); i++ )
    {
      GLuint bo = lis_bos[i];
      assert( lis_pvects[i]->size() == n_instances );
      glBindBuffer(GL_ARRAY_BUFFER,bo);
      if ( first_render )
        glBufferData
          (GL_ARRAY_BUFFER, lis_pvects[i]->size_chars(),
           lis_pvects[i]->data(), GL_STREAM_DRAW);
      glBindBufferBase(GL_SHADER_STORAGE_BUFFER,i+1,bo);
    }
  glBindBuffer(GL_ARRAY_BUFFER,0);

  glUniform4fv(1, 1, color_dim_gray ); // color_front
  glUniform4fv(2, 1, color_salmon ); // color_back
  glUniform2f(4, sides, 0 );
  glUniform1f(5, delta_tee );

  GLboolean l0, l1;
  glGetBooleanv(GL_LIGHT0,&l0);
  glGetBooleanv(GL_LIGHT1,&l1);
  int light_state = l0 | (l1<<1);
  glUniform2i(6, light_state, segments );
  glUniform3i(3, opt_tryout1, opt_tryout2, opt_normal_sphere);

  glDrawArraysInstanced
    (GL_LINE_STRIP, 0, segments+1, n_instances);

  sp_fixed->use();
}


void
World::frame_callback()
{
  // This routine called whenever window needs to be updated.

  frame_timer.phys_start();
  const double time_now = time_wall_fp();
  const int time_step_count_start = time_step_count;
  const int nmp = gpu_info.cuda_prop.multiProcessorCount;
```

```
  int bl_p_sm_max = 1024;
  for ( int i=0; i<gpu_info.num_kernels; i++ )
    set_min
      (bl_p_sm_max,
       gpu_info.get_max_active_blocks_per_mp(i, opt_block_size, 0) );
  const int grid_size = bl_p_sm_max * nmp;

  const bool opt_cuda = opt_physics_method != GP_cpu;
  if ( opt_cuda )
    {
      CE( cudaEventRecord(frame_start_ce,0) );
      data_cpu_to_gpu_constants();
      data_cpu_to_gpu_dynamic();
    }

  if ( !opt_pause || opt_single_frame || opt_single_time_step )
    {
      /// Advance simulation state.

      // Amount of time since the user saw the last frame.
      //
      const double wall_delta_t = time_now - last_frame_wall_time;

      // Compute amount by which to advance simulation state for this frame.
      //
      const double duration =
        opt_single_time_step ? opt_time_step_duration :
        opt_single_frame ? 1/30.0 :
        wall_delta_t;

      const double world_time_target = world_time + duration;
      const double wall_time_limit = time_now + 0.05;

      while ( world_time < world_time_target )
        {
          if ( opt_cuda )
            {
              launch_time_step(opt_time_step_duration,grid_size,opt_block_size);
            }
          else
            {
              time_step_cpu(opt_time_step_duration);
            }
          time_step_count++;
          world_time += opt_time_step_duration;
          const double time_right_now = time_wall_fp();
          if ( time_right_now > wall_time_limit ) break;
        }

      // Reset these, just in case they were set.
      //
      opt_single_frame = opt_single_time_step = false;
    }

  if ( opt_cuda )
    {
      data_gpu_to_cpu_dynamic();
      CE( cudaEventRecord(frame_stop_ce,0) );
    }

  frame_timer.phys_end();
  frame_timer.work_amt_set(time_step_count-time_step_count_start);

  float cuda_time = 0;
  if ( opt_cuda )
```

```
    {
      CE(cudaEventSynchronize(frame_stop_ce));
      CE(cudaEventElapsedTime(&cuda_time,frame_start_ce,frame_stop_ce));
    }
  frame_timer.cuda_frame_time_set(cuda_time);

  last_frame_wall_time = time_now;

  render();
}

int
main(int argv, char **argc)
{
  pOpenGL_Helper popengl_helper(argv,argc);
  World world(popengl_helper,argv,argc);

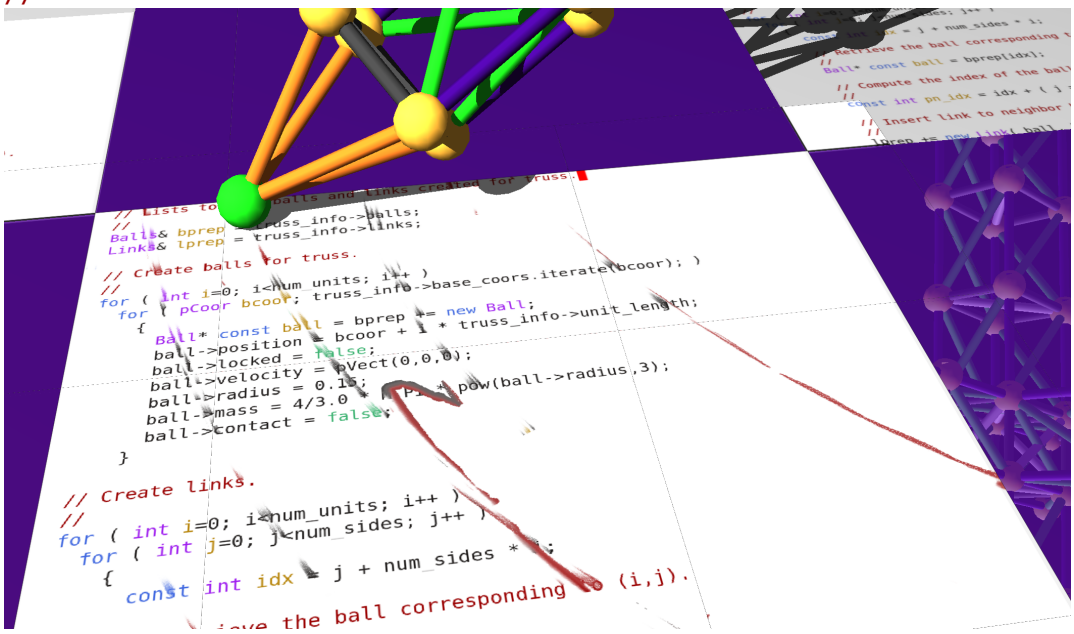
  glDisable(GL_DEBUG_OUTPUT);
  glDebugMessageControl(GL_DONT_CARE,GL_DONT_CARE,
                        GL_DEBUG_SEVERITY_NOTIFICATION,0,NULL,false);

  popengl_helper.rate_set(30);
  popengl_helper.display_cb_set(world.frame_callback_w,&world);
}
```

# LSU EE 4702-1 (Fall 2017), GPU Programming

## Homework 3 -- SOLUTION

Search for SOLUTION in this file to find changes.
In git use git diff to find changes.



## Instructions

Read the assignment: http://www.ece.lsu.edu/koppel/gpup/2017/hw03.pdf

#if 0

# CODE OUTLINE

## Code Outline

- Preparing a Texture
- Division of Platform into Overlays (Use many small textures.)
- Coordinate Spaces: Object (balls) to Overlays to Texture Pixels

## Code Organization

## Platform_Overlay

```
Platform_Overlay* platform_overlays;  // Array of overlays.
```

A data structure holding info about texture that covers part of platform.

Platform is covered by nx * ny (default 40*40) overlays ..
.. arranged in a grid.

Initialized in My_Piece_Of_The_World::init():

## Overlay Rationale

We expect to update texture on CPU and send it back go GPU.

This may happen every frame.

```
//  Suppose minimum scuff size is 1/100 of a tile:
//    19 * 19 platform tiles  * 100 * 100  * 24 bytes / texel
//    = 86640000 B
//
//  Recall: PCIe x16:  15.8 GB/s
//  Suppose we want a frame rate of 60 f/s
//   15.8 10^9 / ( 60 * 86640000 ) = 3.04
//   About 1/3 of the capacity will be used for texture updates.



#endif
/// Purpose
//
//   Demonstrate simulation of point masses connected by springs.

/// What Code Does

// Simulates balls connected by springs over a platform. Balls and
// springs can be initialized in different arrangements (called
// scenes). Currently scene 1 is a simple string of beads, and scenes
// 2, 3, and 4 are trusses. The platform consists of tiles, some are
// purple-tinted mirrors (showing a reflection of the ball), the
// others show the course syllabus.


///  Keyboard Commands
 //
 /// Object (Eye, Light, Ball) Location or Push
 //   Arrows, Page Up, Page Down
 //        Move object or push ball, depending on mode.
 //        With shift key pressed, motion is 5x faster.
 //   'e': Move eye.
 //   'l': Move light.
 //   'b': Move head (first) ball. (Change position but not velocity.)
 //   'B': Push head ball. (Add velocity.)
 //
 /// Eye Direction
 //   Home, End, Delete, Insert
 //   Turn the eye direction.
 //   Home should rotate eye direction up, End should rotate eye
 //   down, Delete should rotate eye left, Insert should rotate eye
 //   right.  The eye direction vector is displayed in the upper left.

 /// Simulation Options
 //  (Also see variables below.)
 //
 //  'c'    Clean the platform.
 //  'w'    Twirl balls around axis formed by head and tail.
 //  '1'    Set up scene 1.
 //  '2'    Set up scene 2.
 //  '3'    Set up scene 3.
 //  'p'    Pause simulation. (Press again to resume.)
 //  ' '    (Space bar.) Advance simulation by 1/30 second.
 //  'S- '  (Shift-space bar.) Advance simulation by one time step.
 //  'h'    Freeze position of first (head) ball. (Press again to release.)
 //  't'    Freeze position of last (tail) ball. (Press again to release.)
 //  's'    Stop balls.
 //  'g'    Turn gravity on and off.
 //  'y'    Toggle value of opt_tryout1. Intended for experiments and debugging.
```

```
//  'Y'    Toggle value of opt_tryout2. Intended for experiments and debugging.
//  'F11'  Change size of green text (in upper left).
//  'F12'  Write screenshot to file.

/// Variables
//   Selected program variables can be modified using the keyboard.
//   Use "Tab" to cycle through the variable to be modified, the
//   name of the variable is displayed next to "VAR" on the bottom
//   line of green text.

//  'Tab' Cycle to next variable.
//  '`'   Cycle to previous variable.
//  '+'   Increase variable value.
//  '-'   Decrease variable value.
//
//  VAR Spring Constant - Set spring constant.
//  VAR Time Step Duration - Set physics time step.
//  VAR Air Resistance - Set air resistance.
//  VAR Light Intensity - The light intensity.
//  VAR Gravity - Gravitational acceleration. (Turn on/off using 'g'.)


#define GL_GLEXT_PROTOTYPES
#define GLX_GLXEXT_PROTOTYPES

#include <GL/gl.h>
#include <GL/glext.h>
#include <GL/glx.h>
#include <GL/glxext.h>
#include <GL/glu.h>
#include <GL/freeglut.h>

#include <Magick++.h>

#include <gp/util.h>
#include <gp/glextfuncs.h>
#include <gp/coord.h>
#include <gp/shader.h>
#include <gp/pstring.h>
#include <gp/misc.h>
#include <gp/gl-buffer.h>
#include <gp/texture-util.h>
#include <gp/colors.h>
#include <gp/util-containers.h>
#include "shapes.h"

// Make ImageMagick symbols visible.
using namespace Magick;


///
/// Main Data Structures
///
//
// class World: All data about scene.


class World;


class Platform_Overlay {
public:
  Platform_Overlay():data(NULL){}
```

```
  // Note: When using the array below as an argument to glTexImage2D
  // use data format GL_RGBA and type GL_FLOAT.
  //
  pColor *data;
  GLuint txid;
  float overlay_xmin, overlay_zmin;

  bool texture_modified;
  bool texture_object_initialized;

  pCoor vertices[4];

};

//   Use this class to define variables and member functions.
//   Don't modify hw03-graphics.cc.
//
class My_Piece_Of_The_World {
public:
  My_Piece_Of_The_World(World& wp):w(wp){};
  World& w;
  Platform_Overlay* platform_overlays;  // Array of overlays.
  Platform_Overlay sample_overlay;
  int nx, nz;           // Number of overlays along each dimension.
  int num_overlays;
  int twid_x, twid_z;  // Dimensions of each texture in texels.
  int num_texels;
  float wid_x, wid_z;  // Width of each overlay in object space units.
  float wid_x_inv, wid_z_inv;  // Their inverses.

  float scale_x_obj_to_texel;
  float scale_z_obj_to_texel;

  int syl_wd, syl_ht;
  vector<PixelPacket> syl_pixels;

  void init();
  void sample_tex_make();

  // Return platform tile coordinate in x and z components. Tile
  // coordinates range from 0 to platform_tile_count in the x and z
  // directions. For example, (0.1,0,3.5) indicates that the
  // coordinate is near the left (low x) side of the first tile in the
  // x direction and in the middle of the fourth tile in the z
  // direction.
  //
  // If w component is 1 then tile shows texture (syllabus),
  // if w component is 0 then tile is mirrored.
  //
  pCoor platform_obj_to_tile(pCoor obj_coord);

  float opt_smear_factor;

  pCoor platform_tile_to_obj(pCoor tile_coord);

  pCoor overlay_obj_to_ovr(pCoor pos_obj);

  // Return the platform overlay that includes pos, or NULL if pos is
  // not on platform.
  //
  Platform_Overlay* po_get(pCoor pos);

  // Convert object space coordinate to texel coordinate relative
  // to overlay po.
```

```
  pCoor overlay_obj_to_tex(Platform_Overlay *po, pCoor pos);

  void render();
  void clean();
};


// Object Holding Ball State
//
class Ball {
public:
  Ball():velocity(pVect(0,0,0)),locked(false),
         color(color_lsu_spirit_gold),contact(false){};
  pCoor position;
  pVect velocity;

  float mass;
  float mass_min; // Mass below which simulation is unstable.
  float radius;

  bool locked;

  pVect force;
  pColor color;
  bool contact;                  // When true, ball rendered in gray.
  float spring_constant_sum;     // Used to compute minimum mass.

  vector<pColor> glop;

  void push(pVect amt);
  void translate(pVect amt);
  void stop();
  void freeze();
};

class Link {
public:
  Link(Ball *b1, Ball *b2):ball1(b1),ball2(b2),
     distance_relaxed(pDistance(b1->position,b2->position)), snapped(false),
     natural_color(color_lsu_spirit_purple),color(color_lsu_spirit_purple){}
  Link(Ball *b1, Ball *b2, pColor colorp):ball1(b1),ball2(b2),
     distance_relaxed(pDistance(b1->position,b2->position)), snapped(false),
     natural_color(colorp),color(colorp){}
  Ball* const ball1;
  Ball* const ball2;
  float distance_relaxed;
  bool snapped;
  pColor natural_color;
  pColor color;
};

// Declare containers and iterators for Balls and Links.
// (See util_container.h.)
//
typedef pVectorI<Link> Links;
typedef pVectorI<Ball> Balls;

struct Truss_Info {

  // See make_truss for a description of what the members are for.

  // Inputs
  PStack<pCoor> base_coors;  // Coordinates of first set of balls.
```

```
  pVect unit_length;
  int num_units;

  // Output
  Balls balls;
  Links links;
};

#include "hw03-graphics.cc"

void
My_Piece_Of_The_World::init()
{
  // Number of overlays along each direction.
  //
  nx = 40; nz = 40;
  num_overlays = nx * nz;

  // Number of texels along each dimension of each overlay's texture.
  //
  twid_x = 512; twid_z = 512;
  num_texels = twid_x * twid_z;

  sample_tex_make();

  platform_overlays = new Platform_Overlay[num_overlays];

  // Dimensions of overlay in object-space coordinates.
  //
  wid_x = ( w.platform_xmax - w.platform_xmin ) / nx;
  wid_z = ( w.platform_zmax - w.platform_zmin ) / nz;
  wid_x_inv = 1.0 / wid_x;
  wid_z_inv = 1.0 / wid_z;

  //  Compute variables that will come in handy when converting
  //  from object space coordinates to texel coordinates (for our overlay).
  //
  scale_x_obj_to_texel = wid_x_inv * twid_x;
  scale_z_obj_to_texel = wid_z_inv * twid_z;

  /// SOLUTION - Minor Change - Choose a better looking smear factor.
  opt_smear_factor = 1.43;
  w.variable_control.insert(opt_smear_factor,"Smear Factor");

}

void
My_Piece_Of_The_World::sample_tex_make()
{
  /// Homework 3 -- Sample Code

  // Code in this routine creates a texture with a big red X, and
  // loads it into a texture object.  The texture object can
  // be used as a substitute for the "scuffed" texture before
  // the scuffed texture part of this assignment is finished.

  Platform_Overlay* const po = &sample_overlay;

  // Allocate storage for the array of texels.
  //
  if ( !po->data ) po->data = new pColor[ num_texels ];

  // Initialize the texels to black and transparent. (Alpha = 0)
  //
  memset(po->data,0,num_texels*sizeof(po->data[0]));
```

```
//
// Put syllabus in part of texture.
//

// ImageMagick C++ Binding Documentation:
//     http://www.imagemagick.org/Magick++/Image++.html
Magick::Image image(w.platform_tex_path);

const int img_wd = image.columns();
const int img_ht = image.rows();
if ( img_wd )
  {
    // Image was read, include it in texture.

    syl_wd = img_wd * ( w.platform_trmax - w.platform_trmin );
    syl_ht = img_ht * ( w.platform_tsmax - w.platform_tsmin );
    const int cor_c = img_wd * w.platform_trmin;
    const int cor_r = img_ht * w.platform_tsmin;

    PixelPacket* pp = image.getPixels(cor_c,cor_r,syl_wd,syl_ht);
    syl_pixels.resize(syl_wd*syl_ht);
    for ( int c = 0;  c < syl_wd;  c++ )
      for ( int r = 0;  r < syl_ht;  r++ )
        syl_pixels[ syl_wd - c - 1 + ( syl_ht -r - 1 ) * syl_wd ]
          = pp[ c + r * syl_wd ];

    const double ixtx = syl_wd * 2.0 / twid_x;
    const double iytz = syl_ht * 2.0 / twid_z;
    const double sc = 1.0 / MaxRGB;
    for ( int tx = 0;  tx < twid_x / 2;  tx++ )
      for ( int tz = 0;  tz < twid_z / 2;  tz++ )
        {
          int ix = tx * ixtx;
          int iy = tz * iytz;
          PixelPacket p = syl_pixels[ix + iy * syl_wd];
          po->data[tx + tz * twid_x] =
            pColor( p.red * sc, p.green * sc, p.blue * sc );
        }
  }

// Thickness of the strokes making up the letter ex.
//
const int thickness = max(2, twid_x/10);

// Write the letter ex, a big one, in the texture.
//
for ( int tx=0; tx<twid_x-thickness; tx++ )
  {
    // Note: Compute tz without assuming twid_x == twid_z.
    int tz_raw = float(tx)/twid_x * twid_z;
    int tz = min(tz_raw,twid_z-1);

    // Array index of texel at (tx,tz).
    //
    int idx  = tx              + tz * twid_x; // Lower left to upper right.
    int idx2 = twid_x - 1 - tx + tz * twid_x; // Lower right to upper left.

    // Write colors to texels.
    //
    for ( int i=0; i<thickness; i++ )
      for ( pColor* c: { &po->data[idx+i], &po->data[idx2-i] } )
        {
          *c = 0.5 * ( *c + color_red );
          c->a = 1;
```

```
            }
        }

    // Create a new texture object.
    //
    glGenTextures(1,&po->txid);

    // Make our new texture object the current texture object.
    //
    glBindTexture(GL_TEXTURE_2D,po->txid);
    //
    // Subsequent OpenGL calls operating on GL_TEXTURE_2D ..
    // .. will now operate on po->txid, our new texture object.

    // Tell OpenGL to generate MIPMAP levels for us.
    //
    glTexParameteri(GL_TEXTURE_2D, GL_GENERATE_MIPMAP, 1);

    // Load our texture into the texture object.
    //
    glTexImage2D
      (GL_TEXTURE_2D,
       0,                 // Level of Detail (0 is base).
       GL_RGBA,           // Internal format to be used for texture.
       twid_x, twid_z,
       0,                 // Border
       GL_RGBA,           // GL_BGRA: Format of data read by this call.
       GL_FLOAT,          // Size of component.
       (void*)po->data    // Pointer to the texture data.
       );
}

pCoor
My_Piece_Of_The_World::platform_obj_to_tile(pCoor pos_obj)
{
  /// Problem 1 Solution Goes Here

  const float x = ( pos_obj.x - w.platform_xmin ) / w.platform_tile_sz_x;
  const float z = ( pos_obj.z - w.platform_zmin ) / w.platform_tile_sz_z;
  pCoor rv(x,0,z);

  // Make sure that conversion is correct by applying the inverse conversion.
  pCoor check = platform_tile_to_obj(rv);
  assert( fabs(pos_obj.x - check.x) < 0.0001 );
  assert( fabs(pos_obj.z - check.z) < 0.0001 );

  bool mirrored = ( int(x) + int(z) * w.platform_tile_count ) & 1;
  rv.w = !mirrored;

  return rv;
}

pCoor
My_Piece_Of_The_World::platform_tile_to_obj(pCoor tile_coord)
{
  pCoor rv =
    pVect(w.platform_xmin, 0, w.platform_zmin) +
    pCoor( tile_coord.x * w.platform_tile_sz_x,
           tile_coord.y,
           tile_coord.z * w.platform_tile_sz_z );

  return rv;
}
```

```
pCoor
My_Piece_Of_The_World::overlay_obj_to_tex(Platform_Overlay *po, pCoor pos_obj)
{
  pCoor lc;
  // Convert object space coordinates to texel coordinates.
  // Variables overlay_xmin and overlay_zmin were set when the overlay
  // was retrieved.
  //
  lc.x = ( pos_obj.x - po->overlay_xmin ) * scale_x_obj_to_texel;
  lc.z = ( pos_obj.z - po->overlay_zmin ) * scale_z_obj_to_texel;
  lc.y = 0;
  lc.w = 0;
  return lc;
}

pCoor
My_Piece_Of_The_World::overlay_obj_to_ovr(pCoor pos_obj)
{
  // Convert object-space coordinate, pos, into an overlay space, (x,z) ..
  // .. in which (0,0) is the lower-left overlay, etc.

  pCoor pos_ovr;
  pos_ovr.w = 1; // Will be set back to 0 if within platform.

  pos_ovr.x = ( pos_obj.x - w.platform_xmin ) * wid_x_inv;
  if ( pos_ovr.x < 0 || pos_ovr.x >= nx ) pos_ovr.w = 0;
  pos_ovr.z = ( pos_obj.z - w.platform_zmin ) * wid_z_inv;
  if ( pos_ovr.z < 0 || pos_ovr.z >= nz ) pos_ovr.w = 0;
  return pos_ovr;
}

Platform_Overlay*
My_Piece_Of_The_World::po_get(pCoor pos)
{
  // Convert object-space coordinate, pos, into an overlay space, (x,z) ..
  // .. in which (0,0) is the lower-left overlay, etc.

  pCoor pos_ovr = overlay_obj_to_ovr(pos);
  if ( pos_ovr.w == 0 ) return NULL;

  const int x = pos_ovr.x;
  const int z = pos_ovr.z;

  // Retrieve the overlay in which pos lies.
  //
  Platform_Overlay* const po = &platform_overlays[x + z * nz];

  po->overlay_xmin = w.platform_xmin + x * wid_x;
  po->overlay_zmin = w.platform_zmin + z * wid_z;

  if ( !po->data )
    {
      // This overlay has never been visited, initialize texel array.

      // Allocate the color array.
      po->data = new pColor[num_texels];

      // Initialize it. Note that this makes the colors transparent.
      memset(po->data,0,num_texels*sizeof(po->data[0]));

      po->texture_object_initialized = false;

      // Pre-compute the object-space coordinates of the corners of
      // the overlay. These will be used when constructing the primitive
```

```
      // on which the texture will be applied.
      //
      pCoor* const vertices = po->vertices;
      vertices[0] = pCoor( po->overlay_xmin, 0.01, po->overlay_zmin );
      vertices[1] = vertices[0] + pVect(wid_x,0,0);
      vertices[2] = vertices[1] + pVect(0,0,wid_z);
      vertices[3] = vertices[0] + pVect(0,0,wid_z);
      po->texture_modified = true;

      /// SOLUTION: Homework 3  Problem 3 -- Initialize texture to match image tile.

#if 0
      // Copy sample texture.
      //
      memcpy(po->data,sample_overlay.data,num_texels*sizeof(pCoor));
      //
      /// Please remove the memcpy when this part is solved.
#endif

      // Object space coordinates of overlay lower-left (00) and
      // upper-right (11) corners.
      //
      pCoor ovr_obj_00 = vertices[0];

      /// SOLUTION Homework 3 Problem 3

      // Factors to scale overlay texel space coordinates to object space.
      //
      const float oxtx = wid_x / twid_x;
      const float oztz = wid_z / twid_z;

      // Scale ImageMagick color components to range [0,1].
      //
      const float sc = 1.0 / MaxRGB;

      // Convenience function for extracting the fractional part of a number.
      //
      auto fpart = [] ( float f ) { float i; return modf(f,&i); };

      for ( int tx = 0;  tx < twid_x;  tx++ )
        for ( int tz = 0;  tz < twid_z;  tz++ )
          {
            // Get object-space coordinates of this texel ..
            //
            pCoor coor_obj = ovr_obj_00 + pVect( tx * oxtx, 0, tz * oztz );
            //
            // .. and convert that to platform tile space.
            //
            pCoor coor_tile = platform_obj_to_tile(coor_obj);

            // Do nothing if this texel is over a mirror tile.
            //
            if ( !coor_tile.w ) continue;

            // From tile-space coordinate compute location in syl_pixels array
            // and retrieve the pixel, if it's not out of range.
            //
            const int idx_syl =
              int ( fpart(coor_tile.x) * syl_wd )
              + int ( fpart(coor_tile.z) * syl_ht ) * syl_wd;
            if ( idx_syl >= syl_wd * syl_ht ) continue;
            PixelPacket p = syl_pixels[ idx_syl ];

            // Convert ImageMagick pixel to a pColor and write it to our
            // texture.
```

```
            //
            const int idx_otex = tx + tz * twid_x;
            po->data[idx_otex] = pColor( p.red, p.green, p.blue ) * sc;
          }
      }

    return po;
  }

  void
  My_Piece_Of_The_World::render()
  {
    // See demo-8-texture.cc for examples.

    // Turn on texturing.
    glEnable(GL_TEXTURE_2D);

    // Specify which texture unit to use.
    glActiveTexture(GL_TEXTURE0);

    // Specify how texel and the primitive's lighted color should be
    // combined. Use modulation, so that scuffs are affected by
    // lighting.
    glTexEnvi(GL_TEXTURE_ENV, GL_TEXTURE_ENV_MODE, GL_MODULATE);

    // Turn on the alpha test. This is not needed if blending is in use.
    if ( w.opt_tryout1 ) glEnable(GL_ALPHA_TEST);

    // Only write fragments corresponding to scuffed parts of the texture.
    glAlphaFunc(GL_GREATER,0.4);

    // Turn on blending so that scuffs are combined with what's already there.
    glEnable(GL_BLEND);

    // Set the blend equation to blend colors, but to write the alpha value
    // carried by the fragment (from our texture).
    glBlendFuncSeparate
      (GL_SRC_ALPHA,GL_ONE_MINUS_SRC_ALPHA, GL_ZERO, GL_ONE);

    for ( int i=0; i<num_overlays; i++ )
      {
        Platform_Overlay* const po = &platform_overlays[i];
        if ( !po->data ) continue;

        if ( !po->texture_object_initialized )
          {
            po->texture_object_initialized = true;

            //  This is the first time we've used this particular overlay,
            //  so we need to create a texture object for it and set its
            //  parameters.
            //
            glGenTextures(1,&po->txid);
            glBindTexture(GL_TEXTURE_2D,po->txid);
            glTexParameteri(GL_TEXTURE_2D, GL_GENERATE_MIPMAP, 1);
            glTexParameterf(GL_TEXTURE_2D,GL_TEXTURE_MIN_FILTER,
                            GL_LINEAR_MIPMAP_LINEAR);
            glTexParameterf(GL_TEXTURE_2D,GL_TEXTURE_MAG_FILTER,GL_LINEAR);
          }

        glBindTexture(GL_TEXTURE_2D,po->txid);

        if ( po->texture_modified )
          {
            // We've modified the texture since the last render, so
```

```
            // we need to update OpenGL's copy.

            po->texture_modified = false;

            glTexImage2D
              (GL_TEXTURE_2D,
               0,                 // Level of Detail (0 is base).
               GL_RGBA,           // Internal format to be used for texture.
               twid_x, twid_z,
               0,                 // Border
               GL_RGBA,     // GL_BGRA: Format of data read by this call.
               GL_FLOAT,    // Size of component.
               (void*)po->data);
            pError_Check();
          }


        //  If tryout2 is active (press 'Y' to toggle) show the red x texture
        //  instead of our own.
        //
        if ( w.opt_tryout2 ) glBindTexture(GL_TEXTURE_2D, sample_overlay.txid);

        //  Apply the texture to a quad. Note that we are using
        //  our pre-computed vertices.
        //
        glBegin(GL_QUADS);
        glNormal3f(0,-1,0);
        glColor3fv(color_white);
        glTexCoord2f(0,0);
        glVertex3fv(po->vertices[0]);
        glTexCoord2f(1,0);
        glVertex3fv(po->vertices[1]);
        glTexCoord2f(1,1);
        glVertex3fv(po->vertices[2]);
        glTexCoord2f(0,1);
        glVertex3fv(po->vertices[3]);
        glEnd();
      }

    glDisable(GL_ALPHA_TEST);
    glDisable(GL_BLEND);
    glDisable(GL_TEXTURE_2D);
  }

  void
  My_Piece_Of_The_World::clean()
  {
    /// Remove scuffs from dirty textures.

    //  Free any textures that are present.

    for ( int i=0; i<num_overlays; i++ )
      {
        Platform_Overlay* const po = &platform_overlays[i];
        if ( !po->data ) continue;

        // Tell OpenGL to free the texture object, including the
        // texture data that it has copied from us.
        //
        glDeleteTextures(1,&po->txid);

        // Free our own texel array.
        //
        free( po->data );
```

```
      // Update variables to reflect clean state.
      //
      po->data = NULL;
      po->texture_modified = false;
      po->texture_object_initialized = false;
    }

  /// SOLUTION - Remove glop from balls when clean pressed.
  //             Note: This was not specifically requested.

  for ( Ball* b: w.balls ) b->glop.clear();
}


void
World::init()
{
  chain_length = 14;

  opt_time_step_duration = 0.0003;

  distance_relaxed = 15.0 / chain_length;
  opt_spring_constant = 15000;
  variable_control.insert(opt_spring_constant,"Spring Constant");

  opt_gravity_accel = 9.8;
  opt_gravity = true;
  gravity_accel = pVect(0,-opt_gravity_accel,0);
  variable_control.insert(opt_gravity_accel,"Gravity");

  opt_air_resistance = 0.04;

  world_time = 0;
  time_step_count = 0;
  last_frame_wall_time = time_wall_fp();
  frame_timer.work_unit_set("Steps / s");

  ball_eye = NULL;
  opt_ride = false;

  // File to use for texture on platform files. Image can
  // be in any format that ImageMagick can handle, which is alot.
  //
  platform_tex_path = "code_shot.png";

  // Part of texture that covers a platform tile.
  platform_trmin = 0;
  platform_trmax = 1;
  platform_tsmin = 0;
  platform_tsmax = 1;

  init_graphics();

  mp.init();

  ball_setup_2();
  lock_update();
}

Ball*
World::make_marker(pCoor position, pColor color)
{
  Ball* const ball = new Ball;
  ball->position = position;
  ball->locked = true;
```

```
  ball->velocity = pVect(0,0,0);
  ball->radius = 0.2;
  ball->mass = 0;
  ball->contact = false;
  ball->color = color;
  return ball;
}

void
World::lock_update()
{
  // This routine called when options like opt_head_lock might have
  // changed.

  // Update locked status.
  //
  if ( head_ball ) head_ball->locked = opt_head_lock;
  if ( tail_ball ) tail_ball->locked = opt_tail_lock;

  // Re-compute minimum mass needed for stability.
  //
  for ( Ball* const ball: balls ) ball->spring_constant_sum = 0;
  const double dtis = pow( opt_time_step_duration, 2 );
  for ( Link* const link: links )
    {
      Ball* const b1 = link->ball1;
      Ball* const b2 = link->ball2;
      b1->spring_constant_sum += opt_spring_constant;
      b2->spring_constant_sum += opt_spring_constant;
    }
  for ( Ball* const ball: balls )
    ball->mass_min = ball->spring_constant_sum * dtis;
}

void
World::balls_twirl()
{
  if ( !head_ball || !tail_ball ) return;

  pCoor centroid(0,0,0);
  if ( opt_head_lock && head_ball )
    centroid = head_ball->position;
  else if ( opt_tail_lock && tail_ball )
    centroid = tail_ball->position;
  else {
    for ( Ball* const ball: balls ) centroid += pCoor(ball->position,1);
    centroid.homogenize();
  }
  for ( Ball* const ball: balls )
    ball->velocity += cross( pVect(centroid,ball->position), pVect(0,3,0) );

  return;

  pNorm axis(head_ball->position, tail_ball->position);

  for ( Ball* const ball: balls )
    {
      pVect b_to_top(ball->position,head_ball->position);
      const float dist_along_axis = dot(b_to_top,axis);
      const float lsq = b_to_top.mag_sq() - dist_along_axis * dist_along_axis;
      if ( lsq <= 1e-5 ) { ball->velocity = pVect(0,0,0); continue; }
      const float dist_to_axis = sqrt(lsq);
      pNorm rot_dir = cross(b_to_top,axis);
      ball->velocity += 10 * dist_to_axis * rot_dir;
    }
```

```
}

void
World::objects_erase()
{
  ball_eye = NULL;
  balls.erase();
  links.erase();
}

///
/// Physical Simulation Code
///

 /// Initialize Simulation
//

void
World::ball_setup_1()
{
  // Arrange and size balls to form a pendulum.

  pCoor first_pos(13.4,10,-9.2);
  pVect delta_pos = pVect(distance_relaxed,0,0);

  // Remove objects from the simulated objects lists, balls and links.
  // The delete operator is used on objects in the lists.
  //
  objects_erase();

  for ( int i=0; i<chain_length; i++ )
    {
      // Construct a new ball and add it to the simulated objects list (balls).
      //
      Ball* const ball = balls += new Ball;

      // Initialize position and other information.
      //
      ball->position = first_pos + i * delta_pos;
      ball->locked = false;
      ball->velocity = cross(pVect(ball->position,first_pos),pVect(0,1,0));
      ball->radius = 0.3;
      ball->mass = 4/3.0 * M_PI * pow(ball->radius,3);
      ball->contact = false;

      // If it's not the first ball link it to the previous ball.
      if ( i > 0 ) links += new Link( ball, balls[i-1] );
    }

  // The balls pointed to by head_ball and tail_ball can be manipulated
  // using the user interface (by pressing 'h' or 't', for example).
  // Set these variables.
  //
  head_ball = balls[0];
  tail_ball = balls[balls-1];

  opt_head_lock = true;    // Head ball will be frozen in space.
  opt_tail_lock = false;   // Tail ball can move freely.
}


void
World::make_truss(Truss_Info *truss_info)
{
```

```
/// Construct a truss based on members of truss_info.
//

//            <---- num_units (=9) ----------->
//  j
//  0         O---O---O---O---O---O---O---O---O     ^
//            |   |   |   |   |   |   |   |   |     |
//  1         O---O---O---O---O---O---O---O---O   num_sides (=3)
//            |   |   |   |   |   |   |   |   |     |
//  2         O---O---O---O---O---O---O---O---O     v
//
//      i ->  0   1   2   3   4   5   6   7   8
//
//  Note: Not all links are shown in the diagram above.

/// Truss_Info Inputs
//
//  truss_info->num_units
//    The number of sections in the truss (see diagram above).
//
//  truss_info->base_coors
//    A list containing num_sides coordinates, the coordinates of
//    balls at i=0 (see diagram above). (num_sides is the number of
//    elements in this list.)  These coordinates should all be
//    in the same plane.
//
//  truss_info->unit_length
//    A vector pointing from the ball at (i,j) to the ball at (i+1,j).
//
/// Truss_Info Outputs
//
//  truss_info->balls
//    A list that should be filled with the balls making up the truss.
//
//  truss_info->links
//    A list that should be filled with the links making up the truss.


const int num_sides = truss_info->base_coors.occ();
const int num_units = truss_info->num_units;

// Lists to hold balls and links created for truss.
//
Balls& bprep = truss_info->balls;
Links& lprep = truss_info->links;

// Create balls for truss.
//
for ( int i=0; i<num_units; i++ )
  for ( pCoor bcoor; truss_info->base_coors.iterate(bcoor); )
    {
      Ball* const ball = bprep += new Ball;
      ball->position = bcoor + i * truss_info->unit_length;
      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;
    }

// Create links.
//
for ( int i=0; i<num_units; i++ )
  for ( int j=0; j<num_sides; j++ )
    {
```

```
        const int idx = j + num_sides * i;

        // Retrieve the ball corresponding to (i,j).
        //
        Ball* const ball = bprep[idx];

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

        // Insert link to neighbor ball with name i.
        //
        lprep += new Link( ball, bprep[pn_idx], color_gray );

        // Insert links to balls with same i that are not neighbors.
        //
        if ( j == i % num_sides )
          for ( int k = 2; k < num_sides-1; k++ )
            lprep += new Link
              ( ball, bprep[ idx + (k+j)%num_sides - j ], color_white );

        if ( i == 0 ) continue;

        // Insert link to ball at (i-1,j).
        //
        lprep +=
          new Link( ball, bprep[idx-num_sides], color_lsu_official_purple );

        // Insert link to ball at (i-1, j-1 mod num_sides ).
        //
        lprep += new Link( ball, bprep[pn_idx-num_sides], color_green );
      }
}

void
World::ball_setup_2()
{
  const int sides = 4;
  const double delta_angle = 2*M_PI/sides;
  const double rot = delta_angle / 2;
  const float ball_radius = 0.15;
  const float spacing = distance_relaxed;
  pCoor first_pos
    (13.4 + drand48(),0.5*spacing*cos(rot)+ball_radius,-9.2+drand48());
  pVect delta_pos = pVect(spacing,0,0);
  pNorm loc_y = delta_pos;
  pNorm loc_x = pVect(0,0,1);
  pNorm loc_z = cross(loc_y,loc_x);

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

  Truss_Info truss_info;

  const int truss_units = 5;

  truss_info.num_units = truss_units;
  truss_info.unit_length = delta_pos;

  for ( int j=0; j<sides; j++ )
    {
      const double angle = rot + double(j)/sides*2*M_PI;
      pCoor chain_first_pos =
        first_pos
```

```
        + 0.5 * spacing * cos(angle) * loc_x
        + 0.5 * spacing * sin(angle) * loc_z;

      truss_info.base_coors += chain_first_pos;
    }

  make_truss(&truss_info);

  // 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 + truss_units * 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 ( Ball* const ball: balls )
    {
      ball->locked = false;
      ball->velocity = pVect(0,0,0);
      ball->radius = ball_radius;
      ball->mass = 10 * 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 = true;

  balls_twirl();

}

void
World::ball_setup_3()
{
  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.
  //
  objects_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);

const int idx_center = chain_length / 2 * sides;

for ( int i=0; i<sides; i++ )
  {
    Truss_Info ti;
    ti.num_units = chain_length / 2;
    const int idx_1 = idx_center + ( i == 0 ? sides - 1 : i - 1 );
    const int idx_2 = idx_center + i;

    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];
    ti.base_coors += b0->position;
    ti.base_coors += b1->position;
    ti.base_coors += b2->position;
    ti.base_coors += b3->position;

    pNorm v_head = cross(b1->position,b2->position,b3->position);
    ti.unit_length = delta_dir.magnitude * v_head;
    make_truss(&ti);
    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);
    links += ti.links;
    balls += ti.balls;

    int tsz = ti.balls.size();

    if ( i == 2 )
      {
        ball_eye = ti.balls[tsz-2];
        ball_down = ti.balls[tsz-1];
      }
    else if ( i == 1 )
      {
        ball_gaze = ti.balls[tsz/2];
      }

  }

// Insert links to balls at either end.
//
head_ball = balls += new Ball;
head_ball->color = color_green;
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->color = color_green;
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 ( Ball* const ball: balls )
    {
      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;
}


void
World::ball_setup_4()
{
}

void
World::ball_setup_5()
{
}


 /// Advance Simulation State by delta_t Seconds
//
void
World::time_step_cpu(double delta_t)
{
  time_step_count++;

  // Smoothly move ball in response to user input.
  //
  if ( adj_t_stop )
    {
      const double dt = min(world_time,adj_t_stop) - adj_t_prev;
      pVect adj = dt/adj_duration * adj_vector;
      balls_translate(adj,0);
      adj_t_prev = world_time;
      if ( world_time >= adj_t_stop ) adj_t_stop = 0;
    }

  for ( Ball* const ball: balls )
    ball->force = ball->mass * gravity_accel;

  for ( Link* const link: links )
    {
      // Spring Force from Neighbor Balls
      //
      Ball* const ball1 = link->ball1;
      Ball* const ball2 = link->ball2;

      // Construct a normalized (Unit) Vector from ball to neighbor.
```

```
      //
      pNorm ball_to_neighbor(ball1->position,ball2->position);

      const float distance_between_balls = ball_to_neighbor.magnitude;

      // Compute the speed of ball towards neighbor_ball.
      //
      pVect delta_v = ball2->velocity - ball1->velocity;
      float delta_s = dot( delta_v, ball_to_neighbor );

      // Compute by how much the spring is stretched (positive value)
      // or compressed (negative value).
      //
      const float spring_stretch =
        distance_between_balls - link->distance_relaxed;

      // Determine whether spring is gaining energy (whether its length
      // is getting further from its relaxed length).
      //
      const bool gaining_e = ( delta_s > 0.0 ) == ( spring_stretch > 0 );

      // Use a smaller spring constant when spring is loosing energy,
      // a quick and dirty way of simulating energy loss due to spring
      // friction.
      //
      const float spring_constant =
        gaining_e ? opt_spring_constant : opt_spring_constant * 0.7;

      ball1->force += spring_constant * spring_stretch * ball_to_neighbor;
      ball2->force -= spring_constant * spring_stretch * ball_to_neighbor;
    }

  ///
  /// Update Position of Each Ball
  ///

  for ( Ball* const ball: balls )
    {
      if ( ball->locked )
        {
          ball->velocity = pVect(0,0,0);
          continue;
        }

      // Update Velocity
      //
      // This code assumes that force on ball is constant over time
      // step. This is clearly wrong when balls are moving with
      // respect to each other because the springs are changing
      // length. This inaccuracy will make the simulation unstable
      // when spring constant is large for the time step.
      //
      const float mass = max( ball->mass, ball->mass_min );

      pVect delta_v = ( ball->force / mass ) * delta_t;

      const float dist_above = ball->position.y - ball->radius;

      const bool collision =
        platform_collision_possible(ball->position) && dist_above < 0;

      if ( collision )
        {
          const float spring_constant_plat =
            ball->velocity.y < 0 ? 100000 : 50000;
```

```
      const float fric_coefficient = 0.1;
      const float force_up = -dist_above * spring_constant_plat;
      const float delta_v_up = force_up / mass * delta_t;
      const float fric_force_mag = fric_coefficient * force_up;
      pNorm surface_v(ball->velocity.x,0,ball->velocity.z);
      const float delta_v_surf = fric_force_mag / mass * delta_t;
      if ( delta_v_surf > surface_v.magnitude )
        {
          // Ignoring other forces?
          delta_v =
            pVect(-ball->velocity.x,delta_v.y,-ball->velocity.z);
        }
      else
        {
          delta_v -= delta_v_surf * surface_v;
        }
      delta_v.y += delta_v_up;
    }

  ball->velocity += delta_v;

  // Air Resistance
  //
  const double fs = pow(1+opt_air_resistance,-delta_t);
  ball->velocity *= fs;

  // Update Position
  //
  // Assume that velocity is constant.
  //

  //  Remember previous ball position, used for scuff mark.
  //
  pCoor pos_prev = ball->position;

  ball->position += ball->velocity * delta_t;

  if ( !collision ) continue;

  Platform_Overlay* const po = mp.po_get(ball->position);

  if ( !po ) continue;

  //  Find the location of all of the texels scuffed
  //  by this ball during this timestep. The length
  //  of the scuff mark is based on the old and new positions,
  //  the width of the scuff mark is based on the intersection
  //  of the ball with the platform.

  // Current ball location in texel coordinates.
  //
  pCoor ball_tex = mp.overlay_obj_to_tex(po,ball->position);

  // Previous location of ball in texel coordinates.
  //
  pCoor prev_tex = mp.overlay_obj_to_tex(po,pos_prev);

  float width = mp.scale_x_obj_to_texel *
    4 * sqrt( ball->radius * ball->radius - pos_prev.y * pos_prev.y );

  // Direction along scuff mark (based on sliding motion).
  //
  pNorm skid(prev_tex,ball_tex);

  // Find direction along width of scuff mark.
```

```
//
pNorm nskid = cross( skid, pVect(0,1,0) );

// Iterate over texel locations scuffed by ball.
//
for ( float t = 0; t <= skid.magnitude; t++ )
  for ( float u = -width; u < max(width,1.0f); u++ )
  {
    pCoor tex_pos = prev_tex + t * skid + u * nskid;
    const int idx = int(tex_pos.x) + mp.twid_x * int(tex_pos.z);
    if ( idx < 0 || idx >= mp.num_texels ) continue;

    pColor& texel = po->data[ idx ];

    /// SOLUTION -- Homework 3 Problem 2

    if ( !texel.a ) continue;

    po->texture_modified = true;

    pColor cur_color = texel;

    // Determine which element of glop array corresponds to this
    // value of u.
    //
    size_t glop_idx = u < 0 ? - 2 * int(u) : 1 + 2 * int(u);
    //
    // Note: u is perpendicular to the direction of motion. Say,
    // u = -width is the leftmost part of the smear, u=0 is the
    // center, and u=width-1 is the rightmost part.

    // Initialize any new glop array elements.
    //
    while ( ball->glop.size() <= glop_idx )
      ball->glop.emplace_back( 0, 0, 0, 0 );

    pColor& glop = ball->glop[glop_idx];

    // Save the prior glop.a. Here, glop.a is the amount of ink
    // accumulated by the ball. It's not used as an alpha
    // channel.
    //
    const float glop_a = glop.a;

    // Compute amount of the glop int to deposit at this location.
    //
    const float dropoff = glop_a/(1+glop_a);

    // Compute the amount of ink that is before we arrived here.
    // Base that on the smallest (darkest) of the three color
    // components.
    //
    const float amt_ink =
      1 - min(min(cur_color.r,cur_color.g),cur_color.b);

    // Compute the amount of ink at this location that will be
    // added to the glop.
    //
    const float pickup_factor = 0.7;
    const float pickup = pickup_factor * amt_ink;

    texel = ( 1 - dropoff ) * texel + dropoff * glop;
    texel.a = 1;

    // Compute the amount of ink the glop looses.
```

```
            //
            const float glop_dropoff = dropoff/( 2 * mp.opt_smear_factor );
            //
            // Note that there is no conservation of ink law being applied.
            // That is, it's okay of glop_dropoff != dropoff.

            // Compute the amount of ink on the glop after accounting for
            // pickup and drop off.
            //
            const float glop_a_new = glop_a - glop_dropoff + pickup;

            // Compute new glop color, taking care to re-normalize it.
            //
            glop =
              glop_a_new < 0.0001 ? pColor(0.5,0.5,0.5) :
              ( 1 / glop_a_new )
              * ( ( glop_a - glop_dropoff ) * glop + pickup * cur_color );

            glop.a = glop_a_new;

            continue;

            /// SOLUTION ABOVE


            // Don't bother if already scuffed. If we wanted to be
            // fancy we could add this balls color onto what is already
            // there. But we don't want to be fancy.
            //
            if ( texel.a ) continue;

          }

      }
}

bool
World::platform_collision_possible(pCoor pos)
{
  // Assuming no motion in x or z axes.
  //
  return pos.x >= platform_xmin && pos.x <= platform_xmax
    && pos.z >= platform_zmin && pos.z <= platform_zmax;
}

 /// External Modifications to State
//
//   These allow the user to play with state while simulation
//   running.

// Move the ball.
//
void Ball::translate(pVect amt) {position += amt;}

// Add velocity to the ball.
//
void Ball::push(pVect amt) {velocity += amt;}

// Set the velocity to zero.
//
void Ball::stop() {velocity = pVect(0,0,0); }

// Set the velocity and rotation (not yet supported) to zero.
//
```

```
void Ball::freeze() {velocity = pVect(0,0,0); }



void World::balls_translate(pVect amt,int b){head_ball->translate(amt);}
void World::balls_push(pVect amt,int b){head_ball->push(amt);}
void World::balls_translate(pVect amt)
{ for ( Ball* const ball: balls ) ball->translate(amt); }
void World::balls_push(pVect amt)
{ for ( Ball* const ball: balls ) ball->push(amt); }
void World::balls_stop()
{ for ( Ball* const ball: balls ) ball->stop(); }
void World::balls_freeze(){balls_stop();}
void World::render_my_piece() {mp.render();}


void
World::frame_callback()
{
  // This routine called whenever window needs to be updated.

  const double time_now = time_wall_fp();

  if ( !opt_pause || opt_single_frame || opt_single_time_step )
    {
      /// Advance simulation state.

      // Amount of time since the user saw the last frame.
      //
      const double wall_delta_t = time_now - last_frame_wall_time;

      // Compute amount by which to advance simulation state for this frame.
      //
      const double duration =
        opt_single_time_step ? opt_time_step_duration :
        opt_single_frame ? 1/30.0 :
        wall_delta_t;

      const double world_time_target = world_time + duration;

      while ( world_time < world_time_target )
        {
          time_step_cpu(opt_time_step_duration);
          world_time += opt_time_step_duration;
        }

      // Reset these, just in case they were set.
      //
      opt_single_frame = opt_single_time_step = false;
    }

  last_frame_wall_time = time_now;

  if ( opt_ride && ball_eye )
    {
      pNorm b_eye_down(ball_eye->position,ball_down->position);
      pVect b_eye_up = -b_eye_down;
      pCoor eye_pos = ball_eye->position + 2.2 * ball_eye->radius * b_eye_up;
      pNorm b_eye_direction(eye_pos,ball_gaze->position);

      pNorm b_eye_left = cross(b_eye_direction,b_eye_up);
      pMatrix_Translate center_eye(-eye_pos);
      pMatrix rotate; rotate.set_identity();
```

```
      for ( int i=0; i<3; i++ ) rotate.rc(0,i) = b_eye_left.elt(i);
      for ( int i=0; i<3; i++ ) rotate.rc(1,i) = b_eye_up.elt(i);
      for ( int i=0; i<3; i++ ) rotate.rc(2,i) = -b_eye_direction.elt(i);
      modelview = rotate * center_eye;
      pMatrix reflect; reflect.set_identity(); reflect.rc(1,1) = -1;
      transform_mirror = modelview * reflect * invert(modelview);
    }

  render();
}

int
main(int argv, char **argc)
{
  pOpenGL_Helper popengl_helper(argv,argc);
  World world(popengl_helper);

  popengl_helper.rate_set(30);
  popengl_helper.display_cb_set(world.frame_callback_w,&world);
}
```

```
/// LSU EE 4702-1 (Fall 2017), GPU Programming
//
 /// Homework 4 -- SOLUTION
 //

 /// Instructions
 //
 //  Read the assignment: http://www.ece.lsu.edu/koppel/gpup/2017/hw04.pdf
 //

// Specify version of OpenGL Shading Language.
//
#version 450 compatibility


layout ( binding = 1 ) buffer sr { mat4 sphere_rot[]; };
layout ( binding = 2 ) buffer spr { vec4 sphere_pos_rad[]; };
layout ( binding = 3 ) buffer sc { vec4 sphere_color[]; };

// Use this variable to debug your code. Press 'y' to toggle tryout.x
// and 'Y' to toggle debug_bool.y (between true and false).
//
layout ( location = 3 ) uniform bvec3 tryout;
layout ( location = 2 ) uniform int lighting_options;

layout ( location = 4 ) uniform int opt_spirals;

const vec4 color_lsu_spirit_purple =
  vec4(0x58 / 256.0, 0x0d/ 256.0, 0xa6/256.0, 1);

#ifdef _VERTEX_SHADER_

in vec2 gl_MultiTexCoord0;

// Interface block for vertex shader output / geometry shader input.
//
out Data_to_GS
{
  vec3 normal_e;
  vec4 vertex_e;
  vec2 gl_TexCoord[1];
  vec4 gl_Position;

  vec4 pos_loc_theta;

  int instance_id;

  // Any changes here must also be made to the fragment shader input.
};


void
vs_main_p0()
{
  /// Do not put solution here.  Use this file for comparison or minor experiments.
  //  Instead, make vertex shader changes to vs_main_p1.

  vec4 pos_rad = sphere_pos_rad[gl_InstanceID];

  vec3 sphere_center = pos_rad.xyz;
  float radius = pos_rad.w;

  // Sphere orientation. Transpose is needed to convert C ordering to Fortran.
  mat4 rot = transpose(sphere_rot[gl_InstanceID]);
```

```
  // Coordinate of slice in the sphere's local coordinate space.
  vec4 vtx_local = vec4(gl_Vertex.xyz,1);

  // Transform slice normal / local coordinate to an object-space
  // normal and coordinate
  //
  vec4 normr = rot * vtx_local;
  vec3 normal_o = normr.xyz;
  vec4 vertex_o = vec4( sphere_center + radius * normal_o, 1 );

  // Transform coordinate to clip and eye space, and transform normal to
  // eye space.
  //
  gl_Position = gl_ModelViewProjectionMatrix * vertex_o;
  vertex_e = gl_ModelViewMatrix * vertex_o;
  normal_e = normalize(gl_NormalMatrix * normal_o );

  // Compute texture coordinates using theta and local y coordinate.
  //
  float theta = gl_Vertex.w;
  float y_loc = gl_Vertex.y;
  float tex_s = -theta / ( 2 * 3.1415926535f );
  float tex_t = 0.5f - 0.5f * y_loc;
  gl_TexCoord[0] = vec2(tex_s,tex_t);

  // Pass instance id down pipeline.
  //
  instance_id = gl_InstanceID;
}

void
vs_main_p1()
{
  /// Vertex Shader for Homework 4 Solution
  //
  // Homework 4 solution goes here, and other places.
  /// SOLUTION
  //
  //  Move code that computes vertex coordinates to the geometry shader.
  //  Pass along vertex coordinates/theta and instance number.

  pos_loc_theta = gl_Vertex;
  instance_id = gl_InstanceID;
}

#endif


#ifdef _GEOMETRY_SHADER_

in Data_to_GS
{

  vec3 normal_e;
  vec4 vertex_e;
  vec2 gl_TexCoord[1];
  vec4 gl_Position;

  /// SOLUTION
  //
  // The declarations above are not needed and will be optimized
  // out by the compiler.  They were not removed because they are
  // still needed by gs_main_p0, which shares this interface block.
  //
```

```
  // The declarations below are needed for the solution.

  vec4 pos_loc_theta;
  int instance_id;
} In[];

out Data_to_FS
{
  vec3 normal_e;
  vec4 vertex_e;
  vec2 gl_TexCoord[1];
  flat int instance_id;
};

// Type of primitive at geometry shader input.
//
layout ( triangles ) in;

// Type of primitives emitted geometry shader output.
//
 /// Homework 4 solution goes here, and other places.
//
// SOLUTION - Increase max_vertices from 3 to 60.
//
layout ( triangle_strip, max_vertices = 60 ) out;


void
gs_main_p0()
{
  /// Do not put solution here.  Use this file for comparison or minor experiments.
  //  Instead, make geometry shader changes to gs_main_p1.

  // If true use sphere normal, otherwise use one normal for whole triangle.
  const bool opt_normal_sphere = tryout.z;

  for ( int i=0; i<3; i++ )
    {
      // Optionally use same normal for each vertex.
      normal_e = In[opt_normal_sphere?i:0].normal_e;

      // Pass along values unchanged.
      vertex_e = In[i].vertex_e;
      gl_Position = In[i].gl_Position;
      gl_TexCoord[0] = In[i].gl_TexCoord[0];
      instance_id = In[i].instance_id;
      EmitVertex();
    }
  EndPrimitive();
}

void
gs_main_p1()
{
  /// Geometry Shader for Homework 4 Solution
  //
  //  Put Homework 4 solution here, and other places.

  // If true use sphere normal, otherwise use one normal for whole triangle.
  //
  const bool opt_normal_sphere = tryout.z;

  /// SOLUTION

  const float delta_theta = 2f * 3.1415926535f / opt_spirals;
```

```
  // Load rotation matrix. Note that all 3 vertices have same instance ID,
  // so we can use the instance ID of vertex 0, In[0].instance_id.
  //
  const mat4 rot4 = transpose(sphere_rot[In[0].instance_id]);

  // Make into a 3x3 since local coordinates won't have a w component.
  //
  mat3 rot = { rot4[0].xyz, rot4[1].xyz, rot4[2].xyz };

  const vec4 pos_rad = sphere_pos_rad[In[0].instance_id];

  // Extract object-space coordinate of center of sphere, and its radius.
  //
  const vec3 ctr_o = pos_rad.xyz;
  const float radius = pos_rad.w;

  // Compute a rotation matrix that will rotate by 2 π / opt_spirals
  // around the xz plane.
  //
  const float cosdtheta = cos(delta_theta);
  const float sindtheta = sin(delta_theta);
  const vec3 ax = vec3(cosdtheta,0,sindtheta);
  const vec3 az = vec3(-sindtheta,0,cosdtheta);
  const mat3 rotdt = mat3( ax, vec3(0,1,0), az );

  for ( int s=0; s<opt_spirals; s++ )
    {
      const float theta0 = s * delta_theta;

      for ( int i=0; i<3; i++ )
        {
          const vec3 pos_loc = In[i].pos_loc_theta.xyz;
          const float theta = In[i].pos_loc_theta.w + theta0;

          const float tex_s = -theta / ( 2 * 3.1415926535f );
          const float tex_t = 0.5f - 0.5f * pos_loc.y;
          gl_TexCoord[0] = vec2( tex_s, tex_t );

          const vec3 normal_o = rot * pos_loc;
          const vec4 vertex_o = vec4( ctr_o + radius * normal_o, 1 );

          gl_Position = gl_ModelViewProjectionMatrix * vertex_o;
          vertex_e = gl_ModelViewMatrix * vertex_o;

          if ( opt_normal_sphere || i == 0 )
            normal_e = normalize(gl_NormalMatrix * normal_o );

          instance_id = bool(s & 1) ? -1 : In[i].instance_id;

          EmitVertex();
        }

      EndPrimitive();

      // Rotate rotation matrix by delta_theta. With a decent compiler
      // this will take 8 multiply/adds because all but 4 elements of
      // rotdt are 0 or 1.
      //
      rot = rot * rotdt;
    }
}

#endif
```

```
#ifdef _FRAGMENT_SHADER_

uniform sampler2D tex_unit_0;

in Data_to_FS
{
  vec3 normal_e;
  vec4 vertex_e;
  vec2 gl_TexCoord[1];
  flat int instance_id;
};

vec4 generic_lighting(vec4 vertex_e, vec4 color, vec3 normal_e);


void
fs_main_p0()
{
  /// Do not put solution here. Use this for minor experiments.
  //
  //  Use fs_main_p1 for the Homework 4 solution.

  // Perform lighting, fetch and blend texture, then emit fragment.
  //

  // Get filtered texel.
  //
  vec4 texel = texture(tex_unit_0,gl_TexCoord[0]);

  vec4 color = sphere_color[instance_id];
  vec4 color2 = gl_FrontFacing ? color : vec4(0.5,0,0,1);

  // Multiply filtered texel color with lighted color of fragment.
  //
  gl_FragColor = texel * generic_lighting(vertex_e, color2, normal_e);

  // Copy fragment depth unmodified.
  //
  gl_FragDepth = gl_FragCoord.z;
}

void
fs_main_p1()
{
  /// Problem 1 Solution goes here and other places.

  // Perform lighting, fetch and blend texture, then emit fragment.
  //

  // Get filtered texel.
  //
  vec4 texel = texture(tex_unit_0,gl_TexCoord[0]);

  /// SOLUTION: Substitute purple.
  vec4 color =
    instance_id < 0 ? color_lsu_spirit_purple : sphere_color[instance_id];
  vec4 color2 = gl_FrontFacing ? color : vec4(0.5,0,0,1);

  // Multiply filtered texel color with lighted color of fragment.
  //
  gl_FragColor = texel * generic_lighting(vertex_e, color2, normal_e);

  // Copy fragment depth unmodified.
```

```
  //
  gl_FragDepth = gl_FragCoord.z;
}


///
/// Routine used by Either Vertex or Fragment Shader
///

vec4
generic_lighting(vec4 vertex_e, vec4 color, vec3 normal_er)
{
  // Return lighted color of vertex_e.
  //

  vec3 nspc_color = color.rgb * gl_LightModel.ambient.rgb;
  vec3 spec_color = vec3(0);
  vec3 normal_e = normalize(normal_er);

  for ( int i=0; i<2; i++ )
    {
      if ( ( lighting_options & ( 1 << i ) ) == 0 ) continue;
      vec4 light_pos = gl_LightSource[i].position;
      vec3 v_vtx_light = light_pos.xyz - vertex_e.xyz;
      float dist = length(v_vtx_light);
      float dist_vl_inv = 1.0 / dist;
      vec3 v_vtx_l_n = v_vtx_light * dist_vl_inv;

      float d_n_vl = dot(normal_e, v_vtx_l_n);
      float phase_light = max(0,gl_FrontFacing ? d_n_vl : -d_n_vl );

      vec3 ambient_light = gl_LightSource[i].ambient.rgb;
      vec3 diffuse_light = gl_LightSource[i].diffuse.rgb;
      float distsq = dist * dist;
      float atten_inv =
        gl_LightSource[i].constantAttenuation +
        gl_LightSource[i].linearAttenuation * dist +
        gl_LightSource[i].quadraticAttenuation * distsq;
      vec3 lighted_color =
        color.rgb
        * ( ambient_light + phase_light * diffuse_light ) / atten_inv;
      nspc_color += lighted_color;

      vec3 h = normalize( v_vtx_l_n - normalize(vertex_e.xyz) );

      spec_color +=
        pow(max(0.0,dot(normal_e,h)),16)
        * color.rgb
        * gl_LightSource[i].specular.rgb / atten_inv;
    }

  return vec4(nspc_color+spec_color,1);
}

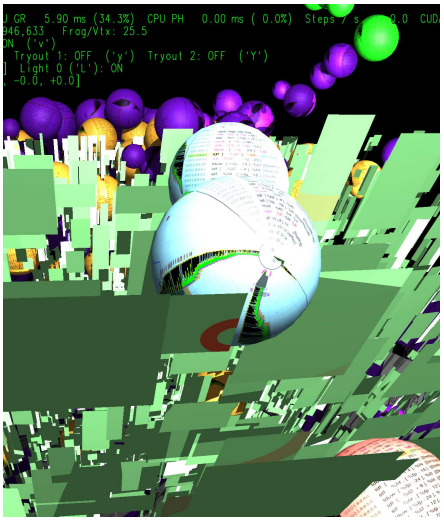

#endif
```

# LSU EE 4702-1 Homework 5 Solution Due: 27 November 2017

The solution code is in the repository in file `hw05-shdr-sol.cc`. Running the makefile will generate an executable for the solution named `hw05-sol`. (Strictly speaking, it's a second name for `hw05`. When run it checks its own name and if it starts `hw05-sol` it uses the solution shader, otherwise it uses the pre-solution shader.)

Colorized solution files can be found at `http://www.ece.lsu.edu/koppel/gpup/2017/hw05-shdr-sol.cc.html`. The CPU code (which is not modified for the solution) can be found at `http://www.ece.lsu.edu/koppel/gpup/2017/hw05.cc.html`.

**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 Scene 1, a cloud of green rectangles, *tiles*, with balls dropping on them. (The familiar water wheel is in scene 2.)



## Scene Interaction and User Interface

Pressing `d` (drip) will toggle ball dripping on and off. Pressing `x` will toggle a shower of balls. Pressing digits `1` through `2` will initialize different scenes, the program starts with scene 1. Pressing `t` starts the lots of balls scene (t is for test). Pressing `T` drops fewer balls.

The scenes differ in the number of objects, which include spheres, tiles, 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, tiles, 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.

## Display of Performance-Related Data

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).

## User Interface for Navigation and Interaction

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 `hw05.cc` and `hw05-shdr.cc` for documentation on other keys.

### Switching Between Shaders

The program can use three different sets of shaders for the spheres, PLAIN (tiles_0), PROBLEM 1 (tiles_1), and PROBLEM 2 (tiles_2). Pressing `v` switches between them.

### Description of Rendering

Scene 1 is filled with 8000 light green tiles arranged randomly in some volume, but all facing one point. The CPU code rendering tiles (all tiles, not just the light green ones described above) is in routine `World::render_tiles` in file `hw05.cc`. This routine launches one of the three shaders for rendering tiles. Shader `s_hw05_tiles_0`, which will be called `tiles_0` here, is used with a rendering pass in which the vertices are grouped as individual triangles. The shaders for this version are in file `hw05-shdr.cc` and have names ending in `tiles_0`. The `tiles_0` work correctly in the unmodified code, and they provide one special effect: a tile changes to red when touced by a ball and remains red for two seconds. The `tiles_0` shaders are for at most experimentation, they don't need to be modified for the homework. The `tiles_1` shaders are used in a rendering pass that also groups vertices as individual triangles, but the values used for `glVertex` and `glColor` are unconventional (see the code). The `tiles_2` shaders are used in a rendering pass in which vertices are not grouped (meaning points). For `tiles_2` the vertex data is placed in buffer objects, and the buffer objects are only updated when the underlying data changes. The only vertex shader input for the `tiles_2` is `gl_VertexID`. Initially the `tiles_1` and `tiles_2` shaders just call their `tiles_0` counterparts and so they won't render properly.

### Debugging Help

Boolean variables `opt_tryout1` and `opt_tryout2` can be used for debugging on the CPU. Pressing `y` and `Y` toggles their state. In the shader code use `tryout.x` and `tryout.y` for these two variables.

To see the difference between your code and the original code issue the Emacs command `Ctrl x v =` or from the shell the command `git diff`.

**Problem 1:** Modify the `tiles_1` shaders in `hw05-shdr.cc` so that they correctly render the tiles. They must operate with the data provided by `World::render_tiles` case 1, unconventional though it is. Do not change what data is provided by the CPU for this rendering pass.

- Work with the data provided from the CPU, don't send additional information.
- The code must be reasonably efficient.
- Pay attention to the geometry shader layout declarations **including** `max_vertices`.
- Pay attention to the `w` component of vertex shader inputs. *Note: This warning was not in the original assignment.*

First, consider the CPU code for setting up the rendering pass, which is case 1 in `World::render_tiles(bool simple)`:

```
case 1:
  {
    pShader_Use use(s_hw05_tiles_1);
    glUniform2i(1, opt_tryout1, opt_tryout2);
    glUniform1i(2, light_state_get());
    glUniform1f(3, world_time);

    glBegin(GL_TRIANGLES);
    for ( Tile* tile: tiles )
      {
        glColor3fv(tile->color);
        glVertex3fv(tile->pt_00);
        glColor4fv(tile->tact_pos);
        glVertex3fv(tile->ax);
        glVertex3fv(tile->ay);
      }
    glEnd();
  }
  break;
```

Inspection of the CPU code for case 1 reveals that:

- The primitive used in the rendering pass is individual triangles.
- Vertex 0 of each triangle is given the color in the `gl_Color` vertex shader input and the coordinate of `pt_00` in the `gl_Vertex` input.
- Vertex 1 of each triangle is given the contact position and time in the `gl_Color` input and the tile's `ax` **vector** in the `gl_Vertex` input
- Vertex 2 of each triangle is given the tile's `ay` **vector** in the `gl_Vertex` input
- Unlike the case 0 code, the **tile normal is not provided**.

From this we need to plan the design of the shaders.

Vertex Shader: Typically the vertex shader performs operations that can be done on individual vertices, such as transforming the vertex coordinate from object to clip space. But that won't work here because the kind of data in the shader input is different for each triangle vertex. For example, a transformation from object to clip space using the modelview matrix would work for the coordinate in vertex 0 but not for the vectors in vertices 1 and 2 of each triangle. Therefore all the vertex shader can do is pass the data to the geometry shader. The vertex shader code appears below:

```
vs_main_tiles_1() {
  // Pass through inputs unchanged.
  vertex_e = gl_Vertex;  // Either pt_00, ax, or ay.
  color = gl_Color;      // Either color or tact_pos_time.
}
```

Geometry Shader: The geometry shader needs to compute the coordinates of the tile corners (vertices) and also compute the normal.

Because the vectors `ax` and `ay` were sent using `glVertex3f` their `w` component was automatically set to 1. That's correct for coordinates but wrong for vectors, for which the `w` component, if present at all, should be 0. To see why consider adding coordinate $\begin{bmatrix}1\\2\\3\end{bmatrix}$ to vector $\begin{bmatrix}0\\40\\0\end{bmatrix}$. The sum is $\begin{bmatrix}1\\2\\3\end{bmatrix}+\begin{bmatrix}0\\40\\0\end{bmatrix}=\begin{bmatrix}1\\42\\3\end{bmatrix}$. This works if the

coordinate is a homogenized homogeneous coordinate $\begin{bmatrix}1\\2\\3\\1\end{bmatrix} + \begin{bmatrix}0\\40\\0\end{bmatrix} = \begin{bmatrix}1\\42\\3\\1\end{bmatrix}$. If the vector is represented by a four-component data type, such as `vec4`, the fourth, `w`, component must be set to zero. If it were set to 1 then the **sum would be wrong:** $\begin{bmatrix}1\\2\\3\\1\end{bmatrix} + \begin{bmatrix}0\\40\\0\\1\end{bmatrix} = \begin{bmatrix}1\\42\\3\\2\end{bmatrix} = \begin{bmatrix}0.5\\21\\1.5\\1\end{bmatrix}$.

In the geometry shader below the `w` components of the vectors are set to 0. Then the coordinates of the tile corners are computed and placed in an array. The tile normal is computed using the `ax` and `ay` vectors, converted to eye space and assigned to the shader output `normal_e`. The color from triangle vertex 0, which is actually a color, is assigned to shader output `color`. Finally a loop transforms and emits the four vertex coordinates.

```
void
gs_main_tiles_1()
{
  /// SOLUTION -- Problem 1

  vec4 vtx_o[4];

  // Write tile information into appropriately named variables.
  vec4 pt_00 = In[0].vertex_e;
  vec4 ax_o = vec4(In[1].vertex_e.xyz,0);
  vec4 ay_o = vec4(In[2].vertex_e.xyz,0);

  // Compute the object-space coordinates of the tile corners.
  vtx_o[0] = pt_00;
  vtx_o[1] = pt_00 + ax_o;
  vtx_o[2] = pt_00 + ay_o;
  vtx_o[3] = pt_00 + ay_o + ax_o;
  // The order is chosen for a triangle strip.

  // Compute the tile normal.
  vec3 normal_o = normalize(cross(ax_o.xyz,ay_o.xyz));
  normal_e = normalize(gl_NormalMatrix * normal_o);

  color = In[0].color;

  for ( int i=0; i<4; i++ ) {
      gl_Position = gl_ModelViewProjectionMatrix * vtx_o[i];
      vertex_e = gl_ModelViewMatrix * vtx_o[i];
      EmitVertex();
    }

  EndPrimitive();
}
```

**Problem 2:** Modify the code in `tiles_2` so that it renders the tiles correctly. They must operate with the data provided by `World::render_tiles` case 2, which is much more straightforward than

case 1. Note that case 2 uses points as the primitive.

- Work with the data provided from the CPU, don't send additional information.
- The code must be reasonably efficient.
- Pay attention to the geometry shader layout declarations **including** `max_vertices`.

Inspection of the CPU code for case 2 and the shader code reveals the following:

- All of the data needed to render a tile have been placed in buffer objects which have been bound to arrays in the shader code. For example, the `pt_00` coordinates are bound to array `pt_00s`.
- A rendering pass is performed in which the primitives are points.
- The case 2 code does not provide any vertex shader inputs. Not even `gl_Vertex`.
- The rendering pass is initiated using `glDrawArrays(GL_POINTS,0,tiles.size());`, and so the vertex shader will be invoked with `gl_VertexID` ranging from 0 to `tiles.size()-1`.

As in the solution to Problem 1, all of the work will be done in the geometry shader. All the vertex shader will do is pass the vertex id to the geometry shader. Unlike the situation in Problem 1, we could in principle compute and transform tile coordinates in the vertex shader and then send them to the geometry shader. However, that would unnecessarily increase the amount of data sent between the vertex and geometry shaders, which would waste resources and perhaps slow down execution.

Vertex Shader: The vertex shader send the vertex id to the geometry shader using new vertex shader output `vertex_id`. The interface block and vertex shader appear below.

```
in Data_to_GS {
  // Note: Unneeded variables removed.
  int vertex_id;
} In[1];

void vs_main_tiles_2() {
  vertex_id = gl_VertexID;
}
```

Geometry Shader: The geometry shader reads tile data from the arrays rather than from shader inputs. The geometry shader also includes code for the solution to Problem 3. Otherwise it is the same as the geometry shader in the Problem 1 solution.

```
void gs_main_tiles_2() {
  int vertex_id = In[0].vertex_id;

  // Retrieve information about the tile from buffer objects instead of shader inputs.
  vec4 vtx_o[4];
  vec4 pt_00 = pt_00s[vertex_id];
  vec4 ax_o = vec4(axs[vertex_id].xyz,0);
  vec4 ay_o = vec4(ays[vertex_id].xyz,0);

  // Compute tile object space vertex coordinates.
  vtx_o[0] = pt_00;
  vtx_o[1] = pt_00 + ax_o;
  vtx_o[2] = pt_00 + ay_o;
```

```
  vtx_o[3] = pt_00 + ay_o + ax_o;

  // Compute normal.
  vec3 normal_o = normalize(cross(ax_o.xyz,ay_o.xyz));
  normal_e = normalize(gl_NormalMatrix * normal_o);

  // Convert contact position to eye space and write it to a
  // geometry shader output. Also write time. (Part of Problem 3 solution.)
  //
  vec4 tact_pos_time = tact_pos_times[vertex_id];
  tact_pos_time_e = gl_ModelViewMatrix * vec4(tact_pos_time.xyz,1);
  tact_pos_time_e.w = tact_pos_time.w;

  color = colors[vertex_id];

  for ( int i=0; i<4; i++ ) {
      gl_Position = gl_ModelViewProjectionMatrix * vtx_o[i];
      vertex_e = gl_ModelViewMatrix * vtx_o[i];
      EmitVertex();
    }
  EndPrimitive();
}
```

**Problem 3:** Modify the code in any of the shaders so that when a tile is struck by a ball an expanding red ring is drawn centered at the point of contact. The ring should be rendered using existing primitives, don't emit new primitives for the ring. The screenshot at the top of the assignment shows one ring in the center of the image, and some larger faded regions below.

Each tile has a `tact_pos` member that holds the coordinates of the point on the tile that the ball has struck, that in the `x`, `y`, and `z` components, and the time that the contact occurred, in the `w` component. All three rendering passes provide this data in one way or another, see `World::render_tiles`. Also, notice that `world_time` is provided to shaders.

The ring should be easily visible for a few seconds, until it expands to the point where the tile is completely inside (and so the ring is no longer visible).

- Don't omit new primitives, instead use the fragment shader operating on existing primitives for rendering the tile.
- The width of the ring should be some fixed object-space size, don't make it one pixel wide.
- Try to reduce the amount of work done by the fragment shader by doing extra work in the vertex or geometry shaders.
- Also try reducing the amount of work done by the rasterization stage.

To solve this the contact position coordinate and time will be sent to the fragment shader. The fragment shader can then compute the distance between the fragment coordinate (which will be in `vertex_e`) and the contact coordinate. The problem statement did not give a particular ring expansion speed or width. The solution uses a speed of 0.5 object-space units per second and a width of 0.1 object space units. Both look reasonable. The fragment shader will also compute the radius of the ring based on the contact time, the current time (`world_time`), and the speed. The width will be used to compute an outer radius. If the distance from the fragment to the contact point is between these two radii then the fragment shader will emit a red color, otherwise it will emit the input color.

Two aspects of the solution were chosen to reduce the amount of work performed by the rasterizer and fragment shader. Since the fragment shader needs the eye-space vertex coordinate for lighting, it would make sense to transform the contact position to eye space *in the geometry shader* and then send it to the fragment shader. (Remember that for each primitive emitted by the geometry shader there can be dozens or more fragments.) Note that the contact position applies to the entire tile. It is being emitted as a vertex attribute (named `tact_pos_time_e`), but the same value is emitted for all four vertices making up the tile's triangle strip. It would be a waste of effort for the rasterizer to interpolate the values. For that reason the interface block declares `tact_pos_time_e` as `flat`.
Geometry Shader: The geometry shader given in the solution to Problem 2 converts the contact position to eye space and emits it, along with the time, in a new geometry shader output `tact_pos_time_e`. The interface block appears below, see the Problem 2 solution for the geometry shader code.

```
in Data_to_FS {
  flat vec3 normal_e;
  vec4 vertex_e;
  flat vec4 color;

  // Pass contact position and time to fragment shader to draw the ring.
  flat vec4 tact_pos_time_e;
};
```

Fragment Shader:

```
void fs_main_tiles_2() {
  float age = world_time - tact_pos_time_e.w;

  float speed = 0.5;   // Speed of increase of the ring radius.
  float width = 0.1;   // Width of ring.

  // Compute distance from this fragment to the contact point.
  float dist = distance(vertex_e.xyz,tact_pos_time_e.xyz);

  float inner_radius = age * speed;
  float outer_radius = inner_radius + width;

  // Determine if this fragment is in the ring, and set color appropriately.
  bool ring = dist >= inner_radius && dist <= outer_radius;
  vec4 color_red = vec4(1,0,0,1);
  vec4 color2 = ring ? color_red : color;

  gl_FragColor = generic_lighting(vertex_e, color2, normal_e);
  gl_FragDepth = gl_FragCoord.z;
}
```

```
/// LSU EE 4702-1 (Fall 2017), GPU Programming
//
 /// Homework 5 -- SOLUTION

 /// Instructions
 //
 //  Solution Discussion: http://www.ece.lsu.edu/koppel/gpup/2017/hw05_sol.pdf
 //
 ///  Put the solution in this file.
 //   To solve the problem code in hw05.cc needs to be inspected.


// Specify version of OpenGL Shading Language.
//
#version 450 compatibility

// The _GEOMETRY_SHADER_ define is put there by code in shader.h.
//
#ifdef _GEOMETRY_SHADER_
#extension GL_EXT_geometry_shader4 : enable
#endif

 // ----------------------------------------------------------------------------
 /// PROBLEM 1 - 3    ↓↓ Code can be placed below. ↓↓


layout ( location = 1 ) uniform bvec2 opt_tryout;
layout ( location = 2 ) uniform int lighting_options;
layout ( location = 3 ) uniform float world_time;

layout ( binding = 1 ) buffer b1 { vec4 pt_00s[]; };
layout ( binding = 2 ) buffer b2 { vec4 axs[]; };
layout ( binding = 3 ) buffer b3 { vec4 ays[]; };
layout ( binding = 4 ) buffer b4 { vec4 colors[]; };
layout ( binding = 5 ) buffer b5 { vec4 tact_pos_times[]; };


#ifdef _VERTEX_SHADER_

 /// PROBLEM 1 - 3    ↑↑ Code can be placed above. ↑↑
 // ----------------------------------------------------------------------------


#ifndef _PROBLEM_2_

 // ----------------------------------------------------------------------------
 /// PROBLEM 1 and 3  ↓↓ Code can be placed below. ↓↓


// Interface block for vertex shader output / geometry shader input.
//
out Data_to_GS
{
  vec4 gl_Position;
  vec3 normal_e;
  vec4 vertex_e;
  vec4 color;

  // Any changes here must also be made to the geometry shader input.
};

void
vs_main_tiles_0()
{
```

```
  gl_Position = gl_ModelViewProjectionMatrix * gl_Vertex;
  vertex_e = gl_ModelViewMatrix * gl_Vertex;
  normal_e = normalize(gl_NormalMatrix * gl_Normal);
  color = gl_Color;
}

void
vs_main_tiles_1()
{
  /// SOLUTION - Problem 1

  // Pass through inputs unchanged.
  //
  vertex_e = gl_Vertex;  // Either pt_00, ax, or ay.
  color = gl_Color;      // Either color or tact_pos_time.
}

 /// PROBLEM 1 and 3 ↑↑ Code can be placed above. ↑↑
 // ------------------------------------------------------------------------
#else
 // ------------------------------------------------------------------------
 /// PROBLEM 2 and 3 ↓↓ Code can be placed below. ↓↓

// Interface block for vertex shader output / geometry shader input.
//
out Data_to_GS
{
  vec4 gl_Position;
  vec3 normal_e;
  vec4 vertex_e;
  vec4 color;

  // Any changes here must also be made to the geometry shader input.

  /// SOLUTION -- Problem 2

  int vertex_id;
};

void
vs_main_tiles_2()
{
  /// SOLUTION -- Problem 2

  // Pass vertex id unchanged.
  //
  vertex_id = gl_VertexID;
  //
  // Note: Doing more work here and sending the results to the
  // geometry shader might result in exactly the same amount of
  // computation, but it will definitely increase the amount of data
  // sent between the vertex and geometry shaders, which is not
  // a good thing.
}

#endif

 /// PROBLEM 2 and 3 ↑↑ Code can be placed above. ↑↑
 // ------------------------------------------------------------------------

#endif

 // ------------------------------------------------------------------------
 /// PROBLEM 1 - 3 ↓↓ Code can be placed below. ↓↓
```

```
#ifdef _GEOMETRY_SHADER_

out Data_to_FS
{
  flat vec3 normal_e;
  vec4 vertex_e;
  flat vec4 color;

  /// SOLUTION -- Problem 3

  // Pass contact position and time to fragment shader to draw the ring.
  //
  flat vec4 tact_pos_time_e;
  //
  // Note the use of a flat qualifier since the values will be the
  // same at all three vertices and so there is no point in doing
  // interpolation, which isn't free.
};

 /// PROBLEM 1 - 3      ↑↑ Code can be placed above. ↑↑
 // ---------------------------------------------------------------------------

#ifndef _PROBLEM_2_

 // ---------------------------------------------------------------------------
 /// PROBLEM 1 and 3   ↓↓ Code can be placed below. ↓↓

// Interface block for vertex shader output / geometry shader input.
//
in Data_to_GS
{
  vec4 gl_Position;
  vec3 normal_e;
  vec4 vertex_e;
  vec4 color;

  // Any changes here must also be made to the vertex shader output.
} In[3];


// Type of primitive at geometry shader input.
//
layout ( triangles ) in;

// Type of primitives emitted geometry shader output.
//
 /// SOLUTION -- Problem 1 and 2
//
// Increase max_vertices by 1.
//
layout ( triangle_strip, max_vertices = 4 ) out;


void
gs_main_tiles_0()
{
  /// Do not put solution here.  Use this file for comparison or minor experiments.
  //  Instead, make geometry shader changes to gs_main_p1.

  // Color and normal will be the same for each vertex.
  normal_e = In[0].normal_e;
  color = In[0].color;

  for ( int i=0; i<3; i++ )
```

```
    {
      // Pass along values unchanged.
      vertex_e = In[i].vertex_e;
      gl_Position = In[i].gl_Position;
      EmitVertex();
    }
  EndPrimitive();
}

void
gs_main_tiles_1()
{
  /// SOLUTION -- Problem 1

  vec4 vtx_o[4];

  // Write tile information into appropriately named variables.
  //
  vec4 pt_00 = In[0].vertex_e;
  vec4 ax_o = vec4(In[1].vertex_e.xyz,0);
  vec4 ay_o = vec4(In[2].vertex_e.xyz,0);
  //
  // Note: The w component of ax and ay are set to zero. The OpenGL
  // code on the CPU would have set them to one since they were
  // specified using glVertex3f(FOO). Setting w to 1 is correct for
  // coordinates (when only x, y, and z are given) but wrong for
  // vectors.
  //
  // By setting the w components of vectors to 0 expressions like
  // pt_00 + ax_o compute the correct results.


  // Compute the object-space coordinates of the tile corners.
  //
  vtx_o[0] = pt_00;
  vtx_o[1] = pt_00 + ax_o;
  vtx_o[2] = pt_00 + ay_o;
  vtx_o[3] = pt_00 + ay_o + ax_o;
  //
  // The order is chosen for a triangle strip.

  // Compute the tile normal.
  //
  vec3 normal_o = normalize(cross(ax_o.xyz,ay_o.xyz));
  normal_e = normalize(gl_NormalMatrix * normal_o);
  //
  // Note that the normal is set once here and used for all four
  // vertices.
  //
  // Note that the CPU case 1 code does NOT send a normal. The vertex
  // shader still has an input named gl_Normal, but since the CPU code
  // does not set it, its value will be arbitrary.

  color = In[0].color;
  //
  // The same color is also used for all four vertices.

  for ( int i=0; i<4; i++ )
    {
      gl_Position = gl_ModelViewProjectionMatrix * vtx_o[i];
      vertex_e = gl_ModelViewMatrix * vtx_o[i];
      EmitVertex();
    }

  EndPrimitive();
```

```
}

 /// PROBLEM 1 and 3   ↑↑ Code can be placed above. ↑↑
 // ----------------------------------------------------------------------------

#else

 // ----------------------------------------------------------------------------
 /// PROBLEM 2 and 3   ↓↓ Code can be placed below. ↓↓

// Interface block for vertex shader output / geometry shader input.
//
in Data_to_GS
{
  vec4 gl_Position;
  vec3 normal_e;
  vec4 vertex_e;
  vec4 color;

  // Any changes here must also be made to the vertex shader output.

  /// SOLUTION -- Problem 2

  int vertex_id;

} In[1];


// Type of primitive at geometry shader input.
//
layout ( points ) in;

// Type of primitives emitted geometry shader output.
//
 /// SOLUTION -- Problem 2
//
layout ( triangle_strip, max_vertices = 4 ) out;

void
gs_main_tiles_2()
{
  /// SOLUTION -- Problem 2

  int vertex_id = In[0].vertex_id;

  // Retrieve information about the tile from buffer objects instead
  // of shader inputs.
  //
  vec4 vtx_o[4];
  vec4 pt_00 = pt_00s[vertex_id];
  vec4 ax_o = vec4(axs[vertex_id].xyz,0);
  vec4 ay_o = vec4(ays[vertex_id].xyz,0);

  // Compute tile object space vertex coordinates.
  //
  vtx_o[0] = pt_00;
  vtx_o[1] = pt_00 + ax_o;
  vtx_o[2] = pt_00 + ay_o;
  vtx_o[3] = pt_00 + ay_o + ax_o;

  // Compute normal.
  //
  vec3 normal_o = normalize(cross(ax_o.xyz,ay_o.xyz));
  normal_e = normalize(gl_NormalMatrix * normal_o);
  //
```

```
  // Don't write comments like "compute normal" in real-world code
  // for situations when it's obvious what the code is doing.

  /// SOLUTION -- Problem 3

  // Convert contact position to eye space and write it to a
  // geometry shader output. Also write time.
  //
  vec4 tact_pos_time = tact_pos_times[vertex_id];
  tact_pos_time_e = gl_ModelViewMatrix * vec4(tact_pos_time.xyz,1);
  tact_pos_time_e.w = tact_pos_time.w;
  //
  // Using one variable for position and time is not the best coding
  // style. But, in GPUs movement of 4-element vectors is more
  // efficient than scalars or smaller vectors.


  /// SOLUTION -- Problem 2
  //
  //  The code below is identical to the corresponding code in Problem 1.

  color = colors[vertex_id];

  for ( int i=0; i<4; i++ )
    {
      gl_Position = gl_ModelViewProjectionMatrix * vtx_o[i];
      vertex_e = gl_ModelViewMatrix * vtx_o[i];
      EmitVertex();
    }
  EndPrimitive();
}


 /// PROBLEM 2 and 3    ↑↑ Code can be placed above. ↑↑
 // ------------------------------------------------------------------------

#endif

#endif

 // ------------------------------------------------------------------------
 /// PROBLEM 1 - 3     ↓↓ Code can be placed below. ↓↓



#ifdef _FRAGMENT_SHADER_

in Data_to_FS
{
  flat vec3 normal_e;
  vec4 vertex_e;
  flat vec4 color;

  /// SOLUTION -- Problem 3

  // Pass contact position and time to fragment shader to draw the ring.
  //
  flat vec4 tact_pos_time_e;
  //
  // Note the use of a flat qualifier since the values will be the
  // same at all three vertices and so there is no point in doing
  // interpolation, which isn't free.
};

vec4 generic_lighting(vec4 vertex_e, vec4 color, vec3 normal_e);
```

```
void
fs_main_tiles_0()
{
  /// Do not put solution here. Use this for minor experiments.
  //

  // Multiply filtered texel color with lighted color of fragment.
  //
  gl_FragColor = generic_lighting(vertex_e, color, normal_e);

  // Copy fragment depth unmodified.
  //
  gl_FragDepth = gl_FragCoord.z;
}

void
fs_main_tiles_1()
{
  fs_main_tiles_0();
}

void
fs_main_tiles_2()
{
  /// SOLUTION -- Problem 3

  float age = world_time - tact_pos_time_e.w;

  // Quantities below are in object space.
  //
  float speed = 0.5;   // Speed of increase of the ring radius.
  float width = 0.1;   // Width of ring.
  //
  // Note: The distance between two points in object-space coordinates
  // is (or should be) the same as the distance in eye-space
  // coordinates and so it's okay to use an object-space width and
  // speed on eye-space distances.

  // Compute distance from this fragment to the contact point.
  //
  float dist = distance(vertex_e.xyz,tact_pos_time_e.xyz);

  float inner_radius = age * speed;
  float outer_radius = inner_radius + width;

  // Determine if this fragment is in the ring, and set color appropriately.
  //
  bool ring = dist >= inner_radius && dist <= outer_radius;
  vec4 color_red = vec4(1,0,0,1);
  vec4 color2 = ring ? color_red : color;

  gl_FragColor = generic_lighting(vertex_e, color2, normal_e);

  // Copy fragment depth unmodified.
  //
  gl_FragDepth = gl_FragCoord.z;
}


vec4
generic_lighting(vec4 vertex_e, vec4 color, vec3 normal_er)
{
  // Return lighted color of vertex_e.
```

```
  //
  vec3 nspc_color = color.rgb * gl_LightModel.ambient.rgb;
  vec3 spec_color = vec3(0);
  vec3 normal_e = normalize(normal_er);

  for ( int i=0; i<2; i++ )
    {
      if ( ( lighting_options & ( 1 << i ) ) == 0 ) continue;
      vec4 light_pos = gl_LightSource[i].position;
      vec3 v_vtx_light = light_pos.xyz - vertex_e.xyz;
      float dist = length(v_vtx_light);
      float dist_vl_inv = 1.0 / dist;
      vec3 v_vtx_l_n = v_vtx_light * dist_vl_inv;

      float d_n_vl = dot(normal_e, v_vtx_l_n);
      float phase_light = max(0,gl_FrontFacing ? d_n_vl : -d_n_vl );

      vec3 ambient_light = gl_LightSource[i].ambient.rgb;
      vec3 diffuse_light = gl_LightSource[i].diffuse.rgb;
      float distsq = dist * dist;
      float atten_inv =
        gl_LightSource[i].constantAttenuation +
        gl_LightSource[i].linearAttenuation * dist +
        gl_LightSource[i].quadraticAttenuation * distsq;
      vec3 lighted_color =
        color.rgb
        * ( ambient_light + phase_light * diffuse_light ) / atten_inv;
      nspc_color += lighted_color;

      vec3 h = normalize( v_vtx_l_n - normalize(vertex_e.xyz) );

      spec_color +=
        pow(max(0.0,dot(normal_e,h)),16)
        * color.rgb
        * gl_LightSource[i].specular.rgb / atten_inv;
    }

  return vec4(nspc_color+spec_color,1);
}

#endif
```

# 26 Fall 2016 Solutions

```
/// LSU EE 4702-1 (Fall 2016), GPU Programming
//
 /// Homework 1 --- SOLUTION
 //
 ///
 /// Instructions
 //
 //  Read the assignment: http://www.ece.lsu.edu/koppel/gpup/2016/hw01.pdf
 //
 //  Edit this file only. Contact the instructor if you believe that
 //  code in other files, such as hw01-graphics.cc needs to be edited.
 //
 //  Answers to questions that don't require writing code (for
 //  example, a question like "Explain why...") can be put in this
 //  file as comments or can be submitted separately (on paper or view
 //  E-mail).
 //
 //  Only add comments where necessary, for example, to explain code
 //  that's not obvious or to describe what the code is based on (if
 //  it's not yours).
 //
 //  There is no need to use comments to mark your code.
 //
 //  Try to clean up your code by removing any commented-out
 //  variations.


/// Purpose
//
//   Demonstrate simulation of string modeled as point masses and springs

/// What Code Does

// Simulates a string of beads over a platform. The string is modeled
// as point masses connected by springs with a long relaxed
// length. The platform consists of tiles, some are purple-tinted
// mirrors (showing a reflection of the ball), the others show the
// course syllabus.

///  Keyboard Commands
 //
 /// Object (Eye, Light, Ball) Location or Push
 //   Arrows, Page Up, Page Down
 //        Move object or push ball, depending on mode.
 //        With shift key pressed, motion is 5x faster.
 //   'e': Move eye.
 //   'l': Move light.
 //   'b': Move head (first) ball. (Change position but not velocity.)
 //   'B': Push head ball. (Add velocity.)
 //
 /// Eye Direction
 //   Home, End, Delete, Insert
 //   Turn the eye direction.
 //   Home should rotate eye direction up, End should rotate eye
 //   down, Delete should rotate eye left, Insert should rotate eye
 //   right.  The eye direction vector is displayed in the upper left.

 /// Simulation Options
 //  (Also see variables below.)
```

```
//
//  '1'    Set up scene 1.
//  '2'    Set up scene 2.
//  'p'    Pause simulation. (Press again to resume.)
//  ' '    (Space bar.) Advance simulation by 1/30 second.
//  'S- '  (Shift-space bar.) Advance simulation by one time step.
//  'h'    Freeze position of first (head) ball. (Press again to release.)
//  't'    Freeze position of last (tail) ball. (Press again to release.)
//  's'    Stop balls.
//  'g'    Turn gravity on and off.
//  'F12'  Write screenshot to file.

/// Variables
//   Selected program variables can be modified using the keyboard.
//   Use "Tab" to cycle through the variable to be modified, the
//   name of the variable is displayed next to "VAR" on the bottom
//   line of green text.

//  'Tab' Cycle to next variable.
//  '`'   Cycle to previous variable.
//  '+'   Increase variable value.
//  '-'   Decrease variable value.
//
//  VAR Spring Constant - Set spring constant.
//  VAR Air Resistance - Set air resistance.
//  VAR Light Intensity - The light intensity.
//  VAR Gravity - Gravitational acceleration. (Turn on/off using 'g'.)


#define GL_GLEXT_PROTOTYPES
#define GLX_GLXEXT_PROTOTYPES

#include <math.h>
#include <GL/gl.h>
#include <GL/glext.h>
#include <GL/glx.h>
#include <GL/glxext.h>
#include <GL/glu.h>
#include <GL/freeglut.h>

#include <gp/util.h>
#include <gp/glextfuncs.h>
#include <gp/coord.h>
#include <gp/shader.h>
#include <gp/pstring.h>
#include <gp/misc.h>
#include <gp/gl-buffer.h>
#include <gp/texture-util.h>

#include "shapes.h"


///
/// Main Data Structures
///
//
// class World: All data about scene.


class World;


// Object Holding Ball State
//
```

```
class Ball {
public:
  pCoor position;
  pVect velocity;

  float mass;
  float radius;

  bool contact;                 // Can be used for special effects.

  void push(pVect amt);
  void translate(pVect amt);
  void stop();
  void freeze();
};

#include "hw01-graphics.cc"


void
World::init()
{
  chain_length = 20;
  balls = new Ball[chain_length];

  opt_globe = true;
  globe.radius = 0.5;
  globe.position = pCoor(15.2,12,-18.5);
  opt_globe_charge = 250;
  variable_control.insert(opt_globe_charge,"Globe's Electric Charge");

  distance_relaxed = 15.0 / chain_length;
  opt_spring_constant = 1000;
  variable_control.insert(opt_spring_constant,"Spring Constant");

  opt_gravity_accel = 9.8;
  opt_gravity = true;
  gravity_accel = pVect(0,-opt_gravity_accel,0);
  variable_control.insert(opt_gravity_accel,"Gravity");

  opt_time_step_easy = false;

  opt_air_resistance = 0.001;
  variable_control.insert(opt_air_resistance,"Air Resistance");

  world_time = 0;
  time_step_count = 0;
  last_frame_wall_time = time_wall_fp();
  frame_timer.work_unit_set("Steps / s");

  init_graphics();

  ball_setup_2();
}

///
/// Physical Simulation Code
///

 /// Initialize Simulation
//
void
World::ball_setup_1()
```

```
{
  /// Arrange balls horizontally

  // Desired position of top ball.
  //
  pCoor top_pos(15.2, distance_relaxed * ( chain_length + 1 ), -18.5);

  // Desired distance between adjacent balls.
  //
  pVect ball_separation(distance_relaxed, 0, 0);

  for ( int i=0; i<chain_length; i++ )
    {
      Ball* const ball = &balls[i];
      ball->position = top_pos - i * ball_separation;

      ball->velocity = pVect(0,0,0);
      ball->radius = 0.3 * distance_relaxed;
      ball->mass = 4/3.0 * M_PI * pow(ball->radius,3);
      ball->contact = false;
    }

  opt_head_lock = true;

}

void
World::ball_setup_2()
{
  /// Arrange balls in direction dir.

  /// HOMEWORK 1: Solutions to Problems 1 and 2 here.

  // Desired position of top ball.
  //
  pCoor top_pos(15.2, distance_relaxed * ( chain_length + 1 ), -18.5);

  // Randomly chosen normal vector.
  //
  pNorm dir( drand48() - 0.5, -drand48() * 0.5, drand48() - 0.5 );


  /// SOLUTION --- PROBLEM 1
  //
  //  Use unit vector dir and distance_relaxed to compute a vector
  //  giving the desired ball-to-ball spacing.
  //
  pVect ball_to_ball = distance_relaxed * dir;

  for ( int i=0; i<chain_length; i++ )
    {
      Ball* const ball = &balls[i];

      /// SOLUTION --- PROBLEM 1, continued.
      //
      //  Use ball_to_ball to find difference between top ball and
      //  desired position of i'th ball.
      //
      pVect top_to_ball = i * ball_to_ball;
      //
      //  And use this to find the position of the i'th ball.
      //
      ball->position = top_pos + top_to_ball;

      /// SOLUTION --- PROBLEM 2
```

```
      //
      //  To obtain smooth rotation, the velocity direction must be
      //  orthogonal to the axis of rotation (the y axis in this case)
      //  and orthogonal to a vector connecting the ball to the axis
      //  of rotation.
      //
      pVect rot_axis(0,-1,0);  // From problem description.
      //
      //  So that all the balls rotate smoothly, the magnitude of the
      //  velocity must be proportional to the distance between the
      //  ball and the closest point on the axis. The distance from
      //  the axis is dist = i * distance_relaxed * cos(theta), where
      //  theta is the smallest angle between dir and rot_axis.
      //  Fortunately, the magnitude of cross( rot_axis, top_to_ball )
      //  is dist, so the cross product provides both the correct
      //  direction and the correct magnitude.
      //
      ball->velocity = 0.8 * cross( rot_axis, top_to_ball );


      ball->radius = 0.3 * distance_relaxed;
      ball->mass = 4/3.0 * M_PI * pow(ball->radius,3);
      ball->contact = false;
    }

  opt_head_lock = true;

}

void
World::ball_setup_3()
{
  float delta_theta = 2 * M_PI / chain_length;

  const float circum = chain_length * distance_relaxed;
  const float circ_radius = circum / ( 2 * M_PI );

  // Given:
  pNorm n(0,1,1);
  pCoor C(15.2,17.8,-18.5);
  pCoor P(C.x + circ_radius,17.8,C.z);

  // Compute:
  pNorm ax(C,P);  // ax is a normal vector from C to P.
  pNorm ay = cross(ax,n);
  float r = ax.magnitude;

  // Construct points on circle:
  //  for ( float theta = 0; theta < 2 * M_PI; theta += delta_theta )
  for ( int i=0; i<chain_length; i++ )
    {
      const float theta = i * delta_theta;

      pCoor pos = C + r * cos(theta) * ax + r * sin(theta) * ay;
      // Do something with pos..


      Ball* const ball = &balls[i];
      ball->position = pos;

      ball->velocity = pVect(0,0,0);
      ball->radius = ( i == chain_length - 1 ? 0.6 : 0.3 ) * distance_relaxed;
      ball->mass = 4/3.0 * M_PI * pow(ball->radius,3);
      ball->contact = false;
    }
```

```
  opt_head_lock = true;

}

void
World::ball_setup_4()
{
}

void
World::ball_setup_5()
{
}


 /// Advance Simulation State by delta_t Seconds
//
void
World::time_step_cpu_full(double delta_t)
{
  time_step_count++;

  /// HOMEWORK 1: Solution to Problem 3 in this routine.

  //
  /// Compute force and update velocity of each ball.
  //
  for ( int i=0; i<chain_length; i++ )
    {
      Ball* const ball = &balls[i];

      // Skip locked balls.
      //
      if ( opt_head_lock && i == 0 || opt_tail_lock && i == chain_length - 1 )
        {
          ball->velocity = pVect(0,0,0);
          continue;
        }

      pVect force(0,0,0);

      // Gravitational Force
      //
      force += ball->mass * gravity_accel;


      /// SOLUTION --- PROBLEM 3
      //
      //  Find the direction between the globe and the ball. The
      //  repulsion force will be in this direction.
      //
      pNorm gdist(globe.position,ball->position);
      //
      //  Compute the magnitude of the repulsion force. Note that
      //  the pNorm::mag_sq member is being used to avoid re-computing
      //  the square of the distance (magnitude).
      //
      const float eforce = opt_globe_charge * ball->mass / gdist.mag_sq;
      //
      //  Apply the force.
      //
      if ( opt_globe ) force += eforce * gdist;
```

```
    // Spring Force from Neighbor Balls
    //
    for ( int n_idx: { i-1, i+1 } )
      {
        if ( n_idx < 0 ) continue;
        if ( n_idx == chain_length ) break;

        Ball* const neighbor_ball = &balls[n_idx];

        // Construct a normalized (Unit) Vector from ball to neighbor.
        //
        pNorm ball_to_neighbor( ball->position, neighbor_ball->position );

        // Get distance between balls using pNorm member magnitude.
        //
        const float distance_between_balls = ball_to_neighbor.magnitude;

        // Compute by how much the spring is stretched (positive value)
        // or compressed (negative value).
        //
        const float spring_stretch =
          distance_between_balls - distance_relaxed;

        // Compute the speed of ball towards neighbor_ball.
        //
        pVect delta_v = neighbor_ball->velocity - ball->velocity;
        float delta_s = dot( delta_v, ball_to_neighbor );


        // Determine whether spring is gaining energy (whether its length
        // is getting further from its relaxed length).
        //
        const bool gaining_e = ( delta_s > 0.0 ) == ( spring_stretch > 0 );

        // Use a smaller spring constant when spring is loosing energy,
        // a quick and dirty way of simulating energy loss due to spring
        // friction.
        //
        const float spring_constant =
          gaining_e ? opt_spring_constant : opt_spring_constant * 0.7;

        force += spring_constant * spring_stretch * ball_to_neighbor;
      }

    // Update Velocity
    //
    // This code assumes that force on ball is constant over time
    // step. This is clearly wrong when balls are moving with
    // respect to each other because the springs are changing
    // length. This inaccuracy will make the simulation unstable
    // when spring constant is large for the time step.
    //
    ball->velocity += ( force / ball->mass ) * delta_t;

    // Air Resistance
    //
    const double fs = pow(1+opt_air_resistance,-delta_t);
    ball->velocity *= fs;
  }

///
/// Update Position of Each Ball
///

for ( int i=0; i<chain_length; i++ )
```

```
    {
      Ball* const ball = &balls[i];

      // Update Position
      //
      // Assume that velocity is constant.
      //
      ball->position += ball->velocity * delta_t;

      // Possible Collision with Platform
      //

      // Skip if collision impossible.
      //
      if ( !platform_collision_possible(ball->position) ) continue;
      if ( ball->position.y >= 0 ) continue;

      // Snap ball position to surface.
      //
      ball->position.y = 0;

      // Reflect y (vertical) component of velocity, with a reduction
      // due to energy lost in the collision.
      //
      if ( ball->velocity.y < 0 )
        ball->velocity.y = - 0.9 * ball->velocity.y;
    }
}

bool
World::platform_collision_possible(pCoor pos)
{
  // Assuming no motion in x or z axes.
  //
  return pos.x >= platform_xmin && pos.x <= platform_xmax
    && pos.z >= platform_zmin && pos.z <= platform_zmax;
}

 /// External Modifications to State
//
//   These allow the user to play with state while simulation
//   running.

// Move the ball.
//
void Ball::translate(pVect amt) {position += amt;}

// Add velocity to the ball.
//
void Ball::push(pVect amt) {velocity += amt;}

// Set the velocity to zero.
//
void Ball::stop() {velocity = pVect(0,0,0); }

// Set the velocity and rotation (not yet supported) to zero.
//
void Ball::freeze() {velocity = pVect(0,0,0); }



void World::balls_translate(pVect amt,int b){balls[b].translate(amt);}
void World::balls_push(pVect amt,int b){balls[b].push(amt);}
```

```
void World::balls_translate(pVect amt)
{ for(int i=0;i<chain_length;i++)balls[i].translate(amt);}
void World::balls_push(pVect amt)
{ for(int i=0;i<chain_length;i++)balls[i].push(amt);}
void World::balls_stop()
{ for(int i=0;i<chain_length;i++)balls[i].stop();}
void World::balls_freeze(){balls_stop();}


void
World::frame_callback()
{
  // This routine called whenever window needs to be updated.

  const double time_now = time_wall_fp();

  if ( !opt_pause || opt_single_frame || opt_single_time_step )
    {
      /// Advance simulation state.

      // Amount of time since the user saw the last frame.
      //
      const double wall_delta_t = time_now - last_frame_wall_time;

      const double time_step_duration = 0.0001;

      // Compute amount by which to advance simulation state for this frame.
      //
      const double duration =
        opt_single_time_step ? time_step_duration :
        opt_single_frame ? 1/30.0 :
        wall_delta_t;

      const double world_time_target = world_time + duration;

      while ( world_time < world_time_target )
        {
          time_step_cpu_full(time_step_duration);
          world_time += time_step_duration;
        }

      // Reset these, just in case they were set.
      //
      opt_single_frame = opt_single_time_step = false;
    }

  last_frame_wall_time = time_now;
  render();
}

int
main(int argv, char **argc)
{
  pOpenGL_Helper popengl_helper(argv,argc);
  World world(popengl_helper);

  popengl_helper.rate_set(30);
  popengl_helper.display_cb_set(world.frame_callback_w,&world);
}
```

```
/// LSU EE 4702-1 (Fall 2016), GPU Programming
//
 /// Homework 2 -- SOLUTION
 //

 /// Instructions
 //
 //  Read the assignment: http://www.ece.lsu.edu/koppel/gpup/2016/hw02.pdf


/// Purpose
//
//   Demonstrate simulation of point masses connected by springs.


/// What Code Does

// Simulates balls connected by springs over a platform. Balls and
// springs can be initialized in different arrangements (called
// scenes). Currently scene 1 is a simple string of beads. The
// platform consists of tiles, some are purple-tinted mirrors (showing
// a reflection of the ball), the others show the course syllabus.



///  Keyboard Commands
 //
 /// Object (Eye, Light, Ball) Location or Push
 //   Arrows, Page Up, Page Down
 //        Move object or push ball, depending on mode.
 //        With shift key pressed, motion is 5x faster.
 //   'e': Move eye.
 //   'l': Move light.
 //   'b': Move head (first) ball. (Change position but not velocity.)
 //   'B': Push head ball. (Add velocity.)
 //
 /// Eye Direction
 //   Home, End, Delete, Insert
 //   Turn the eye direction.
 //   Home should rotate eye direction up, End should rotate eye
 //   down, Delete should rotate eye left, Insert should rotate eye
 //   right.  The eye direction vector is displayed in the upper left.

 /// Simulation Options
 //  (Also see variables below.)
 //
 //  'w'    Twirl balls around axis formed by head and tail. (Prob 2 soln).
 //  '1'    Set up scene 1.
 //  '2'    Set up scene 2.
 //  '3'    Set up scene 3.
 //  'p'    Pause simulation. (Press again to resume.)
 //  ' '    (Space bar.) Advance simulation by 1/30 second.
 //  'S- '  (Shift-space bar.) Advance simulation by one time step.
 //  'h'    Freeze position of first (head) ball. (Press again to release.)
 //  't'    Freeze position of last (tail) ball. (Press again to release.)
 //  's'    Stop balls.
 //  'g'    Turn gravity on and off.
 //  'y'    Toggle value of opt_tryout1. Intended for experiments and debugging.
 //  'Y'    Toggle value of opt_tryout2. Intended for experiments and debugging.
 //  'F11'  Change size of green text (in upper left).
 //  'F12'  Write screenshot to file.
```

```
/// Variables
//   Selected program variables can be modified using the keyboard.
//   Use "Tab" to cycle through the variable to be modified, the
//   name of the variable is displayed next to "VAR" on the bottom
//   line of green text.

//  'Tab' Cycle to next variable.
//  '`'   Cycle to previous variable.
//  '+'   Increase variable value.
//  '-'   Decrease variable value.
//
//  VAR Spring Constant - Set spring constant.
//  VAR Time Step Duration - Set physics time step.
//  VAR Air Resistance - Set air resistance.
//  VAR Light Intensity - The light intensity.
//  VAR Gravity - Gravitational acceleration. (Turn on/off using 'g'.)


#define GL_GLEXT_PROTOTYPES
#define GLX_GLXEXT_PROTOTYPES

#include <GL/gl.h>
#include <GL/glext.h>
#include <GL/glx.h>
#include <GL/glxext.h>
#include <GL/glu.h>
#include <GL/freeglut.h>

#include <gp/util.h>
#include <gp/glextfuncs.h>
#include <gp/coord.h>
#include <gp/shader.h>
#include <gp/pstring.h>
#include <gp/misc.h>
#include <gp/gl-buffer.h>
#include <gp/texture-util.h>
#include <gp/colors.h>

#include "util-containers.h"
#include "shapes.h"


///
/// Main Data Structures
///
//
// class World: All data about scene.


class World;


// Object Holding Ball State
//
class Ball {
public:
  Ball():velocity(pVect(0,0,0)),locked(false),
         color(color_lsu_spirit_gold),contact(false){};
  pCoor position;
  pVect velocity;

  float mass;
  float mass_min; // Mass below which simulation is unstable.
  float radius;
```

```
  bool locked;

  pVect force;
  pColor color;
  bool contact;                 // When true, ball rendered in gray.
  float spring_constant_sum;    // Used to compute minimum mass.

  void push(pVect amt);
  void translate(pVect amt);
  void stop();
  void freeze();
};

class Link {
public:
  Link(Ball *b1, Ball *b2):ball1(b1),ball2(b2),
     distance_relaxed(pDistance(b1->position,b2->position)),
     color(color_lsu_spirit_purple){}
  Link(Ball *b1, Ball *b2, pColor colorp):ball1(b1),ball2(b2),
     distance_relaxed(pDistance(b1->position,b2->position)),
     color(colorp){}
  Ball* const ball1;
  Ball* const ball2;
  float distance_relaxed;
  pColor color;
};

// Declare containers and iterators for Balls and Links.
// (See util_container.h.)
//
typedef pVectorI<Link> Links;
typedef pVectorI_Iter<Link> LIter;
typedef pVectorI<Ball> Balls;
typedef pVectorI_Iter<Ball> BIter;
typedef pVector<pCoor> pCoors;
typedef pVector<pVect> pVects;

#include "hw02-graphics.cc"


void
World::init()
{
  chain_length = 14;

  opt_time_step_duration = 0.0003;
  variable_control.insert(opt_time_step_duration,"Time Step Duration");

  distance_relaxed = 15.0 / chain_length;
  opt_spring_constant = 15000;
  variable_control.insert(opt_spring_constant,"Spring Constant");

  opt_gravity_accel = 9.8;
  opt_gravity = true;
  gravity_accel = pVect(0,-opt_gravity_accel,0);
  variable_control.insert(opt_gravity_accel,"Gravity");

  opt_air_resistance = 0.04;
  variable_control.insert(opt_air_resistance,"Air Resistance");

  world_time = 0;
  time_step_count = 0;
```

```
  last_frame_wall_time = time_wall_fp();
  frame_timer.work_unit_set("Steps / s");

  init_graphics();

  pNorm dir_right = cross(pVect(0,-1,0),eye_direction);

  pCoor first_pos(13.4,15.8,-9.2);

  const float sample_size = 4;
  const float sample_thickness = sample_size / 10;
  pVect sample_x = sample_size * dir_right;
  pCoor sample_start = first_pos - 3.3 * 1.5 * sample_x;
  for ( int i=0; i<3; i++ )
    {
      sample_a[i] = sample_start + sample_x * i * 1.5;
      sample_b[i] = sample_a[i] - pVect(0,sample_size,0);
      sample_c[i] = sample_a[i] + sample_thickness * dir_right;
    }

  auto tr = [&](pCoor *co) { co[0] = co[-3] + pVect(0,-first_pos.y/2,0); };
  for ( int i=3; i<6; i++ ) { tr(sample_a+i);tr(sample_b+i);tr(sample_c+i);}

  ball_setup_2();
  lock_update();
}

Ball*
World::make_marker(pCoor position, pColor color)
{
  Ball* const ball = new Ball;
  ball->position = position;
  ball->locked = true;
  ball->velocity = pVect(0,0,0);
  ball->radius = 0.2;
  ball->mass = 0;
  ball->contact = false;
  ball->color = color;
  return ball;
}

void
World::lock_update()
{
  // This routine called when options like opt_head_lock might have
  // changed.

  // Update locked status.
  //
  if ( head_ball ) head_ball->locked = opt_head_lock;
  if ( tail_ball ) tail_ball->locked = opt_tail_lock;

  // Re-compute minimum mass needed for stability.
  //
  for ( BIter ball(balls); ball; ) ball->spring_constant_sum = 0;
  const double dtis = pow( opt_time_step_duration, 2 );
  for ( LIter link(links); link; )
    {
      Ball* const b1 = link->ball1;
      Ball* const b2 = link->ball2;
      b1->spring_constant_sum += opt_spring_constant;
      b2->spring_constant_sum += opt_spring_constant;
    }
  for ( BIter ball(balls); ball; )
    ball->mass_min = ball->spring_constant_sum * dtis;
```

```
}

void
World::render_it_demo(pCoor a, pCoor b, pCoor c, int version)
{
  /// DO NOT put homework solution in this routine.
  //
  //  This routine can be modified for experimentation, but don't
  //  put solution here. The solution should go in to render_it.

  pNorm dir_z = cross(c,b,a);
  glColorMaterial(GL_FRONT,GL_AMBIENT_AND_DIFFUSE);
  glMaterialfv(GL_BACK,GL_AMBIENT_AND_DIFFUSE,color_gray);

  glBegin(GL_TRIANGLES);
  glColor3fv(color_red);
  glNormal3fv(dir_z);
  glVertex3fv(a);
  glVertex3fv(b);
  glVertex3fv(c);
  glEnd();

  if ( version == 0 ) return;

  // Add coordinates to a container (something based on a standard C
  // library vector) so they can be easily iterated over.
  pCoors coords;
  coords += a;
  coords += b;
  coords += c;

  // Create a second set of coordinates by translating first set.
  pCoors more_coords;
  for ( pCoor co: coords ) more_coords += co + pVect(0.5,0,0);

  // Take the original coordinates and rotate them, and
  // put these new coordinates in more_coords.
  //
  pMatrix_Rotation rot(dir_z, M_PI/10);  // Rotation transform around dir_z.
  pMatrix m = pMatrix_Translate(b) * rot * pMatrix_Translate(-b);
  if ( version == 2 )
    for ( pCoor co: coords ) more_coords += m * co;

  // Render the transformed coordinates.
  glBegin(GL_TRIANGLES);
  glColor3fv(color_cyan);
  glNormal3fv(dir_z);
  for ( pCoor co: more_coords ) glVertex3fv( co );
  glEnd();

}

void
World::render_it(pCoor a, pCoor b, pCoor c, int version)
{
  pNorm dir_z = cross(c,b,a);

  pNorm dir_dn(a,b);
  pNorm dir_rt(a,c);  // Right

  /// SOLUTION

  //
  /// Construct L shape.
  //
```

```
  float edge_len = dir_dn.magnitude;

  float width = dir_rt.magnitude;
  float hwidth = width / 2;

  // Naming scheme:
  //   l0: Point on left edge,
  //   l1: point a bit over to the right of l0,...
  //   r0, t0, b0: point on right, top, bottom edge respectively.
  //
  pCoor p_l0t2 = a + width * dir_dn;
  pCoor p_l1t1 = a + hwidth * ( dir_rt + dir_dn );
  pCoor p_l2t0 = a + width * dir_rt;
  pCoor p_l2t2 = p_l2t0 + width * dir_dn;
  pCoor p_l2b2 = c + (edge_len - width ) * dir_dn;
  pCoor p_r2b0 = b + (edge_len - width ) * dir_rt;
  pCoor p_r2b2 = p_r2b0 - width * dir_dn;
  pCoor p_r1b1 = p_r2b0 + hwidth * ( dir_rt - dir_dn );
  pCoor p_r0b0 = b + edge_len * dir_rt;

  pCoors coords;
  coords += p_l2t0;
  coords += a;
  coords += p_l1t1;
  coords += p_l0t2;
  coords += p_l2t2;
  coords += b;
  coords += p_l2b2;
  coords += p_r2b0;
  coords += p_r2b2;
  coords += p_r1b1;

  glColorMaterial(GL_FRONT,GL_AMBIENT_AND_DIFFUSE);
  glMaterialfv(GL_BACK,GL_AMBIENT_AND_DIFFUSE,color_gray);

  // Find center of square to be formed using two L shapes.
  //
  pCoor p_ctr = a + 0.5 * pVect(a,p_r0b0);

  if ( version > 0 )
    {
      // Construct transformation matrix that will rotate L shape into
      // position.

      pMatrix m =
        pMatrix_Translate(p_ctr)
        * pMatrix_Rotation(dir_z,M_PI) * pMatrix_Translate(-p_ctr);

      // Use transformation to add rotated points on to list.
      //
      for ( pCoor co: pCoors(coords) ) coords +=  m * co;
    }

  if ( version < 2 )
    {
      // Render the L or the square.

      glBegin(GL_TRIANGLE_STRIP);
      glColor3fv(color_yellow);
      glNormal3fv(dir_z);
      for ( pCoor co: coords ) glVertex3fv( co );
      glEnd();
      return;
    }
```

```
  pCoor cube_ctr = p_ctr - 0.5 * edge_len * dir_z;

  // Rotate the square to form a cube.

  for ( int axis_num = 0; axis_num < 2; axis_num++ )
    {
      pVect axis = axis_num == 0 ? dir_dn : dir_rt;
      for ( int face = 0;  face < 4;  face++ )
        {
          if ( axis_num == 1 &&  ( ( face & 1 ) == 0 ) ) continue;
          const double angle = face * M_PI / 2;
          pMatrix_Rotation rot(axis,angle);
          pMatrix m =
            pMatrix_Translate(cube_ctr) * rot * pMatrix_Translate(-cube_ctr);

          glBegin(GL_TRIANGLE_STRIP);
          glColor3fv(color_yellow);

          // Use rot, not m, to rotate normal. (Vectors don't need to
          // be centered.)
          glNormal3fv(rot * dir_z);

          // Rotate coordinates.
          //
          for ( pCoor co: coords ) glVertex3fv( m * co );

          glEnd();
        }
    }
}


void
World::objects_erase()
{
  balls.erase();
  links.erase();
  chain_starts.clear();
}

///
/// Physical Simulation Code
///

 /// Initialize Simulation
//

void
World::ball_setup_1()
{
  // Arrange and size balls to form a pendulum.

  pCoor first_pos(13.4,17.8,-9.2);
  pVect delta_pos = pVect(distance_relaxed,0,0);

  // Remove objects from the simulated objects lists, balls and links.
  // The delete operator is used on objects in the lists.
  //
  objects_erase();

  for ( int i=0; i<chain_length; i++ )
    {
      // Construct a new ball and add it to the simulated objects list (balls).
      //
```

```
      Ball* const ball = balls += new Ball;

      // Initialize position and other information.
      //
      ball->position = first_pos + i * delta_pos;
      ball->locked = false;
      ball->velocity = pVect(0,0,0);
      ball->radius = 0.3;
      ball->mass = 4/3.0 * M_PI * pow(ball->radius,3);
      ball->contact = false;

      // If it's not the first ball link it to the previous ball.
      if ( i > 0 ) links += new Link( ball, balls[i-1] );
    }

  // The balls pointed to by head_ball and tail_ball can be manipulated
  // using the user interface (by pressing 'h' or 't', for example).
  // Set these variables.
  //
  head_ball = balls[0];
  tail_ball = balls[balls-1];

  opt_head_lock = true;    // Head ball will be frozen in space.
  opt_tail_lock = false;   // Tail ball can move freely.
}


void
World::ball_setup_2()
{
  // Arrange and size balls to form a pendulum.

  pCoor first_pos(5.1,17.8,-13.1);
  pNorm delta_dir = pVect(1,-1,0);
  pVect delta_pos = distance_relaxed * delta_dir;

  // Remove objects from the simulated objects lists, balls and links.
  // The delete operator is used on objects in the lists.
  //
  objects_erase();

  int i;
  for ( i=0; i<chain_length/2; i++ )
    {
      // Construct a new ball and add it to the simulated objects list (balls).
      //
      Ball* const ball = balls += new Ball;

      // Initialize position and other information.
      //
      ball->position = first_pos + i * delta_pos;

      // If it's not the first ball link it to the previous ball.
      if ( i > 0 ) links += new Link( ball, balls[i-1] );
    }

  int sides = 4;
  pNorm hx = delta_pos.x
    ? pVect(-delta_pos.y,delta_pos.x,0) : pVect(0, -delta_pos.z,delta_pos.y);
  pNorm hy = cross(delta_pos,hx);
  Ball* const l1 = balls.back();
  for ( int i=0; i<sides; i++ )
    {
      const double angle = i * 2 * M_PI / sides;
      Ball* const ball = balls += new Ball;
```

```
      chain_starts += ball;
      ball->position =
        l1->position + delta_pos
        + 2 * distance_relaxed * cos(angle) * hx
        + 2 * distance_relaxed * sin(angle) * hy;
      links += new Link( ball, l1 );
      if ( i > 0 )
        links += new Link( ball, chain_starts[i-1] );
      if ( i > 1 && i == sides - 1 )
        links += new Link( ball, chain_starts[0] );
      if ( sides > 3 && i > sides/2 )
        links += new Link( ball, chain_starts[i-sides/2] );
    }

  for ( int i=1; i<chain_length/2; i++ )
    for ( int j=0; j<sides; j++ )
      {
      // Construct a new ball and add it to the simulated objects list (balls).
      //
      Ball* const ball = balls += new Ball;

      // Initialize position and other information.
      //
      ball->position = chain_starts[j]->position + i * delta_pos;

      // If it's not the first ball link it to the previous ball.
      links += new Link( ball, balls[balls-sides-1] );
    }

  for ( Ball* ball : balls )
    {
      ball->locked = false;
      ball->velocity = pVect(0,0,0);
      ball->radius = 0.3;
      ball->mass = 4/3.0 * M_PI * pow(ball->radius,3);
      ball->contact = false;
    }

  // The balls pointed to by head_ball and tail_ball can be manipulated
  // using the user interface (by pressing 'h' or 't', for example).
  // Set these variables.
  //
  head_ball = balls[0];
  tail_ball = balls[balls-1];

  opt_head_lock = true;    // Head ball will be frozen in space.
  opt_tail_lock = false;   // Tail ball can move freely.

}


void
World::ball_setup_3()
{
}

void
World::ball_setup_4()
{
}


void
World::ball_setup_5()
```

```
{
}


 /// Advance Simulation State by delta_t Seconds
//
void
World::time_step_cpu(double delta_t)
{
  time_step_count++;

  for ( BIter ball(balls); ball; )
    ball->force = ball->mass * gravity_accel;

  for ( LIter link(links); link; )
    {
      // Spring Force from Neighbor Balls
      //
      Ball* const ball1 = link->ball1;
      Ball* const ball2 = link->ball2;

      // Construct a normalized (Unit) Vector from ball to neighbor.
      //
      pNorm ball_to_neighbor(ball1->position,ball2->position);

      const float distance_between_balls = ball_to_neighbor.magnitude;

      // Compute the speed of ball towards neighbor_ball.
      //
      pVect delta_v = ball2->velocity - ball1->velocity;
      float delta_s = dot( delta_v, ball_to_neighbor );

      // Compute by how much the spring is stretched (positive value)
      // or compressed (negative value).
      //
      const float spring_stretch =
        distance_between_balls - link->distance_relaxed;

      // Determine whether spring is gaining energy (whether its length
      // is getting further from its relaxed length).
      //
      const bool gaining_e = ( delta_s > 0.0 ) == ( spring_stretch > 0 );

      // Use a smaller spring constant when spring is loosing energy,
      // a quick and dirty way of simulating energy loss due to spring
      // friction.
      //
      const float spring_constant =
        gaining_e ? opt_spring_constant : opt_spring_constant * 0.7;

      ball1->force += spring_constant * spring_stretch * ball_to_neighbor;
      ball2->force -= spring_constant * spring_stretch * ball_to_neighbor;
    }

  ///
  /// Update Position of Each Ball
  ///

  for ( BIter ball(balls); ball; )
    {
      if ( ball->locked )
        {
          ball->velocity = pVect(0,0,0);
          continue;
        }
```

```
      // Update Velocity
      //
      // This code assumes that force on ball is constant over time
      // step. This is clearly wrong when balls are moving with
      // respect to each other because the springs are changing
      // length. This inaccuracy will make the simulation unstable
      // when spring constant is large for the time step.
      //
      float mass = std::max(ball->mass, ball->mass_min );
      ball->velocity += ( ball->force / mass ) * delta_t;

      // Air Resistance
      //
      const double fs = pow(1+opt_air_resistance,-delta_t);
      ball->velocity *= fs;

      // Update Position
      //
      // Assume that velocity is constant.
      //
      ball->position += ball->velocity * delta_t;

      // Possible Collision with Platform
      //

      // Skip if collision impossible.
      //
      if ( !platform_collision_possible(ball->position) ) continue;
      if ( ball->position.y >= 0 ) continue;

      // Snap ball position to surface.
      //
      ball->position.y = 0;

      // Reflect y (vertical) component of velocity, with a reduction
      // due to energy lost in the collision.
      //
      if ( ball->velocity.y < 0 )
        ball->velocity.y = - 0.9 * ball->velocity.y;
    }
}

bool
World::platform_collision_possible(pCoor pos)
{
  // Assuming no motion in x or z axes.
  //
  return pos.x >= platform_xmin && pos.x <= platform_xmax
    && pos.z >= platform_zmin && pos.z <= platform_zmax;
}

 /// External Modifications to State
//
//   These allow the user to play with state while simulation
//   running.

// Move the ball.
//
void Ball::translate(pVect amt) {position += amt;}

// Add velocity to the ball.
//
void Ball::push(pVect amt) {velocity += amt;}
```

```
// Set the velocity to zero.
//
void Ball::stop() {velocity = pVect(0,0,0); }

// Set the velocity and rotation (not yet supported) to zero.
//
void Ball::freeze() {velocity = pVect(0,0,0); }



void World::balls_translate(pVect amt,int b){head_ball->translate(amt);}
void World::balls_push(pVect amt,int b){head_ball->push(amt);}
void World::balls_translate(pVect amt)
{ for ( BIter ball(balls); ball; ) ball->translate(amt); }
void World::balls_push(pVect amt)
{ for ( BIter ball(balls); ball; ) ball->push(amt); }
void World::balls_stop()
{ for ( BIter ball(balls); ball; ) ball->stop(); }
void World::balls_freeze(){balls_stop();}


void
World::frame_callback()
{
  // This routine called whenever window needs to be updated.

  const double time_now = time_wall_fp();

  if ( !opt_pause || opt_single_frame || opt_single_time_step )
    {
      /// Advance simulation state.

      // Amount of time since the user saw the last frame.
      //
      const double wall_delta_t = time_now - last_frame_wall_time;

      // Compute amount by which to advance simulation state for this frame.
      //
      const double duration =
        opt_single_time_step ? opt_time_step_duration :
        opt_single_frame ? 1/30.0 :
        wall_delta_t;

      const double world_time_target = world_time + duration;

      while ( world_time < world_time_target )
        {
          time_step_cpu(opt_time_step_duration);
          world_time += opt_time_step_duration;
        }

      // Reset these, just in case they were set.
      //
      opt_single_frame = opt_single_time_step = false;
    }

  last_frame_wall_time = time_now;
  render();
}

int
main(int argv, char **argc)
```

```
{
  pOpenGL_Helper popengl_helper(argv,argc);
  World world(popengl_helper);

  popengl_helper.rate_set(30);
  popengl_helper.display_cb_set(world.frame_callback_w,&world);
}
```

```
/// LSU EE 4702-1 (Fall 2016), GPU Programming
//
 /// Homework 3 and 4 --- SOLUTION PRELIMINARY
 //
 //  See http://www.ece.lsu.edu/koppel/gpup/2016/hw03.pdf

/// Purpose
//
//   Demonstrate simulation of point masses connected by springs.

/// What Code Does

// Simulates balls connected by springs over a platform. Balls and
// springs can be initialized in different arrangements (called
// scenes). Currently scene 1 is a simple string of beads, and scenes
// 2, 3, and 4 are trusses. The platform consists of tiles, some are
// purple-tinted mirrors (showing a reflection of the ball), the
// others show the course syllabus.


///  Keyboard Commands
 //
 /// Object (Eye, Light, Ball) Location or Push
 //   Arrows, Page Up, Page Down
 //        Move object or push ball, depending on mode.
 //        With shift key pressed, motion is 5x faster.
 //   'e': Move eye.
 //   'l': Move light.
 //   'b': Move head (first) ball. (Change position but not velocity.)
 //   'B': Push head ball. (Add velocity.)
 //
 /// Eye Direction
 //   Home, End, Delete, Insert
 //   Turn the eye direction.
 //   Home should rotate eye direction up, End should rotate eye
 //   down, Delete should rotate eye left, Insert should rotate eye
 //   right.  The eye direction vector is displayed in the upper left.

 /// Simulation Options
 //  (Also see variables below.)
 //
 //  'w'    Twirl balls around axis formed by head and tail. (Prob 2 soln).
 //  '1'    Set up scene 1.
 //  '2'    Set up scene 2.
 //  '3'    Set up scene 3.
 //  'p'    Pause simulation. (Press again to resume.)
 //  ' '    (Space bar.) Advance simulation by 1/30 second.
 //  'S- '  (Shift-space bar.) Advance simulation by one time step.
 //  'h'    Freeze position of first (head) ball. (Press again to release.)
 //  't'    Freeze position of last (tail) ball. (Press again to release.)
 //  's'    Stop balls.
 //  'g'    Turn gravity on and off.
 //  'y'    Toggle value of opt_tryout1. Intended for experiments and debugging.
 //  'Y'    Toggle value of opt_tryout2. Intended for experiments and debugging.
 //  'F11'  Change size of green text (in upper left).
 //  'F12'  Write screenshot to file.

 //  'z'    Switch between render_link_1 and render_link_2.
```

```
/// Variables
//   Selected program variables can be modified using the keyboard.
//   Use "Tab" to cycle through the variable to be modified, the
//   name of the variable is displayed next to "VAR" on the bottom
//   line of green text.

//  'Tab' Cycle to next variable.
//  '`'   Cycle to previous variable.
//  '+'   Increase variable value.
//  '-'   Decrease variable value.
//
//  VAR Spring Constant - Set spring constant.
//  VAR Time Step Duration - Set physics time step.
//  VAR Air Resistance - Set air resistance.
//  VAR Light Intensity - The light intensity.
//  VAR Gravity - Gravitational acceleration. (Turn on/off using 'g'.)


#define GL_GLEXT_PROTOTYPES
#define GLX_GLXEXT_PROTOTYPES

#include <GL/gl.h>
#include <GL/glext.h>
#include <GL/glx.h>
#include <GL/glxext.h>
#include <GL/glu.h>
#include <GL/freeglut.h>

#include <gp/util.h>
#include <gp/glextfuncs.h>
#include <gp/coord.h>
#include <gp/shader.h>
#include <gp/pstring.h>
#include <gp/misc.h>
#include <gp/gl-buffer.h>
#include <gp/texture-util.h>
#include <gp/colors.h>

#include "util-containers.h"
#include "shapes.h"


///
/// Main Data Structures
///
//
// class World: All data about scene.


class World;


// Object Holding Ball State
//
class Ball {
public:
  Ball():velocity(pVect(0,0,0)),omega(pVect(0,0,0)),
         density(1.00746),
         fdt_to_do(0),
         locked(false),
         color(color_lsu_spirit_gold),contact(false)
         {
           orientation_set(pVect(0,1,0),0);
         }
```

```
  pCoor position;
  pVect velocity;
  pQuat orientation;
  pMatrix3x3 omatrix;
  pVect omega;                  // Spin rate and axis.

  int idx; // Position in balls_pos_rad;

  float mass;
  float mass_min; // Mass below which simulation is unstable.
  float radius;
  float radius_sq, radius_inv;  // Pre-computed based on radius.
  float density;
  float mass_inv;
  float fdt_to_do; // Radius / moment of inertia.

  bool locked;

  pVect force;
  pVect torque;
  pColor color;
  bool contact;                 // When true, ball rendered in gray.
  float spring_constant_sum;    // Used to compute minimum mass.

  void orientation_set(pNorm dir, float angle)
    { orientation_set(pQuat(dir,angle)); }
  void orientation_set(pQuat ori)
    {
      orientation = ori;
      omatrix = pMatrix3x3_Rotation(orientation);
    }
  void constants_update();
  pVect point_rot_vel(pNorm tact_dir);
  pVect point_rot_vel(pVect tact_dir);
  void apply_tan_force(pNorm tact_dir, pNorm force_dir, double force_mag);
  void push(pVect amt);
  void translate(pVect amt);
  void stop();
  void freeze();
};


void
Ball::constants_update()
{
  assert( radius );
  radius_sq = radius * radius;
  mass = 4.0 / 3 * M_PI * radius_sq * radius * density;
  radius_inv = 1 / radius;
  mass_inv = 1 / mass;
  // FYI, notice simplifications:
  //   const float mo_inertia = 0.4 * mass * radius_sq;
  //   fdt_to_do = radius / mo_inertia;
  fdt_to_do = radius_inv * 2.5 * mass_inv;
}

pVect
Ball::point_rot_vel(pNorm direction)
{
  /// Return velocity of point on surface of ball.
  //
  return radius * cross( omega, direction );
}
```

```
pVect
Ball::point_rot_vel(pVect rel_pos)
{
  /// Return velocity of point relative to center.
  //
  return cross( omega, rel_pos );
}

void
Ball::apply_tan_force(pNorm tact_dir, pNorm force_dir, double force_mag)
{
  torque += cross(tact_dir, force_mag * force_dir);
}

class Link {
public:
  Link(Ball *b1, Ball *b2):ball1(b1),ball2(b2),
     snapped(false),
     natural_color(color_lsu_spirit_purple),color(color_lsu_spirit_purple)
  { init(); }
  Link(Ball *b1, Ball *b2, pColor colorp):ball1(b1),ball2(b2),
     snapped(false),
     natural_color(colorp),color(colorp)
  { init(); }
  Link(Link *link, pVect cb1p, pVect cb2p, float drp):
    ball1(link->ball1), ball2(link->ball2), cb1(cb1p), cb2(cb2p),
    is_surface_connection(true), is_renderable(false),
    is_simulatable(true), distance_relaxed(drp),
    snapped(false){ link->is_simulatable = false; }
  void init()
    {
      assert( ball1->radius > 0 );
      assert( ball2->radius > 0 );
      is_simulatable = true;
      is_renderable = true;
      pNorm n12(ball1->position,ball2->position);
      const float rad_sum = ball1->radius + ball2->radius;
      is_surface_connection = n12.magnitude >= rad_sum;
      if ( !is_surface_connection )
        {
          distance_relaxed = n12.magnitude;
          cb1 = cb2 = pVect(0,0,0);
          return;
        }
      distance_relaxed = n12.magnitude - rad_sum;
      cb1 = ball1->radius * mult_MTV( ball1->omatrix, n12 );
      cb2 = ball2->radius * mult_MTV( ball2->omatrix, -n12 );
    }
  int serial;
  Ball* const ball1;
  Ball* const ball2;
  pVect cb1, cb2, b1_dir, b2_dir;
  bool is_surface_connection;
  bool is_renderable, is_simulatable;
  float distance_relaxed;
  bool snapped;
  pColor natural_color;
  pColor color;
};

// Declare containers and iterators for Balls and Links.
// (See util_container.h.)
```

```
//
typedef pVectorI<Link> Links;
typedef pVectorI_Iter<Link> LIter;
typedef pVectorI<Ball> Balls;
typedef pVectorI_Iter<Ball> BIter;

typedef pVector<pVect> pVects;
typedef pVector<pCoor> pCoors;


enum Render_Option { RO_Normally, RO_Simple, RO_Shadow_Volumes };
enum Shader_Option { SO_Fixed, SO_Phong, SO_ENUM_SIZE };

class World {
public:
  World(pOpenGL_Helper &fb):ogl_helper(fb){init();}
  void init();
  void init_graphics();
  static void frame_callback_w(void *moi){((World*)moi)->frame_callback();}
  void frame_callback();
  void render();
  void render_objects(Render_Option render_option);
  void objects_erase();
  void cb_keyboard();
  void modelview_update();

  pOpenGL_Helper& ogl_helper;
  pVariable_Control variable_control;
  pFrame_Timer frame_timer;
  double world_time;
  double last_frame_wall_time;
  float opt_time_step_duration;
  int time_step_count;
  float opt_gravity_accel;      // Value chosen by user.
  pVect gravity_accel;          // Set to zero when opt_gravity is false;
  bool opt_gravity;
  bool opt_head_lock, opt_tail_lock;
  bool opt_ride; // When true, move eye to ball_eye.
  bool opt_tryout1, opt_tryout2;  // For ad-hoc experiments.

  bool opt_ball_texture;

  // Tiled platform for ball.
  //
  float platform_xmin, platform_xmax, platform_zmin, platform_zmax;
  float platform_pi_xwidth_inv;
  pBuffer_Object<pVect> platform_tile_coords;
  pBuffer_Object<float> platform_tex_coords;
  pVect platform_normal;
  GLuint texid_hw;
  GLuint texid_syl;
  GLuint texid_emacs;
  bool opt_platform_texture;
  void platform_update();
  bool platform_collision_possible(pCoor pos);

  pCoor light_location;
  float opt_light_intensity;
  enum { MI_Eye, MI_Light, MI_Ball, MI_Ball_V, MI_COUNT } opt_move_item;
  bool opt_pause;
  bool opt_single_frame;      // Simulate for one frame.
  bool opt_single_time_step;  // Simulate for one time step.
  int viewer_shadow_volume;
```

```
pCoor eye_location;
pVect eye_direction;
pMatrix modelview;
pMatrix transform_mirror;

pVect adj_vector;
double adj_t_prev;
double adj_t_stop;
double adj_duration;

int opt_shader;
bool opt_shadows;
pShader *sp_fixed;          // Fixed functionality.
pShader *sp_phong;          // Basic stuff.

GLuint balls_pos_rad_bo;
GLuint balls_color_bo;
GLuint links_indices_bo;

pColor *balls_color;
pCoor *balls_pos_rad;
size_t balls_size;
size_t links_size;
int last_setup; // Last scene set up.
bool link_change;

void ball_setup_1();
void ball_setup_2();
void ball_setup_3();
void ball_setup_4();
void ball_setup_5();
void setup_at_end();
void time_step_cpu(double);
void balls_stop();
void balls_freeze();
void balls_translate(pVect amt, int idx);
void balls_translate(pVect amt);
void balls_push(pVect amt, int idx);
void balls_push(pVect amt);
void balls_twirl();
void lock_update();

Ball *make_marker(pCoor pos, pColor col);

Ball *ball_eye, *ball_gaze, *ball_down;

float opt_spring_constant;
float opt_air_resistance;
float distance_relaxed;
int chain_length;
Balls balls;
Ball *head_ball, *tail_ball;
Links links;
Sphere sphere;
Cylinder cyl;

/// 2016 Homework 3
//
Links link_new(Ball *ball1, Ball *ball2);

void render_link_1(Link *link);
```

```
  void render_link_2(Link *link);

  int opt_render_links;

  /// Homework 4:  This is a good place to declare items needed across calls.

  /// SOLUTION -- Homework 4
  //
  //  Buffer objects for attributes of curved links.
  GLuint link_bo_vtx, link_bo_nrm, link_bo_tco;
  //  Buffer objects for attributes of straight links.
  GLuint link_bo_vtx_s, link_bo_nrm_s;
  //  Number of vertices in curved links.
  int bo_s_num_vertices;
  //  Transformation matrix transforming saved straight link to a local space.
  pMatrix link_s_to_local;
};


void
World::init_graphics()
{
  ///
  /// Graphical Model Initialization
  ///

  balls_pos_rad = NULL;
  balls_pos_rad_bo = 0;
  balls_color = NULL;
  balls_color_bo = 0;
  balls_size = 0;
  links_size = 0;
  link_change = true;

  opt_platform_texture = true;
  opt_head_lock = false;
  opt_tail_lock = false;
  opt_ball_texture = true;
  opt_tryout1 = opt_tryout2 = false;

  eye_location = pCoor(24.2,11.6,-38.7);
  eye_direction = pVect(-0.42,-0.09,0.9);

  platform_xmin = -40; platform_xmax = 40;
  platform_zmin = -40; platform_zmax = 40;
  texid_syl = pBuild_Texture_File("gpup.png",false,255);
  texid_emacs = pBuild_Texture_File("mult.png", false,-1);
  texid_hw = pBuild_Texture_File("hw03-assign.png", false,255);

  opt_light_intensity = 100.2;
  light_location = pCoor(platform_xmax,platform_xmax,platform_zmin);

  variable_control.insert(opt_light_intensity,"Light Intensity");

  opt_move_item = MI_Eye;
  opt_pause = false;
  opt_single_time_step = false;
  opt_single_frame = false;

  sphere.init(35);

  platform_update();
  modelview_update();

  adj_t_stop = 0;
```

```
  adj_duration = 0.25;

  opt_render_links = 0;

  /// SOLUTION -- Homework 4
  link_bo_vtx = 0;
  bo_s_num_vertices = 0;
}


void
World::platform_update()
{
  const float tile_count = 19;
  const float ep = 1.00001;
  const float delta_x = ( platform_xmax - platform_xmin ) / tile_count * ep;
  const float zdelta = ( platform_zmax - platform_zmin ) / tile_count * ep;

  const float trmin = 0.05;
  const float trmax = 0.7;
  const float tsmin = 0;
  const float tsmax = 0.4;

  platform_normal = pVect(0,1,0);

  PStack<pVect> p_tile_coords;
  PStack<pVect> p1_tile_coords;
  PStack<float> p_tex_coords;
  bool even = true;

  for ( int i = 0; i < tile_count; i++ )
    {
      const float x0 = platform_xmin + i * delta_x;
      const float x1 = x0 + delta_x;
      const float y = 0;
      for ( float z = platform_zmin; z < platform_zmax; z += zdelta )
        {
          PStack<pVect>& t_coords = even ? p_tile_coords : p1_tile_coords;
          p_tex_coords += trmax; p_tex_coords += tsmax;
          t_coords += pVect(x0,y,z);
          p_tex_coords += trmax; p_tex_coords += tsmin;
          t_coords += pVect(x0,y,z+zdelta);
          p_tex_coords += trmin; p_tex_coords += tsmin;
          t_coords += pVect(x1,y,z+zdelta);
          p_tex_coords += trmin; p_tex_coords += tsmax;
          t_coords += pVect(x1,y,z);
          even = !even;
        }
    }

  while ( pVect* const v = p1_tile_coords.iterate() ) p_tile_coords += *v;

  platform_tile_coords.re_take(p_tile_coords);
  platform_tile_coords.to_gpu();
  platform_tex_coords.re_take(p_tex_coords);
  platform_tex_coords.to_gpu();
}

void
World::modelview_update()
{
  pMatrix_Translate center_eye(-eye_location);
  pMatrix_Rotation rotate_eye(eye_direction,pVect(0,0,-1));
  modelview = rotate_eye * center_eye;
  pMatrix reflect; reflect.set_identity(); reflect.rc(1,1) = -1;
```

```
  transform_mirror = modelview * reflect * invert(modelview);
}

void
World::render_objects(Render_Option option)
{
  const float shininess_ball = 5;
  pColor spec_color(0.2,0.2,0.2);

  if ( option == RO_Shadow_Volumes )
    viewer_shadow_volume = 0;

  if ( option == RO_Normally )
    {
      glLightf(GL_LIGHT0, GL_LINEAR_ATTENUATION, 1.0);
      if ( opt_ball_texture )
        {
          glEnable(GL_TEXTURE_2D);
          glBindTexture(GL_TEXTURE_2D,texid_emacs);
        }
      else
        {
          glDisable(GL_TEXTURE_2D);
        }
      glMaterialf(GL_FRONT_AND_BACK,GL_SHININESS,shininess_ball);
      glMaterialfv(GL_FRONT_AND_BACK,GL_SPECULAR,spec_color);
      glLightModeli(GL_LIGHT_MODEL_COLOR_CONTROL, GL_SEPARATE_SPECULAR_COLOR);
      glEnable(GL_COLOR_SUM);
    }

  cyl.apex_radius = 1; cyl.set_color(color_lsu_spirit_purple);
  if ( option == RO_Shadow_Volumes )
    {
#ifdef DB_SV
      glLightf(GL_LIGHT0, GL_LINEAR_ATTENUATION, 1.0);
      glEnable(GL_COLOR_SUM);
      glColor3f(0.5,0,0);
#endif
      cyl.light_pos = light_location;
      sphere.light_pos = light_location;

      for ( BIter ball(balls); ball; )
        sphere.render_shadow_volume(ball->radius,ball->position);

      for ( LIter link(links); link; )
        {
          if ( link->snapped ) continue;
          if ( !link->is_renderable ) continue;
          if ( !link->is_simulatable ) continue; // Should check if straight.
          Ball *const ball1 = link->ball1;
          Ball *const ball2 = link->ball2;
          cyl.render_shadow_volume
            (ball1->position,0.3*ball1->radius,
             ball2->position-ball1->position);
        }
    }
  else
    {
      sphere.opt_texture = opt_ball_texture;

      if ( opt_shader == SO_Phong )
        sp_phong->use();
      else if ( opt_shader == SO_Fixed )
        sp_fixed->use();
```

```
    if ( opt_shader != SO_Fixed )
      {
        glUniform2i(3, opt_tryout1, opt_tryout2);
      }
    for ( BIter ball(balls); ball; )
      {
        if ( ball->contact )
          sphere.color = color_gray;
        else if ( ball->mass > 0 && ball->mass < ball->mass_min )
          sphere.color = color_red;
        else if ( ball->mass > 0 && ball->locked )
          sphere.color = color_pale_green;
        else
          sphere.color = ball->color;
        sphere.render
          (ball->radius,ball->position,
           pMatrix_Rotation(ball->orientation));
      }
    glBindTexture(GL_TEXTURE_2D,texid_hw);
    for ( Link *link: links )
      {
        if ( !link->is_renderable ) continue;
        if ( opt_render_links == 0 )
          render_link_1(link); else render_link_2(link);
        continue;

        Ball *const ball1 = link->ball1;
        Ball *const ball2 = link->ball2;
        pVect dir1 = ball1->omatrix * link->cb1;
        pCoor pos1 = ball1->position + dir1;

        pVect dir2 = ball2->omatrix * link->cb2;
        pCoor pos2 = ball2->position + dir2;

        cyl.set_color(link->color);
        cyl.render(pos1,0.3*ball1->radius, pos2-pos1);
      }
  }

sp_fixed->use();

glDisable(GL_COLOR_SUM);
glLightModeli(GL_LIGHT_MODEL_COLOR_CONTROL, GL_SINGLE_COLOR);
glLightfv(GL_LIGHT0, GL_SPECULAR, color_black);

//
// Render Platform
//
const int half_elements = platform_tile_coords.elements >> 3 << 2;

glEnable(GL_TEXTURE_2D);

// Set up attribute (vertex, normal, etc.) arrays.
//
glBindTexture(GL_TEXTURE_2D,texid_syl);
platform_tile_coords.bind();
glVertexPointer(3, GL_FLOAT, sizeof(platform_tile_coords.data[0]), 0);
glEnableClientState(GL_VERTEX_ARRAY);
glNormal3fv(platform_normal);
platform_tex_coords.bind();
glTexCoordPointer(2, GL_FLOAT,2*sizeof(float), 0);
glEnableClientState(GL_TEXTURE_COORD_ARRAY);

// Write lighter-colored, textured tiles.
//
```

```
  glMaterialfv(GL_FRONT_AND_BACK,GL_SPECULAR,spec_color);
  glMaterialf(GL_FRONT_AND_BACK,GL_SHININESS,2.0);
  glColor3f(0.35,0.35,0.35);
  glColor3f(0.55,0.55,0.55);
  glDrawArrays(GL_QUADS,0,half_elements+4);

  glDisableClientState(GL_VERTEX_ARRAY);
  glDisableClientState(GL_TEXTURE_COORD_ARRAY);
  glBindBuffer(GL_ARRAY_BUFFER,0);
}

void
World::render()
{
  // Get any waiting keyboard commands.
  //
  cb_keyboard();

  // Start a timer object used for tuning this code.
  //
  frame_timer.frame_start();

  /// Emit a Graphical Representation of Simulation State
  //

  // Understanding of the code below not required for introductory
  // lectures.

  // That said, much of the complexity of the code is to show
  // the ball shadow and reflection.


  const int win_width = ogl_helper.get_width();
  const int win_height = ogl_helper.get_height();
  const float aspect = float(win_width) / win_height;

  glMatrixMode(GL_MODELVIEW);
  glLoadTransposeMatrixf(modelview);

  glMatrixMode(GL_PROJECTION);
  glLoadIdentity();
  // Frustum: left, right, bottom, top, near, far
  glFrustum(-.8,.8,-.8/aspect,.8/aspect,1,5000);

  glViewport(0, 0, win_width, win_height);
  pError_Check();

  glClearColor(0,0,0,0.5);
  glClearDepth(1.0);
  glClearStencil(0);
  glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT | GL_STENCIL_BUFFER_BIT );

  glEnable(GL_DEPTH_TEST);
  glDepthFunc(GL_LESS);
  glDisable(GL_BLEND);

  glLightModeli(GL_LIGHT_MODEL_TWO_SIDE,1);
  glLightfv(GL_LIGHT0, GL_POSITION, light_location);

  glLightf(GL_LIGHT0, GL_CONSTANT_ATTENUATION, 0.5);
  glLightf(GL_LIGHT0, GL_LINEAR_ATTENUATION, 1.0);
  glLightf(GL_LIGHT0, GL_QUADRATIC_ATTENUATION, 0);

  pColor ambient_color(0x555555);
```

```
  glLightModelfv(GL_LIGHT_MODEL_AMBIENT, ambient_color);
  glLightfv(GL_LIGHT0, GL_DIFFUSE, color_white * opt_light_intensity);
  glLightfv(GL_LIGHT0, GL_AMBIENT, color_black);
  glLightfv(GL_LIGHT0, GL_SPECULAR, color_white * opt_light_intensity);

  glEnable(GL_LIGHT0);
  glEnable(GL_LIGHTING);

  glEnable(GL_COLOR_MATERIAL);
  glColorMaterial(GL_FRONT_AND_BACK,GL_AMBIENT_AND_DIFFUSE);

  glShadeModel(GL_SMOOTH);

  // Common to all textures.
  //
  glActiveTexture(GL_TEXTURE0);
  glEnable(GL_TEXTURE_2D);
  glTexParameterf(GL_TEXTURE_2D,GL_TEXTURE_MIN_FILTER,
                  GL_LINEAR_MIPMAP_LINEAR);
  glTexParameterf(GL_TEXTURE_2D,GL_TEXTURE_MAG_FILTER,GL_LINEAR);
  glTexEnvi(GL_TEXTURE_ENV,GL_TEXTURE_ENV_MODE,GL_MODULATE);
  glTexParameterf(GL_TEXTURE_2D,GL_TEXTURE_WRAP_S,GL_REPEAT);
  glTexParameterf(GL_TEXTURE_2D,GL_TEXTURE_WRAP_T,GL_REPEAT);

  glEnable(GL_RESCALE_NORMAL);
  glEnable(GL_NORMALIZE);

  const double time_now = time_wall_fp();
  const bool blink_visible = int64_t(time_now*3) & 1;
# define BLINK(txt,pad) ( blink_visible ? txt : pad )

  ogl_helper.fbprintf("%s\n",frame_timer.frame_rate_text_get());

  ogl_helper.fbprintf
    ("Code Compiled: %s\n",
#ifdef __OPTIMIZE__
     "WITH OPTIMIZATION"
#else
     BLINK("WITHOUT OPTIMIZATION","")
#endif
     );

  ogl_helper.fbprintf
    ("Time Step: %8d  World Time: %11.6f  %s\n",
     time_step_count, world_time,
     opt_pause ? BLINK("PAUSED, 'p' to unpause, SPC or S-SPC to step.","") :
     "Press 'p' to pause."
     );

  ogl_helper.fbprintf
    ("Eye location: [%5.1f, %5.1f, %5.1f]  "
     "Eye direction: [%+.2f, %+.2f, %+.2f]\n",
     eye_location.x, eye_location.y, eye_location.z,
     eye_direction.x, eye_direction.y, eye_direction.z);

  Ball& ball = *balls[0];

  ogl_helper.fbprintf
    ("Head Ball Pos  [%5.1f,%5.1f,%5.1f] Vel [%+5.1f,%+5.1f,%+5.1f]\n",
     ball.position.x,ball.position.y,ball.position.z,
     ball.velocity.x,ball.velocity.y,ball.velocity.z );

  ogl_helper.fbprintf
    ("Links: %s  ('z' to change)  "
     "Tryout 1: %s  ('y' to change)  Tryout 2: %s  ('Y' to change)\n",
```

```
    opt_render_links == 0 ? "render_link_1" : "render_link_2",
    opt_tryout1 ? BLINK("ON","  ") : "OFF",
    opt_tryout2 ? BLINK("ON","  ") : "OFF");

  pVariable_Control_Elt* const cvar = variable_control.current;
  ogl_helper.fbprintf("VAR %s = %.5f  (TAB or `~' to change, +/- to adjust)\n",
                      cvar->name,cvar->get_val());

  const int half_elements = platform_tile_coords.elements >> 3 << 2;

  //
  // Render ball reflection.  (Will be blended with dark tiles.)
  //

  // Write stencil at location of dark (mirrored) tiles.
  //
  glDisable(GL_LIGHTING);
  glEnable(GL_STENCIL_TEST);
  glStencilFunc(GL_NEVER,2,2);
  glStencilOp(GL_REPLACE,GL_KEEP,GL_KEEP);
  platform_tile_coords.bind();
  glVertexPointer(3, GL_FLOAT, sizeof(platform_tile_coords.data[0]), 0);
  glEnableClientState(GL_VERTEX_ARRAY);
  glDrawArrays(GL_QUADS,half_elements+4,half_elements-4);
  glEnable(GL_LIGHTING);
  glDisableClientState(GL_VERTEX_ARRAY);
  glBindBuffer(GL_ARRAY_BUFFER,0);

  // Prepare to write only stenciled locations.
  //
  glStencilFunc(GL_EQUAL,2,2);
  glStencilOp(GL_KEEP,GL_KEEP,GL_KEEP);

  // Use a transform that reflects objects to other side of platform.
  //
  glMatrixMode(GL_PROJECTION);
  glPushMatrix();
  glMultTransposeMatrixf(transform_mirror);

  // Reflected front face should still be treated as the front face.
  //
  glFrontFace(GL_CW);

  render_objects(RO_Normally);

  glFrontFace(GL_CCW);
  glMatrixMode(GL_PROJECTION);
  glPopMatrix();
  glDisable(GL_STENCIL_TEST);


  // Setup texture for platform.
  //
  glBindTexture(GL_TEXTURE_2D,texid_syl);

  // Blend dark tiles with existing ball reflection.
  //
  glEnable(GL_STENCIL_TEST);
  glBlendEquation(GL_FUNC_ADD);
  glBlendFunc(GL_CONSTANT_ALPHA,GL_ONE_MINUS_CONSTANT_ALPHA); // src, dst
  glBlendColor(0,0,0,0.5);

  glDepthFunc(GL_ALWAYS);

  if ( opt_platform_texture )
```

```
  {
    glEnable(GL_TEXTURE_2D);
    platform_tex_coords.bind();
    glTexCoordPointer(2, GL_FLOAT,2*sizeof(float), 0);
    glEnableClientState(GL_TEXTURE_COORD_ARRAY);
  }

platform_tile_coords.bind();
glVertexPointer
  (3, GL_FLOAT,sizeof(platform_tile_coords.data[0]), 0);
glEnableClientState(GL_VERTEX_ARRAY);
glNormal3fv(platform_normal);

if ( opt_platform_texture ) glEnable(GL_TEXTURE_2D);

// Write lighter-colored, textured tiles.
//
glMaterialfv(GL_FRONT_AND_BACK,GL_SPECULAR,color_gray);
glMaterialf(GL_FRONT_AND_BACK,GL_SHININESS,2.0);
glColor3f(0.35,0.35,0.35);
glDrawArrays(GL_QUADS,0,half_elements+4);

// Write darker-colored, untextured, mirror tiles.
//
glEnable(GL_BLEND);
glMaterialfv(GL_FRONT_AND_BACK,GL_SPECULAR,color_white);
glMaterialf(GL_FRONT_AND_BACK,GL_SHININESS,20);
glDisable(GL_TEXTURE_2D);
glColor3fv(color_lsu_spirit_purple);
glDrawArrays(GL_QUADS,half_elements+4,half_elements-4);
glDisable(GL_BLEND);

glDisableClientState(GL_TEXTURE_COORD_ARRAY);
glDisableClientState(GL_VERTEX_ARRAY);
glBindBuffer(GL_ARRAY_BUFFER,0);
glDisable(GL_STENCIL_TEST);
glDepthFunc(GL_LESS);

glLightModelfv(GL_LIGHT_MODEL_AMBIENT, ambient_color);

if ( !opt_shadows )
  {
    // Render.
    //
    render_objects(RO_Normally);
  }
else
  {
    //
    // First pass, render using only ambient light.
    //
    glDisable(GL_LIGHT0);

    // Send balls, tiles, and platform to opengl.
    // Do occlusion test too.
    //
    render_objects(RO_Normally);

    //
    // Second pass, add on light0.
    //

    // Turn off ambient light, turn on light 0.
    //
    glLightModelfv(GL_LIGHT_MODEL_AMBIENT, color_black);
```

```
      glEnable(GL_LIGHT0);


      glClear(GL_STENCIL_BUFFER_BIT);

      // Make sure that only stencil buffer written.
      //
#ifndef DB_SV
      glColorMask(false,false,false,false);
      glDepthMask(false);

      // Don't waste time computing lighting.
      //
      glDisable(GL_LIGHTING);
#endif
      glDisable(GL_TEXTURE_2D);

      // Set up stencil test to count shadow volume surfaces: plus 1 for
      // entering the shadow volume, minus 1 for leaving the shadow
      // volume.
      //
      glEnable(GL_STENCIL_TEST);
      // sfail, dfail, dpass
      glStencilOpSeparate(GL_FRONT,GL_KEEP,GL_KEEP,GL_INCR_WRAP);
      glStencilOpSeparate(GL_BACK,GL_KEEP,GL_KEEP,GL_DECR_WRAP);
      glStencilFuncSeparate(GL_FRONT_AND_BACK,GL_ALWAYS,1,-1); // ref, mask

      // Write stencil with shadow locations based on shadow volumes
      // cast by light0 (light_location).  Shadowed locations will
      // have a positive stencil value.  Routine will set viewer_shadow_volume
      // to the number of view volumes containing the eye.
      //
      render_objects(RO_Shadow_Volumes);

      glEnable(GL_LIGHTING);
      glColorMask(true,true,true,true);
      glDepthMask(true);

      // Use stencil test to prevent writes to shadowed areas.
      //
      glStencilOp(GL_KEEP,GL_KEEP,GL_KEEP);
      glStencilFunc(GL_EQUAL,viewer_shadow_volume,-1); // ref, mask

      // Allow pixels to be re-written.
      //
      glDepthFunc(GL_LEQUAL);
      glEnable(GL_BLEND);
      glBlendEquation(GL_FUNC_ADD);
      glBlendFunc(GL_ONE,GL_ONE);

      // Render.
      //
      render_objects(RO_Normally);

      glDisable(GL_BLEND);
      glDisable(GL_STENCIL_TEST);

    }


  // Render Marker for Light Source
  //
  insert_tetrahedron(light_location,0.5);

  pError_Check();
```

```
  glColor3f(0.5,1,0.5);

  frame_timer.frame_end();

  ogl_helper.user_text_reprint();

  glutSwapBuffers();
}


void
World::cb_keyboard()
{
  if ( !ogl_helper.keyboard_key ) return;
  pVect adjustment(0,0,0);
  pVect user_rot_axis(0,0,0);
  const bool shift = ogl_helper.keyboard_shift;
  const float move_amt = shift ? 2.0 : 0.4;

  switch ( ogl_helper.keyboard_key ) {
  case FB_KEY_LEFT: adjustment.x = -move_amt; break;
  case FB_KEY_RIGHT: adjustment.x = move_amt; break;
  case FB_KEY_PAGE_UP: adjustment.y = move_amt; break;
  case FB_KEY_PAGE_DOWN: adjustment.y = -move_amt; break;
  case FB_KEY_DOWN: adjustment.z = move_amt; break;
  case FB_KEY_UP: adjustment.z = -move_amt; break;
  case FB_KEY_DELETE: user_rot_axis.y = 1; break;
  case FB_KEY_INSERT: user_rot_axis.y =  -1; break;
  case FB_KEY_HOME: user_rot_axis.x = 1; break;
  case FB_KEY_END: user_rot_axis.x = -1; break;
  case '1': ball_setup_1(); break;
  case '2': ball_setup_2(); break;
  case '3': ball_setup_3(); break;
  case '4': ball_setup_4(); break;
  case '5': ball_setup_5(); break;
  case 'b': opt_move_item = MI_Ball; break;
  case 'B': opt_move_item = MI_Ball_V; break;
  case 'e': case 'E': opt_move_item = MI_Eye; break;
  case 'g': case 'G': opt_gravity = !opt_gravity; break;
  case 'h': case 'H': opt_head_lock = !opt_head_lock; break;
  case 't': case 'T': opt_tail_lock = !opt_tail_lock; break;
  case 'l': case 'L': opt_move_item = MI_Light; break;
  case 'n': case 'N': opt_platform_texture = !opt_platform_texture; break;
  case 'p': case 'P': opt_pause = !opt_pause; break;
  case 'r': case 'R': opt_ride = !opt_ride; break;
  case 's': case 'S': balls_stop(); break;
  case 'v': case 'V': opt_shader++; if ( opt_shader > 1 ) opt_shader = 0; break;
  case 'w': case 'W': balls_twirl(); break;
  case 'y': opt_tryout1 = !opt_tryout1; break;
  case 'Y': opt_tryout2 = !opt_tryout2; break;
  case 'z': opt_render_links++;
    if ( opt_render_links > 1 ) opt_render_links = 0;
    break;
  case ' ':
    if ( shift ) opt_single_time_step = true; else opt_single_frame = true;
    opt_pause = true;
    break;
  case 9: variable_control.switch_var_right(); break;
  case 96: variable_control.switch_var_left(); break; // `, until S-TAB works.
  case '-':case '_': variable_control.adjust_lower(); break;
  case '+':case '=': variable_control.adjust_higher(); break;
  default: printf("Unknown key, %d\n",ogl_helper.keyboard_key); break;
  }
```

```
  gravity_accel.y = opt_gravity ? -opt_gravity_accel : 0;

  lock_update();

  // Update eye_direction based on keyboard command.
  //
  if ( user_rot_axis.x || user_rot_axis.y )
    {
      pMatrix_Rotation rotall(eye_direction,pVect(0,0,-1));
      user_rot_axis *= invert(rotall);
      eye_direction *= pMatrix_Rotation(user_rot_axis, M_PI * 0.03);
      modelview_update();
    }

  // Update eye_location based on keyboard command.
  //
  if ( adjustment.x || adjustment.y || adjustment.z )
    {
      const double angle =
        fabs(eye_direction.y) > 0.99
        ? 0 : atan2(eye_direction.x,-eye_direction.z);
      pMatrix_Rotation rotall(pVect(0,1,0),-angle);
      adjustment *= rotall;

      switch ( opt_move_item ){
      case MI_Ball:
        adj_vector = adjustment;
        if ( adj_t_stop == 0 )
          adj_t_prev = adj_t_stop = world_time;
        adj_t_stop += adj_duration;
        break;
      case MI_Ball_V: balls_push(adjustment,0); break;
      case MI_Light: light_location += adjustment; break;
      case MI_Eye: eye_location += adjustment; break;
      default: break;
      }
      modelview_update();
    }

}



void
World::init()
{
  chain_length = 14;

  opt_time_step_duration = 0.0003;
  variable_control.insert(opt_time_step_duration,"Time Step Duration");

  distance_relaxed = 15.0 / chain_length;
  opt_spring_constant = 150;
  variable_control.insert(opt_spring_constant,"Spring Constant");

  opt_gravity_accel = 9.8;
  opt_gravity = true;
  gravity_accel = pVect(0,-opt_gravity_accel,0);
  variable_control.insert(opt_gravity_accel,"Gravity");

  opt_air_resistance = 0.04;
  variable_control.insert(opt_air_resistance,"Air Resistance");

  world_time = 0;
```

```
  time_step_count = 0;
  last_frame_wall_time = time_wall_fp();
  frame_timer.work_unit_set("Steps / s");

  opt_shadows = true;
  opt_shader = SO_Fixed;

  ball_eye = NULL;
  opt_ride = false;

  init_graphics();

  // Declared like a programmable shader, but used for fixed-functionality.
  //
  sp_fixed = new pShader();

  sp_phong = new pShader
    ("hw03-shdr.cc",// File holding shader program.
     "vs_main(); ",     // Name of vertex shader main routine.
     "gs_main_simple();",
     "fs_main();"       // Name of fragment shader main routine.
     );
  ball_setup_2();
  lock_update();
}

Ball*
World::make_marker(pCoor position, pColor color)
{
  Ball* const ball = new Ball;
  ball->position = position;
  ball->locked = true;
  ball->velocity = pVect(0,0,0);
  ball->radius = 0.2;
  ball->mass = 0;
  ball->contact = false;
  ball->color = color;
  return ball;
}

void
World::lock_update()
{
  // This routine called when options like opt_head_lock might have
  // changed.

  // Update locked status.
  //
  if ( head_ball ) head_ball->locked = opt_head_lock;
  if ( tail_ball ) tail_ball->locked = opt_tail_lock;

  // Re-compute minimum mass needed for stability.
  //
  for ( BIter ball(balls); ball; ) ball->spring_constant_sum = 0;
  const double dtis = pow( opt_time_step_duration, 2 );
  for ( LIter link(links); link; )
    {
      Ball* const b1 = link->ball1;
      Ball* const b2 = link->ball2;
      b1->spring_constant_sum += opt_spring_constant;
      b2->spring_constant_sum += opt_spring_constant;
    }
  for ( BIter ball(balls); ball; )
    {
      ball->mass_min = ball->spring_constant_sum * dtis;
```

```
      ball->constants_update();
    }
}

void
World::balls_twirl()
{
  if ( !head_ball || !tail_ball ) return;

  pNorm axis(head_ball->position, tail_ball->position);

  for ( BIter ball(balls); ball; )
    {
      pVect b_to_top(ball->position,head_ball->position);
      const float dist_along_axis = dot(b_to_top,axis);
      const float lsq = b_to_top.mag_sq() - dist_along_axis * dist_along_axis;
      if ( lsq <= 1e-5 ) { ball->velocity = pVect(0,0,0); continue; }
      const float dist_to_axis = sqrt(lsq);
      pNorm rot_dir = cross(b_to_top,axis);
      ball->velocity += 2 * dist_to_axis * rot_dir;
    }
}

void
World::objects_erase()
{
  ball_eye = NULL;
  link_change = true;
  balls.erase();
  links.erase();
}

Links
World::link_new(Ball *ball1, Ball *ball2)
{
  Links links_rv;
  assert( ball1->radius > 0 );
  assert( ball2->radius > 0 );

  Link* const rlink = new Link(ball1,ball2);
  links_rv += rlink;

  pNorm n12(ball1->position,ball2->position);
  const float rad_sum = ball1->radius + ball2->radius;
  pMatrix3x3 b1rot = ball1->omatrix;
  pMatrix3x3 b2rot = ball2->omatrix;
  pCoor ctr = ball1->position
    + ( ball1->radius + 0.5 * ( n12.magnitude - rad_sum ) ) * n12;

  pNorm b1_y = b1rot * pVect(0,1,0);
  pNorm b1_x = b1rot * pVect(1,0,0);
  bool b1_dir_is_y = fabs(dot(b1_y,n12)) < 0.999;
  pNorm b1_dir = b1_dir_is_y ? b1_y : b1_x;

  pNorm con_x = cross(b1_dir,n12);
  pVect con_y = cross(n12,con_x);
  rlink->b1_dir = mult_MTV( b1rot, con_y );
  rlink->b2_dir = mult_MTV( b2rot, con_y );

  const float lrad = 0.2 * ball1->radius;

  for ( int i=0; i<3; i++ )
    {
      const double theta = i * 2 * M_PI / 3;
```

```
      pVect convec = lrad * ( cos(theta) * con_x + sin(theta) * con_y );
      pCoor con = ctr + convec;
      pVect cb1 = mult_MTV( b1rot, con - ball1->position );
      pVect cb2 = mult_MTV( b2rot, con - ball2->position );
      links_rv += new Link(rlink, cb1, cb2, 0);
    }
  return links_rv;
}

///
/// Physical Simulation Code
///

 /// Initialize Simulation
//

void
World::ball_setup_1()
{
  // Arrange and size balls to form a rectangular spiral.

  last_setup = 1;

  pCoor first_pos(13.4,10.8f,-9.2);

  // Remove objects from the simulated objects lists, balls and links.
  // The delete operator is used on objects in the lists.
  //
  objects_erase();

  pNorm to_eye(first_pos,eye_location);
  pNorm to_rt = pVect(1,0,0);
  pNorm to_up = pVect(0,0,1);

  for ( int i=0; i<chain_length; i++ )
    {
      // Construct a new ball and add it to the simulated objects list (balls).
      //
      Ball* const ball = balls += new Ball;

      pCoor ref_pos = i == 0 ? first_pos : balls[balls-2]->position;

      pVect dirs[4] = {to_rt,to_up,-to_rt,-to_up};

      pVect dir = dirs[i&3];

      // Initialize position and other information.
      //
      ball->position = ref_pos + (1+i/2) * distance_relaxed * dir;
      ball->locked = false;
      ball->velocity = pVect(0,0,0);
      ball->radius = 0.3;
      ball->contact = false;

      // If it's not the first ball link it to the previous ball.
      if ( i > 0 ) links += link_new( ball, balls[i-1] );
    }

  // The balls pointed to by head_ball and tail_ball can be manipulated
  // using the user interface (by pressing 'h' or 't', for example).
  // Set these variables.
  //
  head_ball = balls[0];
  tail_ball = balls[balls-1];
```

```
  opt_head_lock = true;    // Head ball will be frozen in space.
  opt_tail_lock = false;   // Tail ball can move freely.
  setup_at_end();
}



void
World::ball_setup_2()
{
  // Arrange balls to form sort of an umbrella.

  last_setup = 2;

  pCoor first_pos(17.2,14.8f,-20.2);

  pVect dir_dn(0.5,first_pos.y/8,0.5);
  pVect dir_x(3*distance_relaxed,0,0);
  pVect dir_z(0,0,3*distance_relaxed);

  // Remove objects from the simulated objects lists, balls and links.
  // The delete operator is used on objects in the lists.
  //
  objects_erase();

  auto nb =
    [&] (pCoor pos) {
      Ball* const ball = new Ball;
      ball->position = pos;
      ball->locked = false;
      ball->velocity = pVect(0,0,0);
      ball->radius = 0.3;
      ball->contact = false;
      balls += ball;
      return ball;
      };

  head_ball = nb(first_pos);
  for ( int j: {-1,1} )
    for ( int i: {-1,0,1} )
      {
        Ball* const ball = nb(first_pos - dir_dn + i * dir_x + j * dir_z );
        if ( i != 0 ) links += link_new(head_ball,ball);
        if ( i != -1 ) links += link_new(balls[balls-2],ball);
        if ( j == 1 && i != 0 )  links += link_new(balls[balls-4],ball);
      }

  Ball* const tail_start = nb(first_pos - dir_dn);
  links += link_new(balls[balls-6],tail_start);
  links += link_new(balls[balls-3],tail_start);
  pCoor last_pos = tail_start->position;
  for ( int i=1; i<5; i++ )
    {
      Ball* const ball = nb(last_pos - dir_dn * (0.5 + drand48()) );
      links += link_new(balls[balls-2],ball);
      last_pos = ball->position;
    }

  tail_ball = balls[balls-1];

  opt_head_lock = true;    // Head ball will be frozen in space.
  opt_tail_lock = false;   // Tail ball can move freely.
  setup_at_end();
}
```

```
void
World::ball_setup_3()
{
  last_setup = 3;
  objects_erase();
  setup_at_end();
}


void
World::ball_setup_4()
{
  last_setup = 4;
  objects_erase();
  setup_at_end();
}

void
World::ball_setup_5()
{
  last_setup = 5;
  objects_erase();
  setup_at_end();
}


void
World::setup_at_end()
{
}


 /// Advance Simulation State by delta_t Seconds
//
void
World::time_step_cpu(double delta_t)
{
  time_step_count++;

  // Smoothly move ball in response to user input.
  //
  if ( adj_t_stop )
    {
      const double dt = min(world_time,adj_t_stop) - adj_t_prev;
      pVect adj = dt/adj_duration * adj_vector;
      balls_translate(adj,0);
      adj_t_prev = world_time;
      if ( world_time >= adj_t_stop ) adj_t_stop = 0;
    }

  for ( BIter ball(balls); ball; )
    {
      assert( ball->fdt_to_do );
      ball->force = ball->mass * gravity_accel;
      ball->torque = pVect(0,0,0);
    }

  for ( Link *link: links )
    {
      if ( !link->is_simulatable ) continue;
      // Spring Force from Neighbor Balls
      //
      Ball* const ball1 = link->ball1;
```

```
    Ball* const ball2 = link->ball2;

    // Find position and velocity of the point where the link touches
    // the surface of ball 1 ...
    //
    pVect dir1 = ball1->omatrix * link->cb1;
    pCoor pos1 = ball1->position + dir1;
    pVect vel1 = ball1->velocity + ball1->point_rot_vel(dir1);

    // ... and ball 2.
    //
    pVect dir2 = ball2->omatrix * link->cb2;
    pCoor pos2 = ball2->position + dir2;
    pVect vel2 = ball2->velocity + ball2->point_rot_vel(dir2);

    // Construct a normalized (Unit) Vector from ball to neighbor
    // based on link connection points and ball centers.
    //
    pNorm link_dir(pos1,pos2);
    pNorm c_to_c(ball1->position,ball2->position);

    const float link_length = link_dir.magnitude;

    // Compute the speed of ball's end of link towards neighbor's end of link.
    //
    pVect delta_v = vel2 - vel1;
    float delta_s = dot( delta_v, link_dir );

    // Compute by how much the spring is stretched (positive value)
    // or compressed (negative value).
    //
    const float spring_stretch = link_length - link->distance_relaxed;

    // Determine whether spring is gaining energy (whether its length
    // is getting further from its relaxed length).
    //
    const bool gaining_e = ( delta_s > 0.0 ) == ( spring_stretch > 0 );

    // Use a smaller spring constant when spring is loosing energy,
    // a quick and dirty way of simulating energy loss due to spring
    // friction.
    //
    const float spring_constant =
      gaining_e ? opt_spring_constant : opt_spring_constant * 0.7;

    const float force_mag = spring_constant * spring_stretch;
    pVect spring_force_12 = force_mag * link_dir;

    // Apply forces affecting linear momentum.
    //
    ball1->force += spring_force_12;
    ball2->force -= spring_force_12;

    if ( ! link->is_surface_connection ) continue;

    // Apply torque.
    //
    ball1->apply_tan_force(dir1,link_dir,force_mag);
    ball2->apply_tan_force(dir2,link_dir,-force_mag);

  }

///
/// Update Position of Each Ball
///
```

```
  for ( BIter ball(balls); ball; )
    {
      if ( ball->locked )
        {
          ball->velocity = pVect(0,0,0);
          ball->omega = pVect(0,0,0);
          continue;
        }

      // Update Velocity
      //
      // This code assumes that force on ball is constant over time
      // step. This is clearly wrong when balls are moving with
      // respect to each other because the springs are changing
      // length. This inaccuracy will make the simulation unstable
      // when spring constant is large for the time step.
      //
      const float mass = max( ball->mass, ball->mass_min );

      pVect delta_v = ( ball->force / mass ) * delta_t;

      if ( platform_collision_possible(ball->position) && ball->position.y < 0 )
        {
          const float spring_constant_plat =
            ball->velocity.y < 0 ? 100000 : 50000;
          const float fric_coefficient = 0.1;
          const float force_up = -ball->position.y * spring_constant_plat;
          const float delta_v_up = force_up / mass * delta_t;
          const float fric_force_mag = fric_coefficient * force_up;
          pNorm surface_v(ball->velocity.x,0,ball->velocity.z);
          const float delta_v_surf = fric_force_mag / mass * delta_t;

          if ( delta_v_surf > surface_v.magnitude )
            {
              // Ignoring other forces?
              delta_v =
                pVect(-ball->velocity.x,delta_v.y,-ball->velocity.z);
            }
          else
            {
              delta_v -= delta_v_surf * surface_v;
            }
          delta_v.y += delta_v_up;
        }


      ball->velocity += delta_v;

      // Air Resistance
      //
      const double fs = pow(1+opt_air_resistance,-delta_t);
      ball->velocity *= fs;

      // Update Position
      //
      // Assume that velocity is constant.
      //
      ball->position += ball->velocity * delta_t;

      ball->omega += delta_t * ball->fdt_to_do * ball->torque;

      pNorm axis(ball->omega);

      // Update Orientation
```

```
      //
      // If ball isn't spinning fast skip expensive rotation.
      //
      if ( axis.mag_sq < 0.000001 ) continue;

      // Update ball orientation.
      //
      ball->orientation_set
        ( pQuat(axis,delta_t * axis.magnitude) * ball->orientation );
    }
}

bool
World::platform_collision_possible(pCoor pos)
{
  // Assuming no motion in x or z axes.
  //
  return pos.x >= platform_xmin && pos.x <= platform_xmax
    && pos.z >= platform_zmin && pos.z <= platform_zmax;
}

 /// External Modifications to State
//
//   These allow the user to play with state while simulation
//   running.

// Move the ball.
//
void Ball::translate(pVect amt) {position += amt;}

// Add velocity to the ball.
//
void Ball::push(pVect amt) {velocity += amt;}

// Set the velocity to zero.
//
void Ball::stop() {velocity = pVect(0,0,0); }

// Set the velocity and rotation (not yet supported) to zero.
//
void Ball::freeze() {velocity = pVect(0,0,0); }



void World::balls_translate(pVect amt,int b){head_ball->translate(amt);}
void World::balls_push(pVect amt,int b){head_ball->push(amt);}
void World::balls_translate(pVect amt)
{ for ( BIter ball(balls); ball; ) ball->translate(amt); }
void World::balls_push(pVect amt)
{ for ( BIter ball(balls); ball; ) ball->push(amt); }
void World::balls_stop()
{ for ( BIter ball(balls); ball; ) ball->stop(); }
void World::balls_freeze(){balls_stop();}

void
World::render_link_1(Link *link)
{
  /// HOMEWORK 3:  Put solution to Problem 1 here.

  Ball *const ball1 = link->ball1;
  Ball *const ball2 = link->ball2;

  // Direction of link relative to center off ball.
```

```
//
pVect dir1 = ball1->omatrix * link->cb1;

// Place on surface of ball where link connects.
//
pCoor pos1 = ball1->position + dir1;

// Direction and connection location for ball 2.
//
pVect dir2 = ball2->omatrix * link->cb2;
pCoor pos2 = ball2->position + dir2;

pVect p1p2(pos1,pos2);
pNorm p1p2n(p1p2);

// Number of segments used to construct link.  Each segment is
// approximately a cylinder.
//
const int segments = 15;

// Number of sides of each cylinder.
//
const int sides = 20;

const float delta_tee = 1.0 / segments;

// Radius of link.
//
const float rad = ball1->radius * 0.3;

/// SOLUTION -- Homework 3
// Compute scale factors for texture.
const float tex_margin = 0.1;
const float tex_aspect_ratio = 8.5 / 11;
const float page_width_o = 2.5;  // Width of texture in object-space coords.
const float tex_t_scale = link->distance_relaxed / page_width_o;
const float tex_angle_scale = rad * tex_aspect_ratio / page_width_o;

pNorm dirn1(dir1);
pNorm dirn2(dir2);

// Vectors used to describe the cubic Hermite curve.
//
pVect v1 = p1p2n.magnitude * dirn1;
pVect v2 = p1p2n.magnitude * dirn2;

// Convert link's local x and y axes to global coordinates.
//
pVect b1_ydir = ball1->omatrix * link->b1_dir;

pVect b2_ydir = ball2->omatrix * link->b2_dir;

pCoor pts[sides+1];
pVect vecs[sides+1];

glColorMaterial(GL_FRONT,GL_AMBIENT_AND_DIFFUSE);
glMaterialfv(GL_BACK,GL_AMBIENT_AND_DIFFUSE,color_red);

glBegin(GL_TRIANGLE_STRIP);
glColor3fv( color_light_gray );

for ( int i=0; i<=segments; i++ )
  {
    const float t = i * delta_tee;
    const float t2 = t * t;
```

```
      const float t3 = t2 * t;

      // Cubic Hermite interpolation.
      // Compute point along core of link.
      //
      pCoor ctr =
        ( 2*t3 - 3*t2 + 1 ) * pos1 + (-2*t3 + 3*t2 + 0 ) * pos2
        + (t3 - 2*t2 + t ) * v1 - (t3 - t2 ) * v2;

      // Compute direction of link at this point.
      //
      pNorm tan =
        ( 6*t2 - 6*t ) * pos1 + (-6*t2 + 6*t ) * pos2
        + (3*t2 - 4*t + 1 ) * v1 - (3*t2 - 2*t ) * v2;

      // Compute local x and y axes for drawing a cylinder.
      //
      pVect ydir = (1-t) * b1_ydir + t * b2_ydir;
      pNorm norm = cross(tan,ydir);
      pVect binorm = cross(tan,norm);

      for ( int j=0; j<=sides; j++ )
        {
          const float theta = j * ( 2 * M_PI / sides );
          pCoor pt = pts[j];
          pVect vec = vecs[j];
          vecs[j] = cosf(theta) * norm + sinf(theta) * binorm;
          pts[j] = ctr + rad * vecs[j];
          if ( i == 0 ) continue;

          /// SOLUTION -- Homework 3
          glTexCoord2d
            (tex_margin + t * tex_t_scale, 0.5 + theta * tex_angle_scale );

          glNormal3fv(vecs[j]);
          glVertex3fv(pts[j]);

          /// SOLUTION -- Homework 3
          glTexCoord2d
            (tex_margin + (t-delta_tee)*tex_t_scale,
             0.5 + theta * tex_angle_scale );

          glNormal3fv(vec);
          glVertex3fv(pt);
        }
    }
  glEnd();
}

void
World::render_link_2(Link *link)
{
  /// HOMEWORK 4:  Put solution to Problem 1 in this routine
  //  and else where.
  //
  Ball *const ball1 = link->ball1;
  Ball *const ball2 = link->ball2;

  // Direction of link relative to center off ball.
  //
  pVect dir1 = ball1->omatrix * link->cb1;

  // Place on surface of ball where link connects.
  //
  pCoor pos1 = ball1->position + dir1;
```

```
// Direction and connection location for ball 2.
//
pVect dir2 = ball2->omatrix * link->cb2;
pCoor pos2 = ball2->position + dir2;

pVect p1p2(pos1,pos2);
pNorm p1p2n(p1p2);

// Number of segments used to construct link.  Each segment is
// approximately a cylinder.
//
const int segments = 15;

// Number of sides of each cylinder.
//
const int sides = 20;

const float delta_tee = 1.0 / segments;

// Radius of link.
//
const float rad = ball1->radius * 0.3;

// Compute scale factors for texture.
const float tex_aspect_ratio = 8.5 / 11;
const float page_width_o = 2.5;  // Width of texture in object-space coords.
const float tex_t_scale = link->distance_relaxed / page_width_o;
const float tex_angle_scale = rad * tex_aspect_ratio / page_width_o;

pNorm dirn1(dir1);
pNorm dirn2(dir2);

// Vectors used to describe the cubic Hermite curve.
//
pVect v1 = p1p2n.magnitude * dirn1;
pVect v2 = p1p2n.magnitude * dirn2;

// Convert link's local x and y axes to global coordinates.
//
pVect b1_ydir = ball1->omatrix * link->b1_dir;
pVect b1_xdir = cross(b1_ydir,p1p2n);

pVect b2_ydir = ball2->omatrix * link->b2_dir;

pCoor pts[sides+1];
pVect vecs[sides+1];

/// SOLUTION -- Homework 4
pNorm c1c2(ball1->position,ball2->position);
const float thr = 0.999;
const bool straight_link =
  dot( c1c2, dirn1 ) > thr && dot( c1c2, dirn2 ) < -thr;

/// SOLUTION -- Homework 4 Remove OpenGL glBegin command.

pCoors coords;
pVects norms;
pVector<float> tcoords;

if ( !straight_link || bo_s_num_vertices == 0 ) // SOLUTION
for ( int i=0; i<=segments; i++ )
  {
    const float t = i * delta_tee;
    const float t2 = t * t;
```

```
      const float t3 = t2 * t;

      // Cubic Hermite interpolation.
      // Compute point along core of link.
      //
      pCoor ctr =
        ( 2*t3 - 3*t2 + 1 ) * pos1 + (-2*t3 + 3*t2 + 0 ) * pos2
        + (t3 - 2*t2 + t ) * v1 - (t3 - t2 ) * v2;

      // Compute direction of link at this point.
      //
      pNorm tan =
        ( 6*t2 - 6*t ) * pos1 + (-6*t2 + 6*t ) * pos2
        + (3*t2 - 4*t + 1 ) * v1 - (3*t2 - 2*t ) * v2;

      // Compute local x and y axes for drawing a cylinder.
      //
      pVect ydir = (1-t) * b1_ydir + t * b2_ydir;
      pNorm norm = cross(tan,ydir);
      pVect binorm = cross(tan,norm);

      for ( int j=0; j<=sides; j++ )
        {
          const float theta = j * ( 2 * M_PI / sides );
          pCoor pt = pts[j];
          pVect vec = vecs[j];
          vecs[j] = cosf(theta) * norm + sinf(theta) * binorm;
          pts[j] = ctr + rad * vecs[j];
          if ( i == 0 ) continue;

          /// SOLUTION -- Homework 4
          //
          //  Remove calls to glNormal3fv and glVertex3fv and replace with
          //  the code that saves normals and coordinates in lists. Also
          //  add code saving texture coordinates.

          tcoords += t * tex_t_scale;
          tcoords += 0.5 + theta * tex_angle_scale;

          norms += vecs[j];
          coords += pts[j];

          tcoords += ( t - delta_tee ) * tex_t_scale;
          tcoords += 0.5 + theta * tex_angle_scale;

          norms += vec;
          coords += pt;

        }
    }

  /// SOLUTION -- Homework 4

  const bool bo_tco_empty = !link_bo_vtx;
  if ( !link_bo_vtx )
    glGenBuffers(5,&link_bo_vtx);

  GLuint bo_vtx = straight_link ? link_bo_vtx_s : link_bo_vtx;
  GLuint bo_nrm = straight_link ? link_bo_nrm_s : link_bo_nrm;

  if ( !straight_link || !bo_s_num_vertices )
    {
      if ( straight_link )
        {
          bo_s_num_vertices = coords.size();
```

```
        pMatrix_Translate center(-pos1);
        pMatrix_Rows rot_to_local
          ((1/rad)*b1_xdir, (1/rad)*b1_ydir, p1p2n/p1p2n.magnitude);
        link_s_to_local = rot_to_local * center;
      }

    glBindBuffer(GL_ARRAY_BUFFER, bo_vtx);
    glBufferData
      (GL_ARRAY_BUFFER,           // Kind of buffer object.
       sizeof(coords[0])*coords.size(), coords.data(),
       GL_STATIC_DRAW);
    glBindBuffer(GL_ARRAY_BUFFER, bo_nrm);
    glBufferData
      (GL_ARRAY_BUFFER,           // Kind of buffer object.
       sizeof(norms[0])*norms.size(), norms.data(),
       GL_STATIC_DRAW);
  }
if ( bo_tco_empty )
  {
    glBindBuffer(GL_ARRAY_BUFFER, link_bo_tco);
    glBufferData
      (GL_ARRAY_BUFFER,           // Kind of buffer object.
       sizeof(tcoords[0])*tcoords.size(), tcoords.data(),
       GL_STATIC_DRAW);
  }

const int num_vtx = straight_link ? bo_s_num_vertices : coords.size();

if ( straight_link )
  {
    // Multiply modelview matrix by a matrix that will transform the
    // coordinates of the straight links that we saved in the buffer
    // objects link_bo_vtx_s and link_bo_nrm_s to approximately
    // match the coordinates of the straight link we need to render
    // now.

    glMatrixMode(GL_MODELVIEW);
    glPushMatrix();
    pMatrix_Translate center(pos1);
    pMatrix_Cols rot_to_global(rad*b1_xdir, rad*b1_ydir, p1p2);
    pMatrix m = center * rot_to_global* link_s_to_local;
    glMultTransposeMatrixf(m);
  }

glColor3fv( straight_link ? color_pale_green : color_light_gray );
glEnableClientState(GL_VERTEX_ARRAY);
glBindBuffer(GL_ARRAY_BUFFER, bo_vtx);
glVertexPointer( 3, GL_FLOAT, sizeof(coords[0]), NULL);
glEnableClientState(GL_NORMAL_ARRAY);
glBindBuffer(GL_ARRAY_BUFFER, bo_nrm);
glNormalPointer( GL_FLOAT, 0, NULL );
glEnableClientState(GL_TEXTURE_COORD_ARRAY);
glBindBuffer(GL_ARRAY_BUFFER, link_bo_tco);
glTexCoordPointer(2,GL_FLOAT,0,NULL);

glDrawArrays(GL_TRIANGLE_STRIP, 0, num_vtx );

glDisableClientState(GL_NORMAL_ARRAY);
glDisableClientState(GL_VERTEX_ARRAY);
glDisableClientState(GL_TEXTURE_COORD_ARRAY);
glBindBuffer(GL_ARRAY_BUFFER, 0);

if ( straight_link )
  {
    glPopMatrix();
```

```
    }

  /// SOLUTION -- Homework 4, above
}


void
World::frame_callback()
{
  // This routine called whenever window needs to be updated.

  frame_timer.phys_start();
  const double time_now = time_wall_fp();

  if ( !opt_pause || opt_single_frame || opt_single_time_step )
    {
      /// Advance simulation state.

      // Amount of time since the user saw the last frame.
      //
      const double wall_delta_t = time_now - last_frame_wall_time;

      // Compute amount by which to advance simulation state for this frame.
      //
      const double duration =
        opt_single_time_step ? opt_time_step_duration :
        opt_single_frame ? 1/30.0 :
        wall_delta_t;

      const double world_time_target = world_time + duration;
      const double wall_time_limit = time_now + 0.05;

      while ( world_time < world_time_target )
        {
          time_step_cpu(opt_time_step_duration);
          world_time += opt_time_step_duration;
          const double time_right_now = time_wall_fp();
          if ( time_right_now > wall_time_limit ) break;
        }

      // Reset these, just in case they were set.
      //
      opt_single_frame = opt_single_time_step = false;
    }

  last_frame_wall_time = time_now;

  if ( opt_ride && ball_eye )
    {
      pNorm b_eye_down(ball_eye->position,ball_down->position);
      pVect b_eye_up = -b_eye_down;
      pCoor eye_pos = ball_eye->position + 2.2 * ball_eye->radius * b_eye_up;
      pNorm b_eye_direction(eye_pos,ball_gaze->position);

      pNorm b_eye_left = cross(b_eye_direction,b_eye_up);
      pMatrix_Translate center_eye(-eye_pos);
      pMatrix rotate; rotate.set_identity();
      for ( int i=0; i<3; i++ ) rotate.rc(0,i) = b_eye_left.elt(i);
      for ( int i=0; i<3; i++ ) rotate.rc(1,i) = b_eye_up.elt(i);
      for ( int i=0; i<3; i++ ) rotate.rc(2,i) = -b_eye_direction.elt(i);
      modelview = rotate * center_eye;
      pMatrix reflect; reflect.set_identity(); reflect.rc(1,1) = -1;
      transform_mirror = modelview * reflect * invert(modelview);
    }
```

```
  render();
}

int
main(int argv, char **argc)
{
  pOpenGL_Helper popengl_helper(argv,argc);
  World world(popengl_helper);

  glDisable(GL_DEBUG_OUTPUT);
  glDebugMessageControl(GL_DONT_CARE,GL_DONT_CARE,
                        GL_DEBUG_SEVERITY_NOTIFICATION,0,NULL,false);

  popengl_helper.rate_set(30);
  popengl_helper.display_cb_set(world.frame_callback_w,&world);
}
```

```
/// LSU EE 4702-1 (Fall 2016), GPU Programming
//
 /// Homework 5 -- SOLUTION
 //
 //  See http://www.ece.lsu.edu/koppel/gpup/2016/hw05.pdf


// Specify version of OpenGL Shading Language.
//
#version 450 compatibility

layout ( location = 1 ) uniform vec4 color_bottom;
layout ( location = 2 ) uniform vec4 color_back;
layout ( location = 4 ) uniform vec4 color_side;
layout ( location = 5 ) uniform vec4 color_top;
layout ( location = 6 ) uniform ivec2 seg_sides;
layout ( location = 7 ) uniform float thickness;

// Use this variable to debug your code. Press 'y' to toggle tryout.x
// and 'Y' to toggle tryout.y (between true and false).
//
layout ( location = 3 ) uniform bvec2 tryout;



#ifdef _VERTEX_SHADER_

in vec2 gl_MultiTexCoord0;

// Interface block for vertex shader output / geometry shader input.
//
out Data_to_GS
{
  vec3 normal_e;
  vec4 vertex_e;
  vec2 gl_TexCoord[1];
  vec4 gl_Position;

  /// SOLUTION -- Problem 1
  //
  // Carry the value of theta to the geometry shader where it will be
  // used to determine whether triangles should be discarded.
  //
  float theta;
};


void
vs_main()
{
  /// Problem 1 solution here and other places.

  normal_e = normalize(gl_NormalMatrix * gl_Normal);
  gl_TexCoord[0] = gl_MultiTexCoord0;

  /// SOLUTION -- Problem 1
  //
  //  Move the theta value to the newly declared vertex shader output
  //  named theta and re-construct the object space vertex coordinate
  //  in vtx_o
  //
  theta = gl_Vertex.w;
  vec4 vtx_o = vec4(gl_Vertex.xyz,1);
  gl_Position = gl_ModelViewProjectionMatrix * vtx_o;
```

```
  vertex_e = gl_ModelViewMatrix * vtx_o;
}

#endif


#ifdef _GEOMETRY_SHADER_

in Data_to_GS
{
  vec3 normal_e;
  vec4 vertex_e;
  vec2 gl_TexCoord[1];
  vec4 gl_Position;

  /// SOLUTION -- Problem 1
  //
  // Carry the value of theta to the geometry shader where it will be
  // used to determine whether triangles should be discarded.
  //
  float theta;
} In[];

out Data_to_FS
{
  vec3 normal_e;
  vec4 vertex_e;
  vec2 gl_TexCoord[1];

  /// SOLUTION -- Problem 2
  //
  //  See in Data_to_FS for detailed comments describing these declarations.
  //
  flat bool is_side;
  flat vec3 normal_f;
  float height;
};

// Type of primitive at geometry shader input.
//
layout ( triangles ) in;

// Type of primitives emitted geometry shader output.
//
 /// SOLUTION -- Problem 2
//
// Specify just enough vertices to render the raised or surface
// triangle (3) and the sides (2*4=8)
//
layout ( triangle_strip, max_vertices = 11 ) out;


void
gs_main_1()
{

  /// SOLUTION -- Problem 1
  //
  //  Discard primitives in which the provoking vertex is at theta = 0,
  //  because for those primitives vertex In[0] belongs to a previous
  //  segment.
  //
  if ( In[2].theta == 0 ) return;

  for ( int i=0; i<3; i++ )
```

```
    {
      normal_e = In[i].normal_e;
      vertex_e = In[i].vertex_e;
      gl_Position = In[i].gl_Position;
      gl_TexCoord[0] = In[i].gl_TexCoord[0];
      EmitVertex();
    }
  EndPrimitive();
}

void
gs_main_2()
{
  /// Problem 2 solution goes here, and other places.

  // Don't a render triangle for first two vertices of a segment since
  // such a triangle would cross between segments. Note: this is an
  // alternative method of fixing the issue described in Problem 1.
  //
  int vtx_num = gl_PrimitiveIDIn + 2;
  int side_num = vtx_num % ( 2 * seg_sides.y );
  if ( side_num < 2 ) return;

  /// SOLUTION -- Problem 2

  // Compute offset, the vector to use to raise the triangle.
  //
  vec3 v01 = In[1].vertex_e.xyz - In[0].vertex_e.xyz;
  vec3 v02 = In[2].vertex_e.xyz - In[0].vertex_e.xyz;
  vec3 norm = normalize(cross( v01, v02 ));
  vec3 offset = thickness * norm;

  is_side = true;

  // Raise one out of every four triangles.
  //
  const bool raised = ( gl_PrimitiveIDIn & 0x3 ) == 0;

  /// Draw Side Triangles
  //
  // Side triangles are drawn as one triangle strip. Note that In[0]
  // is examined for the first (i=0) and last (i=3) iteration of the loop.
  //
  if ( raised )
  for ( int i=0; i<=3; i++ )
    {
      const int this_vtx_idx = i % 3;
      const int prev_vtx_idx = ( i + 2 ) % 3;

      // Use same texture coordinate for surface and raised vertices.
      //
      gl_TexCoord[0] = In[this_vtx_idx].gl_TexCoord[0];

      vec4 vertex_e_surface = In[this_vtx_idx].vertex_e;
      vec4 vertex_e_raised = vertex_e_surface + vec4(offset,0);
      vec4 vertex_e_prev = In[prev_vtx_idx].vertex_e;

      // Compute normal of side triangles and assign to flat normal output.
      //
      vec3 v_this_prev = vertex_e_prev.xyz - vertex_e_surface.xyz;
      normal_f = cross(norm,v_this_prev);

      // Emit the side triangle vertex shared with the raised triangle.
      //
      height = 1;
```

```
      vertex_e = vertex_e_raised;
      gl_Position = gl_ProjectionMatrix * vertex_e_raised;
      EmitVertex();

      // Emit the side triangle vertex shared with the surface triangle.
      //
      height = 0;
      vertex_e = vertex_e_surface;
      gl_Position = In[this_vtx_idx].gl_Position;
      EmitVertex();
    }
  EndPrimitive();

  is_side = false;
  height = float(raised);

  // Emit either a raised or surface triangle.
  //
  for ( int i=0; i<3; i++ )
    {
      normal_e = normalize(In[i].normal_e);
      vertex_e = In[i].vertex_e + ( raised ? vec4(offset,0) : vec4(0) );
      gl_Position = gl_ProjectionMatrix * vertex_e;
      gl_TexCoord[0] = In[i].gl_TexCoord[0];
      EmitVertex();
    }
  EndPrimitive();
}



#endif


#ifdef _FRAGMENT_SHADER_

uniform sampler2D tex_unit_0;

in Data_to_FS
{
  vec3 normal_e;
  vec4 vertex_e;
  vec2 gl_TexCoord[1];

  /// SOLUTION -- Problem 2

  // If true, fragment is part of a side triangle.
  //
  flat bool is_side;

  // Surface normal to be used for side triangles. Since it is declared
  // flat the value for a fragment is taken from the provoking vertex
  // (3rd vertex for triangles) rather than an interpolation of the values
  // at all three triangle vertices.
  //
  flat vec3 normal_f;

  // Distance from surface. Zero means at the surface, one means at
  // the raised triangle.  We want it's value to be interpolated.
  //
  float height;
};

vec4 generic_lighting(vec4 vertex_e, vec4 color, vec3 normal_e);
```

```
void
fs_main_1()
{
  /// Do not modify this module. Use fs_main_2 for Problem 2

  // Perform lighting, fetch and blend texture, then emit fragment.
  //

  // Get filtered texel, unless the fragment belongs to an edge primitive.
  //
  vec4 texel = texture(tex_unit_0,gl_TexCoord[0]);

  vec4 color2 = gl_FrontFacing ? color_bottom : color_back;

  // Multiply filtered texel color with lighted color of fragment.
  //
  gl_FragColor = texel * generic_lighting(vertex_e, color2, normal_e);

  // Copy fragment depth unmodified.
  //
  gl_FragDepth = gl_FragCoord.z;
}

void
fs_main_2()
{
  /// Problem 2 solution goes here and other places.

  // Perform lighting, fetch and blend texture, then emit fragment.
  //

  // Get filtered texel, unless the fragment belongs to an edge primitive.
  //
  vec4 texel = texture(tex_unit_0,gl_TexCoord[0]);

  /// SOLUTION -- Problem 2
  //
  float marg = 1.0/3;
  float omm = 1 - marg;  // One Minus Margin

  // Blend appropriate set of colors together.  Note that for
  // non-side triangles height = 0.
  //
  vec4 color2 = !gl_FrontFacing ? color_back :
    height > omm ?  (   ( height - omm ) * color_top
                      + ( 1.0 - height ) * color_side   ) / marg :
    height < marg ? (             height * color_side
                    + ( marg - height ) * color_bottom ) / marg :
    color_side;

  // Compute a texture blend amount and use it to blend the texel color
  // with the color white.
  //
  float tblend = 2 * abs(height-0.5);  // Note: Value is between 0 and 1.
  vec4 white = vec4(1,1,1,1);
  vec4 texel2 = (1-tblend) * white + tblend * texel;

  // For side triangles use the non-interpolated normal.
  //
  vec3 norm = is_side ? normal_f : normal_e;

  // Multiply filtered texel color with lighted color of fragment.
  //
  gl_FragColor = texel2 * generic_lighting(vertex_e, color2, norm);
```

```
  // Copy fragment depth unmodified.
  //
  gl_FragDepth = gl_FragCoord.z;
}


///
/// Routine used by Either Vertex or Fragment Shader
///


vec4
generic_lighting(vec4 vertex_e, vec4 color, vec3 normal_e)
{
  /// It should not be necessary to modify this module for Homework 5.

  // Return lighted color of vertex_e.
  //
  vec4 light_pos = gl_LightSource[0].position;
  vec3 v_vtx_light = light_pos.xyz - vertex_e.xyz;
  float dist = length(v_vtx_light);

  float d_n_vl = dot(normalize(normal_e), v_vtx_light) / dist;
  float phase_light = max(0,gl_FrontFacing ? d_n_vl : -d_n_vl );

  vec3 ambient_light = gl_LightSource[0].ambient.rgb;
  vec3 diffuse_light = gl_LightSource[0].diffuse.rgb;
  float distsq = dist * dist;
  float atten_inv =
    gl_LightSource[0].constantAttenuation +
    gl_LightSource[0].linearAttenuation * dist +
    gl_LightSource[0].quadraticAttenuation * distsq;
  vec4 lighted_color;
  lighted_color.rgb =
    color.rgb * gl_LightModel.ambient.rgb
    + 0.3 * color.rgb
    * ( ambient_light + phase_light * diffuse_light ) / atten_inv;
  lighted_color.a = color.a;
  return lighted_color;
}


#endif
```

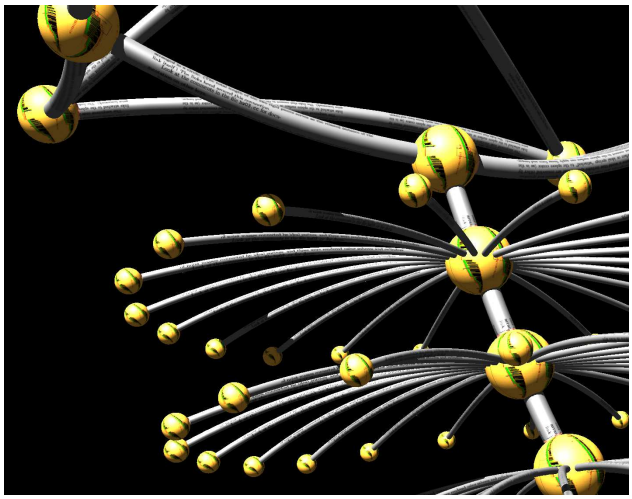
# LSU EE 4702-1 Homework 6 Solution Due: 16 November 2016

The solution has been checked into the repo. For a syntax highlighted version of the shader code see `http://www.ece.lsu.edu/koppel/gpup/2016/hw06-shdr-links-sol.cc.html` and for the CPU code see `http://www.ece.lsu.edu/koppel/gpup/2016/hw06-sol.cc.html`.

**Problem 0:** If necessary, follow the instructions on the `http://www.ece.lsu.edu/koppel/gpup/proc.html` page for account setup and programming homework work flow. For this assignment only edit files `hw06-shdr-links.cc` and `hw06.cc`. Compile and run the homework code unmodified. It should initially show Scene 3, a spikey ball. The vaguely umbrella-shaped object hanging in space from prior assignments but with bristles or something on the vertical tail of balls is in Scene 2 and Scene 1 shows a tree (the kind that grows outdoors).

Pressing `v` will cycle through three different sets of shaders. The shader set that is being used is shown in the penultimate line of green text. Shader set PLAIN is a conventional set of shaders, and is there for comparison purposes. Shader set SET 1 is comprised of vertex shader `vs_main_1`, geometry shader `gs_main_1`, and fragment shader `fs_main`, all in file `hw06-shdr-links.cc`, and is used for Problem 1. Shader set SET 2 is comprised of vertex shader `vs_main_2`, geometry shader `gs_main_2`, and fragment shader `fs_main`, also in file `hw06-shdr-links.cc`, and is used for Problem 2.

The plain set of shaders should be slow, while the other sets show the links in a crude manner. Options that affect performance are: turning shadows on and off `o` and turning mirroring on and off `r`. (Pressing `O` will show shadow volumes.) Note that the links still don't cast shadows.



Press digits `1` through `2` to initialize different scenes, the program starts with scene 2. Scene 1 shows a balls connected in a rectangular spiral. *Promptly report any problems.*

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.

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).

When using the arrow and other keys to move the eye, light, or ball using `Shift` will move by a 5× greater amount and using `Ctrl` will move by one $\frac{1}{5}$ the amount than the motion without either modifier.

Look at the comments in the file `hw06.cc` for documentation on other keys.

A goal of Homework 3 was to reduce the amount of communication between the CPU and GPU by using buffer objects, especially for almost straight links. In this assignment two methods will be used to improve performance: computing the surface of the curved link in the GPU, and using an instanced draw to reduce the number of rendering passes.

See the OpenGL Shading language documentation linked to the references page, `http://www.ece.lsu.edu/koppel/gpup/ref.html`, for syntax and library functions that can be used in shader code.

**Problem 1:** The code in `vs_main_1` and `gs_main_1` are used to crudely render a link with an instanced draw. Examine the code in `vs_main_1` and `render_link_2_render`. Built-in variable `gl_InstanceID` indicates which link is being rendered, its value can range from 0 to one minus the number of links (or instances). The number of instances is determined by the last argument to `glDrawArraysInstanced`. The built-in vertex shader variable `gl_VertexID` indicates the vertex number being rendered, its range of values is determined by the third argument to `glDrawArraysInstanced`.

Note that in the rendering pass set up in `render_link_2_render` *no vertex coordinates nor any other vertex shader inputs* are sent to the vertex shader. The shader must rely on `gl_InstanceID` and `gl_VertexID` to find its inputs.

Notice that `vs_main_1` reads link endpoint coordinates from buffer objects `pos1` and `pos2`, and using a scaled version of `gl_VertexID`, computes a point partway between the endpoints, `vertex_o`. That coordinate is transformed and sent to the geometry shader for constructing triangles.

Since it renders a straight link there is no reason to use more than two vertices per link. (The curved link will be rendered using `vs_main_2` an `gs_main_2`.)

(*a*) Modify `vs_main_1`, `gs_main_1`, and `render_link_2_render` so that the complete straight link is rendered using only two vertices per instance without changing the primitive at the input to the geometry shader (a line strip).

- Split work between the vertex shader and the geometry shader to void duplication of effort.
- Make sure that changes to `render_link_2_render` don't affect `vs_main_2`.

See the code checked into the repo for details. In the original code the vertex shader computed a point partway between `pos1` and `pos2` based on $t$, which varied from 0 to 1. Since there are only two vertices now $t$ would be either 0 or 1 and there's no point in computing it. Instead, of the vertex ID is 0 `pos1` is used, otherwise `pos2` is used. The vertex shader appears below. The geometry shader is modified to adjust the width of the links. See the code in the repo or at the link above for details.

```
void vs_main_1() {
  iid = gl_InstanceID;
  vec3 vertex_1 = pos1[iid].xyz;
  vec3 vertex_2 = pos2[iid].xyz;
  vec4 vertex_o = vec4( gl_VertexID == 0 ? vertex_1 : vertex_2, 1 );
  gl_Position = gl_ModelViewProjectionMatrix * vertex_o;
  vertex_e = gl_ModelViewMatrix * vertex_o;
}
```

**Problem 2:** The code in `vs_main_2` and `gs_main_2` which also crudely render a link, are placeholders for code rendering the curved link.

(*a*) Modify `vs_main_2` and `gs_main_2` so that they render the curved link based on the data copied into buffer objects by routines `render_link_2_gather` and `render_link_2_render`. Base your code on the code in `render_link_1`. Be sure that any changes made for the prior problem don't interfere with this one.

- Split work between the vertex shader and the geometry shader to void duplication of effort.

- Write your routine for a link with a maximum of 20 sides.

See the code for details. In the solution the vertex shader computes a point on the curve (the cylinder axis) and vectors orthogonal to the cylinder axis. These are sent to the geometry shader which draws the cylinder.

**Problem 3:** Estimate the amount of data sent from CPU to GPU for each link for the plain and set 1 and set 2 shaders. (The set 1 and set 2 shaders should send the same amount of data.) Do this by examining the code in `render_link_2_gather` and `render_link_2_render`.

The plain links are rendered in `render_link_1`. As can be seen by the `glEnableClientState` calls, each vertex in the link consists of a coordinate (`GL_VERTEX_ARRAY`, $16\,\mathrm{B}$), a normal ($12\,\mathrm{B}$), and a texture coordinate ($8\,\mathrm{B}$). The number of vertices is $2gs$, where $g$ is the number of segments and $s$ is the number of sides (defined in the code). So the total amount of data per link is $(16\,\mathrm{B} + 12\,\mathrm{B} + 8\,\mathrm{B})sg = 36sg\,\mathrm{B}$. For the default values of $g = 15$ and $s = 20$ that's $10800\,\mathrm{B}$ per link. (The color would add just 12 more bytes.)

The code in `render_link_2_render` sends over 7 buffer objects (for `pos1`, `pos2`, `v1`, `v2`, `b1_ydir`, `b2_ydir`, and `misc`). Each buffer object holds an array of `vec4`s. (A `vec4` is $16\,\mathrm{B}$.) There is one array element per link, so the total amount of buffer object data per link is $7 \times 16 = 112\,\mathrm{B}$. The call to `glDrawArraysInstanced`, which initiates the rendering of many links, does not send any per-vertex data because no client arrays were set up. It would need to send over the type of primitive (a line strip), the number of vertices per instance, and the number of instances. That might take 12 bytes for the entire rendering pass. Finally, in `render_link_2_render` we see that several uniforms are sent over, such as the colors. The total uniform data is 12 floats and 3 integers, which total to $(12 + 3) \times 4 = 60\,\mathrm{B}$. So if $n$ is the number of links to be rendered, the amount of data per link would be $112\,\mathrm{B} + \frac{12\,\mathrm{B}+60\,\mathrm{B}}{n}$.

Obviously $112\,\mathrm{B}$ per link (for large $n$) is much lower than $10800\,\mathrm{B}$ per link required by the plain shader. Much less data is needed because the shaders compute points on the link given only data about its endpoints.

```
/// LSU EE 4702-1 (Fall 2016), GPU Programming
//
 /// Homework 6 -- SOLUTION
 //
 //  See http://www.ece.lsu.edu/koppel/gpup/2016/hw06.pdf


// Specify version of OpenGL Shading Language.
//
#version 450 compatibility

layout ( location = 1 ) uniform vec4 color_front;
layout ( location = 2 ) uniform vec4 color_back;

// A bit vector indicating which lights are on.
layout ( location = 6 ) uniform int lighting_options;

// Use this variable to debug your code. Press 'y' to toggle tryout.x
// and 'Y' to toggle debug_bool.y (between true and false).
//
layout ( location = 3 ) uniform bvec2 tryout;


layout ( location = 4 ) uniform vec2 sides_rad;
layout ( location = 5 ) uniform float delta_tee;

layout ( binding = 1 ) buffer sr { vec4 pos1[]; };
layout ( binding = 2 ) buffer sr2 { vec4 pos2[]; };
layout ( binding = 3 ) buffer spr { vec4 v1[]; };
layout ( binding = 4 ) buffer sc { vec4 v2[]; };
layout ( binding = 5 ) buffer sc5 { vec4 b1_ydir[]; };
layout ( binding = 6 ) buffer sc6 { vec4 b2_ydir[]; };
layout ( binding = 7 ) buffer sc7 { vec4 tex_rad[]; };



#ifdef _VERTEX_SHADER_

// Interface block for vertex shader output / geometry shader input.
//
out Data_to_GS
{
  int iid; // Instance ID
  vec4 gl_Position;
  vec4 vertex_e;

  /// SOLUTION -- Problem 2
  //
  float t;
  vec4 ctr;
  vec3 norm, binorm;

  // Any changes here must also be made to the fragment shader input.
};

void
vs_main_1()
{
  iid = gl_InstanceID;

  vec3 vertex_1 = pos1[iid].xyz;
  vec3 vertex_2 = pos2[iid].xyz;

  /// SOLUTION -- Problem 1
```

```
  //
  //  Use the vertex number to select either the first or second vertex.
  //
  vec4 vertex_o = vec4( gl_VertexID == 0 ? vertex_1 : vertex_2, 1 );

  gl_Position = gl_ModelViewProjectionMatrix * vertex_o;
  vertex_e = gl_ModelViewMatrix * vertex_o;
}

void
vs_main_2()
{
  /// SOLUTION -- Problem 2

  // In the vertex shader compute the coordinate of the axis of the
  // cylinder (ctr) and the vectors orthogonal to the axis. The
  // geometry shader will use a pair of these to draw the cylinder
  // surface.

  iid = gl_InstanceID;

  // Compute the values of t, t^2 and t^3 needed to compute point on
  // curve (core of link).
  //
  const int i = gl_VertexID;
  t = i * delta_tee;
  const float t2 = t * t;
  const float t3 = t2 * t;

  // Cubic Hermite interpolation.
  // Compute point along core of link in object space.
  //
  ctr.xyz =
    ( 2*t3 - 3*t2 + 1 ) * pos1[iid].xyz + (-2*t3 + 3*t2 ) * pos2[iid].xyz
    + ( t3 - 2*t2 + t ) * v1[iid].xyz - ( t3 - t2 ) * v2[iid].xyz;
  ctr.w = 1;

  // Compute direction of link at this point.
  //
  vec3 tan_raw =
    ( 6*t2 - 6*t ) * pos1[iid].xyz + ( -6*t2 + 6*t ) * pos2[iid].xyz
    + ( 3*t2 - 4*t + 1 ) * v1[iid].xyz - ( 3*t2 - 2*t ) * v2[iid].xyz;
  vec3 tan = normalize(tan_raw);

  // Compute local x and y axes for drawing a cylinder.
  //
  vec3 ydir = mix(b1_ydir[iid].xyz, b2_ydir[iid].xyz, t);
  vec3 norm_raw = cross(tan,ydir);
  norm = normalize(norm_raw);
  binorm = cross(tan,norm);
}

#endif


#ifdef _GEOMETRY_SHADER_

in Data_to_GS
{
  int iid; // Instance ID
  vec4 gl_Position;
  vec4 vertex_e;

  /// SOLUTION -- Problem 2
  float t;
```

```
  vec4 ctr;
  vec3 norm, binorm;
} In[];

out Data_to_FS
{
  vec3 normal_e;
  vec4 vertex_e;
  vec2 gl_TexCoord[1];
};

// Type of primitive at geometry shader input.
//
layout ( lines ) in;

// Type of primitives emitted geometry shader output.
//

 /// SOLUTION -- Problem 2
//   Increase max_vertices to 42.
layout ( triangle_strip, max_vertices = 42 ) out;


void
gs_main_1()
{
  vec3 v01 = In[1].vertex_e.xyz - In[0].vertex_e.xyz;
  vec3 ax = normalize(v01.x == 0 ? vec3(0,v01.z,-v01.y) : vec3(v01.y,-v01.x,0));

  /// SOLUTION -- Problem 1
  //
  //  Since we are only rendering using two vertices v01 will be longer,
  //  and so for the links to be the same width as when we used more
  //  segments multiply by delta_tee.
  //
  float l = length(v01) * delta_tee;

  for ( int i=0; i<2; i++ )
    {
      vertex_e = In[i].vertex_e;
      gl_Position = In[i].gl_Position;
      EmitVertex();
      vertex_e = In[i].vertex_e + vec4( ax * l * 0.1, 1 );
      gl_Position = gl_ProjectionMatrix * vertex_e;
      EmitVertex();
    }
  EndPrimitive();
}

void
gs_main_2()
{
  /// SOLUTION -- Problem 2

  // Emit primitives for a cylinder. One end of the cylinder is
  // defined by data in In[0], the other end by data in In[1].

  const float M_PI = 3.1415926535;
  const vec2 tex_scale = tex_rad[In[1].iid].xy;
  const float rad = tex_rad[In[1].iid].z;
  const float sides = sides_rad.x;

  for ( int j=0; j<=sides; j++ )
    {
      const float theta = j * ( 2 * M_PI / sides );
```

```
      vec3 vect0 = cos(theta) * In[0].norm + sin(theta) * In[0].binorm;
      vec3 vect1 = cos(theta) * In[1].norm + sin(theta) * In[1].binorm;
      vec4 pt0 = In[0].ctr + vec4(rad * vect0,0);
      vec4 pt1 = In[1].ctr + vec4(rad * vect1,0);

      normal_e = gl_NormalMatrix * vect1;
      vertex_e = gl_ModelViewMatrix * pt1;
      gl_TexCoord[0] =
        vec2( In[1].t * tex_scale.x, 0.5 + theta * tex_scale.y );
      gl_Position = gl_ModelViewProjectionMatrix * pt1;
      EmitVertex();

      normal_e = gl_NormalMatrix * vect0;
      vertex_e = gl_ModelViewMatrix * pt0;
      gl_TexCoord[0] =
        vec2( In[0].t * tex_scale.x, 0.5 + theta * tex_scale.y );
      gl_Position = gl_ModelViewProjectionMatrix * pt0;
      EmitVertex();
    }
  EndPrimitive();
}


#endif


#ifdef _FRAGMENT_SHADER_

uniform sampler2D tex_unit_0;

in Data_to_FS
{
  vec3 normal_e;
  vec4 vertex_e;
  vec2 gl_TexCoord[1];
};

vec4 generic_lighting(vec4 vertex_e, vec4 color, vec3 normal_e);


void
fs_main()
{
  // Perform lighting, fetch and blend texture, then emit fragment.
  //

  // Get filtered texel, unless the fragment belongs to an edge primitive.
  //
  vec4 texel = texture(tex_unit_0,gl_TexCoord[0]);

  vec4 color2 = gl_FrontFacing ? color_front : color_back;

  // Multiply filtered texel color with lighted color of fragment.
  //
  gl_FragColor = texel * generic_lighting(vertex_e, color2, normal_e);

  // Copy fragment depth unmodified.
  //
  gl_FragDepth = gl_FragCoord.z;
}


///
```

```
/// Routine used by Either Vertex or Fragment Shader
///


vec4
generic_lighting(vec4 vertex_e, vec4 color, vec3 normal_er)
{
  // Return lighted color of vertex_e.
  //

  vec3 nspc_color = color.rgb * gl_LightModel.ambient.rgb;
  vec3 spec_color = vec3(0);
  vec3 normal_e = normalize(normal_er);

  for ( int i=0; i<2; i++ )
    {
      if ( ( lighting_options & ( 1 << i ) ) == 0 ) continue;
      vec4 light_pos = gl_LightSource[i].position;
      vec3 v_vtx_light = light_pos.xyz - vertex_e.xyz;
      float dist = length(v_vtx_light);
      float dist_vl_inv = 1.0 / dist;
      vec3 v_vtx_l_n = v_vtx_light * dist_vl_inv;

      float d_n_vl = dot(normal_e, v_vtx_l_n);
      float phase_light = max(0,gl_FrontFacing ? d_n_vl : -d_n_vl );

      vec3 ambient_light = gl_LightSource[i].ambient.rgb;
      vec3 diffuse_light = gl_LightSource[i].diffuse.rgb;
      float distsq = dist * dist;
      float atten_inv =
        gl_LightSource[i].constantAttenuation +
        gl_LightSource[i].linearAttenuation * dist +
        gl_LightSource[i].quadraticAttenuation * distsq;
      vec3 lighted_color =
        color.rgb
        * ( ambient_light + phase_light * diffuse_light ) / atten_inv;
      nspc_color += lighted_color;

      vec3 h = normalize( v_vtx_l_n - normalize(vertex_e.xyz) );

      spec_color +=
        pow(max(0.0,dot(normal_e,h)),16)
        * color.rgb
        * gl_LightSource[i].specular.rgb / atten_inv;
    }

  return vec4(nspc_color+spec_color,1);
}


#endif
```

```
/// LSU EE 4702-1 (Fall 2016), GPU Programming
//
 /// Homework 6 -- SOLUTION
 //
 //  See http://www.ece.lsu.edu/koppel/gpup/2016/hw06_sol.pdf

 //  For Homework 6 edit this file and hw06-shdr-links.cc.

/// Purpose
//
//   Demonstrate simulation of point masses connected by springs.


/// What Code Does

// Simulates balls connected by springs over a platform. Balls and
// springs can be initialized in different arrangements (called
// scenes). Currently scene 1 is a simple string of beads, and scenes
// 2, 3, and 4 are trusses. The platform consists of tiles, some are
// purple-tinted mirrors (showing a reflection of the ball), the
// others show the course syllabus.



///  Keyboard Commands
 //
 /// Object (Eye, Light, Ball) Location or Push
 //   Arrows, Page Up, Page Down
 //        Move object or push ball, depending on mode.
 //        With shift key pressed, motion is 5x faster.
 //        With control key pressed, motion is 5x slower.
 //   'e': Move eye.
 //   'l': Move light.
 //   'b': Move head (first) ball. (Change position but not velocity.)
 //   'B': Push head ball. (Add velocity.)
 //
 /// Eye Direction
 //   Home, End, Delete, Insert
 //   Turn the eye direction.
 //   Home should rotate eye direction up, End should rotate eye
 //   down, Delete should rotate eye left, Insert should rotate eye
 //   right.  The eye direction vector is displayed in the upper left.

 /// Simulation Options
 //  (Also see variables below.)
 //
 //  'w'    Twirl balls around axis formed by head and tail.
 //  '1'    Set up scene 1.
 //  '2'    Set up scene 2.
 //  '3'    Set up scene 3.
 //  'p'    Pause simulation. (Press again to resume.)
 //  ' '    (Space bar.) Advance simulation by 1/30 second.
 //  'S- '  (Shift-space bar.) Advance simulation by one time step.
 //  'h'    Freeze position of first (head) ball. (Press again to release.)
 //  't'    Freeze position of last (tail) ball. (Press again to release.)
 //  's'    Stop balls.
 //  'g'    Turn gravity on and off.
 //  'y'    Toggle value of opt_tryout1. Intended for experiments and debugging.
 //  'Y'    Toggle value of opt_tryout2. Intended for experiments and debugging.
 //  'F11'  Change size of green text (in upper left).
 //  'F12'  Write screenshot to file.
```

```
/// Variables
//   Selected program variables can be modified using the keyboard.
//   Use "Tab" to cycle through the variable to be modified, the
//   name of the variable is displayed next to "VAR" on the bottom
//   line of green text.

//  'Tab' Cycle to next variable.
//  '`'   Cycle to previous variable.
//  '+'   Increase variable value.
//  '-'   Decrease variable value.
//
//  VAR Spring Constant - Set spring constant.
//  VAR Time Step Duration - Set physics time step.
//  VAR Air Resistance - Set air resistance.
//  VAR Light Intensity - The light intensity.
//  VAR Gravity - Gravitational acceleration. (Turn on/off using 'g'.)


#define GL_GLEXT_PROTOTYPES
#define GLX_GLXEXT_PROTOTYPES

#include <GL/gl.h>
#include <GL/glext.h>
#include <GL/glx.h>
#include <GL/glxext.h>
#include <GL/glu.h>
#include <GL/freeglut.h>

#include <gp/util.h>
#include <gp/glextfuncs.h>
#include <gp/coord.h>
#include <gp/shader.h>
#include <gp/pstring.h>
#include <gp/misc.h>
#include <gp/gl-buffer.h>
#include <gp/texture-util.h>
#include <gp/colors.h>

#include <gp/util-containers.h>
#include <gp/coord-containers.h>
#include "shapes.h"


///
/// Main Data Structures
///
//
// class World: All data about scene.


class World;


// Object Holding Ball State
//
class Ball {
public:
  Ball():velocity(pVect(0,0,0)),omega(pVect(0,0,0)),
         density(1.00746),
         fdt_to_do(0),
         locked(false),
         color(color_lsu_spirit_gold),contact(false)
         {
           orientation_set(pVect(0,1,0),0);
```

```
        }
  pCoor position;
  pVect velocity;
  pQuat orientation;
  pMatrix3x3 omatrix;
  pVect omega;                  // Spin rate and axis.

  int idx; // Position in balls_pos_rad;

  float mass;
  float mass_min; // Mass below which simulation is unstable.
  float radius;
  float radius_sq, radius_inv;  // Pre-computed based on radius.
  float density;
  float mass_inv;
  float fdt_to_do; // Radius / moment of inertia.

  bool locked;

  pVect force;
  pVect torque;
  pColor color;
  bool contact;                 // When true, ball rendered in gray.
  float spring_constant_sum;    // Used to compute minimum mass.

  void orientation_set(pNorm dir, float angle)
    { orientation_set(pQuat(dir,angle)); }
  void orientation_set(pQuat ori)
    {
      orientation = ori;
      omatrix = pMatrix3x3_Rotation(orientation);
    }
  void constants_update();
  pVect point_rot_vel(pNorm tact_dir);
  pVect point_rot_vel(pVect tact_dir);
  void apply_tan_force(pNorm tact_dir, pNorm force_dir, double force_mag);
  void apply_tan_force(pNorm tact_dir, pVect force);
  void push(pVect amt);
  void translate(pVect amt);
  void stop();
  void freeze();
};


void
Ball::constants_update()
{
  assert( radius );
  radius_sq = radius * radius;
  mass = 4.0 / 3 * M_PI * radius_sq * radius * density;
  radius_inv = 1 / radius;
  mass_inv = 1 / mass;
  // FYI, notice simplifications:
  //   const float mo_inertia = 0.4 * mass * radius_sq;
  //   fdt_to_do = radius / mo_inertia;
  fdt_to_do = radius_inv * 2.5 * mass_inv;
}

pVect
Ball::point_rot_vel(pNorm direction)
{
  /// Return velocity of point on surface of ball.
```

```
  //
  return radius * cross( omega, direction );
}

pVect
Ball::point_rot_vel(pVect rel_pos)
{
  /// Return velocity of point relative to center.
  //
  return cross( omega, rel_pos );
}

void
Ball::apply_tan_force(pNorm tact_dir, pVect force)
{
  torque += cross(tact_dir, force);
}

void
Ball::apply_tan_force(pNorm tact_dir, pNorm force_dir, double force_mag)
{
  apply_tan_force(tact_dir, force_mag * force_dir);
}

class Link {
public:
  Link(Ball *b1, Ball *b2):ball1(b1),ball2(b2),
     snapped(false),
     natural_color(color_lsu_spirit_purple),color(color_lsu_spirit_purple)
  { init(); }
  Link(Ball *b1, Ball *b2, pColor colorp):ball1(b1),ball2(b2),
     snapped(false),
     natural_color(colorp),color(colorp)
  { init(); }
  Link(Link *link, pVect cb1p, pVect cb2p, float drp):
    ball1(link->ball1), ball2(link->ball2), cb1(cb1p), cb2(cb2p),
    is_surface_connection(true), is_renderable(false),
    is_simulatable(true), distance_relaxed(drp),
    snapped(false){ link->is_simulatable = false; }
  void init()
    {
      assert( ball1->radius > 0 );
      assert( ball2->radius > 0 );
      is_simulatable = true;
      is_renderable = true;
      pNorm n12(ball1->position,ball2->position);
      const float rad_sum = ball1->radius + ball2->radius;
      is_surface_connection = n12.magnitude >= rad_sum;
      if ( !is_surface_connection )
        {
          distance_relaxed = n12.magnitude;
          cb1 = cb2 = pVect(0,0,0);
          return;
        }
      distance_relaxed = n12.magnitude - rad_sum;
      cb1 = ball1->radius * mult_MTV( ball1->omatrix, n12 );
      cb2 = ball2->radius * mult_MTV( ball2->omatrix, -n12 );
    }
  int serial;
  Ball* const ball1;
  Ball* const ball2;
  pVect cb1, cb2, b1_dir, b2_dir;
  bool is_surface_connection;
```

```
  bool is_renderable, is_simulatable;
  float distance_relaxed;
  bool snapped;
  pColor natural_color;
  pColor color;

  pVect spring_force_12;
  pVect torque1,torque2;

};

// Declare containers for Balls and Links. The containers are derived
// from std::vector and use overloaded operator += for push_back. (See
// include/util-containers.h.)
//
typedef pVectorI<Link> Links;
typedef pVectorI<Ball> Balls;


enum Render_Option { RO_Normally, RO_Simple, RO_Shadow_Volumes };
enum Shader_Option { SO_Phong, SO_Set_1, SO_Set_2, SO_ENUM_SIZE };

class World {
public:
  World(pOpenGL_Helper &fb, int argc, char **argv):ogl_helper(fb)
  {init(argc,argv);}
  void init(int argc, char **argv);
  void init_graphics();
  static void frame_callback_w(void *moi){((World*)moi)->frame_callback();}
  void frame_callback();
  void render();
  void render_objects(Render_Option render_option);
  void objects_erase();
  void cb_keyboard();
  void modelview_update();

  pOpenGL_Helper& ogl_helper;
  pVariable_Control variable_control;
  pFrame_Timer frame_timer;
  double world_time;
  double last_frame_wall_time;
  float opt_time_step_duration;
  int time_step_count;
  float opt_gravity_accel;      // Value chosen by user.
  pVect gravity_accel;          // Set to zero when opt_gravity is false;
  bool opt_gravity;
  bool opt_head_lock, opt_tail_lock;
  bool opt_tryout1, opt_tryout2;  // For ad-hoc experiments.

  // Tiled platform for ball.
  //
  float platform_xmin, platform_xmax, platform_zmin, platform_zmax;
  float platform_pi_xwidth_inv;
  pBuffer_Object<pVect> platform_tile_coords;
  pBuffer_Object<float> platform_tex_coords;
  pVect platform_normal;
  GLuint texid_hw;
  GLuint texid_syl;
  GLuint texid_emacs;
  void platform_update();
  bool platform_collision_possible(pCoor pos);

  pCoor light_location;
```

```
float opt_light_intensity;
enum { MI_Eye, MI_Light, MI_Ball, MI_Ball_V, MI_COUNT } opt_move_item;
bool opt_pause;
bool opt_single_frame;      // Simulate for one frame.
bool opt_single_time_step;  // Simulate for one time step.

pCoor eye_location;
pVect eye_direction;
pMatrix modelview;
pMatrix transform_mirror;

pVect adj_vector;
double adj_t_prev;
double adj_t_stop;
double adj_duration;

int opt_shader;
bool opt_mirror;
bool opt_shadows;
bool opt_shadow_volumes;      // Make shadow volumes visible.
pShader *sp_fixed;          // Fixed functionality.
pShader *sp_phong;          // Basic stuff.
pShader *sp_instances_sphere;
pShader *sp_instances_sv;

GLuint balls_pos_rad_bo;
GLuint balls_color_bo;
GLuint links_indices_bo;

pColor *balls_color;
pCoor *balls_pos_rad;
size_t balls_size;
size_t links_size;
int last_setup; // Last scene set up.
bool link_change;

void ball_setup_1();
void ball_setup_2();
void ball_setup_3();
void ball_setup_4();
void ball_setup_5();
void setup_at_end();
void time_step_cpu(double);
void balls_stop();
void balls_freeze();
void balls_translate(pVect amt, int idx);
void balls_translate(pVect amt);
void balls_push(pVect amt, int idx);
void balls_push(pVect amt);
void balls_twirl();
void lock_update();

Ball *make_marker(pCoor pos, pColor col);

float opt_spring_constant;
float opt_air_resistance;
float distance_relaxed;
int chain_length;
Balls balls;
Ball *head_ball, *tail_ball;
Links links;
Sphere sphere;
```

```
  Cylinder cyl;

  Links link_new(Ball *ball1, Ball *ball2, float stiffness = 0.2);

  // Homework 6

  void render_link_1(Link *link);
  void render_link_2_start();
  void render_link_2_gather(Link *link);
  void render_link_2_render();

  pShader *sp_set_1;
  pShader *sp_set_2;
  GLuint link_bo_vtx, link_bo_nrm, link_bo_tco;
  double world_time_link_update;

  int opt_sides, opt_segments;

  pVectorI<pCoors> lis_pvects;

  pCoors lis_pos1, lis_pos2;
  pCoors lis_v1, lis_v2, lis_b1_ydir, lis_b2_ydir, lis_tex_rad;
  GLuint *lis_bos;
};


void
World::init_graphics()
{
  ///
  /// Graphical Model Initialization
  ///

  balls_pos_rad = NULL;
  balls_pos_rad_bo = 0;
  balls_color = NULL;
  balls_color_bo = 0;
  balls_size = 0;
  links_size = 0;
  link_change = true;

  opt_head_lock = false;
  opt_tail_lock = false;
  opt_tryout1 = opt_tryout2 = false;

  eye_location = pCoor(24.2,11.6,-38.7);
  eye_direction = pVect(-0.42,-0.09,0.9);

  platform_xmin = -40; platform_xmax = 40;
  platform_zmin = -40; platform_zmax = 40;
  texid_syl = pBuild_Texture_File("gpup.png",false,255);
  texid_emacs = pBuild_Texture_File("mult.png", false,-1);
  texid_hw = pBuild_Texture_File("hw05-assign.png", false,255);

  opt_light_intensity = 100.2;
  light_location = pCoor(platform_xmax,platform_xmax,platform_zmin);

  variable_control.insert(opt_light_intensity,"Light Intensity");

  opt_move_item = MI_Eye;
  opt_pause = false;
  opt_single_time_step = false;
  opt_single_frame = false;
```

```
  sphere.init(35);

  platform_update();
  modelview_update();

  adj_t_stop = 0;
  adj_duration = 0.25;

  link_bo_vtx = 0;
  opt_sides = 20;
  opt_segments = 15;
  variable_control.insert(opt_sides,"Link Sides");
  variable_control.insert(opt_segments,"Link Segments");
}


void
World::platform_update()
{
  const float tile_count = 19;
  const float ep = 1.00001;
  const float delta_x = ( platform_xmax - platform_xmin ) / tile_count * ep;
  const float zdelta = ( platform_zmax - platform_zmin ) / tile_count * ep;

  const float trmin = 0.05;
  const float trmax = 0.7;
  const float tsmin = 0;
  const float tsmax = 0.4;

  platform_normal = pVect(0,1,0);

  PStack<pVect> p_tile_coords;
  PStack<pVect> p1_tile_coords;
  PStack<float> p_tex_coords;
  bool even = true;

  for ( int i = 0; i < tile_count; i++ )
    {
      const float x0 = platform_xmin + i * delta_x;
      const float x1 = x0 + delta_x;
      const float y = 0;
      for ( float z = platform_zmin; z < platform_zmax; z += zdelta )
        {
          PStack<pVect>& t_coords = even ? p_tile_coords : p1_tile_coords;
          p_tex_coords += trmax; p_tex_coords += tsmax;
          t_coords += pVect(x0,y,z);
          p_tex_coords += trmax; p_tex_coords += tsmin;
          t_coords += pVect(x0,y,z+zdelta);
          p_tex_coords += trmin; p_tex_coords += tsmin;
          t_coords += pVect(x1,y,z+zdelta);
          p_tex_coords += trmin; p_tex_coords += tsmax;
          t_coords += pVect(x1,y,z);
          even = !even;
        }
    }

  while ( pVect* const v = p1_tile_coords.iterate() ) p_tile_coords += *v;

  platform_tile_coords.re_take(p_tile_coords);
  platform_tile_coords.to_gpu();
  platform_tex_coords.re_take(p_tex_coords);
  platform_tex_coords.to_gpu();
}

void
```

```
World::modelview_update()
{
  pMatrix_Translate center_eye(-eye_location);
  pMatrix_Rotation rotate_eye(eye_direction,pVect(0,0,-1));
  modelview = rotate_eye * center_eye;
  pMatrix reflect; reflect.set_identity(); reflect.rc(1,1) = -1;
  transform_mirror = modelview * reflect * invert(modelview);
}

void
World::render_objects(Render_Option option)
{
  const float shininess_ball = 5;
  pColor spec_color(1,1,1);
  pColor spec_light = color_white * opt_light_intensity;
  GLboolean l0, l1;
  glGetBooleanv(GL_LIGHT0,&l0);
  glGetBooleanv(GL_LIGHT1,&l1);
  int light_state = l0 | (l1<<1);
  glLightfv(GL_LIGHT0, GL_SPECULAR, spec_light);
  const double wt = world_time;

  if ( option == RO_Normally )
    {
      glLightf(GL_LIGHT0, GL_LINEAR_ATTENUATION, 1.0);
      glEnable(GL_TEXTURE_2D);
      glBindTexture(GL_TEXTURE_2D,texid_emacs);
      glMaterialf(GL_FRONT_AND_BACK,GL_SHININESS,shininess_ball);
      glMaterialfv(GL_FRONT_AND_BACK,GL_SPECULAR,spec_color);
      glLightModeli(GL_LIGHT_MODEL_COLOR_CONTROL, GL_SEPARATE_SPECULAR_COLOR);
      glEnable(GL_COLOR_SUM);
    }

  cyl.apex_radius = 1; cyl.set_color(color_lsu_spirit_purple);
  if ( option == RO_Shadow_Volumes )
    {
      if ( opt_shadow_volumes )
        {
          glEnable(GL_LIGHTING);
          glColorMask(true,true,true,true);
          glLightf(GL_LIGHT0, GL_LINEAR_ATTENUATION, 1.0);
          glEnable(GL_BLEND);
          glBlendEquation(GL_FUNC_ADD);
          glBlendFunc(GL_ONE,GL_ONE);
          glColor3f(0.8,0,0);
        }
      cyl.light_pos = light_location;
      sphere.light_pos = light_location;

      sphere.render_bunch_gather_sv(wt);
      for ( Ball *ball: balls )
        sphere.render_shadow_volume(ball->radius,ball->position);

      sp_instances_sv->use();
      glUniform2i(3, opt_tryout1, opt_tryout2);
      sphere.render_bunch_render_sv();
      sp_fixed->use();

      for ( Link *link: links )
        {
          if ( link->snapped ) continue;
          if ( !link->is_renderable ) continue;
          if ( !link->is_simulatable ) continue; // Should check if straight.
          Ball *const ball1 = link->ball1;
          Ball *const ball2 = link->ball2;
```

```
          cyl.render_shadow_volume
            (ball1->position,0.3*ball1->radius,
             ball2->position-ball1->position);
        }
    }
  else
    {
      sphere.opt_texture = true;

      sphere.render_bunch_gather(wt);
      for ( Ball *ball: balls )
        {
          if ( ball->contact )
            sphere.color = color_gray;
          else if ( ball->mass > 0 && ball->mass < ball->mass_min )
            sphere.color = color_red;
          else if ( ball->mass > 0 && ball->locked )
            sphere.color = color_pale_green;
          else
            sphere.color = ball->color;
          sphere.render
            (ball->radius,ball->position,
             pMatrix_Rotation(ball->orientation));
        }

      sp_instances_sphere->use();
      glUniform1i(2, light_state);

      glUniform2i(3, opt_tryout1, opt_tryout2);
      sphere.render_bunch_render();

      glBindTexture(GL_TEXTURE_2D,texid_hw);

      const bool rl2 = opt_shader >= SO_Set_1;

      if ( rl2 )
        {
          render_link_2_start();
        }
      else
        {
          sp_phong->use();
          glUniform1i(2, light_state);
          glUniform2i(3, opt_tryout1, opt_tryout2);
        }

      for ( Link *link: links )
        {
          if ( !link->is_renderable ) continue;
          if ( !rl2 )
            render_link_1(link);
          else
            render_link_2_gather(link);
          continue;

          Ball *const ball1 = link->ball1;
          Ball *const ball2 = link->ball2;
          pVect dir1 = ball1->omatrix * link->cb1;
          pCoor pos1 = ball1->position + dir1;

          pVect dir2 = ball2->omatrix * link->cb2;
          pCoor pos2 = ball2->position + dir2;

          cyl.set_color(link->color);
          cyl.render(pos1,0.3*ball1->radius, pos2-pos1);
```

```
        }

      if ( rl2 )
        render_link_2_render();

    }

  sp_fixed->use();

  if ( option == RO_Shadow_Volumes ) return;

  sp_phong->use();
  glUniform1i(2, light_state);
  glUniform2i(3, opt_tryout1, opt_tryout2);

  //
  // Render Platform
  //
  const int half_elements = platform_tile_coords.elements >> 3 << 2;

  glEnable(GL_TEXTURE_2D);

  // Set up attribute (vertex, normal, etc.) arrays.
  //
  glBindTexture(GL_TEXTURE_2D,texid_syl);
  platform_tile_coords.bind();
  glVertexPointer(3, GL_FLOAT, sizeof(platform_tile_coords.data[0]), 0);
  glEnableClientState(GL_VERTEX_ARRAY);
  glNormal3fv(platform_normal);
  platform_tex_coords.bind();
  glTexCoordPointer(2, GL_FLOAT,2*sizeof(float), 0);
  glEnableClientState(GL_TEXTURE_COORD_ARRAY);

  // Write lighter-colored, textured tiles.
  //
  glMaterialfv(GL_FRONT_AND_BACK,GL_SPECULAR,spec_color);
  glMaterialf(GL_FRONT_AND_BACK,GL_SHININESS,5.0);
  glColor3f(0.55,0.55,0.55);
  glDrawArrays(GL_QUADS,0,half_elements+4);

  glDisableClientState(GL_VERTEX_ARRAY);
  glDisableClientState(GL_TEXTURE_COORD_ARRAY);
  glBindBuffer(GL_ARRAY_BUFFER,0);

  sp_fixed->use();
}

void
World::render()
{
  // Get any waiting keyboard commands.
  //
  cb_keyboard();

  // Start a timer object used for tuning this code.
  //
  frame_timer.frame_start();

  /// Emit a Graphical Representation of Simulation State
  //

  // Understanding of the code below not required for introductory
  // lectures.

  // That said, much of the complexity of the code is to show
```

```
// the ball shadow and reflection.


const int win_width = ogl_helper.get_width();
const int win_height = ogl_helper.get_height();
const float aspect = float(win_width) / win_height;

glMatrixMode(GL_MODELVIEW);
glLoadTransposeMatrixf(modelview);

glMatrixMode(GL_PROJECTION);
glLoadIdentity();
// Frustum: left, right, bottom, top, near, far
const float xr = .08;
glFrustum(-xr,xr,-xr/aspect,xr/aspect,.1,5000);

glViewport(0, 0, win_width, win_height);
pError_Check();

glClearColor(0,0,0,0.5);
glClearDepth(1.0);
glClearStencil(0);
glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT | GL_STENCIL_BUFFER_BIT );

glEnable(GL_DEPTH_TEST);
glDepthFunc(GL_LESS);
glDisable(GL_BLEND);

glLightModeli(GL_LIGHT_MODEL_TWO_SIDE,1);
glLightfv(GL_LIGHT0, GL_POSITION, light_location);

glLightf(GL_LIGHT0, GL_CONSTANT_ATTENUATION, 0.5);
glLightf(GL_LIGHT0, GL_LINEAR_ATTENUATION, 1.0);
glLightf(GL_LIGHT0, GL_QUADRATIC_ATTENUATION, 0);

pColor ambient_color(0x555555);

glLightModelfv(GL_LIGHT_MODEL_AMBIENT, ambient_color);
glLightfv(GL_LIGHT0, GL_DIFFUSE, color_white * opt_light_intensity);
glLightfv(GL_LIGHT0, GL_AMBIENT, color_black);
glLightfv(GL_LIGHT0, GL_SPECULAR, color_white * opt_light_intensity);

glEnable(GL_LIGHT0);
glEnable(GL_LIGHTING);

glEnable(GL_COLOR_MATERIAL);
glColorMaterial(GL_FRONT_AND_BACK,GL_AMBIENT_AND_DIFFUSE);

glShadeModel(GL_SMOOTH);

// Common to all textures.
//
glActiveTexture(GL_TEXTURE0);
glEnable(GL_TEXTURE_2D);
glTexParameterf(GL_TEXTURE_2D,GL_TEXTURE_MIN_FILTER,
                GL_LINEAR_MIPMAP_LINEAR);
glTexParameterf(GL_TEXTURE_2D,GL_TEXTURE_MAG_FILTER,GL_LINEAR);
glTexEnvi(GL_TEXTURE_ENV,GL_TEXTURE_ENV_MODE,GL_MODULATE);
glTexParameterf(GL_TEXTURE_2D,GL_TEXTURE_WRAP_S,GL_REPEAT);
glTexParameterf(GL_TEXTURE_2D,GL_TEXTURE_WRAP_T,GL_REPEAT);

glEnable(GL_RESCALE_NORMAL);
glEnable(GL_NORMALIZE);

const double time_now = time_wall_fp();
```

```
  const bool blink_visible = int64_t(time_now*3) & 1;
# define BLINK(txt,pad) ( blink_visible ? txt : pad )

  ogl_helper.fbprintf("%s\n",frame_timer.frame_rate_text_get());

  ogl_helper.fbprintf
    ("Code Compiled: %s\n",
#ifdef __OPTIMIZE__
     "WITH OPTIMIZATION"
#else
     BLINK("WITHOUT OPTIMIZATION","")
#endif
     );

  ogl_helper.fbprintf
    ("Time Step: %8d  World Time: %11.6f  %s\n",
     time_step_count, world_time,
     opt_pause ? BLINK("PAUSED, 'p' to unpause, SPC or S-SPC to step.","") :
     "Press 'p' to pause."
     );

  ogl_helper.fbprintf
    ("Eye location: [%5.1f, %5.1f, %5.1f]  "
     "Eye direction: [%+.2f, %+.2f, %+.2f]\n",
     eye_location.x, eye_location.y, eye_location.z,
     eye_direction.x, eye_direction.y, eye_direction.z);

  Ball& ball = *balls[0];

  ogl_helper.fbprintf
    ("Head Ball Pos  [%5.1f,%5.1f,%5.1f] Vel [%+5.1f,%+5.1f,%+5.1f]\n",
     ball.position.x,ball.position.y,ball.position.z,
     ball.velocity.x,ball.velocity.y,ball.velocity.z );

  const char *sh_names[] = { "PLAIN", "SET 1", "SET 2" };

  ogl_helper.fbprintf
    ("Shader: %s  ('v' to change)  "
     "Tryout 1: %s  ('y' to change)  Tryout 2: %s  ('Y' to change)\n",
     sh_names[opt_shader],
     opt_tryout1 ? BLINK("ON","  ") : "OFF",
     opt_tryout2 ? BLINK("ON","  ") : "OFF");

  pVariable_Control_Elt* const cvar = variable_control.current;
  ogl_helper.fbprintf("VAR %s = %.5f  (TAB or '`' to change, +/- to adjust)\n",
                      cvar->name,cvar->get_val());

  const int half_elements = platform_tile_coords.elements >> 3 << 2;

  //
  // Render ball reflection.  (Will be blended with dark tiles.)
  //

  if ( opt_mirror )
    {
      // Write stencil value 2 at location of dark (mirrored) tiles.
      //
      glDisable(GL_LIGHTING);
      glEnable(GL_STENCIL_TEST);
      glStencilFunc(GL_NEVER,2,2);
      glStencilOp(GL_REPLACE,GL_KEEP,GL_KEEP);
      platform_tile_coords.bind();
      glVertexPointer(3, GL_FLOAT, sizeof(platform_tile_coords.data[0]), 0);
      glEnableClientState(GL_VERTEX_ARRAY);
      glDrawArrays(GL_QUADS,half_elements+4,half_elements-4);
```

```
      glEnable(GL_LIGHTING);
      glDisableClientState(GL_VERTEX_ARRAY);
      glBindBuffer(GL_ARRAY_BUFFER,0);

      // Prepare to write only stenciled locations.
      //
      glStencilFunc(GL_EQUAL,2,2);
      glStencilOp(GL_KEEP,GL_KEEP,GL_KEEP);

      // Use a transform that reflects objects to other side of platform.
      //
      glMatrixMode(GL_PROJECTION);
      glPushMatrix();
      glMultTransposeMatrixf(transform_mirror);

      // Reflected front face should still be treated as the front face.
      //
      glFrontFace(GL_CW);

      render_objects(RO_Normally);

      glFrontFace(GL_CCW);
      glMatrixMode(GL_PROJECTION);
      glPopMatrix();
    }

  // Blend mirror tiles with possibly existing ball reflection.
  //
  glDepthFunc(GL_ALWAYS);

  glDisable(GL_STENCIL_TEST);
  glDisable(GL_TEXTURE_2D);

  glEnable(GL_LIGHTING);
  glLightModeli(GL_LIGHT_MODEL_COLOR_CONTROL, GL_SINGLE_COLOR);
  glLightfv(GL_LIGHT0, GL_SPECULAR, color_black);

  glColor3fv(color_lsu_spirit_purple);

  glEnable(GL_BLEND);
  glBlendEquation(GL_FUNC_ADD);
  glBlendFunc(GL_CONSTANT_ALPHA,GL_ONE_MINUS_CONSTANT_ALPHA); // src, dst
  glBlendColor(0,0,0,0.5);

  platform_tile_coords.bind();
  glVertexPointer
    (3, GL_FLOAT,sizeof(platform_tile_coords.data[0]), 0);
  glEnableClientState(GL_VERTEX_ARRAY);

  glDrawArrays(GL_QUADS,half_elements+4,half_elements-4);

  glDisableClientState(GL_VERTEX_ARRAY);
  glBindBuffer(GL_ARRAY_BUFFER,0);

  glDepthFunc(GL_LESS);
  glDisable(GL_BLEND);

  glLightModelfv(GL_LIGHT_MODEL_AMBIENT, ambient_color);

  if ( !opt_shadows )
    {
      // Render.
      //
      render_objects(RO_Normally);
    }
```

```
  else
    {
      /// Render With Shadows

      //
      // First pass, render using only ambient light.
      //
      glDisable(GL_LIGHT0);

      // Send balls, tiles, and platform to opengl.
      //
      render_objects(RO_Normally);

      //
      // Second pass, add on light0.
      //

      // Turn off ambient light, turn on light 0.
      //
      glLightModelfv(GL_LIGHT_MODEL_AMBIENT, color_black);
      glEnable(GL_LIGHT0);


      glClear(GL_STENCIL_BUFFER_BIT);

      // Make sure that only stencil buffer written.
      //
      glDepthMask(false);

      glColorMask(false,false,false,false);

      // Don't waste time computing lighting.
      //
      glDisable(GL_LIGHTING);
      glDisable(GL_TEXTURE_2D);

      // Set up stencil test to count shadow volume surfaces: plus 1 for
      // entering the shadow volume, minus 1 for leaving the shadow
      // volume.
      //
      glEnable(GL_STENCIL_TEST);
      // sfail, dfail, dpass
      glStencilOpSeparate(GL_FRONT,GL_KEEP,GL_KEEP,GL_INCR_WRAP);
      glStencilOpSeparate(GL_BACK,GL_KEEP,GL_KEEP,GL_DECR_WRAP);
      glStencilFuncSeparate(GL_FRONT_AND_BACK,GL_ALWAYS,1,-1); // ref, mask

      // Write stencil with shadow locations based on shadow volumes
      // cast by light0 (light_location).  Shadowed locations will
      // have a positive stencil value.
      //
      render_objects(RO_Shadow_Volumes);

      glEnable(GL_LIGHTING);
      glColorMask(true,true,true,true);
      glDepthMask(true);

      // Use stencil test to prevent writes to shadowed areas.
      //
      glStencilOp(GL_KEEP,GL_KEEP,GL_KEEP);
      glStencilFunc(GL_EQUAL,0,-1); // ref, mask

      // Allow pixels to be re-written.
      //
      glDepthFunc(GL_LEQUAL);
      glEnable(GL_BLEND);
```

```
      glBlendEquation(GL_FUNC_ADD);
      glBlendFunc(GL_ONE,GL_ONE);

      // Render.
      //
      render_objects(RO_Normally);

      glDisable(GL_BLEND);
      glDisable(GL_STENCIL_TEST);

    }

  glDepthFunc(GL_LESS);

  // Render Marker for Light Source
  //
  insert_tetrahedron(light_location,0.5);

  pError_Check();

  glColor3f(0.5,1,0.5);

  frame_timer.frame_end();

  ogl_helper.user_text_reprint();

  glutSwapBuffers();
}


void
World::cb_keyboard()
{
  if ( !ogl_helper.keyboard_key ) return;
  pVect adjustment(0,0,0);
  pVect user_rot_axis(0,0,0);
  const bool kb_mod_s = ogl_helper.keyboard_shift;
  const bool kb_mod_c = ogl_helper.keyboard_control;
  const float move_amt = kb_mod_s ? 2.0 : kb_mod_c ? 0.08 : 0.4;

  switch ( ogl_helper.keyboard_key ) {
  case FB_KEY_LEFT: adjustment.x = -move_amt; break;
  case FB_KEY_RIGHT: adjustment.x = move_amt; break;
  case FB_KEY_PAGE_UP: adjustment.y = move_amt; break;
  case FB_KEY_PAGE_DOWN: adjustment.y = -move_amt; break;
  case FB_KEY_DOWN: adjustment.z = move_amt; break;
  case FB_KEY_UP: adjustment.z = -move_amt; break;
  case FB_KEY_DELETE: user_rot_axis.y = 1; break;
  case FB_KEY_INSERT: user_rot_axis.y =  -1; break;
  case FB_KEY_HOME: user_rot_axis.x = 1; break;
  case FB_KEY_END: user_rot_axis.x = -1; break;
  case '1': ball_setup_1(); break;
  case '2': ball_setup_2(); break;
  case '3': ball_setup_3(); break;
  case '4': ball_setup_4(); break;
  case '5': ball_setup_5(); break;
  case 'b': opt_move_item = MI_Ball; break;
  case 'B': opt_move_item = MI_Ball_V; break;
  case 'e': case 'E': opt_move_item = MI_Eye; break;
  case 'g': case 'G': opt_gravity = !opt_gravity; break;
  case 'h': case 'H': opt_head_lock = !opt_head_lock; break;
  case 't': case 'T': opt_tail_lock = !opt_tail_lock; break;
  case 'l': case 'L': opt_move_item = MI_Light; break;
  case 'r': case 'R': opt_mirror = !opt_mirror; break;
  case 'o': opt_shadows = !opt_shadows; break;
```

```
  case 'O':
    opt_shadow_volumes = !opt_shadow_volumes;
    if ( opt_shadow_volumes ) opt_shadows = true;
    break;
  case 'p': case 'P': opt_pause = !opt_pause; break;
  case 's': case 'S': balls_stop(); break;
  case 'v': case 'V': opt_shader++; if ( opt_shader > 2 ) opt_shader = 0; break;
  case 'w': case 'W': balls_twirl(); break;
  case 'y': opt_tryout1 = !opt_tryout1; break;
  case 'Y': opt_tryout2 = !opt_tryout2; break;
  case ' ':
    if ( kb_mod_s ) opt_single_time_step = true; else opt_single_frame = true;
    opt_pause = true;
    break;
  case 9: variable_control.switch_var_right(); break;
  case 96: variable_control.switch_var_left(); break; // `, until S-TAB works.
  case '-':case '_': variable_control.adjust_lower(); break;
  case '+':case '=': variable_control.adjust_higher(); break;
  default: printf("Unknown key, %d\n",ogl_helper.keyboard_key); break;
  }

  gravity_accel.y = opt_gravity ? -opt_gravity_accel : 0;

  lock_update();
  sphere.bunch_invalidate();

  // Update eye_direction based on keyboard command.
  //
  if ( user_rot_axis.x || user_rot_axis.y )
    {
      pMatrix_Rotation rotall(eye_direction,pVect(0,0,-1));
      user_rot_axis *= invert(rotall);
      eye_direction *= pMatrix_Rotation(user_rot_axis, M_PI * 0.03);
      modelview_update();
    }

  // Update eye_location based on keyboard command.
  //
  if ( adjustment.x || adjustment.y || adjustment.z )
    {
      const double angle =
        fabs(eye_direction.y) > 0.99
        ? 0 : atan2(eye_direction.x,-eye_direction.z);
      pMatrix_Rotation rotall(pVect(0,1,0),-angle);
      adjustment *= rotall;

      switch ( opt_move_item ){
      case MI_Ball:
        adj_vector = adjustment;
        if ( adj_t_stop == 0 )
          adj_t_prev = adj_t_stop = world_time;
        adj_t_stop += adj_duration;
        break;
      case MI_Ball_V: balls_push(adjustment,0); break;
      case MI_Light: light_location += adjustment; break;
      case MI_Eye: eye_location += adjustment; break;
      default: break;
      }
      modelview_update();
    }

}
```

```
void
World::init(int argc, char **argv)
{
  chain_length = 14;

  opt_time_step_duration = 0.0003;
  variable_control.insert(opt_time_step_duration,"Time Step Duration");

  distance_relaxed = 15.0 / chain_length;
  opt_spring_constant = 500;
  variable_control.insert(opt_spring_constant,"Spring Constant");

  opt_gravity_accel = 9.8;
  opt_gravity = true;
  gravity_accel = pVect(0,-opt_gravity_accel,0);
  variable_control.insert(opt_gravity_accel,"Gravity");

  opt_air_resistance = 0.04;
  variable_control.insert(opt_air_resistance,"Air Resistance");

  world_time = 0;
  time_step_count = 0;
  last_frame_wall_time = time_wall_fp();
  frame_timer.work_unit_set("Steps / s");

  world_time_link_update = 0;

  opt_mirror = true;
  opt_shadows = true;
  opt_shadow_volumes = false;
  opt_shader = SO_Set_1;

  init_graphics();

  // Declared like a programmable shader, but used for fixed-functionality.
  //
  sp_fixed = new pShader();

  sp_phong = new pShader
    ("hw06-shdr.cc",// File holding shader program.
     "vs_main(); ",       // Name of vertex shader main routine.
     "gs_main_simple();", // Name of geometry shader main routine.
     "fs_main();"         // Name of fragment shader main routine.
     );

  sp_instances_sphere = new pShader
    ("hw06-shdr.cc",// File holding shader program.
     "vs_main_instances_sphere(); ", // Used to render many spheres at once.
     "gs_main_simple();", // Name of geometry shader main routine.
     "fs_main();"         // Name of fragment shader main routine.
     );

  sp_instances_sv = new pShader
    ("hw06-shdr.cc",// File holding shader program.
     "vs_main_instances_sv();" // Instances of vtx coord only. Use gl_Color.
     ,"gs_main_sv();",
     "fs_main_sv();"
     );


  PSplit exe_pieces(argv[0],'/');
  pString this_exe_name(exe_pieces.pop());

  const char* const links_shader_code_path =
```

```
    true ? "hw06-shdr-links-sol.cc" : "hw06-shdr-links.cc";

  sp_set_1 = new pShader
    (links_shader_code_path,
     "vs_main_1(); ",     // Name of vertex shader main routine.
     "gs_main_1();",      // Name of geometry shader main routine.
     "fs_main();"       // Name of fragment shader main routine.
     );
  sp_set_2 = new pShader
    (links_shader_code_path,
     "vs_main_2(); ",     // Name of vertex shader main routine.
     "gs_main_2();",      // Name of geometry shader main routine.
     "fs_main();"       // Name of fragment shader main routine.
     );

  ball_setup_3();
  lock_update();
}

Ball*
World::make_marker(pCoor position, pColor color)
{
  Ball* const ball = new Ball;
  ball->position = position;
  ball->locked = true;
  ball->velocity = pVect(0,0,0);
  ball->radius = 0.2;
  ball->mass = 0;
  ball->contact = false;
  ball->color = color;
  return ball;
}

void
World::lock_update()
{
  // This routine called when options like opt_head_lock might have
  // changed.

  // Update locked status.
  //
  if ( head_ball ) head_ball->locked = opt_head_lock;
  if ( tail_ball ) tail_ball->locked = opt_tail_lock;

  // Re-compute minimum mass needed for stability.
  //
  for ( Ball *ball: balls ) ball->spring_constant_sum = 0;
  const double dtis = pow( opt_time_step_duration, 2 );
  for ( Link *link: links )
    {
      Ball* const b1 = link->ball1;
      Ball* const b2 = link->ball2;
      b1->spring_constant_sum += opt_spring_constant;
      b2->spring_constant_sum += opt_spring_constant;
    }
  for ( Ball *ball: balls )
    {
      ball->mass_min = ball->spring_constant_sum * dtis;
      ball->constants_update();
    }
}

void
World::balls_twirl()
```

```
{
  if ( !head_ball || !tail_ball ) return;

  pNorm axis(head_ball->position, tail_ball->position);

  for ( Ball *ball: balls )
    {
      pVect b_to_top(ball->position,head_ball->position);
      const float dist_along_axis = dot(b_to_top,axis);
      const float lsq = b_to_top.mag_sq() - dist_along_axis * dist_along_axis;
      if ( lsq <= 1e-5 ) { ball->velocity = pVect(0,0,0); continue; }
      const float dist_to_axis = sqrt(lsq);
      pNorm rot_dir = cross(b_to_top,axis);
      ball->velocity += 2 * dist_to_axis * rot_dir;
    }
}

void
World::objects_erase()
{
  link_change = true;
  balls.erase();
  links.erase();
}

Links
World::link_new(Ball *ball1, Ball *ball2, float stiffness)
{
  Links links_rv;
  assert( ball1->radius > 0 );
  assert( ball2->radius > 0 );

  Link* const rlink = new Link(ball1,ball2);
  links_rv += rlink;

  pNorm n12(ball1->position,ball2->position);
  const float rad_sum = ball1->radius + ball2->radius;
  pMatrix3x3 b1rot = ball1->omatrix;
  pMatrix3x3 b2rot = ball2->omatrix;
  pCoor ctr = ball1->position
    + ( ball1->radius + 0.5 * ( n12.magnitude - rad_sum ) ) * n12;

  pNorm b1_y = b1rot * pVect(0,1,0);
  pNorm b1_x = b1rot * pVect(1,0,0);
  bool b1_dir_is_y = fabs(dot(b1_y,n12)) < 0.999;
  pNorm b1_dir = b1_dir_is_y ? b1_y : b1_x;

  pNorm con_x = cross(b1_dir,n12);
  pVect con_y = cross(n12,con_x);
  rlink->b1_dir = mult_MTV( b1rot, con_y );
  rlink->b2_dir = mult_MTV( b2rot, con_y );

  const float lrad = stiffness * ball1->radius;

  for ( int i=0; i<3; i++ )
    {
      const double theta = i * 2 * M_PI / 3;
      pVect convec = lrad * ( cos(theta) * con_x + sin(theta) * con_y );
      pCoor con = ctr + convec;
      pVect cb1 = mult_MTV( b1rot, con - ball1->position );
      pVect cb2 = mult_MTV( b2rot, con - ball2->position );
      links_rv += new Link(rlink, cb1, cb2, 0);
    }
  return links_rv;
}
```

```
///
/// Physical Simulation Code
///

 /// Initialize Simulation
//

void
World::ball_setup_1()
{
  // Arrange balls to form sort of an umbrella.

  last_setup = 1;

  float ball_rad = 1;  // Warning: Re-assigned, changes behavior of nb.

  pCoor first_pos(17.2,ball_rad,-20.2);

  pVect dir_up(0.1,3,0.1);
  pVect dir_z(3*distance_relaxed,0,0);
  pVect dir_x(0,0,3*distance_relaxed);

  // Remove objects from the simulated objects lists, balls and links.
  // The delete operator is used on objects in the lists.
  //
  objects_erase();

  auto nb =
    [&] (pCoor pos) {
      Ball* const ball = new Ball;
      ball->position = pos;
      ball->locked = false;
      ball->velocity = pVect(0,0,0);
      ball->radius = ball_rad;
      ball->contact = false;
      balls += ball;
      return ball;
      };

  tail_ball = nb(first_pos);

  pCoor last_pos = tail_ball->position;

  Balls tail_balls;
  for ( int i=1; i<8; i++ )
    {
      ball_rad *= 0.8;
      Ball* const ball = nb(last_pos + (i==1?1.0:ball_rad) * dir_up);
      tail_balls += ball;
      links += link_new(balls[balls-2],ball,0.8);
      last_pos = ball->position;
    }

  head_ball = balls[balls-1];

  pNorm dir_zn(dir_z);
  pNorm dir_xn(dir_x);
  for ( Ball *ball: tail_balls )
    {
      if ( ball == head_ball ) break;
      pNorm to_top(ball->position,head_ball->position);
      const float r = to_top.magnitude;
      int num_hairs = max(4,int(7*r));
```

```
      ball_rad = ball->radius / 4;
      const float delta_theta = 2 * M_PI / num_hairs;
      for ( int i=0; i<num_hairs; i++ )
        {
          const float theta = i * delta_theta;
          pCoor pos = ball->position +
            r * cos(theta) * dir_xn + r * sin(theta) * dir_zn;
          Ball* const ballr = nb(pos);
          links += link_new(ballr,ball);
        }
    }

  opt_head_lock = false;
  opt_tail_lock = true;
  setup_at_end();
}

void
World::ball_setup_2()
{
  // Arrange balls to form sort of an umbrella.

  last_setup = 2;

  pCoor first_pos(17.2,14.8f,-20.2);

  pVect dir_dn(0.5,first_pos.y/8,0.5);
  pVect dir_z(3*distance_relaxed,0,0);
  pVect dir_x(0,0,3*distance_relaxed);

  // Remove objects from the simulated objects lists, balls and links.
  // The delete operator is used on objects in the lists.
  //
  objects_erase();

  float ball_rad = 0.3;  // Warning: Re-assigned, changes behavior of nb.

  auto nb =
    [&] (pCoor pos) {
      Ball* const ball = new Ball;
      ball->position = pos;
      ball->locked = false;
      ball->velocity = pVect(0,0,0);
      ball->radius = ball_rad;
      ball->contact = false;
      balls += ball;
      return ball;
      };

  head_ball = nb(first_pos);
  for ( int j: {-1,1} )
    for ( int i: {-1,0,1} )
      {
        Ball* const ball = nb(first_pos - dir_dn + i * dir_x + j * dir_z );
        if ( i != 0 ) links += link_new(head_ball,ball);
        if ( i != -1 ) links += link_new(balls[balls-2],ball);
        if ( j == 1 && i != 0 )  links += link_new(balls[balls-4],ball);
      }

  Ball* const tail_start = nb(first_pos - dir_dn);
  links += link_new(balls[balls-6],tail_start);
  links += link_new(balls[balls-3],tail_start);
  pCoor last_pos = tail_start->position;
  Balls tail_balls;
  for ( int i=1; i<5; i++ )
```

```
    {
      Ball* const ball = nb(last_pos - 0.5 * dir_dn);
      tail_balls += ball;
      links += link_new(balls[balls-2],ball);
      last_pos = ball->position;
    }

  tail_ball = balls[balls-1];

  pNorm dir_zn(dir_z);
  pNorm dir_xn(dir_x);
  int num_hairs = 20;
  ball_rad = 0.1;
  for ( Ball *ball: tail_balls )
    {
      const float delta_theta = 2 * M_PI / num_hairs;
      for ( int i=0; i<num_hairs; i++ )
        {
          const float theta = i * delta_theta;
          const float r = ball->radius * 10;
          pCoor pos = ball->position +
            r * cos(theta) * dir_xn + r * sin(theta) * dir_zn;
          Ball* const ballr = nb(pos);
          links += link_new(ballr,ball);
        }
    }

  opt_head_lock = true;    // Head ball will be frozen in space.
  opt_tail_lock = false;   // Tail ball can move freely.
  setup_at_end();
}

void
World::ball_setup_3()
{
  last_setup = 3;
  objects_erase();

  pCoor first_pos(17.2,14.8f,-10.2);

  float ball_radius = 1;

  auto nb =
    [&] (pCoor pos) {
      Ball* const ball = new Ball;
      ball->position = pos;
      ball->locked = false;
      ball->velocity = pVect(0,0,0);
      ball->radius = ball_radius;
      ball->contact = false;
      balls += ball;
      return ball;
      };

  head_ball = nb(first_pos);
  ball_radius = ball_radius * .1;

  for ( int i=0; i<200; i++ )
    {
      pNorm dir(0.5-drand48(),0.5-drand48(),0.5-drand48());
      Ball* const ball = nb(first_pos + head_ball->radius * 3 * dir);
      links += link_new(ball,head_ball,2);
    }

  tail_ball = balls[1];
```

```
  opt_head_lock = false;    // Head ball will be frozen in space.
  opt_tail_lock = false;   // Tail ball can move freely.

  setup_at_end();
}


void
World::ball_setup_4()
{
  last_setup = 4;
  objects_erase();
  setup_at_end();
}

void
World::ball_setup_5()
{
  last_setup = 5;
  objects_erase();
  setup_at_end();
}


void
World::setup_at_end()
{
}


 /// Advance Simulation State by delta_t Seconds
//
void
World::time_step_cpu(double delta_t)
{
  time_step_count++;
  const int n_balls = balls;
  const int n_links = links;

  // Smoothly move ball in response to user input.
  //
  if ( adj_t_stop )
    {
      const double dt = min(world_time,adj_t_stop) - adj_t_prev;
      pVect adj = dt/adj_duration * adj_vector;
      balls_translate(adj,0);
      adj_t_prev = world_time;
      if ( world_time >= adj_t_stop ) adj_t_stop = 0;
    }

  for ( int bi=0; bi<n_balls; bi++ )
    {
      Ball* const ball = balls[bi];
      assert( ball->fdt_to_do );
      ball->force = ball->mass * gravity_accel;
      ball->torque = pVect(0,0,0);
    }

#pragma omp parallel for
  for ( int i=0; i<n_links; i++ )
    {
      Link* const link = links[i];
      if ( !link->is_simulatable ) continue;
```

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

      // Find position and velocity of the point where the link touches
      // the surface of ball 1 ...
      //
      pVect dir1 = ball1->omatrix * link->cb1;
      pCoor pos1 = ball1->position + dir1;
      pVect vel1 = ball1->velocity + ball1->point_rot_vel(dir1);

      // ... and ball 2.
      //
      pVect dir2 = ball2->omatrix * link->cb2;
      pCoor pos2 = ball2->position + dir2;
      pVect vel2 = ball2->velocity + ball2->point_rot_vel(dir2);

      // Construct a normalized (Unit) Vector from ball to neighbor
      // based on link connection points and ball centers.
      //
      pNorm link_dir(pos1,pos2);
      pNorm c_to_c(ball1->position,ball2->position);

      const float link_length = link_dir.magnitude;

      // Compute the speed of ball's end of link towards neighbor's end of link.
      //
      pVect delta_v = vel2 - vel1;
      float delta_s = dot( delta_v, link_dir );

      // Compute by how much the spring is stretched (positive value)
      // or compressed (negative value).
      //
      const float spring_stretch = link_length - link->distance_relaxed;

      // Determine whether spring is gaining energy (whether its length
      // is getting further from its relaxed length).
      //
      const bool gaining_e = ( delta_s > 0.0 ) == ( spring_stretch > 0 );

      // Use a smaller spring constant when spring is loosing energy,
      // a quick and dirty way of simulating energy loss due to spring
      // friction.
      //
      const float spring_constant =
        gaining_e ? opt_spring_constant : opt_spring_constant * 0.7;

      const float force_mag = spring_constant * spring_stretch;
      pVect spring_force_12 = force_mag * link_dir;

      // Apply forces affecting linear momentum.
      //
      link->spring_force_12 = spring_force_12;

      if ( ! link->is_surface_connection ) continue;

      // Apply torque.
      //
      link->torque1 = cross(pNorm(dir1), spring_force_12);
      link->torque2 = cross(pNorm(dir2), spring_force_12);
    }

  // Note: Because two links can reference the same ball this should
  // not be done in parallel.
```

```
  for ( Link *link: links )
    {
      if ( !link->is_simulatable ) continue;
      Ball* const ball1 = link->ball1;
      Ball* const ball2 = link->ball2;
      ball1->force += link->spring_force_12;
      ball2->force -= link->spring_force_12;
      ball1->torque += link->torque1;
      ball2->torque -= link->torque2;
    }

  ///
  /// Update Position of Each Ball
  ///

#pragma omp parallel for
  for ( int bi=0; bi<n_balls; bi++ )
    {
      Ball* const ball = balls[bi];
      if ( ball->locked )
        {
          ball->velocity = pVect(0,0,0);
          ball->omega = pVect(0,0,0);
          continue;
        }

      // Update Velocity
      //
      // This code assumes that force on ball is constant over time
      // step. This is clearly wrong when balls are moving with
      // respect to each other because the springs are changing
      // length. This inaccuracy will make the simulation unstable
      // when spring constant is large for the time step.
      //
      const float mass = max( ball->mass, ball->mass_min );

      pVect delta_v = ( ball->force / mass ) * delta_t;

      if ( platform_collision_possible(ball->position) && ball->position.y < 0 )
        {
          const float spring_constant_plat =
            ball->velocity.y < 0 ? 100000 : 50000;
          const float fric_coefficient = 0.1;
          const float force_up = -ball->position.y * spring_constant_plat;
          const float delta_v_up = force_up / mass * delta_t;
          const float fric_force_mag = fric_coefficient * force_up;
          pNorm surface_v(ball->velocity.x,0,ball->velocity.z);
          const float delta_v_surf = fric_force_mag / mass * delta_t;

          if ( delta_v_surf > surface_v.magnitude )
            {
              // Ignoring other forces?
              delta_v =
                pVect(-ball->velocity.x,delta_v.y,-ball->velocity.z);
            }
          else
            {
              delta_v -= delta_v_surf * surface_v;
            }
          delta_v.y += delta_v_up;
        }

      ball->velocity += delta_v;

      // Air Resistance
```

```
      //
      const double fs = pow(1+opt_air_resistance,-delta_t);
      ball->velocity *= fs;

      // Update Position
      //
      // Assume that velocity is constant.
      //
      ball->position += ball->velocity * delta_t;

      ball->omega += delta_t * ball->fdt_to_do * ball->torque;

      pNorm axis(ball->omega);

      // Update Orientation
      //
      // If ball isn't spinning fast skip expensive rotation.
      //
      if ( axis.mag_sq < 0.000001 ) continue;

      // Update ball orientation.
      //
      ball->orientation_set
        ( pQuat(axis,delta_t * axis.magnitude) * ball->orientation );
    }
}

bool
World::platform_collision_possible(pCoor pos)
{
  // Assuming no motion in x or z axes.
  //
  return pos.x >= platform_xmin && pos.x <= platform_xmax
    && pos.z >= platform_zmin && pos.z <= platform_zmax;
}

 /// External Modifications to State
//
//   These allow the user to play with state while simulation
//   running.

// Move the ball.
//
void Ball::translate(pVect amt) {position += amt;}

// Add velocity to the ball.
//
void Ball::push(pVect amt) {velocity += amt;}

// Set the velocity to zero.
//
void Ball::stop() {velocity = pVect(0,0,0); }

// Set the velocity and rotation (not yet supported) to zero.
//
void Ball::freeze() {velocity = pVect(0,0,0); }



void World::balls_translate(pVect amt,int b){head_ball->translate(amt);}
void World::balls_push(pVect amt,int b){head_ball->push(amt);}
void World::balls_translate(pVect amt)
{ for ( Ball *ball: balls ) ball->translate(amt); }
```

```
void World::balls_push(pVect amt)
{ for ( Ball *ball: balls ) ball->push(amt); }
void World::balls_stop()
{ for ( Ball *ball: balls ) ball->stop(); }
void World::balls_freeze(){balls_stop();}


void
World::render_link_1(Link *link)
{
  Ball *const ball1 = link->ball1;
  Ball *const ball2 = link->ball2;

  // Direction of link relative to center off ball.
  //
  pVect dir1 = ball1->omatrix * link->cb1;

  // Place on surface of ball where link connects.
  //
  pCoor pos1 = ball1->position + dir1;

  // Direction and connection location for ball 2.
  //
  pVect dir2 = ball2->omatrix * link->cb2;
  pCoor pos2 = ball2->position + dir2;

  pVect p1p2(pos1,pos2);
  pNorm p1p2n(p1p2);

  // Number of segments used to construct link.  Each segment is
  // approximately a cylinder.
  //
  const int segments = opt_segments;

  // Number of sides of each cylinder.
  //
  const int sides = opt_sides;

  const float delta_tee = 1.0 / segments;

  // Radius of link.
  //
  const float rad = ball1->radius * 0.3;

  // Compute scale factors for texture.
  const float tex_aspect_ratio = 3147 / 1151;
  const float page_width_o = 2.5;  // Width of texture in object-space coords.
  const float tex_t_scale = link->distance_relaxed / page_width_o;
  const float tex_angle_scale = rad * tex_aspect_ratio / page_width_o;

  pNorm dirn1(dir1);
  pNorm dirn2(dir2);

  // Vectors used to describe the cubic Hermite curve.
  //
  pVect v1 = p1p2n.magnitude * dirn1;
  pVect v2 = p1p2n.magnitude * dirn2;

  // Convert link's local x and y axes to global coordinates.
  //
  pVect b1_ydir = ball1->omatrix * link->b1_dir;

  pVect b2_ydir = ball2->omatrix * link->b2_dir;
```

```
  pCoor pts[sides+1];
  pVect vecs[sides+1];

  pCoors coords;
  pVects norms;
  pVector<float> tcoords;

  for ( int i=0; i<=segments; i++ )
    {
      const float t = i * delta_tee;
      const float t2 = t * t;
      const float t3 = t2 * t;

      // Cubic Hermite interpolation.
      // Compute point along core of link.
      //
      pCoor ctr =
        ( 2*t3 - 3*t2 + 1 ) * pos1 + (-2*t3 + 3*t2 ) * pos2
        + (t3 - 2*t2 + t ) * v1 - (t3 - t2 ) * v2;

      // Compute direction of link at this point.
      //
      pNorm tan =
        ( 6*t2 - 6*t ) * pos1 + (-6*t2 + 6*t ) * pos2
        + (3*t2 - 4*t + 1 ) * v1 - (3*t2 - 2*t ) * v2;

      // Compute local x and y axes for drawing a cylinder.
      //
      pVect ydir = (1-t) * b1_ydir + t * b2_ydir;
      pNorm norm = cross(tan,ydir);
      pVect binorm = cross(tan,norm);

      for ( int j=0; j<sides; j++ )
        {
          const float theta = j * ( 2 * M_PI / sides );
          pCoor pt = pts[j];
          pVect vec = vecs[j];
          vecs[j] = cosf(theta) * norm + sinf(theta) * binorm;
          pts[j] = ctr + rad * vecs[j];

          if ( opt_shader == SO_Set_1 ) pts[j].w = theta;

          if ( i == 0 ) continue;

          tcoords += t * tex_t_scale;
          tcoords += theta * tex_angle_scale;

          norms += vecs[j];
          coords += pts[j];

          tcoords += ( t - delta_tee ) * tex_t_scale;
          tcoords += theta * tex_angle_scale;

          norms += vec;
          coords += pt;

        }
    }

  if ( !link_bo_vtx ) glGenBuffers(3,&link_bo_vtx);

  glBindBuffer(GL_ARRAY_BUFFER, link_bo_vtx);
  glBufferData
    (GL_ARRAY_BUFFER,           // Kind of buffer object.
     sizeof(coords[0])*coords.size(), coords.data(),
```

```
      GL_STREAM_DRAW);
  glBindBuffer(GL_ARRAY_BUFFER, link_bo_nrm);
  glBufferData
    (GL_ARRAY_BUFFER,           // Kind of buffer object.
     sizeof(norms[0])*norms.size(), norms.data(),
     GL_STREAM_DRAW);
  glBindBuffer(GL_ARRAY_BUFFER, link_bo_tco);
  glBufferData
    (GL_ARRAY_BUFFER,           // Kind of buffer object.
     sizeof(tcoords[0])*tcoords.size(), tcoords.data(),
     GL_STREAM_DRAW);

  const int num_vtx = coords.size();

  glColor3fv( color_dim_gray );

  glEnableClientState(GL_VERTEX_ARRAY);
  glBindBuffer(GL_ARRAY_BUFFER, link_bo_vtx);
  glVertexPointer( 4, GL_FLOAT, sizeof(coords[0]), NULL);
  glEnableClientState(GL_NORMAL_ARRAY);
  glBindBuffer(GL_ARRAY_BUFFER, link_bo_nrm);
  glNormalPointer( GL_FLOAT, 0, NULL );
  glEnableClientState(GL_TEXTURE_COORD_ARRAY);
  glBindBuffer(GL_ARRAY_BUFFER, link_bo_tco);
  glTexCoordPointer(2,GL_FLOAT,0,NULL);
  glBindBuffer(GL_ARRAY_BUFFER, 0);

  glDrawArrays(GL_TRIANGLE_STRIP, 0, num_vtx );

  glDisableClientState(GL_NORMAL_ARRAY);
  glDisableClientState(GL_VERTEX_ARRAY);
  glDisableClientState(GL_TEXTURE_COORD_ARRAY);

}

void
World::render_link_2_start()
{
  if ( world_time_link_update == world_time ) return;

  if ( lis_pvects.size() == 0 )
    {
      lis_pvects += &lis_pos1;
      lis_pvects += &lis_pos2;

      lis_pvects += &lis_v1;
      lis_pvects += &lis_v2;
      lis_pvects += &lis_b1_ydir;
      lis_pvects += &lis_b2_ydir;
      lis_pvects += &lis_tex_rad;
      const int n_pvects = lis_pvects.size();
      lis_bos = new GLuint[n_pvects];
      glGenBuffers(n_pvects,lis_bos);
    }

  // Empty containers in preparation for a new pass.
  for ( auto vl: lis_pvects ) vl->clear();
}

void
World::render_link_2_gather(Link *link)
{
  // Place data needed to render link (the argument to this function)
  // in arrays. Just before rendering those arrays will be placed in
  // buffer objects.
```

```
// If we've already collected links, just return.  This happens
// when doing multiple passes to render shadows and reflections.
//
if ( world_time_link_update == world_time ) return;

Ball *const ball1 = link->ball1;
Ball *const ball2 = link->ball2;

// Direction of link relative to center off ball.
//
pVect dir1 = ball1->omatrix * link->cb1;

// Place on surface of ball where link connects.
//
pCoor pos1 = ball1->position + dir1;

// Direction and connection location for ball 2.
//
pVect dir2 = ball2->omatrix * link->cb2;
pCoor pos2 = ball2->position + dir2;

pVect p1p2(pos1,pos2);
pNorm p1p2n(p1p2);

// Number of segments used to construct link.  Each segment is
// approximately a cylinder.
//
const int segments = opt_segments;

// Number of sides of each cylinder.
//
const int sides = opt_sides;

const float delta_tee = 1.0 / segments;

// Radius of link.
//
const float rad = ball1->radius * 0.3;

// Compute scale factors for texture.
const float tex_aspect_ratio = 8.5 / 11;  // Yes, inches. Can't remember mm.
const float page_width_o = 2.5;  // Width of texture in object-space coords.
const float tex_t_scale = link->distance_relaxed / page_width_o;
const float tex_angle_scale = rad * tex_aspect_ratio / page_width_o;

pNorm dirn1(dir1);
pNorm dirn2(dir2);

// Vectors used to describe the cubic Hermite curve.
//
pVect v1 = p1p2n.magnitude * dirn1;
pVect v2 = p1p2n.magnitude * dirn2;

// Convert link's local x and y axes to global coordinates.
//
pVect b1_ydir = ball1->omatrix * link->b1_dir;
pVect b2_ydir = ball2->omatrix * link->b2_dir;


// Place data needed to render links in lists. Note that
// the += operator appends the right-hand side to the container
// on the left-hand side.
//
```

```
  lis_pos1 += pos1;
  lis_pos2 += pos2;
  lis_v1 += v1;
  lis_v2 += v2;
  lis_b1_ydir += b1_ydir;
  lis_b2_ydir += b2_ydir;
  pVect misc(tex_t_scale,tex_angle_scale,rad);
  lis_tex_rad += misc;
}



void
World::render_link_2_render()
{
  // Perform a rendering pass to render all the links collected
  // by the routine render_link_2.


  // Check whether this is the first time we are rendering this data.
  //
  const bool first_render = world_time_link_update != world_time;
  world_time_link_update = world_time;

  // Number of segments used to construct link.  Each segment is
  // approximately a cylinder.
  //
  const int segments = opt_segments;

  /// SOLUTION -- Problem 1
  //
  // Number of "vertices" per instance. This is the number of vertices
  // used in a rendering pass (per instance). Geometry shader
  // gs_main_1 will render two triangles from each pair of
  // "vertices". The word "vertices" is in quotes to indicate that we
  // are talking about what OpenGL calls a vertex.
  //
  int n_vertices = -1;  // Set to -1 to provoke an error if we forget to set it.

  if ( opt_shader == SO_Set_1 )
    {
      // Use shaders vs_main_1 and gs_main_1
      //
      sp_set_1->use();

      /// SOLUTION -- Problem 1
      //
      //  Just render two vertices per instance, as required by the problem.
      //
      n_vertices = 2;
    }
  else
    {
      // Use shaders vs_main_2 and gs_main_2
      //
      sp_set_2->use();
      n_vertices = segments + 1;
    }


  // Number of sides of each cylinder.
  //
  const int sides = opt_sides;

  const float delta_tee = 1.0 / segments;
```

```
  const size_t n_instances = lis_pos1.size();

  // Bind buffer objects holding link and ball info to locations
  // specified in file hw06-shdr-links.cc
  //
  for ( size_t i=0; i<lis_pvects.size(); i++ )
    {
      GLuint bo = lis_bos[i];
      assert( lis_pvects[i]->size() == n_instances );
      glBindBuffer(GL_ARRAY_BUFFER,bo);
      if ( first_render )
        glBufferData
          (GL_ARRAY_BUFFER, lis_pvects[i]->size_chars(),
           lis_pvects[i]->data(), GL_STREAM_DRAW);
      glBindBufferBase(GL_SHADER_STORAGE_BUFFER,i+1,bo);
    }
  glBindBuffer(GL_ARRAY_BUFFER,0);

  glUniform4fv(1, 1, color_dim_gray ); // color_front
  glUniform4fv(2, 1, color_salmon ); // color_back
  glUniform2f(4, sides, 0 );
  glUniform1f(5, delta_tee );

  GLboolean l0, l1;
  glGetBooleanv(GL_LIGHT0,&l0);
  glGetBooleanv(GL_LIGHT1,&l1);
  int light_state = l0 | (l1<<1);
  glUniform1i(6, light_state);
  glUniform2i(3, opt_tryout1, opt_tryout2);

  /// SOLUTION -- Problem 1
  //
  // Use variable n_vertices to specifiy number of vertices in 3rd argument.
  // (See code above.)
  //
  glDrawArraysInstanced
    (GL_LINE_STRIP, 0, n_vertices, n_instances);

  sp_fixed->use();
}


void
World::frame_callback()
{
  // This routine called whenever window needs to be updated.

  frame_timer.phys_start();
  const double time_now = time_wall_fp();

  if ( !opt_pause || opt_single_frame || opt_single_time_step )
    {
      /// Advance simulation state.

      // Amount of time since the user saw the last frame.
      //
      const double wall_delta_t = time_now - last_frame_wall_time;

      // Compute amount by which to advance simulation state for this frame.
      //
      const double duration =
        opt_single_time_step ? opt_time_step_duration :
        opt_single_frame ? 1/30.0 :
        wall_delta_t;
```

```
      const double world_time_target = world_time + duration;
      const double wall_time_limit = time_now + 0.05;

      while ( world_time < world_time_target )
        {
          time_step_cpu(opt_time_step_duration);
          world_time += opt_time_step_duration;
          const double time_right_now = time_wall_fp();
          if ( time_right_now > wall_time_limit ) break;
        }

      // Reset these, just in case they were set.
      //
      opt_single_frame = opt_single_time_step = false;
    }

  frame_timer.phys_end();

  last_frame_wall_time = time_now;

  render();
}

int
main(int argv, char **argc)
{
  pOpenGL_Helper popengl_helper(argv,argc);
  World world(popengl_helper,argv,argc);

  glDisable(GL_DEBUG_OUTPUT);
  glDebugMessageControl(GL_DONT_CARE,GL_DONT_CARE,
                        GL_DEBUG_SEVERITY_NOTIFICATION,0,NULL,false);

  popengl_helper.rate_set(30);
  popengl_helper.display_cb_set(world.frame_callback_w,&world);
}
```

# LSU EE 4702-1 Homework 7 Solution Due: 1 December 2016

The solution has been checked into the repo. For a syntax highlighted version of the shader code see `http://www.ece.lsu.edu/koppel/gpup/2016/hw07-cuda-sol.cu.html`.

**Problem 0:** If necessary, follow the instructions on the `http://www.ece.lsu.edu/koppel/gpup/proc.html` page for account setup and programming homework work flow. For this assignment only edit files `hw07-cuda.cu` and `hw07.cc`. **Find a machine with GPUs of CC (compute capability) 3.0 or higher,** see the table on `http://www.ece.lsu.edu/koppel/gpup/sys-status.html`. Compile and run the homework code unmodified. It should initially show the spring from classroom demo `demo-cuda-04-acc-pat.cc`.

The physics for the spring can be performed using three different sets of code, the one in use is shown at the beginning of the last line of green text next to Physics. The physics code can be changed by pressing `a`. Physics code *CPU* uses the CPU and will be very slow. Physics code *CUDA-M1* and *CUDA-M2* use the GPU running code in file `hw07-cuda.cu`. Code *CUDA-M1* uses routine `time_step_intersect_1` to detect and resolve intersection of helix segments, and *CUDA-M2* uses routine `time_step_intersect_2`.

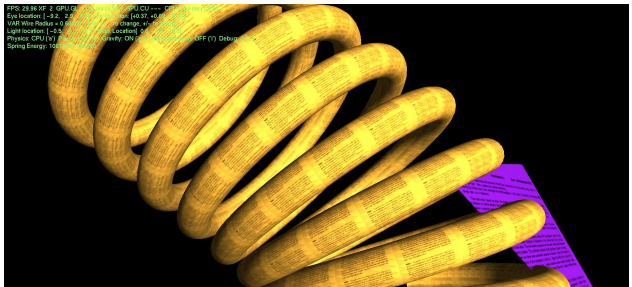
The green line starting with "Intersect" shows information about the execution of the chosen intersection routine. As discussed in class, it is this routine that will dominate performance. On that line "Bl Sz" is the number of blocks. That can be changed using the `Tab` key to find "Intersect Block Size" and then pressing `+` and `-` keys. Green text "N Blocks" shows the number of blocks. The number of blocks is determined by variable "Seg / Block" (`intersect_seg_per_block`). Green text "Bl/SM" shows the number of blocks per SM in two ways. The first value is the number of blocks running the intersect routine that *can* fit on an SM. The second number is how many are assumed running based on the number of blocks and the number of SMs. If the second number is lower than there are not enough blocks. Green text "Wp/SM" shows the number of warps per SM. That's based on the block size and the number of blocks per SM. Finally, "Bl Time" shows the number of cycles it takes to run one block. (The top green line shows timing per frame.)

On the last green line, "Shared Mem" shows how shared memory is being used to buffer the "b" segments in the intersection routine. "NONE" means that shared memory is not being used, "SYNC EXPER" is something to be used in Problem 2, "ONE ITER" means that shared memory is being loaded with "b" segments every `j` iteration (see the intersect routines), and "MULT ITERS" means that shared memory is loaded with enough data for several `j` iterations—when Problem 3 is solved correctly.

The top green line shows performance. The important items to pay attention to are "GPU.CU" and "Steps / s". "GPU.CU" shows the time per frame executing CUDA code. Use this to evaluate the performance of the intersection routines. The last item on the line (you may need to widen the window), "Steps / s", shows how many physics time steps are performed each second. If computation is fast enough it will be about 6000, otherwise it will be lower. "FPS" is the frame rate in frames per second. "XF" shows the frame update interval, an ideal number is 1. A 2 indicates that the frame is updated at half the display's refresh rate. "GPU.GL" is the time the GPU spends executiong OpenGL code. "CPU GR" is the CPU time for graphics. "CPU PH" is the CPU time for physics.

Fun Stuff: Pressing `g` will toggle gravity. The free end of the spring can be grabbed by pressing `b` and it can be moved using the arrow keys.

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 free end of the spring around. (Since the motion is instantaneous, the spring will act like it was plucked.) Press `l` to move the light around and `e` to move the eye (which is what the arrow keys do when the program starts).

When using the arrow and other keys to move the eye, light, or ball using `Shift` will move by a 5× greater amount and using `Ctrl` will move by one $\frac{1}{5}$ the amount than the motion without either modifier.

Look at the comments in the file `hw07.cc` for documentation on other keys. For documentation see the CUDA C Programming Guide linked to the course `http://www.ece.lsu.edu/koppel/gpup/ref.html` page.

**Problem 1:** The goal of this assignment is to understand and improve the performance of the intersection routines. The performance of the intersection code is very sensitive to the block size (variable "Intersect Block Size") and to the number of a segments per block (variable "Seg / Block", called $m$ in our classroom analysis).

($a$) Indicate the model of GPU you were using (it's shown in the text printed when the program starts). Indicate the number of SMs (shown as MP) for that GPU.

The GPU is an NVIDIA K20c which has 13 MPs.

($b$) Adjust these two variables to obtain the best performance on each intersection routine. Do these with shared memory set to NONE. Provide the following information: The settings of those variables. The initial CUDA execution time ("GPU.CU") and the best one obtained.

Initial CUDA time: 290 ms. The best performance is 13.2 ms with a block size of 128 threads and $m = 8$. The tables below show the raw data. Column WP Occ shows the warp occupancy (number of warps per SM), something which performance is sensitive to. The maximum for a K20 (and other CC 3.X through CC 6.x devices) is 64 warps per SM. The first table shows the impact of block size with $m = 1$. With $m = 1$ we have the maximum number of blocks, which in general is good for warp occupancy, but for not when blocks are too small because we hit an SM's block limit (16) before the warp limit (64).

The amount of data read (in units of 16-byte elements) from global memory when accessing the "a" elements is $nB/16$ (where $n$ is the number of segments and $B$ is the block size), and the amount of data read from global memory when accessing the "b" elements is $n^2/m$ for $m \leq 16$. As we've noted in class larger $m$'s mean less data loaded because there are fewer blocks and we know that each block must load at least $n + m$ elements. That means $m = 1$ is the worst value as far as data is concerned. However, the number of blocks is $n/m$ and so smaller $m$'s are better for warp occupancy, up to a point. That is, once we've reached the maximum of 64 warps per SM there is no benefit of having more warps. From the first table we see that we've hit that maximum at a block size of 128, which is where performance peaks. Increasing the block size reduces the number of iterations in the $j$ loop, which increases the impact of the code outside the loop. Also, increasing the block size increases the amount of data loaded for the "a" segments. Both of those reasons might explain the drop in performance with a larger block size.

The second table shows the impact of changing $m$ at three different block sizes. For block sizes of 256 and 128 the best performance is the smallest value of $m$ for which warp occupancy is below 64. If $m$ is increased further then the benefit of less data transfer is overwhelmed by the penalty of lower occupancy. Notice that at a block size of 64 the maximum warp occupancy is 32 and so the performance of those configurations suffers.

Vary Block Size with m = 1:

| B | GPU.gl | WP Occ | j Iters |
|---|---|---|---|
| 32 | 40.9 ms | 16 | 16 |
| 64 | 18.6 ms | 32 | 8 |
| 128 | 15.1 ms | 64 | 4 |
| 256 | 17.7 ms | 64 | 2 |
| 512 | 37 ms | 64 | 1 |
| 1024 | 90 ms | 64 (32) | 1 |

| | -- B = 256 ----- | | -- B = 128 ----- | | -- B = 64 ----- | |
|---|---|---|---|---|---|---|
| m | GPU.gl | WP Occ | GPU.gl | WP Occ | GPU.gl | WP Occ |
| 1 | 17.7 ms | 64 | 14.6 ms | 64 | 18.6 ms | 32 |
| 2 | 14.2 ms | 64 | 14.0 ms | 64 | 20.0 ms | 32 |
| 4 | 14.8 ms | 64 | 13.2 ms | 39.4 | 19.8 ms | 19.7 |
| 8 | 13.6 ms | 39.4 | 19.8 ms | 19.7 | 72 ms | 9.8 |
| 16 | 20 ms | 19.7 | | | | |

($c$) Based on our classroom analysis there was less data communication with a larger value of $m$. Are your results consistent with that? If not, explain why.

Based on the discussion in the solution above, the results are consistent, performance improves with larger $m$. Until, that is, warp occupancy suffers, then performance drops.

**Problem 2:** Set CUDA-M1 to the optimal parameters found in the previous problem. Now, cycle through the different shared memory options. The "ONE ITER" option, which caches enough data for one iteration, should be the slowest. Is it because of the syncthreads?

Modify routine `time_step_intersect_1` so that when `hi.opt_sm_option` is set to value `SMO_sync_experiment` syncthreads is called in a way similar to when `hi.opt_sm_option` is set to `SMO_one_iteration`, but without actually loading or using shared memory. Do so in a way that will determine whether syncthreads is making "ONE ITER" take longer.

Describe what you found by doing the experiment.

The sync experiment version executed like "none" except that syncthreads was called twice. The performance was almost identical to "none" and so it is not syncthreads that was causing the slowdown.

**Problem 3:** Modify `time_step_intersect_1` so that when `hi.opt_sm_option` is set to value `SMO_multiple_iterations` array `pos_cache` is loaded with $B$ items, where $B$ is the block size (value of `blockDim.x`). This should be done inside the `j` loop when data is needed. That is, **don't** load the cache every iteration.

Compare the performance to NONE and ONE ITER.

A new variable `cache_idx_next` is used to keep track of which element of `pos_cache` each thread uses. The range of values is 0 through the block size minus 1. It is incremented each iteration by $d$ (`thd_per_a`):

```
float3 b_position =
  hi.opt_sm_option == SMO_one_iteration
  ? pos_cache[ b_idx_start ] :
  hi.opt_sm_option == SMO_multiple_iterations
  ? pos_cache[ cache_idx_next ]
  : m3( helix_position[j] );

cache_idx_next += thd_per_a;
```

When `cache_idx_next` exceeds the block size it's time to load new data into `pos_cache` and set `cache_idx_next` back to its initial value, `b_idx_start`. Variable `b_idx_next` is used to indicate which "b" segment to load, that's incremented by the block size each time `pos_cache` is reloaded:

```
else if ( hi.opt_sm_option == SMO_multiple_iterations )
  {
    if ( cache_idx_next >= blockDim.x )
      {
        __syncthreads();
        cache_idx_next = b_idx_start;
        pos_cache[ threadIdx.x ] = m3(helix_position[ b_idx_next ] );
        b_idx_next += blockDim.x;
        __syncthreads();
      }
  }
```

Variable **cache_idx_next** is initialized to an out-of-range value so that on the first iteration **pos_cache** is loaded. Variable **b_idx_next** is initialized to the first "b" segment to load:

```
int cache_idx_next = b_idx_start + blockDim.x;
int b_idx_next = threadIdx.x;

for ( int j=b_idx_start; j<hi.phys_helix_segments; j += thd_per_a )
  {
```

Here is an excerpt of the code in order:

```
const float3 a_position = m3(helix_position[a_idx]);

/// SOLUTION -- Problem 3
//
//  The next element of pos_cache to use. Its value should be
//  between 0 and blockDim.x (block size) -1. It is intentionally
//  initialized to an out-of-range value so that the cache will be
//  loaded.
//
int cache_idx_next = b_idx_start + blockDim.x;
//
//  The next element of helix to put into the cache.
//
int b_idx_next = threadIdx.x;

for ( int j=b_idx_start; j<hi.phys_helix_segments; j += thd_per_a )
  {
    if ( hi.opt_sm_option == SMO_one_iteration )
      {
        __syncthreads();
        if ( threadIdx.x < thd_per_a )
          pos_cache[threadIdx.x] =
            m3(helix_position[ j - b_idx_start + threadIdx.x ] );
        __syncthreads();
      }
    else if ( hi.opt_sm_option == SMO_sync_experiment )
      {
        /// SOLUTION -- Problem 2
        //
        //  See if just executing __syncthreads slows things down.
        //
        __syncthreads();
        __syncthreads();
      }
    else if ( hi.opt_sm_option == SMO_multiple_iterations )
      {
        /// SOLUTION -- Problem 3
        //
        // If the next pos_cache element to use is out of range, then
        // load pos_cache with a new batch of data.
        //
        if ( cache_idx_next >= blockDim.x )
```

```
      {
        __syncthreads();
        cache_idx_next = b_idx_start;
        pos_cache[ threadIdx.x ] = m3(helix_position[ b_idx_next ] );
        b_idx_next += blockDim.x;
        __syncthreads();
      }
  }

/// SOLUTION -- Problem 3
//
//  For the multiple iteration case the index to pos_cache
//  is a function of j.
//
float3 b_position =
  hi.opt_sm_option == SMO_one_iteration
  ? pos_cache[ b_idx_start ] :
  hi.opt_sm_option == SMO_multiple_iterations
  ? pos_cache[ cache_idx_next ]
  : m3( helix_position[j] );

/// SOLUTION -- Problem 3
//
cache_idx_next += thd_per_a;

pVect ab = mv(a_position,b_position);

// Skip if segment is too close.
if ( abs(a_idx-j) < min_idx_dist ) continue;

// Skip if no chance of intersection.
if ( mag_sq(ab) >= four_wire_radius_sq ) continue;

// Compute intersection force based on bounding sphere, an
// admittedly crude approximation.
//
pNorm dist = mn(ab);
const float pen = 2 * hi.wire_radius - dist.magnitude;
float3 f = pen * hi.opt_spring_constant * dist;

// Add force to shared variable. This is time consuming
// (especially in CC 3.x and older GPUs) but done
// infrequently. (A segment can normally only intersect a a few
// other segments.)
//
atomicAdd(&force[a_local_idx].x,f.x);
atomicAdd(&force[a_local_idx].y,f.y);
atomicAdd(&force[a_local_idx].z,f.z);
//
// Optimization Note: Could acquire a lock and then update
// all three components.
```

```
}
```

```
/// LSU EE 4702-1 (Fall 2016), GPU Programming
//
 /// Homework 7 -- SOLUTION
 //
 //  See http://www.ece.lsu.edu/koppel/gpup/2016/hw07_sol.pdf

 /// Use this file for your solution.

#include "cuda-coord.cu"
#include "hw07.cuh"
#include <gp/cuda-util-kernel.h>

// Physical State Variables
//
__constant__ float4 *helix_position;
__constant__ float3 *helix_velocity;     // Note: float4 would be faster.
__constant__ float4 *helix_orientation;
__constant__ float3 *helix_omega;        // Note: float4 would be faster.

__constant__ Timing_Data *timing_data;   // Measure execution time of intersect.
__constant__ Helix_Info hi;  // Scalar Constants

__global__ void
time_step_intersect_1()
{
  /// Homework 7  SOLUTION IN THIS ROUTINE

  // Find intersections of one helix segment with some other
  // segments. Each block handles several "a" segments, the threads in
  // the block check for intersection with other segments, called "b"
  // segments.

  __shared__ clock_t time_start;
  if ( !threadIdx.x ) time_start = clock64();


  // Note: The size of the helix_position array is hi.phys_helix_segments.

  // Compute how many "a" elements will be handled by each block.
  //
  const int a_per_block = hi.phys_helix_segments / gridDim.x;

  // Compute how many threads handle each "a" element.
  //
  const int thd_per_a = blockDim.x / a_per_block;

  // Compute the smallest "a" element index that this block will handle.
  //
  const int a_idx_block = blockIdx.x * a_per_block;

  /// Assignment of "a" and "b" Values to Threads
  //
  //  The table below is an example of how this routine
  //  assigns "a" and "b" elements to threads.  The table
  //  is based upon the following values:
  //
  //    blockDim = 8,       blockIdx = 4,     hi.phys_helix_segments = 1024
  //    m:a_per_block = 4,  d:thd_per_a = 2,  a_idx_block = 16
  //
  // tIx     al   a      b --->
  //   0     0    16     0  2  4 ... 1022
  //   1     1    17     0  2  4 ... 1022
  //   2     2    18     0  2  4 ... 1022
  //   3     3    19     0  2  4 ... 1022
```

```
//   4     0    16     1  3  5 ... 1023
//   5     1    17     1  3  5 ... 1023
//   6     2    18     1  3  5 ... 1023
//   7     3    19     1  3  5 ... 1023
//   |     |     |     |
//   |     |     |     |
//   |     |     |     |--------> b_idx_start
//   |     |     |--------------> a_idx
//   |     |--------------------> a_local_idx
//   |------------------------> threadIdx.x

// Compute a_idx and b_idx_start to realize ordering above.
//
const int a_local_idx = threadIdx.x % a_per_block;
const int a_idx = a_idx_block + a_local_idx;
const int b_idx_start = threadIdx.x / a_per_block;

const int min_idx_dist = 0.999f + hi.wire_radius / hi.helix_seg_hlength;
const float four_wire_radius_sq = 4 * hi.wire_radius * hi.wire_radius;

// Declare dynamically allocated shared memory. Will be split
// between array for forces, force, and position cache, pos_cache.
//
extern __shared__ float3 shared[];

pVect* const force = shared;
float3* const pos_cache = &shared[a_per_block];

if ( threadIdx.x < a_per_block ) force[threadIdx.x] = mv(0,0,0);

__syncthreads();

const float3 a_position = m3(helix_position[a_idx]);

/// SOLUTION -- Problem 3
//
//  The next element of pos_cache to use. Its value should be
//  between 0 and blockDim.x (block size) -1. It is intentionally
//  initialized to an out-of-range value so that the cache will be
//  loaded.
//
int cache_idx_next = b_idx_start + blockDim.x;
//
//  The next element of helix to put into the cache.
//
int b_idx_next = threadIdx.x;

for ( int j=b_idx_start; j<hi.phys_helix_segments; j += thd_per_a )
  {
    if ( hi.opt_sm_option == SMO_one_iteration )
      {
        __syncthreads();
        if ( threadIdx.x < thd_per_a )
          pos_cache[threadIdx.x] =
            m3(helix_position[ j - b_idx_start + threadIdx.x ] );
        __syncthreads();
      }
    else if ( hi.opt_sm_option == SMO_sync_experiment )
      {
        /// SOLUTION -- Problem 2
        //
        //  See if just executing __syncthreads slows things down.
        //
        __syncthreads();
        __syncthreads();
```

```
      }
    else if ( hi.opt_sm_option == SMO_multiple_iterations )
      {
        /// SOLUTION -- Problem 3
        //
        // If the next pos_cache element to use is out of range, then
        // load pos_cache with a new batch of data.
        //
        if ( cache_idx_next >= blockDim.x )
          {
            __syncthreads();
            cache_idx_next = b_idx_start;
            pos_cache[ threadIdx.x ] = m3(helix_position[ b_idx_next ] );
            b_idx_next += blockDim.x;
            __syncthreads();
          }
      }

    /// SOLUTION -- Problem 3
    //
    //  For the multiple-iteration case retrieve pos_cache[
    //  cache_idx_next ] (the value of cache_idx_next changes from
    //  iteration to iteration) rather than pos_cache[ b_idx_start ]
    //  (the value of b_idx_start is the same every iteration).
    //
    float3 b_position =
      hi.opt_sm_option == SMO_one_iteration
      ? pos_cache[ b_idx_start ] :
      hi.opt_sm_option == SMO_multiple_iterations
      ? pos_cache[ cache_idx_next ]
      : m3( helix_position[j] );

    /// SOLUTION -- Problem 3
    //
    cache_idx_next += thd_per_a;

    pVect ab = mv(a_position,b_position);

    // Skip if segment is too close.
    if ( abs(a_idx-j) < min_idx_dist ) continue;

    // Skip if no chance of intersection.
    if ( mag_sq(ab) >= four_wire_radius_sq ) continue;

    // Compute intersection force based on bounding sphere, an
    // admittedly crude approximation.
    //
    pNorm dist = mn(ab);
    const float pen = 2 * hi.wire_radius - dist.magnitude;
    float3 f = pen * hi.opt_spring_constant * dist;

    // Add force to shared variable. This is time consuming
    // (especially in CC 3.x and older GPUs) but done
    // infrequently. (A segment can normally only intersect a a few
    // other segments.)
    //
    atomicAdd(&force[a_local_idx].x,f.x);
    atomicAdd(&force[a_local_idx].y,f.y);
    atomicAdd(&force[a_local_idx].z,f.z);
    //
    // Optimization Note: Could acquire a lock and then update
    // all three components.
  }

// Wait for all threads to finish.
```

```
  __syncthreads();

  // Leave it to thread 0 to update velocity.
  if ( threadIdx.x >= a_per_block ) return;

  // Update velocity and write it.
  //
  float3 velocity = helix_velocity[a_idx];
  velocity -= hi.delta_t_mass_inv * force[a_local_idx];
  if ( hi.opt_end_fixed && a_idx + 1 == hi.phys_helix_segments )
    velocity = mv(0,0,0);
  helix_velocity[a_idx] = velocity;

  if ( !threadIdx.x )
    {
      timing_data[blockIdx.x].intersect_time += clock64() - time_start;
      timing_data[blockIdx.x].intersect_count++;
    }
}

__global__ void
time_step_intersect_2()
{
  /// DO NOT MODIFY THIS ROUTINE.

  // Find intersections of one helix segment with some other
  // segments. Each block handles several "a" segments, the threads in the
  // block check for intersection with other segments, called "b"
  // segments.

  __shared__ clock_t time_start;
  if ( !threadIdx.x ) time_start = clock64();

  // Note: The size of the helix_position array is hi.phys_helix_segments.

  // Compute how many "a" elements will be handled by each block.
  //
  const int a_per_block = hi.phys_helix_segments / gridDim.x;

  // Compute how many threads handle each "a" element.
  //
  const int thd_per_a = blockDim.x / a_per_block;

  // Compute the smallest "a" element index that this block will handle.
  //
  const int a_idx_block = blockIdx.x * a_per_block;

  /// Assignment of "a" and "b" Values to Threads
  //
  //  The table below is an example of how this routine
  //  assigns "a" and "b" elements to threads.  The table
  //  is based upon the following values:
  //
  //    blockDim = 8,     blockIdx = 4,   hi.phys_helix_segments = 1024
  //    a_per_block = 4,  thd_per_a = 2,  a_idx_block = 16
  //
  // tIx     al   a      b --->
  //   0     0    16     0  2  4 ...
  //   1     0    16     1  3  5
  //   2     1    17     0  2  4
  //   3     1    17     1  3  5
  //   4     2    18     0  2  4
  //   5     2    18     1  3  5
  //   6     3    19     0  2  4
  //   7     3    19     1  3  5
```

```
//   |     |     |     |
//   |     |     |     |
//   |     |     |     |--------> b_idx_start
//   |     |     |--------------> a_idx
//   |     |--------------------> a_local_idx
//   |--------------------------> threadIdx.x

// Compute a_idx and b_idx_start to realize ordering above.
//
const int a_local_idx = threadIdx.x / thd_per_a;
const int a_idx = a_idx_block + a_local_idx;
const int b_idx_start = threadIdx.x % thd_per_a;

const float3 a_position = m3(helix_position[a_idx]);
const int min_idx_dist = 0.999f + hi.wire_radius / hi.helix_seg_hlength;
const float four_wire_radius_sq = 4 * hi.wire_radius * hi.wire_radius;

// Declare dynamically allocated shared memory. Will be split
// between array for forces, force, and position cache, pos_cache.
//
extern __shared__ float3 shared[];

pVect* const force = shared;
if ( threadIdx.x < a_per_block ) force[threadIdx.x] = mv(0,0,0);

// Wait for thread 0 to initialize force.
__syncthreads();

const bool use_shared =
  hi.opt_sm_option == SMO_one_iteration
  || hi.opt_sm_option == SMO_multiple_iterations;

float3* const pos_cache = &shared[a_per_block];

int cache_elt_remaining = use_shared ? 0 : -1;
int cache_num_refills = 0;
int cache_idx_start = 0;

for ( int j=b_idx_start; j<hi.phys_helix_segments; j += thd_per_a )
  {
    if ( hi.opt_sm_option == SMO_one_iteration )
      {
        /// SOLUTION
        cache_idx_start = j - b_idx_start;
        __syncthreads();
        if ( threadIdx.x < thd_per_a )
          pos_cache[threadIdx.x] = m3(helix_position[j]);
        __syncthreads();
      }
    else if ( hi.opt_sm_option == SMO_multiple_iterations
              && cache_elt_remaining == 0 )
      {
        __syncthreads();
        cache_idx_start = cache_num_refills * blockDim.x;
        pos_cache[threadIdx.x] =
          m3(helix_position[ cache_idx_start + threadIdx.x ] );
        __syncthreads();
        cache_num_refills++;
        cache_elt_remaining = blockDim.x;
      }
    cache_elt_remaining -= thd_per_a;

    float3 b_position =
      use_shared ? pos_cache[j-cache_idx_start] : m3(helix_position[j]);
```

```
      pVect ab = mv(a_position,b_position);

      // Skip if segment is too close.
      if ( abs(a_idx-j) < min_idx_dist ) continue;

      // Skip if no chance of intersection.
      if ( mag_sq(ab) >= four_wire_radius_sq ) continue;

      // Compute intersection force based on bounding sphere, an
      // admittedly crude approximation.
      //
      pNorm dist = mn(ab);
      const float pen = 2 * hi.wire_radius - dist.magnitude;
      float3 f = pen * hi.opt_spring_constant * dist;

      // Add force to shared variable. This is time consuming but
      // done infrequently. (A segment can normally only intersect a
      // a few other segments.)
      //
      atomicAdd(&force[a_local_idx].x,f.x);
      atomicAdd(&force[a_local_idx].y,f.y);
      atomicAdd(&force[a_local_idx].z,f.z);
      //
      // Optimization Note: Could acquire a lock and then update
      // all three components.
    }

  // Wait for all threads to finish.
  __syncthreads();

  // Leave it to thread 0 to update velocity.
  if ( threadIdx.x >= a_per_block ) return;

  {
    // Re-compute a_idx so that first a_per_block threads can write
    // velocities.

    const int a_local_idx = threadIdx.x;
    const int a_idx = a_idx_block + a_local_idx;

    // Update velocity and write it.
    //
    float3 velocity = helix_velocity[a_idx];
    velocity -= hi.delta_t_mass_inv * force[a_local_idx];
    if ( hi.opt_end_fixed && a_idx + 1 == hi.phys_helix_segments )
      velocity = mv(0,0,0);
    helix_velocity[a_idx] = velocity;

    if ( !threadIdx.x )
      {
        timing_data[blockIdx.x].intersect_time += clock64() - time_start;
        timing_data[blockIdx.x].intersect_count++;
      }
  }
}


__global__ void time_step();
__global__ void time_step_intersect_1();
__global__ void time_step_intersect_2();
__global__ void time_step_update_pos();


__host__ cudaError_t
```

```
cuda_setup(GPU_Info *gpu_info)
{
  // Pass the device address to host code. (See gp/cuda-util-kernel.h ).
  CU_SYM(helix_position);
  CU_SYM(helix_velocity);
  CU_SYM(helix_orientation);
  CU_SYM(helix_omega);
  CU_SYM(timing_data);
  CU_SYM(hi);

  // Return attributes of CUDA functions. The code needs the
  // maximum number of threads.

  cudaError_t e1 = cudaSuccess;

  gpu_info->GET_INFO(time_step);
  gpu_info->GET_INFO(time_step_intersect_1);
  gpu_info->GET_INFO(time_step_intersect_2);
  gpu_info->GET_INFO(time_step_update_pos);


  return e1;
}

__host__ void time_step_launch(int grid_size, int block_size)
{
  time_step<<<grid_size,block_size>>>();
}

__device__ void
helix_apply_force_at
(float3 position, float3& force, float3& torque,
 float3 force_pos, pVect dir, float magnitude);


__global__ void
time_step()
{
  int tid = threadIdx.x + blockIdx.x * blockDim.x;
  // Use tid for helix segment number.

  if ( tid + 1 > hi.phys_helix_segments ) return;

  // The position of segment 0 is fixed, so don't evolve it.
  if ( tid == 0 ) return;

  pVect vZero = mv(0,0,0);
  pVect gravity_force = hi.helix_seg_mass_inv * hi.gravity_accel;

  pQuat c_orientation = cq(helix_orientation[tid]);
  float3 c_position = m3(helix_position[tid]);

  pMatrix3x3 c_rot;
  // Initialize c_rot to a rotation matrix based on quaternion c_orientation.
  pMatrix_set_rotation(c_rot,c_orientation);

  float3 c_u = c_rot * mv(0,0,1);  // mv: Make Vector.
  float3 c_v = c_rot * mv(0,1,0);
  float3 c_ctr_to_right_dir = c_rot * mv(1,0,0);
  pVect c_ctr_to_right = hi.helix_seg_hlength * c_ctr_to_right_dir;
  float3 c_pos_right = c_position + c_ctr_to_right;
  float3 c_pos_left = c_position - c_ctr_to_right;

  float3 force = hi.opt_gravity ? gravity_force : vZero;
```

```
  float3 torque = vZero;

  const int pieces = 3;
  const float delta_theta = 2 * M_PI / pieces;

  /// Compute forces due to right neighbor.
  //
  if ( tid + 1 < hi.phys_helix_segments )
    {
      pQuat r_orientation = cq(helix_orientation[tid+1]);
      float3 r_position = m3(helix_position[tid+1]);
      pMatrix3x3 r_rot;
      pMatrix_set_rotation(r_rot,r_orientation);
      float3 r_u = r_rot * mv(0,0,1);
      float3 r_v = r_rot * mv(0,1,0);
      float3 r_ctr_to_right_dir = r_rot * mv(1,0,0);
      pVect r_ctr_to_right = hi.helix_seg_hlength * r_ctr_to_right_dir;
      float3 r_pos_left = r_position - r_ctr_to_right;

      pQuat cn_rot_q = c_orientation * hi.helix_rn_trans;
      pMatrix3x3 cn_rot;
      pMatrix_set_rotation(cn_rot,cn_rot_q);
      pVect n_ru = cn_rot * mv(0,0,1);
      pVect n_rv = cn_rot * mv(0,1,0);

      for ( int j=0; j<pieces; j++ )
        {
          const float theta = delta_theta * j;
          pCoor c_pt = c_pos_right + cosf(theta) * n_ru + sinf(theta) * n_rv;
          pCoor r_pt = r_pos_left + cosf(theta) * r_u + sinf(theta) * r_v;
          pNorm dist = mn(c_pt,r_pt);
          const float force_mag = dist.magnitude * hi.opt_spring_constant;
          helix_apply_force_at(c_position,force,torque,c_pt,dist.v,force_mag);
        }
    }

  /// Compute forces due to left neighbor.
  //
  if ( tid > 0 )
    {
      pQuat l_orientation = cq(helix_orientation[tid-1]);
      float3 l_position = m3(helix_position[tid-1]);
      pMatrix3x3 l_rot;
      pMatrix_set_rotation(l_rot,l_orientation);
      float3 l_u = l_rot * mv(0,0,1);
      float3 l_v = l_rot * mv(0,1,0);
      float3 l_ctr_to_right_dir = l_rot * mv(1,0,0);
      pVect l_ctr_to_right = hi.helix_seg_hlength * l_ctr_to_right_dir;
      float3 l_pos_right = l_position + l_ctr_to_right;

      pQuat ln_rot_q = l_orientation * hi.helix_rn_trans;
      pMatrix3x3 ln_rot;
      pMatrix_set_rotation(ln_rot,ln_rot_q);
      pVect n_cu = ln_rot * mv(0,0,1);
      pVect n_cv = ln_rot * mv(0,1,0);

      for ( int j=0; j<pieces; j++ )
        {
          const float theta = delta_theta * j;
          pCoor c_pt = c_pos_left + cosf(theta) * c_u + sinf(theta) * c_v;
          pCoor l_pt = l_pos_right + cosf(theta) * n_cu + sinf(theta) * n_cv;
          pNorm dist = mn(c_pt,l_pt);
          const float force_mag = dist.magnitude * hi.opt_spring_constant;
          helix_apply_force_at(c_position,force,torque,c_pt,dist.v,force_mag);
        }
```

```
    }

  float3 velocity = helix_velocity[tid];
  velocity *= 0.99999f;
  float3 omega = helix_omega[tid];
  omega *= 0.99999f;
  velocity += hi.delta_t_mass_inv * force;
  const float torque_axial_mag = dot( torque, c_ctr_to_right_dir );
  pVect torque_axial = torque_axial_mag * c_ctr_to_right_dir;
  pVect do_axial = hi.delta_t_ma_axis * torque_axial;
  pVect torque_other = torque - torque_axial;
  pVect do_other = hi.delta_t_ma_perp_axis * torque_other;
  omega += do_axial + do_other;

  // Update velocity and omega. Don't update position or orientation
  // because we don't want threads in this kernel to accidentally read
  // the updated values.

  helix_omega[tid] = omega;
  helix_velocity[tid] = velocity;
}


__device__ void
helix_apply_force_at
(float3 position, float3& force, float3& torque,
 float3 force_pos, pVect dir, float magnitude)
{
  // Update force and torque of segment for a force acting on FORCE_POS
  // pointing in direction DIR of magnitude MAGNITUDE.
  //
  force += magnitude * dir;
  pVect arm = mv(position,force_pos);
  pVect axis = cross( arm, dir );
  pVect amt = magnitude * axis;
  torque += amt;
}



__host__ void
time_step_intersect_launch
(int grid_size, int block_size, int version, int dynamic_sm_amt)
{
  switch ( version ) {
    case 1: time_step_intersect_1<<<grid_size,block_size,dynamic_sm_amt>>>();
    case 2: time_step_intersect_2<<<grid_size,block_size,dynamic_sm_amt>>>();
  }
}

__global__ void
time_step_update_pos()
{
  // Update position and orientation of spring segments.

  // Use tid for helix segment number.
  int tid = threadIdx.x + blockIdx.x * blockDim.x;

  // Skip out-of-range segments.
  if ( tid >= hi.phys_helix_segments ) return;
  if ( tid == 0 ) return;

  // Update Orientation
  //
```

```
  pQuat orientation = cq(helix_orientation[tid]);
  float3 omega = helix_omega[tid];
  pNorm axis = mn(omega);
  helix_orientation[tid] =
    c4( quat_normalize
    ( quat_mult ( mq( axis, hi.delta_t * axis.magnitude ), orientation)));

  // Return if at last segment and it is fixed. Note that even
  // if the segment's position is fixed, it can still rotate.
  //
  if ( hi.opt_end_fixed && tid + 1 == hi.phys_helix_segments ) return;

  // Update Velocity
  //
  float3 position = m3(helix_position[tid]);
  float3 velocity = helix_velocity[tid];
  helix_position[tid] = m4(position + hi.delta_t * velocity,1);
}

__host__ void
time_step_update_pos_launch
(int grid_size, int block_size)
{
  time_step_update_pos<<<grid_size,block_size>>>();
}
```

# 27 Fall 2015 Solutions

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

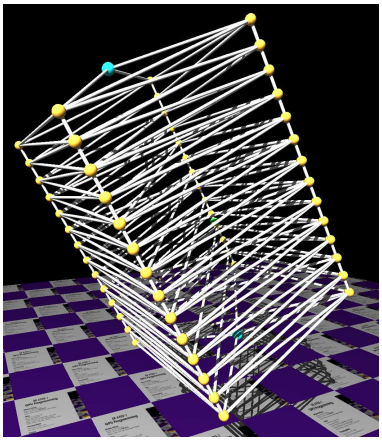
*Some of the effort for this homework assignment is in learning to use the various pieces of software, such as a text editor. Those less familiar with Linux software development procedures might seek out a more knowledgeable classmate to minimize frustration and wasted time.*



**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 $\delta$ 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 $\delta$.)

As one can see by examining the code, the head ball is placed $\delta$ above the start of the truss and the tail ball is placed $\delta$ 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 $\delta$, 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 $\overrightarrow{A_S P_s}$ 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.)*

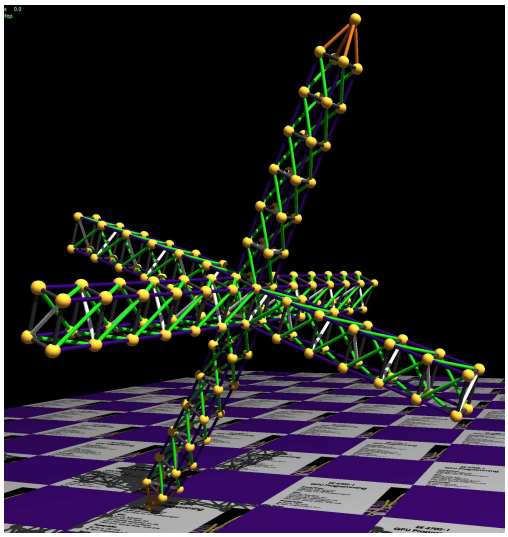
*A common error was to make the change in velocity proportional to the distance from the head ball to the ball (rather than the distance of the ball to the axis).*

```
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();
```

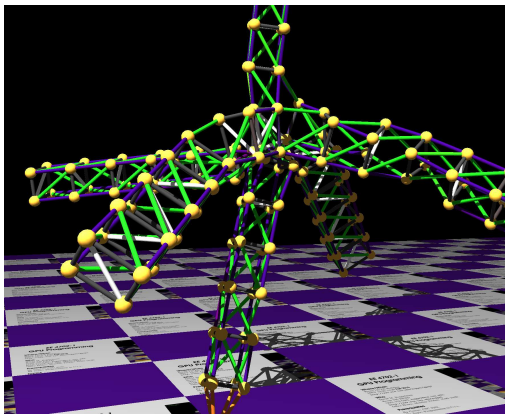
```
  for ( int j=0; j<sides; j++ )
    links += new Link( tail_ball, truss_info.balls[bsize-sides+j],
                       color_chocolate );

  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;
}
```

# 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 $\Delta V$ 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;
```

```
/// LSU EE 4702-1 (Fall 2015), GPU Programming
//
 /// Homework 2 -- SOLUTION
 //

 /// Instructions
 //
 //  Assignment: http://www.ece.lsu.edu/koppel/gpup/2015/hw02.pdf
 //  Solution: http://www.ece.lsu.edu/koppel/gpup/2015/hw02_sol.pdf

/// Purpose
//
//   Demonstrate simulation of point masses connected by springs.

/// What Code Does

// Simulates balls connected by springs over a platform. Balls and
// springs can be initialized in different arrangements (called
// scenes). Currently scene 1 is a simple string of beads, and scenes
// 2, 3, and 4 are trusses. The platform consists of tiles, some are
// purple-tinted mirrors (showing a reflection of the ball), the
// others show the course syllabus.


///  Keyboard Commands
 //
 /// Object (Eye, Light, Ball) Location or Push
 //   Arrows, Page Up, Page Down
 //        Move object or push ball, depending on mode.
 //        With shift key pressed, motion is 5x faster.
 //   'e': Move eye.
 //   'l': Move light.
 //   'b': Move head (first) ball. (Change position but not velocity.)
 //   'B': Push head ball. (Add velocity.)
 //
 /// Eye Direction
 //   Home, End, Delete, Insert
 //   Turn the eye direction.
 //   Home should rotate eye direction up, End should rotate eye
 //   down, Delete should rotate eye left, Insert should rotate eye
 //   right.  The eye direction vector is displayed in the upper left.

 /// Simulation Options
 //  (Also see variables below.)
 //
 //  'v'    Cycle through display of links/balls (SKEL) and triangles (SKIN).
 //  'w'    Twirl balls around axis formed by head and tail.
 //  '1'    Set up scene 1.
 //  '2'    Set up scene 2.
 //  '3'    Set up scene 3.
 //  'p'    Pause simulation. (Press again to resume.)
 //  ' '    (Space bar.) Advance simulation by 1/30 second.
 //  'S- '  (Shift-space bar.) Advance simulation by one time step.
 //  'h'    Freeze position of first (head) ball. (Press again to release.)
 //  't'    Freeze position of last (tail) ball. (Press again to release.)
 //  's'    Stop balls.
 //  'g'    Turn gravity on and off.
 //  'y'    Toggle value of opt_tryout1. Intended for experiments and debugging.
 //  'Y'    Toggle value of opt_tryout2. Intended for experiments and debugging.
```

```
//  'F11'  Change size of green text (in upper left).
//  'F12'  Write screenshot to file.

/// Variables
//   Selected program variables can be modified using the keyboard.
//   Use "Tab" to cycle through the variable to be modified, the
//   name of the variable is displayed next to "VAR" on the bottom
//   line of green text.

//  'Tab' Cycle to next variable.
//  '`'   Cycle to previous variable.
//  '+'   Increase variable value.
//  '-'   Decrease variable value.
//
//  VAR Spring Constant - Set spring constant.
//  VAR Time Step Duration - Set physics time step.
//  VAR Air Resistance - Set air resistance.
//  VAR Light Intensity - The light intensity.
//  VAR Gravity - Gravitational acceleration. (Turn on/off using 'g'.)


#define GL_GLEXT_PROTOTYPES
#define GLX_GLXEXT_PROTOTYPES

#include <GL/gl.h>
#include <GL/glext.h>
#include <GL/glx.h>
#include <GL/glxext.h>
#include <GL/glu.h>
#include <GL/freeglut.h>

#include <gp/util.h>
#include <gp/glextfuncs.h>
#include <gp/coord.h>
#include <gp/shader.h>
#include <gp/pstring.h>
#include <gp/misc.h>
#include <gp/gl-buffer.h>
#include <gp/texture-util.h>
#include <gp/colors.h>

#include "util-containers.h"
#include "shapes.h"


///
/// Main Data Structures
///
//
// class World: All data about scene.


class World;


// Object Holding Ball State
//
class Ball {
public:
  Ball():velocity(pVect(0,0,0)),locked(false),
         color(color_lsu_spirit_gold),contact(false){};
  pCoor position;
  pVect velocity;
```

```
  float mass;
  float mass_min; // Mass below which simulation is unstable.
  float radius;

  bool locked;

  pVect force;
  pColor color;
  bool contact;                 // When true, ball rendered in gray.
  float spring_constant_sum;    // Used to compute minimum mass.

  void push(pVect amt);
  void translate(pVect amt);
  void stop();
  void freeze();
};

class Link {
public:
  Link(Ball *b1, Ball *b2):ball1(b1),ball2(b2),
     distance_relaxed(pDistance(b1->position,b2->position)), snapped(false),
     natural_color(color_lsu_spirit_purple),color(color_lsu_spirit_purple){}
  Link(Ball *b1, Ball *b2, pColor colorp):ball1(b1),ball2(b2),
     distance_relaxed(pDistance(b1->position,b2->position)), snapped(false),
     natural_color(colorp),color(colorp){}
  Ball* const ball1;
  Ball* const ball2;
  float distance_relaxed;
  bool snapped;
  pColor natural_color;
  pColor color;
};

// Declare containers and iterators for Balls and Links.
// (See util_container.h.)
//
typedef pVectorI<Link> Links;
typedef pVectorI_Iter<Link> LIter;
typedef pVectorI<Ball> Balls;
typedef pVectorI_Iter<Ball> BIter;

 /// SOLUTION -- Problem 2

struct Strip {
  Strip(Balls& b, pColor colour):balls(b),color(colour){}
  Strip(pColor colour):color(colour){}
  Balls balls;
  pColor color;
};

typedef pVectorI<Strip> Strips;
typedef pVectorI_Iter<Strip> SIter;

 /// Homework 2 All Problems
//
//   Use this class to define variables and member functions.
//   Don't modify hw02-graphics.cc.
//
class My_Piece_Of_The_World {
public:

  /// SOLUTION -- Problem 1
  //
```

```
  float link_stressed_thd;  // Amount of stretch to be considered stressed.
  float link_snap_thd;      // Amount of stretch needed to snap link.

  pColor color_stressed;

  /// SOLUTION -- Problem 2

  Strips strips;

};


struct Truss_Info {

  // See make_truss for a description of what the members are for.

  // Inputs
  PStack<pCoor> base_coors;  // Coordinates of first set of balls.
  pVect unit_length;
  int num_units;

  // Output
  Balls balls;
  Links links;

  /// SOLUTION -- Problem 2
  //
  Strips strips;
};

#include "hw02-graphics.cc"


void
World::init()
{
  mp.color_stressed = pColor(1,0,0); // Red, what else?

  /// SOLUTION -- Problem 1
  //
  // Define thresholds at which a link will be considered stressed and
  // will snap.
  mp.link_stressed_thd = 1.001;
  mp.link_snap_thd = 1.01;

  chain_length = 14;

  opt_time_step_duration = 0.0003;
  variable_control.insert(opt_time_step_duration,"Time Step Duration");

  distance_relaxed = 15.0 / chain_length;
  opt_spring_constant = 15000;
  variable_control.insert(opt_spring_constant,"Spring Constant");

  opt_gravity_accel = 9.8;
  opt_gravity = true;
  gravity_accel = pVect(0,-opt_gravity_accel,0);
  variable_control.insert(opt_gravity_accel,"Gravity");

  opt_air_resistance = 0.04;
  variable_control.insert(opt_air_resistance,"Air Resistance");

  world_time = 0;
  time_step_count = 0;
  last_frame_wall_time = time_wall_fp();
```

```
  frame_timer.work_unit_set("Steps / s");

  ball_eye = NULL;
  opt_view = 3;
  opt_ride = false;

  init_graphics();

  ball_setup_1();
  lock_update();
}

Ball*
World::make_marker(pCoor position, pColor color)
{
  Ball* const ball = new Ball;
  ball->position = position;
  ball->locked = true;
  ball->velocity = pVect(0,0,0);
  ball->radius = 0.2;
  ball->mass = 0;
  ball->contact = false;
  ball->color = color;
  return ball;
}

void
World::lock_update()
{
  // This routine called when options like opt_head_lock might have
  // changed.

  // Update locked status.
  //
  if ( head_ball ) head_ball->locked = opt_head_lock;
  if ( tail_ball ) tail_ball->locked = opt_tail_lock;

  // Re-compute minimum mass needed for stability.
  //
  for ( BIter ball(balls); ball; ) ball->spring_constant_sum = 0;
  const double dtis = pow( opt_time_step_duration, 2 );
  for ( LIter link(links); link; )
    {
      Ball* const b1 = link->ball1;
      Ball* const b2 = link->ball2;
      b1->spring_constant_sum += opt_spring_constant;
      b2->spring_constant_sum += opt_spring_constant;
    }
  for ( BIter ball(balls); ball; )
    ball->mass_min = ball->spring_constant_sum * dtis;
}

void
World::balls_twirl()
{
  if ( !head_ball || !tail_ball ) return;

  pNorm axis(head_ball->position, tail_ball->position);

  for ( BIter ball(balls); ball; )
    {
      pVect b_to_top(ball->position,head_ball->position);
      const float dist_along_axis = dot(b_to_top,axis);
      const float lsq = b_to_top.mag_sq() - dist_along_axis * dist_along_axis;
```

```
      if ( lsq <= 1e-5 ) { ball->velocity = pVect(0,0,0); continue; }
      const float dist_to_axis = sqrt(lsq);
      pNorm rot_dir = cross(b_to_top,axis);
      ball->velocity += 10 * dist_to_axis * rot_dir;
    }
}

void
World::objects_erase()
{
  ball_eye = NULL;
  balls.erase();
  links.erase();

  /// SOLUTION -- Problem 2
  mp.strips.clear();
}

///
/// Physical Simulation Code
///

 /// Initialize Simulation
//

void
World::ball_setup_1()
{
  // Arrange and size balls to form a pendulum.

  pCoor first_pos(13.4,17.8,-9.2);
  pVect delta_pos = pVect(distance_relaxed,0,0);

  // Remove objects from the simulated objects lists, balls and links.
  // The delete operator is used on objects in the lists.
  //
  objects_erase();

  for ( int i=0; i<chain_length; i++ )
    {
      // Construct a new ball and add it to the simulated objects list (balls).
      //
      Ball* const ball = balls += new Ball;

      // Initialize position and other information.
      //
      ball->position = first_pos + i * delta_pos;
      ball->locked = false;
      ball->velocity = pVect(0,0,0);
      ball->radius = 0.3;
      ball->mass = 4/3.0 * M_PI * pow(ball->radius,3);
      ball->contact = false;

      // If it's not the first ball link it to the previous ball.
      if ( i > 0 ) links += new Link( ball, balls[i-1] );
    }

  // The balls pointed to by head_ball and tail_ball can be manipulated
  // using the user interface (by pressing 'h' or 't', for example).
  // Set these variables.
  //
  head_ball = balls[0];
  tail_ball = balls[balls-1];
```

```
  opt_head_lock = true;    // Head ball will be frozen in space.
  opt_tail_lock = false;   // Tail ball can move freely.
}


void
World::make_truss(Truss_Info *truss_info)
{
  /// Construct a truss based on members of truss_info.
  //

  //            <---- num_units (=9) ----------->
  //  j
  //  0         0---0---0---0---0---0---0---0---0     ^
  //            |   |   |   |   |   |   |   |   |     |
  //  1         0---0---0---0---0---0---0---0---0   num_sides (=3)
  //            |   |   |   |   |   |   |   |   |     |
  //  2         0---0---0---0---0---0---0---0---0     v
  //
  //      i ->  0   1   2   3   4   5   6   7   8
  //
  //  Note: Not all links are shown in the diagram above.

  /// Truss_Info Inputs
  //
  //  truss_info->num_units
  //    The number of sections in the truss (see diagram above).
  //
  //  truss_info->base_coors
  //    A list containing num_sides coordinates, the coordinates of
  //    balls at i=0 (see diagram above). (num_sides is the number of
  //    elements in this list.)  These coordinates should all be
  //    in the same plane.
  //
  //  truss_info->unit_length
  //    A vector pointing from the ball at (i,j) to the ball at (i+1,j).
  //
  /// Truss_Info Outputs
  //
  //  truss_info->balls
  //    A list that should be filled with the balls making up the truss.
  //
  //  truss_info->links
  //    A list that should be filled with the links making up the truss.


  const int num_sides = truss_info->base_coors.occ();
  const int num_units = truss_info->num_units;

  // Lists to hold balls and links created for truss.
  //
  Balls& bprep = truss_info->balls;
  Links& lprep = truss_info->links;

  // Create balls for truss.
  //
  for ( int i=0; i<num_units; i++ )
    for ( pCoor bcoor; truss_info->base_coors.iterate(bcoor); )
      {
        Ball* const ball = bprep += new Ball;
        ball->position = bcoor + i * truss_info->unit_length;
        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;
      }

  /// SOLUTION -- Problem 2
  //
  //  Prepare an array holding pointers to balls in the order needed
  //  to construct a triangle strip.
  //
  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];
        }
    }

  // Create links.
  //
  for ( int i=0; i<num_units; i++ )
    for ( int j=0; j<num_sides; j++ )
      {
        const int idx = j + num_sides * i;

        // Retrieve the ball corresponding to (i,j).
        //
        Ball* const ball = bprep[idx];

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

        // Insert link to neighbor ball with name i.
        //
        lprep += new Link( ball, bprep[pn_idx], color_gray );

        // Insert links to balls with same i that are not neighbors.
        //
        if ( j == i % num_sides )
          for ( int k = 2; k < num_sides-1; k++ )
            lprep += new Link
              ( ball, bprep[ idx + (k+j)%num_sides - j ], color_white );

        if ( i == 0 ) continue;

        // Insert link to ball at (i-1,j).
        //
        lprep +=
          new Link( ball, bprep[idx-num_sides], color_lsu_official_purple );

        // Insert link to ball at (i-1, j-1 mod num_sides ).
        //
        lprep += new Link( ball, bprep[pn_idx-num_sides], color_green );
      }
}
```

```
void
World::ball_setup_2()
{
  pCoor first_pos(13.4,17.8,-9.2);
  const float spacing = distance_relaxed;
  pVect delta_pos = pVect(spacing*0.05,-spacing,0);
  pNorm loc_y = delta_pos;
  pNorm loc_x = pVect(0,0,1);
  pNorm loc_z = cross(loc_y,loc_x);

  // Erase the existing balls and links.
  //
  objects_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) * loc_x
        + 0.5 * spacing * sin(angle) * loc_z;

      truss_info.base_coors += chain_first_pos;
    }

  make_truss(&truss_info);

  // 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;

  /// SOLUTION -- Problem 2
  mp.strips += truss_info.strips;
```

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

void
World::ball_setup_3()
{
  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.
  //
  objects_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);

  const int idx_center = chain_length / 2 * sides;

  for ( int i=0; i<sides; i++ )
    {
      Truss_Info ti;
      ti.num_units = chain_length / 2;
      const int idx_1 = idx_center + ( i == 0 ? sides - 1 : i - 1 );
      const int idx_2 = idx_center + i;

      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];
      ti.base_coors += b0->position;
      ti.base_coors += b1->position;
      ti.base_coors += b2->position;
      ti.base_coors += b3->position;

      pNorm v_head = cross(b1->position,b2->position,b3->position);
      ti.unit_length = delta_dir.magnitude * v_head;
      make_truss(&ti);
      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);
      links += ti.links;
```

```
      balls += ti.balls;

      /// SOLUTION -- Problem 2
      mp.strips += ti.strips;

      int tsz = ti.balls.size();

      if ( i == 2 )
        {
          ball_eye = ti.balls[tsz-2];
          ball_down = ti.balls[tsz-1];
        }
      else if ( i == 1 )
        {
          ball_gaze = ti.balls[tsz/2];
        }

    }

  // 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;
  mp.strips += truss_info.strips;

  opt_tail_lock = false;
  opt_head_lock = false;
}


void
World::ball_setup_4()
{
}

void
World::ball_setup_5()
{
}


 /// Advance Simulation State by delta_t Seconds
```

```
//
void
World::time_step_cpu(double delta_t)
{
  time_step_count++;

  // Smoothly move ball in response to user input.
  //
  if ( adj_t_stop )
    {
      const double dt = min(world_time,adj_t_stop) - adj_t_prev;
      pVect adj = dt/adj_duration * adj_vector;
      balls_translate(adj,0);
      adj_t_prev = world_time;
      if ( world_time >= adj_t_stop ) adj_t_stop = 0;
    }

  for ( BIter ball(balls); ball; )
    ball->force = ball->mass * gravity_accel;

  for ( LIter link(links); link; )
    {
      /// Problem 1 and 3 solutions go here, and other places.

      /// SOLUTION -- Problem 1
      //
      if ( link->snapped ) continue;

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

      // Construct a normalized (Unit) Vector from ball to neighbor.
      //
      pNorm ball_to_neighbor(ball1->position,ball2->position);

      const float distance_between_balls = ball_to_neighbor.magnitude;

      // Compute the speed of ball towards neighbor_ball.
      //
      pVect delta_v = ball2->velocity - ball1->velocity;
      float delta_s = dot( delta_v, ball_to_neighbor );

      // Compute by how much the spring is stretched (positive value)
      // or compressed (negative value).
      //
      const float spring_stretch =
        distance_between_balls - link->distance_relaxed;

      /// SOLUTION -- Problem 1
      //

      // 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 )
        {
          // Take appropriate action if snap threshold exceeded.
          //
          if ( distance_between_balls
               > link->distance_relaxed * mp.link_snap_thd )
            link->snapped = true;
```

```
          // Find an amount by which to blend link's original color
          // and stressed color.
          //
          const float factor =
            ( distance_between_balls / link->distance_relaxed
              - mp.link_stressed_thd )
            / ( mp.link_snap_thd - 1 );
          link->color = max(0.0f,1-factor) * link->natural_color
            + factor * mp.color_stressed;
        }
      else
        {
          link->color = link->natural_color;
        }

      // Determine whether spring is gaining energy (whether its length
      // is getting further from its relaxed length).
      //
      const bool gaining_e = ( delta_s > 0.0 ) == ( spring_stretch > 0 );

      // Use a smaller spring constant when spring is loosing energy,
      // a quick and dirty way of simulating energy loss due to spring
      // friction.
      //
      const float spring_constant =
        gaining_e ? opt_spring_constant : opt_spring_constant * 0.7;

      ball1->force += spring_constant * spring_stretch * ball_to_neighbor;
      ball2->force -= spring_constant * spring_stretch * ball_to_neighbor;
    }

  ///
  /// Update Position of Each Ball
  ///

  for ( BIter ball(balls); ball; )
    {
      /// Problem 3 solution goes here, and other places.

      if ( ball->locked )
        {
          ball->velocity = pVect(0,0,0);
          continue;
        }

      // Update Velocity
      //
      // This code assumes that force on ball is constant over time
      // step. This is clearly wrong when balls are moving with
      // respect to each other because the springs are changing
      // length. This inaccuracy will make the simulation unstable
      // when spring constant is large for the time step.
      //
      const float mass = max( ball->mass, ball->mass_min );

      pVect delta_v = ( ball->force / mass ) * delta_t;

      if ( platform_collision_possible(ball->position) && ball->position.y < 0 )
        {

          // Use a high spring constant for the platform. To damp energy
          // use a smaller constant when ball is moving away from the
          // platform.
          //
          const float spring_constant_plat =
```

```
            ball->velocity.y < 0 ? 100000 : 50000;

        const float force_up = -ball->position.y * spring_constant_plat;
        const float delta_v_up = force_up / mass * delta_t;

        /// SOLUTION -- Problem 3
        //
        // Use delta_v_op to compute a frictional force, and from
        // that a delta_v_surf. If delta_v_surf is so large that it
        // would reverse the direction of the ball then just stop
        // the ball.

        const float fric_coefficient = 0.1;

        // The amount of lateral frictional force is based on the
        // separation force (the force pushing the ball up).
        //
        const float fric_force_mag = fric_coefficient * force_up;

        // Compute the velocity of the ball along the surface.
        //
        pNorm surface_v(ball->velocity.x,0,ball->velocity.z);

        // Compute the change in velocity due to the frictional force.
        //
        const float delta_v_surf = fric_force_mag / mass * delta_t;

        // Check whether the change in velocity would reverse the direction.
        //
        if ( delta_v_surf > surface_v.magnitude )
          {
            // Here the frictional change in velocity that was
            // computed would reverse the direction of the
            // ball. That really can't happen, the ball would just
            // stop (in the x and z directions). So set delta_v so
            // that it stops the ball.
            //
            delta_v =
              pVect(-ball->velocity.x,delta_v.y,-ball->velocity.z);
          }
        else
          {
            // The change in velocity will slow the ball down, so
            // apply it.
            //
            delta_v -= delta_v_surf * surface_v;
          }
        delta_v.y += delta_v_up;
      }

    ball->velocity += delta_v;

    // Air Resistance
    //
    const double fs = pow(1+opt_air_resistance,-delta_t);
    ball->velocity *= fs;

    // Update Position
    //
    // Assume that velocity is constant.
    //
    ball->position += ball->velocity * delta_t;

  }
}
```

```
bool
World::platform_collision_possible(pCoor pos)
{
  // Assuming no motion in x or z axes.
  //
  return pos.x >= platform_xmin && pos.x <= platform_xmax
    && pos.z >= platform_zmin && pos.z <= platform_zmax;
}

 /// External Modifications to State
//
//   These allow the user to play with state while simulation
//   running.

// Move the ball.
//
void Ball::translate(pVect amt) {position += amt;}

// Add velocity to the ball.
//
void Ball::push(pVect amt) {velocity += amt;}

// Set the velocity to zero.
//
void Ball::stop() {velocity = pVect(0,0,0); }

// Set the velocity and rotation (not yet supported) to zero.
//
void Ball::freeze() {velocity = pVect(0,0,0); }



void World::balls_translate(pVect amt,int b){head_ball->translate(amt);}
void World::balls_push(pVect amt,int b){head_ball->push(amt);}
void World::balls_translate(pVect amt)
{ for ( BIter ball(balls); ball; ) ball->translate(amt); }
void World::balls_push(pVect amt)
{ for ( BIter ball(balls); ball; ) ball->push(amt); }
void World::balls_stop()
{ for ( BIter ball(balls); ball; ) ball->stop(); }
void World::balls_freeze(){balls_stop();}


void
World::render_my_piece()
{
  /// SOLUTION -- Problem 2.

  // Declare arrays used to hold vertex coordinates and normals.
  //
  pCoor *coord_buffer = NULL;
  pVect *norm_buffer = NULL;
  int buffer_elts = 0;  // Number of elements in currently allocated array.

  // Iterate over each strip (truss side).
  //
  for ( SIter strip(mp.strips); strip; )
    {
      const int elts = strip->balls;

      // If arrays are not large enough for this truss allocate more
      // storage.
```

```
      //
      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]));
        }

      // Copy ball positions into vertex coordinate array.
      //
      for ( int i=0; i<elts; i++ )
        coord_buffer[i] = strip->balls[i]->position;


      // Compute normals and copy them into normal array.
      //
      for ( int i=0; i<elts-2; i+=2 )
        {
          // Consider coordinates i to i+3. For each one construct a
          // normal based on its two neighbors and add the normal to
          // norm_buffer. When the loop finishes an element of
          // norm_buffer will be the sum of two normals.
          //
          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;
            }
        }

      // Render the strip.
      //
      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);
    }

  free(coord_buffer);
  free(norm_buffer);
}
```

```
void
World::frame_callback()
{
  // This routine called whenever window needs to be updated.

  const double time_now = time_wall_fp();

  if ( !opt_pause || opt_single_frame || opt_single_time_step )
    {
      /// Advance simulation state.

      // Amount of time since the user saw the last frame.
      //
      const double wall_delta_t = time_now - last_frame_wall_time;

      // Compute amount by which to advance simulation state for this frame.
      //
      const double duration =
        opt_single_time_step ? opt_time_step_duration :
        opt_single_frame ? 1/30.0 :
        wall_delta_t;

      const double world_time_target = world_time + duration;

      while ( world_time < world_time_target )
        {
          time_step_cpu(opt_time_step_duration);
          world_time += opt_time_step_duration;
        }

      // Reset these, just in case they were set.
      //
      opt_single_frame = opt_single_time_step = false;
    }

  last_frame_wall_time = time_now;

  if ( opt_ride && ball_eye )
    {
      pNorm b_eye_down(ball_eye->position,ball_down->position);
      pVect b_eye_up = -b_eye_down;
      pCoor eye_pos = ball_eye->position + 2.2 * ball_eye->radius * b_eye_up;
      pNorm b_eye_direction(eye_pos,ball_gaze->position);

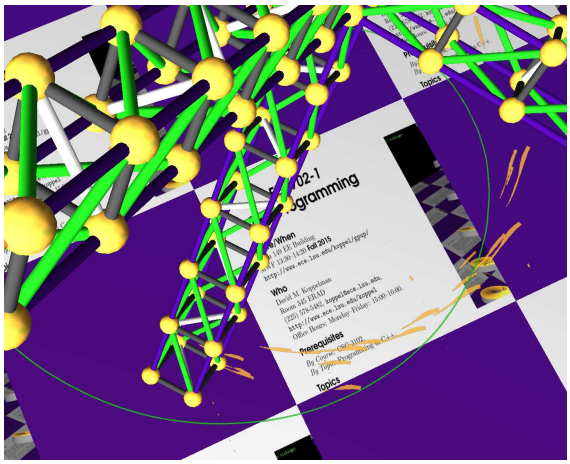
      pNorm b_eye_left = cross(b_eye_direction,b_eye_up);
      pMatrix_Translate center_eye(-eye_pos);
      pMatrix rotate; rotate.set_identity();
      for ( int i=0; i<3; i++ ) rotate.rc(0,i) = b_eye_left.elt(i);
      for ( int i=0; i<3; i++ ) rotate.rc(1,i) = b_eye_up.elt(i);
      for ( int i=0; i<3; i++ ) rotate.rc(2,i) = -b_eye_direction.elt(i);
      modelview = rotate * center_eye;
      pMatrix reflect; reflect.set_identity(); reflect.rc(1,1) = -1;
      transform_mirror = modelview * reflect * invert(modelview);
    }

  render();
}

int
main(int argv, char **argc)
{
  pOpenGL_Helper popengl_helper(argv,argc);
  World world(popengl_helper);
```

```
  popengl_helper.rate_set(30);
  popengl_helper.display_cb_set(world.frame_callback_w,&world);
}
```

# LSU EE 4702-1 Homework 3 Solution Due: 12 October 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.*

The code in this package includes the friction model asked for in Homework 2. The links however are unbreakable and there's no skin. Also, shadows are deactivated by default to make the solution to this assignment easier. Notice that this starts like the code for Homework 2, except that ball at the bottom of the chain hits the platform as it swings by.

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.

*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 `hw03-sol.cc`. For an HTML version visit `http://www.ece.lsu.edu/koppel/gpup/2015/hw03-sol.cc.html`. The repository version will always be most up to date.

**Problem 1:** Notice how the platform is always perfectly clean. That's not realistic! In this problem modify the code so that when the balls slide across the platform they leave scuffs. The shape of a scuff should be the area of the platform touched by the ball (base the width on the intersection of the sphere and the platform) and the color should match the ball. See the illustration to the right. The ball at the base of the gyroscope is green, it leaves a neat green path. The arm balls are yellow. The balls at the arm ends striking the platform leave scuffs that start out thick and get thinner.

The code should be added to class `My_Piece_Of_The_World` and elsewhere as needed. The places in which code needs to be added start with the comment `Homework 3` and are followed by checklists of things to be done.

Scuffs are to be implemented using textures. Rather than using one large texture covering the entire platform, the textures are broken up into a grid of *overlays*. The texture corresponding to an overlay should only be initialized when the area is first scuffed, and textures should only be sent to OpenGL when the textures change. This uses vastly less data then sending a texture for the entire platform at every frame.

(*a*) Code has already been added to `time_step_cpu` to retrieve an overlay, actually a `Platform_Overlay` object, for a ball if it is colliding with the platform. Add code to this routine and elsewhere to write the appropriate texels with scuff marks. For sample code see `sample_tex_make` in the homework file. This routine creates a texture containing a big red ex. (The code in `time_step_cpu` only needs to modify the texels in `po->data`, it should not send the data to OpenGL!)

The key to solving this problem is understanding how to go from object space coordinates, which are of course used for ball positions, to a texel coordinate space, and how to use a texel coordinate to compute an index for the texel array.

The routine `po_get_lcoor` converts from object space to texel space:

```
  // The two lines below are from My_Piece_Of_The_World::init()
  scale_x_obj_to_texel = wid_x_inv * twid_x;
  scale_z_obj_to_texel = wid_z_inv * twid_z;

pCoor My_Piece_Of_The_World::po_get_lcoor(Platform_Overlay *po, pCoor pos) {
  pCoor lc;
  lc.x = ( pos.x - overlay_xmin ) * scale_x_obj_to_texel;
  lc.z = ( pos.z - overlay_zmin ) * scale_z_obj_to_texel;
  lc.y = 0;
  lc.w = 0;
  return lc;
}
```

Variables `overlay_xmin` and `overlay_zmin` are the object space coordinates of a corner of the overlay, these were set in another routine when the overlay was retrieved. The `x` and `z` components of the returned coordinate give texel locations corresponding to `pos`. These will be used to access our texel array, in `data`, which is size `twid_x*twid_z` elements. The index into that array is computed by

```
int My_Piece_Of_The_World::po_get_tidx(pCoor lpos) {
  // Return an index for the texel array corresponding to texel coord lpos.
  const int idx = int(lpos.x) + twid_x * int(lpos.z);
  return idx;}
```

The next step is to find the object coordinates of the area to be scuffed. If we just wanted to scuff a single texel we could call `po_get_lcoor` and `po_get_tidx` to get the index into the array and then set `po->data[idx] = ball->color`, where `idx` is the return value of `po_get_tidx`. But, we want the size of the scuff mark to be based on the force, and we also need to account for motion between time steps. The code below handles both. Notice that the `t/u` loop nest is essentially rasterizing the rectangle corresponding to the scuffed area.

```
      pCoor ball_lcor = mp.po_get_lcoor(po,ball->position);

      // Previous location of ball in texel coordinates.
      pCoor prev_lcor = mp.po_get_lcoor(po,pos_prev);

      float width = mp.scale_x_obj_to_texel *
        sqrt( ball->radius * ball->radius - pos_prev.y * pos_prev.y );

      // Direction along scuff mark (based on sliding motion).
      pNorm skid(prev_lcor,ball_lcor);

      // Find direction along width of scuff mark.
      pNorm nskid = cross( skid, pVect(0,1,0) );
```

```
// Iterate over texel locations scuffed by ball.
for ( float t = -width; t <= skid.magnitude+width; t++ )
  for ( float u = -width; u < max(width,1.0f); u++ )
  {
    pCoor tex_pos = prev_lcor + t * skid + u * nskid;
    pColor* const texel = mp.po_get_texel(po,tex_pos);

    // If texel is not on overlay po then just skip it.
    if ( !texel ) continue;

    // Don't bother if already scuffed. If we wanted to be
    // fancy we could add this balls color onto what is already
    // there. But we don't want to be fancy.
    if ( texel->a ) continue;

    po->texture_modified = true;
    *texel = ball->color;
    texel->a = 0.8;
  }
```

The solution above does not touch an already scuffed region. It would not be too difficult to modify the code so that a second scuff combines colors with any existing scuff.

(*b*) Routine `My_Piece_Of_The_World::render()` is supposed to render the platform overlays. For each scuffed overlay it should emit primitive(s) with the corresponding texture attached. The texture application, alpha test, and blending modes need to be set so that only the scuff marks cover the underlying platform (and semi-transparently if possible). The routine has some code. Finish it, and make changes elsewhere as needed.

See `gpup/demo-8-textures.cc` for examples of how to do texturing, blending, and alpha testing. If you are not sure if your own texture is working you can use the texture created in `sample_tex_make`.

There are several things that need to be done. At the top of the render routine we need to set up texturing and blending. Then for each non-empty overlay we need to do the following: First, if the overlay is new we need to create the buffer object. Second, if our copy of the texture (the one in `data`) has been modified, we need to send it to OpenGL. Next, we need to render a primitive on which the texture will be applied.

To start, the lines below turn on texturing and select the modulate texture application mode. The application mode indicates how the texel color will be combined with the primitive color (actually, the color from the prior texture unit, but in this case there is no prior texture unit). The solution uses modulation, in which the color values are multiplied. My multiplying lighting effects are transferred to the texel values.

```
glEnable(GL_TEXTURE_2D);
glActiveTexture(GL_TEXTURE0);
glTexEnvi(GL_TEXTURE_ENV, GL_TEXTURE_ENV_MODE, GL_MODULATE);
```

The solution uses two methods of applying the textures to the frame buffer. The primary one is through blending, in which the stuff mark and the existing pixel values are combined. That's set up by turning on blending and setting blending to use the texel's alpha value, and to not blend the alpha values themselves.

```
glEnable(GL_BLEND);
glBlendFuncSeparate(GL_SRC_ALPHA,GL_ONE_MINUS_SRC_ALPHA, GL_ZERO, GL_ONE);
```

Inside of the overlay (`i`) loop, a new texture object is initialized if necessary. This should be done once per overlay.

```
for ( int i=0; i<num_overlays; i++ ) {
```

```
Platform_Overlay* const po = &platform_overlays[i];
if ( !po->data ) continue;

if ( !po->texture_object_initialized ) {
    po->texture_object_initialized = true;
    glGenTextures(1,&po->txid);
    glBindTexture(GL_TEXTURE_2D,po->txid);
    glTexParameteri(GL_TEXTURE_2D, GL_GENERATE_MIPMAP, 1);
    glTexParameterf(GL_TEXTURE_2D,GL_TEXTURE_MIN_FILTER,
                    GL_LINEAR_MIPMAP_LINEAR);
    glTexParameterf(GL_TEXTURE_2D,GL_TEXTURE_MAG_FILTER,GL_LINEAR);
  }
```

Next, texture data for this overlay is sent to OpenGL *if the texture changed*.

```
glBindTexture(GL_TEXTURE_2D,po->txid);
if ( po->texture_modified ) {
    po->texture_modified = false;
    glTexImage2D
      (GL_TEXTURE_2D,
       0,                // Level of Detail (0 is base).
       GL_RGBA,          // Internal format to be used for texture.
       twid_x, twid_z,
       0,                // Border
       GL_RGBA,     // GL_BGRA: Format of data read by this call.
       GL_FLOAT,    // Size of component.
       (void*)po->data); }
```

Next a primitive to carry the texture is rendered. The primitive is a quad (quadrilateral), the coordinates were pre-computed when the texture overlay was created.

```
glBegin(GL_QUADS);
glNormal3f(0,-1,0);
glColor3fv(color_white);
glTexCoord2f(0,0);
glVertex3fv(po->vertices[0]);
glTexCoord2f(1,0);
glVertex3fv(po->vertices[1]);
glTexCoord2f(1,1);
glVertex3fv(po->vertices[2]);
glTexCoord2f(0,1);
glVertex3fv(po->vertices[3]);
glEnd();
```

Routine `po_get` in the solution computes the coordinates of the overlay boundaries used above, those are put in array `vertices`. The routine also allocates and initializes the texel array.

The routine `clean` removes scuff marks. It does so by freeing our texel arrays (`data`) and OpenGL's texture objects, making the storage available for other use.

A common mistake was to implement clean with the single line `platform_overlay = new Platform_Overlay[num_overlay`. That would keep the CPU storage for the textures (in `data`) and the GPU storage (in the texture objects) in use, a situation that is called a memory leak. Since the textures can be large, the memory leak would be serious.

**Problem 2:** Answer each question below about performance aspects of texturing.

(*a*) The texel array that we send to OpenGL is an array of `floats`, because that lets us use our familiar `pColor` objects. What would be the impact of using a color representation that matched the depth of our displays, eight bits per color? *Note: This question touches on more issues than were anticipated, and it would not be easy to give a fully correct answer given the material covered in the course.*

Relatively Short Answer: Assuming that the texel is sent to the GPU in the format provided to OpenGL, the use of an 8-bit integer rather than a 32-bit float would reduce the amount of data sent to the GPU by a factor of four, and since the texture arrays we are working with are of moderate size and change frequently the savings would be significant.

Long Answer: The impact that the question asks about can take three forms: the amount of computation needed to convert the texel colors to OpenGL's internal format, the amount of data that needs to be sent from the CPU to the GPU, and the range of representable colors. The OpenGL standard describes behavior in terms of a floating-point representation for colors, and it is likely that modern GPUs use such representations. (Though they may not be 32-bit IEEE 754 floats.) If the representation were 32-bit IEEE floats, then additional computation would be needed if the colors were given in integers. If the conversion were performed on the GPU, then providing the data as integers would reduce the amount of data to be sent by a factor of four, which is a large amount. The actual computation needed to convert to the internal format would be small compared to the computation needed to prepare the different MIPMAP levels. We can assume that a decent OpenGL would do this on the GPU (though the fact that automatic MIPMAP generation is now deprecated might make this less likely in the future).

(*b*) Suppose we vary `nx` and `nz`, and choose `twid_x` and `twid_z` so that the total number of texels across the platform, `nx * twid_x`, is constant. Describe the impact on performance as `nx` and `nz` change vary from 1 to their maximum (say, 10240 for the assignment code).

When `nx` and `nz` are small the textures are huge, and so performance would be hurt when we need to send them over. When `nx` and `nz` are large the textures are small but there are many more textures and especially many more `Platform_Overlay` objects. Though the amount of texture data that is sent to the GPU is smaller with large `nx` and `nz` (since we only send modified textures), the overhead of managing these textures becomes large. At their maximum value each texture is just one texel, that's 16 bytes, but that's comparable to the size of the `Platform_Overlay` object needed to manage them. While the storage for textures is only created when the respective area is scuffed, the `platform_overlays` array is sized to `nx * nz` elements, which would be $10240 \times 10240 = 104\,857\,600$ elements taking over a GB of storage even for a perfectly clean platform. Also, the rendering code performs an exhaustive search of overlays to find the dirty ones. If that code isn't modified there would be a performance hit there too. Even if the `Platform_Overlay` objects were themselves only allocated for scuffed textures, performance would still be hurt because the time needed to prepare to render the overlay would have to be born for each little one-texel texture. If the scuffed region covered 500 texels, that would be 500 times. If `nx` were smaller (and textures larger, say 50 by 50 texels) then perhaps 2 or 3 overlays would be sufficient to cover the scuffed areas, and so the rendering preparation would only be done 2 or 3 times. All off this means that intermediate values of `nx` and `nz` yield the best performance.

Just to be sure that everyone understands, lets start with `nx` and `nz` at 1 and increase them to their maximum value. When they are at 1, we need to send over a $10240 \times 10240 \times 4 \times 4 = 1600\,\mathrm{MiB}$ texture each time the texture changes, which could be every frame. If we increase `nx` and `nz` to 2, now there are four overlays. If new scuff marks are confined to one of those four we only need to send over $400\,\mathrm{MiB}$ of data, which should yield much better performance. Increasing them to 16 means we have $6.25\,\mathrm{MiB}$ textures and $16^2 = 256$ overlays, which ought to improve performance even more. But at some point, maybe when `nx` and `nz` reach 1024 the size of the `platform_overlay` array will start to hurt performance, since we scan the entire thing before rendering. Increasing them further will cause our application to suffer due to the time needed to render all those tiny textures.

```
/// LSU EE 4702-1 (Fall 2015), GPU Programming
//
 /// Homework 3 -- SOLUTION
 //
 /// Your Name:  SOLUTION

 /// Instructions
 //
 //  Read the assignment: http://www.ece.lsu.edu/koppel/gpup/2015/hw03.pdf
 //  Solution discussion: http://www.ece.lsu.edu/koppel/gpup/2015/hw03_sol.pdf

#if 0
/// SOLUTION OUTLINE

 /// Parts of Solution
 //
 //  - Preparing a Texture
 //  - Division of Platform into Overlays (Use many small textures.)
 //  - Coordinate Spaces: Object (balls) to Overlays to Texture Pixels

 /// Code Organization

 /// Platform_Overlay
 //
 Platform_Overlay* platform_overlays;  // Array of overlays.
 //
 //  A data structure holding info about texture that covers part of platform.
 //
 //  Platform is covered by nx * ny (default 40*40) overlays ..
 //  .. arranged in a grid.
 //
 //  Initialized in My_Piece_Of_The_World::init():



 /// Overlay Rationale
 //
 //  We expect to update texture on CPU and send it back go GPU.
 //
 //  This may happen every frame.
 //
 //  Suppose minimum scuff size is 1/100 of a tile:
 //    19 * 19 platform tiles  * 100 * 100  * 24 bytes / texel
 //    = 86640000 B
 //
 //  Recall: PCIe x16:  15.8 GB/s
 //  Suppose we want a frame rate of 60 f/s
 //   15.8 10^9 / ( 60 * 86640000 ) = 3.04
 //   About 1/3 of the capacity will be used for texture updates.



#endif
/// Purpose
//
//   Demonstrate simulation of point masses connected by springs.


/// What Code Does

// Simulates balls connected by springs over a platform. Balls and
```

```
// springs can be initialized in different arrangements (called
// scenes). Currently scene 1 is a simple string of beads, and scenes
// 2, 3, and 4 are trusses. The platform consists of tiles, some are
// purple-tinted mirrors (showing a reflection of the ball), the
// others show the course syllabus.



///  Keyboard Commands
 //
 /// Object (Eye, Light, Ball) Location or Push
 //   Arrows, Page Up, Page Down
 //        Move object or push ball, depending on mode.
 //        With shift key pressed, motion is 5x faster.
 //   'e': Move eye.
 //   'l': Move light.
 //   'b': Move head (first) ball. (Change position but not velocity.)
 //   'B': Push head ball. (Add velocity.)
 //
 /// Eye Direction
 //   Home, End, Delete, Insert
 //   Turn the eye direction.
 //   Home should rotate eye direction up, End should rotate eye
 //   down, Delete should rotate eye left, Insert should rotate eye
 //   right.  The eye direction vector is displayed in the upper left.

 /// Simulation Options
 //  (Also see variables below.)
 //
 //  'c'    Clean the platform.
 //  'w'    Twirl balls around axis formed by head and tail.
 //  '1'    Set up scene 1.
 //  '2'    Set up scene 2.
 //  '3'    Set up scene 3.
 //  'p'    Pause simulation. (Press again to resume.)
 //  ' '    (Space bar.) Advance simulation by 1/30 second.
 //  'S- '  (Shift-space bar.) Advance simulation by one time step.
 //  'h'    Freeze position of first (head) ball. (Press again to release.)
 //  't'    Freeze position of last (tail) ball. (Press again to release.)
 //  's'    Stop balls.
 //  'g'    Turn gravity on and off.
 //  'y'    Toggle value of opt_tryout1. Intended for experiments and debugging.
 //  'Y'    Toggle value of opt_tryout2. Intended for experiments and debugging.
 //  'F11'  Change size of green text (in upper left).
 //  'F12'  Write screenshot to file.

 /// Variables
 //   Selected program variables can be modified using the keyboard.
 //   Use "Tab" to cycle through the variable to be modified, the
 //   name of the variable is displayed next to "VAR" on the bottom
 //   line of green text.

 //  'Tab' Cycle to next variable.
 //  '`'   Cycle to previous variable.
 //  '+'   Increase variable value.
 //  '-'   Decrease variable value.
 //
 //  VAR Spring Constant - Set spring constant.
 //  VAR Time Step Duration - Set physics time step.
 //  VAR Air Resistance - Set air resistance.
 //  VAR Light Intensity - The light intensity.
 //  VAR Gravity - Gravitational acceleration. (Turn on/off using 'g'.)
```

```
#define GL_GLEXT_PROTOTYPES
#define GLX_GLXEXT_PROTOTYPES

#include <GL/gl.h>
#include <GL/glext.h>
#include <GL/glx.h>
#include <GL/glxext.h>
#include <GL/glu.h>
#include <GL/freeglut.h>

#include <gp/util.h>
#include <gp/glextfuncs.h>
#include <gp/coord.h>
#include <gp/shader.h>
#include <gp/pstring.h>
#include <gp/misc.h>
#include <gp/gl-buffer.h>
#include <gp/texture-util.h>
#include <gp/colors.h>

#include "util-containers.h"
#include "shapes.h"


///
/// Main Data Structures
///
//
// class World: All data about scene.


class World;


class Platform_Overlay {
public:
  Platform_Overlay():data(NULL){}

  // Note: When using the array below as an argument to glTexImage2D
  // use data format GL_RGBA and type GL_FLOAT.
  //
  pColor *data;
  GLuint txid;

  bool texture_modified;
  bool texture_object_initialized;

  pCoor vertices[4];

};

 /// Homework 3 All Problems
//
//   Use this class to define variables and member functions.
//   Don't modify hw03-graphics.cc.
//
class My_Piece_Of_The_World {
public:
  My_Piece_Of_The_World(World& wp):w(wp){};
  World& w;
  Platform_Overlay* platform_overlays;  // Array of overlays.
  Platform_Overlay sample_overlay;
  int nx, nz;          // Number of overlays along each dimension.
  int num_overlays;
```

```
  int twid_x, twid_z;  // Dimensions of each texture in texels.
  int num_texels;
  float wid_x, wid_z;  // Width of each overlay in object space units.
  float wid_x_inv, wid_z_inv;  // Their inverses.

  /// SOLUTION
  float scale_x_obj_to_texel;
  float scale_z_obj_to_texel;

  // Minimum x- and z- object space coordinate for most recent overlay.
  float overlay_xmin, overlay_zmin;

  void init();
  void sample_tex_make();

  // Return the platform overlay that includes pos, or NULL if pos is
  // not on platform.
  //
  Platform_Overlay* po_get(pCoor pos);

  //
  // Homework 3: These are suggested functions.
  //

  // Convert object space coordinate to texel coordinate relative
  // to overlay po.
  pCoor po_get_lcoor(Platform_Overlay *po, pCoor pos);

  // Return the texel at lpos in po.
  pColor* po_get_texel(Platform_Overlay *po, pCoor lpos);

  /// SOLUTION -- Problem 1(a)
  int po_get_tidx(pCoor lpos);

  //
  // Homework 3: These functions must be implemented.
  //
  void render();
  void clean();
};


// Object Holding Ball State
//
class Ball {
public:
  Ball():velocity(pVect(0,0,0)),locked(false),
         color(color_lsu_spirit_gold),contact(false){};
  pCoor position;
  pVect velocity;

  float mass;
  float mass_min; // Mass below which simulation is unstable.
  float radius;

  bool locked;

  pVect force;
  pColor color;
  bool contact;                 // When true, ball rendered in gray.
  float spring_constant_sum;    // Used to compute minimum mass.

  void push(pVect amt);
```

```
  void translate(pVect amt);
  void stop();
  void freeze();
};

class Link {
public:
  Link(Ball *b1, Ball *b2):ball1(b1),ball2(b2),
     distance_relaxed(pDistance(b1->position,b2->position)), snapped(false),
     natural_color(color_lsu_spirit_purple),color(color_lsu_spirit_purple){}
  Link(Ball *b1, Ball *b2, pColor colorp):ball1(b1),ball2(b2),
     distance_relaxed(pDistance(b1->position,b2->position)), snapped(false),
     natural_color(colorp),color(colorp){}
  Ball* const ball1;
  Ball* const ball2;
  float distance_relaxed;
  bool snapped;
  pColor natural_color;
  pColor color;
};

// Declare containers and iterators for Balls and Links.
// (See util_container.h.)
//
typedef pVectorI<Link> Links;
typedef pVectorI_Iter<Link> LIter;
typedef pVectorI<Ball> Balls;
typedef pVectorI_Iter<Ball> BIter;

struct Truss_Info {

  // See make_truss for a description of what the members are for.

  // Inputs
  PStack<pCoor> base_coors;  // Coordinates of first set of balls.
  pVect unit_length;
  int num_units;

  // Output
  Balls balls;
  Links links;
};

#include "hw03-graphics.cc"

void
My_Piece_Of_The_World::init()
{
  // Number of overlays along each direction.
  //
  nx = 40; nz = 40;
  num_overlays = nx * nz;

  // Number of texels along each dimension of each overlay's texture.
  //
  twid_x = 256; twid_z = 256;
  num_texels = twid_x * twid_z;

  sample_tex_make();

  platform_overlays = new Platform_Overlay[num_overlays];

  // Dimensions of overlay in object-space coordinates.
```

```
  //
  wid_x = ( w.platform_xmax - w.platform_xmin ) / nx;
  wid_z = ( w.platform_zmax - w.platform_zmin ) / nz;
  wid_x_inv = 1.0 / wid_x;
  wid_z_inv = 1.0 / wid_z;

  /// SOLUTION -- Problem 1a
  //
  //  Compute variables that will come in handy when converting
  //  from object space coordinates to texel coordinates (for our overlay).
  //
  scale_x_obj_to_texel = wid_x_inv * twid_x;
  scale_z_obj_to_texel = wid_z_inv * twid_z;
}

void
My_Piece_Of_The_World::sample_tex_make()
{
  /// Homework 3 -- Sample Code

  // Code in this routine creates a texture with a big red X, and
  // loads it into a texture object.  The texture object can
  // be used as a substitute for the "scuffed" texture before
  // the scuffed texture part of this assignment is finished.

  Platform_Overlay* const po = &sample_overlay;

  // Allocate storage for the array of texels.
  //
  if ( !po->data ) po->data = new pColor[ num_texels ];

  // Initialize the texels to black and transparent. (Alpha = 0)
  //
  memset(po->data,0,num_texels*sizeof(po->data[0]));

  // Thickness of the strokes making up the letter ex.
  //
  const int thickness = max(2, twid_x/10);

  // Write the letter ex, a big one, in the texture.
  //
  for ( int tx=0; tx<twid_x-thickness; tx++ )
    {
      // Note: Compute tz without assuming twid_x == twid_z.
      int tz_raw = float(tx)/twid_x * twid_z;
      int tz = min(tz_raw,twid_z-1);

      // Array index of texel at (tx,tz).
      //
      int idx  = tx              + tz * twid_x; // Lower left to upper right.
      int idx2 = twid_x - 1 - tx + tz * twid_x; // Lower right to upper left.

      // Write colors to texels.
      //
      for ( int i=0; i<thickness; i++ )
        {
          po->data[idx+i] = color_red;
          po->data[idx+i].a = 1;
          po->data[idx2-i] = color_red;
          po->data[idx2-i].a = 1;
        }
    }

  // Create a new texture object.
  //
```

```
  glGenTextures(1,&po->txid);

  // Make our new texture object the current texture object.
  //
  glBindTexture(GL_TEXTURE_2D,po->txid);
  //
  // Subsequent OpenGL calls operating on GL_TEXTURE_2D ..
  // .. will now operate on po->txid, our new texture object.

  // Tell OpenGL to generate MIPMAP levels for us.
  //
  glTexParameteri(GL_TEXTURE_2D, GL_GENERATE_MIPMAP, 1);

  // Load our texture into the texture object.
  //
  glTexImage2D
    (GL_TEXTURE_2D,
     0,                 // Level of Detail (0 is base).
     GL_RGBA,           // Internal format to be used for texture.
     twid_x, twid_z,
     0,                 // Border
     GL_RGBA,           // GL_BGRA: Format of data read by this call.
     GL_FLOAT,          // Size of component.
     (void*)po->data    // Pointer to the texture data.
     );
}

 /// SOLUTION -- Problem 1a
int
My_Piece_Of_The_World::po_get_tidx(pCoor lpos)
{
  // Return an index for the texel array corresponding to texel coord lpos.
  const int idx = int(lpos.x) + twid_x * int(lpos.z);
  return idx;
}

pColor*
My_Piece_Of_The_World::po_get_texel(Platform_Overlay *po, pCoor lpos)
{
  /// SOLUTION -- Problem 1a

  // Use our function to get the array index corresponding to lpos.
  const int idx = po_get_tidx(lpos);

  // If it's out of range return null ..
  if ( idx < 0 || idx >= num_texels ) return NULL;
  // .. otherwise return the texel.
  return &po->data[ idx ];
}

pCoor
My_Piece_Of_The_World::po_get_lcoor(Platform_Overlay *po, pCoor pos)
{
  pCoor lc;
  /// SOLUTION -- Problem 1a.

  // Convert object space coordinates to texel coordinates.
  // Variables overlay_xmin and overlay_zmin were set when the overlay
  // was retrieved.
  //
  lc.x = ( pos.x - overlay_xmin ) * scale_x_obj_to_texel;
  lc.z = ( pos.z - overlay_zmin ) * scale_z_obj_to_texel;
  lc.y = 0;
  lc.w = 0;
```

```
  return lc;
}

Platform_Overlay*
My_Piece_Of_The_World::po_get(pCoor pos)
{
  // Convert object-space coordinate, pos, into an overlay space, (x,z) ..
  // .. in which (0,0) is the lower-left overlay, etc.

  const int x = ( pos.x - w.platform_xmin ) * wid_x_inv;
  if ( x < 0 || x >= nx ) return NULL;
  const int z = ( pos.z - w.platform_zmin ) * wid_z_inv;
  if ( z < 0 || z >= nz ) return NULL;
  overlay_xmin = w.platform_xmin + x * wid_x;
  overlay_zmin = w.platform_zmin + z * wid_z;

  // Retrieve the overlay in which pos lies.
  //
  Platform_Overlay* const po = &platform_overlays[x + z * nz];

  if ( !po->data )
    {
      /// SOLUTION -- Problem 1a
      // This overlay has never been visited, initialize texel array.

      // Allocate the color array.
      po->data = new pColor[num_texels];

      // Initialize it. Note that this makes the colors transparent.
      memset(po->data,0,num_texels*sizeof(po->data[0]));

      po->texture_object_initialized = false;

      // Pre-compute the object-space coordinates of the corners of
      // the overlay. These will be used when constructing the primitive
      // on which the texture will be applied.
      //
      pCoor* const vertices = po->vertices;
      vertices[0] =
        pCoor( w.platform_xmin + x * wid_x, 0.01, w.platform_zmin + z * wid_z );
      vertices[1] = vertices[0] + pVect(wid_x,0,0);
      vertices[2] = vertices[1] + pVect(0,0,wid_z);
      vertices[3] = vertices[0] + pVect(0,0,wid_z);
      po->texture_modified = true;
    }

  return po;
}

void
My_Piece_Of_The_World::render()
{
  /// Homework 3 -- Lots of stuff in this routine.

  // [ ] Enable at least some of the following:
  //     -- Texturing.
  //     -- Texture application mode.
  //     -- The Alpha Test
  //     -- Blending.
  //
  // See demo-8-texture.cc for examples.

  /// SOLUTION -- Problem 1b

  // Turn on texturing.
```

```
  glEnable(GL_TEXTURE_2D);

  // Specify which texture unit to use.
  glActiveTexture(GL_TEXTURE0);

  // Specify how texel and the primitive's lighted color should be
  // combined. Use modulation, so that scuffs are affected by
  // lighting.
  glTexEnvi(GL_TEXTURE_ENV, GL_TEXTURE_ENV_MODE, GL_MODULATE);

  // Turn on the alpha test. This is not needed if blending is in use.
  if ( w.opt_tryout1 ) glEnable(GL_ALPHA_TEST);

  // Only write fragments corresponding to scuffed parts of the texture.
  glAlphaFunc(GL_GREATER,0.4);

  // Turn on blending so that scuffs are combined with what's already there.
  glEnable(GL_BLEND);

  // Set the blend equation to blend colors, but to write the alpha value
  // carried by the fragment (from our texture).
  glBlendFuncSeparate
    (GL_SRC_ALPHA,GL_ONE_MINUS_SRC_ALPHA, GL_ZERO, GL_ONE);

  for ( int i=0; i<num_overlays; i++ )
    {
      Platform_Overlay* const po = &platform_overlays[i];
      if ( !po->data ) continue;

      if ( !po->texture_object_initialized )
        {
          po->texture_object_initialized = true;

          // [x] Do something here.
          /// SOLUTION -- Problem 1b.

          //  This is the first time we've used this particular overlay,
          //  so we need to create a texture object for it and set its
          //  parameters.
          //
          glGenTextures(1,&po->txid);
          glBindTexture(GL_TEXTURE_2D,po->txid);
          glTexParameteri(GL_TEXTURE_2D, GL_GENERATE_MIPMAP, 1);
          glTexParameterf(GL_TEXTURE_2D,GL_TEXTURE_MIN_FILTER,
                          GL_LINEAR_MIPMAP_LINEAR);
          glTexParameterf(GL_TEXTURE_2D,GL_TEXTURE_MAG_FILTER,GL_LINEAR);
        }

      glBindTexture(GL_TEXTURE_2D,po->txid);

      if ( po->texture_modified )
        {
          /// SOLUTION -- Problem 1a.
          // We've modified the texture since the last render, so
          // we need to update OpenGL's copy.

          po->texture_modified = false;

          // [x] Send texel array (po->data) to OpenGL.

          glTexImage2D
            (GL_TEXTURE_2D,
             0,                 // Level of Detail (0 is base).
             GL_RGBA,           // Internal format to be used for texture.
             twid_x, twid_z,
```

```
            0,                  // Border
            GL_RGBA,      // GL_BGRA: Format of data read by this call.
            GL_FLOAT,     // Size of component.
            (void*)po->data);
          pError_Check();
        }


      /// SOLUTION -- 1a
      //
      //  If tryout2 is active (press 'Y' to toggle) show the red x texture
      //  instead of our own.
      //
      if ( w.opt_tryout2 ) glBindTexture(GL_TEXTURE_2D, sample_overlay.txid);

      // [x] Render primitive(s) matching shape of overlay.

      /// SOLUTION -- 1a
      //
      //  Apply the texture to a quad. Note that we are using
      //  our pre-computed vertices.
      //
      glBegin(GL_QUADS);
      glNormal3f(0,-1,0);
      glColor3fv(color_white);
      glTexCoord2f(0,0);
      glVertex3fv(po->vertices[0]);
      glTexCoord2f(1,0);
      glVertex3fv(po->vertices[1]);
      glTexCoord2f(1,1);
      glVertex3fv(po->vertices[2]);
      glTexCoord2f(0,1);
      glVertex3fv(po->vertices[3]);
      glEnd();
    }

  // [x] Disable anything turned on at the start.

  glDisable(GL_ALPHA_TEST);
  glDisable(GL_BLEND);
  glDisable(GL_TEXTURE_2D);
}

void
My_Piece_Of_The_World::clean()
{
  /// Homework 3:  [ ] Remove scuffs from dirty textures.

  /// SOLUTION -- Problem 1.
  //
  //  Free any textures that are present.

  for ( int i=0; i<num_overlays; i++ )
    {
      Platform_Overlay* const po = &platform_overlays[i];
      if ( !po->data ) continue;

      // Tell OpenGL to free the texture object, including the
      // texture data that it has copied from us.
      //
      glDeleteTextures(1,&po->txid);

      // Free our own texel array.
      //
      free( po->data );
```

```
      // Update variables to reflect clean state.
      //
      po->data = NULL;
      po->texture_modified = false;
      po->texture_object_initialized = false;
    }
}


void
World::init()
{
  chain_length = 14;

  opt_time_step_duration = 0.0003;
  variable_control.insert(opt_time_step_duration,"Time Step Duration");

  distance_relaxed = 15.0 / chain_length;
  opt_spring_constant = 15000;
  variable_control.insert(opt_spring_constant,"Spring Constant");

  opt_gravity_accel = 9.8;
  opt_gravity = true;
  gravity_accel = pVect(0,-opt_gravity_accel,0);
  variable_control.insert(opt_gravity_accel,"Gravity");

  opt_air_resistance = 0.04;
  variable_control.insert(opt_air_resistance,"Air Resistance");

  world_time = 0;
  time_step_count = 0;
  last_frame_wall_time = time_wall_fp();
  frame_timer.work_unit_set("Steps / s");

  ball_eye = NULL;
  opt_ride = false;

  init_graphics();

  mp.init();

  ball_setup_1();
  lock_update();
}

Ball*
World::make_marker(pCoor position, pColor color)
{
  Ball* const ball = new Ball;
  ball->position = position;
  ball->locked = true;
  ball->velocity = pVect(0,0,0);
  ball->radius = 0.2;
  ball->mass = 0;
  ball->contact = false;
  ball->color = color;
  return ball;
}

void
World::lock_update()
{
  // This routine called when options like opt_head_lock might have
```

```
  // changed.

  // Update locked status.
  //
  if ( head_ball ) head_ball->locked = opt_head_lock;
  if ( tail_ball ) tail_ball->locked = opt_tail_lock;

  // Re-compute minimum mass needed for stability.
  //
  for ( BIter ball(balls); ball; ) ball->spring_constant_sum = 0;
  const double dtis = pow( opt_time_step_duration, 2 );
  for ( LIter link(links); link; )
    {
      Ball* const b1 = link->ball1;
      Ball* const b2 = link->ball2;
      b1->spring_constant_sum += opt_spring_constant;
      b2->spring_constant_sum += opt_spring_constant;
    }
  for ( BIter ball(balls); ball; )
    ball->mass_min = ball->spring_constant_sum * dtis;
}

void
World::balls_twirl()
{
  if ( !head_ball || !tail_ball ) return;

  pNorm axis(head_ball->position, tail_ball->position);

  for ( BIter ball(balls); ball; )
    {
      pVect b_to_top(ball->position,head_ball->position);
      const float dist_along_axis = dot(b_to_top,axis);
      const float lsq = b_to_top.mag_sq() - dist_along_axis * dist_along_axis;
      if ( lsq <= 1e-5 ) { ball->velocity = pVect(0,0,0); continue; }
      const float dist_to_axis = sqrt(lsq);
      pNorm rot_dir = cross(b_to_top,axis);
      ball->velocity += 10 * dist_to_axis * rot_dir;
    }
}

void
World::objects_erase()
{
  ball_eye = NULL;
  balls.erase();
  links.erase();
}

///
/// Physical Simulation Code
///

 /// Initialize Simulation
//

void
World::ball_setup_1()
{
  // Arrange and size balls to form a pendulum.

  pCoor first_pos(13.4,14,-9.2);
  pVect delta_pos = pVect(distance_relaxed,0,0);
```

```
  // Remove objects from the simulated objects lists, balls and links.
  // The delete operator is used on objects in the lists.
  //
  objects_erase();

  for ( int i=0; i<chain_length; i++ )
    {
      // Construct a new ball and add it to the simulated objects list (balls).
      //
      Ball* const ball = balls += new Ball;

      // Initialize position and other information.
      //
      ball->position = first_pos + i * delta_pos;
      ball->locked = false;
      ball->velocity = pVect(0,0,0);
      ball->radius = 0.3;
      ball->mass = 4/3.0 * M_PI * pow(ball->radius,3);
      ball->contact = false;

      // If it's not the first ball link it to the previous ball.
      if ( i > 0 ) links += new Link( ball, balls[i-1] );
    }

  // The balls pointed to by head_ball and tail_ball can be manipulated
  // using the user interface (by pressing 'h' or 't', for example).
  // Set these variables.
  //
  head_ball = balls[0];
  tail_ball = balls[balls-1];

  opt_head_lock = true;    // Head ball will be frozen in space.
  opt_tail_lock = false;   // Tail ball can move freely.
}


void
World::make_truss(Truss_Info *truss_info)
{
  /// Construct a truss based on members of truss_info.
  //

  //            <---- num_units (=9) ----------->
  //  j
  //  0         O---O---O---O---O---O---O---O---O     ^
  //            |   |   |   |   |   |   |   |   |     |
  //  1         O---O---O---O---O---O---O---O---O   num_sides (=3)
  //            |   |   |   |   |   |   |   |   |     |
  //  2         O---O---O---O---O---O---O---O---O     v
  //
  //      i ->  0   1   2   3   4   5   6   7   8
  //
  //  Note: Not all links are shown in the diagram above.

  /// Truss_Info Inputs
  //
  //  truss_info->num_units
  //    The number of sections in the truss (see diagram above).
  //
  //  truss_info->base_coors
  //    A list containing num_sides coordinates, the coordinates of
  //    balls at i=0 (see diagram above). (num_sides is the number of
  //    elements in this list.)  These coordinates should all be
  //    in the same plane.
  //
```

```
//  truss_info->unit_length
//    A vector pointing from the ball at (i,j) to the ball at (i+1,j).
//
/// Truss_Info Outputs
//
//  truss_info->balls
//    A list that should be filled with the balls making up the truss.
//
//  truss_info->links
//    A list that should be filled with the links making up the truss.


const int num_sides = truss_info->base_coors.occ();
const int num_units = truss_info->num_units;

// Lists to hold balls and links created for truss.
//
Balls& bprep = truss_info->balls;
Links& lprep = truss_info->links;

// Create balls for truss.
//
for ( int i=0; i<num_units; i++ )
  for ( pCoor bcoor; truss_info->base_coors.iterate(bcoor); )
    {
      Ball* const ball = bprep += new Ball;
      ball->position = bcoor + i * truss_info->unit_length;
      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;
    }

// Create links.
//
for ( int i=0; i<num_units; i++ )
  for ( int j=0; j<num_sides; j++ )
    {
      const int idx = j + num_sides * i;

      // Retrieve the ball corresponding to (i,j).
      //
      Ball* const ball = bprep[idx];

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

      // Insert link to neighbor ball with name i.
      //
      lprep += new Link( ball, bprep[pn_idx], color_gray );

      // Insert links to balls with same i that are not neighbors.
      //
      if ( j == i % num_sides )
        for ( int k = 2; k < num_sides-1; k++ )
          lprep += new Link
            ( ball, bprep[ idx + (k+j)%num_sides - j ], color_white );

      if ( i == 0 ) continue;

      // Insert link to ball at (i-1,j).
      //
      lprep +=
```

```
          new Link( ball, bprep[idx-num_sides], color_lsu_official_purple );

        // Insert link to ball at (i-1, j-1 mod num_sides ).
        //
        lprep += new Link( ball, bprep[pn_idx-num_sides], color_green );
      }
}

void
World::ball_setup_2()
{
  pCoor first_pos(13.4,17.8,-9.2);
  const float spacing = distance_relaxed;
  pVect delta_pos = pVect(spacing*0.05,-spacing,0);
  pNorm loc_y = delta_pos;
  pNorm loc_x = pVect(0,0,1);
  pNorm loc_z = cross(loc_y,loc_x);

  // Erase the existing balls and links.
  //
  objects_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) * loc_x
        + 0.5 * spacing * sin(angle) * loc_z;

      truss_info.base_coors += chain_first_pos;
    }

  make_truss(&truss_info);

  // 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;
}

void
World::ball_setup_3()
{
  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.
  //
  objects_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);

  const int idx_center = chain_length / 2 * sides;

  for ( int i=0; i<sides; i++ )
    {
      Truss_Info ti;
      ti.num_units = chain_length / 2;
      const int idx_1 = idx_center + ( i == 0 ? sides - 1 : i - 1 );
      const int idx_2 = idx_center + i;

      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];
      ti.base_coors += b0->position;
      ti.base_coors += b1->position;
      ti.base_coors += b2->position;
      ti.base_coors += b3->position;

      pNorm v_head = cross(b1->position,b2->position,b3->position);
      ti.unit_length = delta_dir.magnitude * v_head;
      make_truss(&ti);
      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);
      links += ti.links;
      balls += ti.balls;

      int tsz = ti.balls.size();

      if ( i == 2 )
        {
          ball_eye = ti.balls[tsz-2];
          ball_down = ti.balls[tsz-1];
        }
      else if ( i == 1 )
        {
          ball_gaze = ti.balls[tsz/2];
        }

    }

  // Insert links to balls at either end.
  //
  head_ball = balls += new Ball;
  head_ball->color = color_green;
  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->color = color_green;
  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;
}


void
World::ball_setup_4()
{
}

void
World::ball_setup_5()
{
}
```

```
 /// Advance Simulation State by delta_t Seconds
//
void
World::time_step_cpu(double delta_t)
{
  time_step_count++;

  // Smoothly move ball in response to user input.
  //
  if ( adj_t_stop )
    {
      const double dt = min(world_time,adj_t_stop) - adj_t_prev;
      pVect adj = dt/adj_duration * adj_vector;
      balls_translate(adj,0);
      adj_t_prev = world_time;
      if ( world_time >= adj_t_stop ) adj_t_stop = 0;
    }

  for ( BIter ball(balls); ball; )
    ball->force = ball->mass * gravity_accel;

  for ( LIter link(links); link; )
    {
      // Spring Force from Neighbor Balls
      //
      Ball* const ball1 = link->ball1;
      Ball* const ball2 = link->ball2;

      // Construct a normalized (Unit) Vector from ball to neighbor.
      //
      pNorm ball_to_neighbor(ball1->position,ball2->position);

      const float distance_between_balls = ball_to_neighbor.magnitude;

      // Compute the speed of ball towards neighbor_ball.
      //
      pVect delta_v = ball2->velocity - ball1->velocity;
      float delta_s = dot( delta_v, ball_to_neighbor );

      // Compute by how much the spring is stretched (positive value)
      // or compressed (negative value).
      //
      const float spring_stretch =
        distance_between_balls - link->distance_relaxed;

      // Determine whether spring is gaining energy (whether its length
      // is getting further from its relaxed length).
      //
      const bool gaining_e = ( delta_s > 0.0 ) == ( spring_stretch > 0 );

      // Use a smaller spring constant when spring is loosing energy,
      // a quick and dirty way of simulating energy loss due to spring
      // friction.
      //
      const float spring_constant =
        gaining_e ? opt_spring_constant : opt_spring_constant * 0.7;

      ball1->force += spring_constant * spring_stretch * ball_to_neighbor;
      ball2->force -= spring_constant * spring_stretch * ball_to_neighbor;
    }

  ///
  /// Update Position of Each Ball
  ///
```

```
    for ( BIter ball(balls); ball; )
      {
        if ( ball->locked )
          {
            ball->velocity = pVect(0,0,0);
            continue;
          }

        // Update Velocity
        //
        // This code assumes that force on ball is constant over time
        // step. This is clearly wrong when balls are moving with
        // respect to each other because the springs are changing
        // length. This inaccuracy will make the simulation unstable
        // when spring constant is large for the time step.
        //
        const float mass = max( ball->mass, ball->mass_min );

        pVect delta_v = ( ball->force / mass ) * delta_t;

        const float dist_above = ball->position.y - ball->radius;

        const bool collision =
          platform_collision_possible(ball->position) && dist_above < 0;

        if ( collision )
          {
            const float spring_constant_plat =
              ball->velocity.y < 0 ? 100000 : 50000;
            const float fric_coefficient = 0.1;
            const float force_up = -dist_above * spring_constant_plat;
            const float delta_v_up = force_up / mass * delta_t;
            const float fric_force_mag = fric_coefficient * force_up;
            pNorm surface_v(ball->velocity.x,0,ball->velocity.z);
            const float delta_v_surf = fric_force_mag / mass * delta_t;
            if ( delta_v_surf > surface_v.magnitude )
              {
                // Ignoring other forces?
                delta_v =
                  pVect(-ball->velocity.x,delta_v.y,-ball->velocity.z);
              }
            else
              {
                delta_v -= delta_v_surf * surface_v;
              }
            delta_v.y += delta_v_up;
          }

        ball->velocity += delta_v;

        // Air Resistance
        //
        const double fs = pow(1+opt_air_resistance,-delta_t);
        ball->velocity *= fs;

        // Update Position
        //
        // Assume that velocity is constant.
        //

        /// SOLUTION -- Problem 1a
        //
        //  Remember previous ball position, used for scuff mark.
        //
```

```
pCoor pos_prev = ball->position;

ball->position += ball->velocity * delta_t;

if ( !collision ) continue;

/// Homework 3
//
//  [x] Retrieve the correct overlay.
//  [ ] Determine area of platform that ball is touching.
//  [ ] Convert to texel coordinate units.
//  [ ] Modify texels in array.
//  [ ] Set po->texture_modified iff any texels change.

Platform_Overlay* const po = mp.po_get(ball->position);

if ( !po ) continue;

/// SOLUTION -- Problem 1a.
//
//  Find the location of all of the texels scuffed
//  by this ball during this timestep. The length
//  of the scuff mark is based on the old and new positions,
//  the width of the scuff mark is based on the intersection
//  of the ball with the platform.

// Current ball location in texel coordinates.
//
pCoor ball_lcor = mp.po_get_lcoor(po,ball->position);

// Previous location of ball in texel coordinates.
//
pCoor prev_lcor = mp.po_get_lcoor(po,pos_prev);

float width = mp.scale_x_obj_to_texel *
  sqrt( ball->radius * ball->radius - pos_prev.y * pos_prev.y );

// Direction along scuff mark (based on sliding motion).
//
pNorm skid(prev_lcor,ball_lcor);

// Find direction along width of scuff mark.
//
pNorm nskid = cross( skid, pVect(0,1,0) );

// Iterate over texel locations scuffed by ball.
//
for ( float t = -width; t <= skid.magnitude+width; t++ )
  for ( float u = -width; u < max(width,1.0f); u++ )
  {
    pCoor tex_pos = prev_lcor + t * skid + u * nskid;
    pColor* const texel = mp.po_get_texel(po,tex_pos);

    // If texel is not on overlay po then just skip it.
    //
    if ( !texel ) continue;

    // Don't bother if already scuffed. If we wanted to be
    // fancy we could add this balls color onto what is already
    // there. But we don't want to be fancy.
    //
    if ( texel->a ) continue;

    po->texture_modified = true;
    *texel = ball->color;
```

```
          texel->a = 0.8;
        }

    }
}

bool
World::platform_collision_possible(pCoor pos)
{
  // Assuming no motion in x or z axes.
  //
  return pos.x >= platform_xmin && pos.x <= platform_xmax
    && pos.z >= platform_zmin && pos.z <= platform_zmax;
}

 /// External Modifications to State
//
//   These allow the user to play with state while simulation
//   running.

// Move the ball.
//
void Ball::translate(pVect amt) {position += amt;}

// Add velocity to the ball.
//
void Ball::push(pVect amt) {velocity += amt;}

// Set the velocity to zero.
//
void Ball::stop() {velocity = pVect(0,0,0); }

// Set the velocity and rotation (not yet supported) to zero.
//
void Ball::freeze() {velocity = pVect(0,0,0); }



void World::balls_translate(pVect amt,int b){head_ball->translate(amt);}
void World::balls_push(pVect amt,int b){head_ball->push(amt);}
void World::balls_translate(pVect amt)
{ for ( BIter ball(balls); ball; ) ball->translate(amt); }
void World::balls_push(pVect amt)
{ for ( BIter ball(balls); ball; ) ball->push(amt); }
void World::balls_stop()
{ for ( BIter ball(balls); ball; ) ball->stop(); }
void World::balls_freeze(){balls_stop();}
void World::render_my_piece() {mp.render();}


void
World::frame_callback()
{
  // This routine called whenever window needs to be updated.

  const double time_now = time_wall_fp();

  if ( !opt_pause || opt_single_frame || opt_single_time_step )
    {
      /// Advance simulation state.

      // Amount of time since the user saw the last frame.
```

```
      //
      const double wall_delta_t = time_now - last_frame_wall_time;

      // Compute amount by which to advance simulation state for this frame.
      //
      const double duration =
        opt_single_time_step ? opt_time_step_duration :
        opt_single_frame ? 1/30.0 :
        wall_delta_t;

      const double world_time_target = world_time + duration;

      while ( world_time < world_time_target )
        {
          time_step_cpu(opt_time_step_duration);
          world_time += opt_time_step_duration;
        }

      // Reset these, just in case they were set.
      //
      opt_single_frame = opt_single_time_step = false;
    }

  last_frame_wall_time = time_now;

  if ( opt_ride && ball_eye )
    {
      pNorm b_eye_down(ball_eye->position,ball_down->position);
      pVect b_eye_up = -b_eye_down;
      pCoor eye_pos = ball_eye->position + 2.2 * ball_eye->radius * b_eye_up;
      pNorm b_eye_direction(eye_pos,ball_gaze->position);

      pNorm b_eye_left = cross(b_eye_direction,b_eye_up);
      pMatrix_Translate center_eye(-eye_pos);
      pMatrix rotate; rotate.set_identity();
      for ( int i=0; i<3; i++ ) rotate.rc(0,i) = b_eye_left.elt(i);
      for ( int i=0; i<3; i++ ) rotate.rc(1,i) = b_eye_up.elt(i);
      for ( int i=0; i<3; i++ ) rotate.rc(2,i) = -b_eye_direction.elt(i);
      modelview = rotate * center_eye;
      pMatrix reflect; reflect.set_identity(); reflect.rc(1,1) = -1;
      transform_mirror = modelview * reflect * invert(modelview);
    }

  render();
}

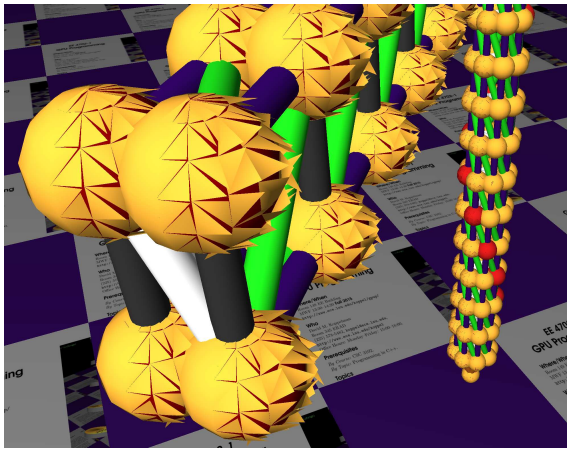
int
main(int argv, char **argc)
{
  pOpenGL_Helper popengl_helper(argv,argc);
  World world(popengl_helper);

  popengl_helper.rate_set(30);
  popengl_helper.display_cb_set(world.frame_callback_w,&world);
}
```

# LSU EE 4702-1 Homework 4 Solution Due: 11 November 2015

Preliminary Solution

The solution has been checked into the repository, look for files that start with hw04-sol. For colorized HTML versions visit `http://www.ece.lsu.edu/koppel/gpup/2015/hw04-sol.cc.html`, `http://www.ece.lsu.edu/koppel/gpup/2015/hw04-sol-shdr.cc.html`, and `http://www.ece.lsu.edu/koppel/gpup/2015/hw04-sol-shdr-links.cc.html`.



**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 start with an 8-arm gyroscope. Press the `+` and `-` keys to change the number of arms. Press digits `1` through `3` to initialize different scenes, the program starts with scene 3.

Pressing 'v' cycles between shaders.

The code in this package includes the friction model asked for in Homework 2. The links however are unbreakable and there's no skin. Also, shadows are deactivated by default to make the solution to this assignment easier.

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.

*Note: There is nothing to turn in for this first problem.*

**Problem 1:** The geometry shader `gs_main_simple` in file `hw04-shdr.cc` passes triangles through unmodified. We know that when rendering spheres and cylinders (at least the cylinders that we use for links) a triangle that's not facing the user will be behind one that is facing the user. It makes sense to discard such triangles early.

(*a*) Modify the geometry shader so that it does not emit triangles based on whether they are facing the user and the value of uniform variable `tri_cull`. If `tri_cull` is zero always emit the triangle. If it is 1, emit the triangle only if the triangle front is facing the user. If `tri_cull` is 2 emit the triangle only if the back is facing the user. Base this on the triangle coordinates, not the normals. The value of `tri_cull` can be changed by pressing `c`.

First, compute the triangle normal in eye space, that's put in variable `gnorm_e` in the code below. The triangle normal should be pointing out of the sphere. Next, compute a vector pointing towards the user. In eye space the user is at the origin, so for an eye-space vertex at $(x, y, z)$ vector $(-x, -y, -z)$ points at the user. The code assigns the dot product of the two vectors to `visibility`. If the dot product is positive then the triangle is facing the user. Finally, the code returns early based on the sign of the dot product and the value of `tri_cull`.

```
// SOLUTION -- Code in file hw04-sol-shdr.cc, routine gs_main_simple();
vec3 gnorm_e = cross
  ( In[1].vertex_e.xyz - In[0].vertex_e.xyz,
```

```
    In[2].vertex_e.xyz - In[1].vertex_e.xyz );

vec3 to_user = -In[0].vertex_e.xyz;

float visibility = dot( gnorm_e, to_user );
if ( tri_cull == 1 && visibility < 0 ) return;
if ( tri_cull == 2 && visibility > 0 ) return;
```

(*b*) Estimate the amount of work saved. Use the following symbols: Let $c_v$, $c_g$, and $c_f$ be the amount of work done by an invocation of the vertex, geometry, and fragment shaders respectively. For example, if we rendered one triangle covering 22 pixels the total work would be $3c_v + c_g + 22c_f$ units. For your answer let $n_t$ denote the number of triangles, and make reasonable choices for other quantities.

Work is saved because we only have to render one half or less of the sphere or cylinder. Since we are culling triangles in the geometry shader we are reducing the amount of work in the fragment shader (since it's after the geometry shader), but we aren't reducing the amount of work performed by the vertex shader (since it's before the geometry shader). We are actually creating more work for the geometry shader since it needs to test whether to cull a triangle.

To determine how much work is saved we need to know the average number of fragments per triangle. Let $n_f$ denote the average number of fragments per triangle. Without culling the fragment shader will be invoked $n_t n_f$ times, performing $n_t n_f c_f$ work. Assuming that we cull half the triangles (which is conservative–we'll usually cull more) and that the average number of fragments per triangle is the same on culled triangles (which is a little unfair, there will usually be less since they are farther away) the amount of work saved for the will be $\frac{1}{2} n_t n_f c_f$.

To see how much of a performance impact that has we need to look at the other shaders. With or without culling the amount of work done by the vertex shader is $n_t c_v$, assuming that triangle strips are used and ignoring the two-vertex startup per sphere. The geometry shader does a little more work to check whether a triangle can be culled, but if a triangle is culled it does less work. Assume that overall, the amount of work does not change. Then the work done is $n_t c_g$ with or without culling.

The absolute amount of work saved is $\frac{1}{2} n_t n_f c_f$. The speedup with culling (the time without culling divided by the time with culling) is given by

$$\frac{n_t c_v + n_t c_g + \frac{1}{2} n_t n_f c_f}{n_t c_v + n_t c_g + n_t n_f c_f}.$$

If $n_f c_f$ is large compared to $c_v$ and $c_g$ a substantial amount of work will be saved.

(*c*) Try to determine actual values for $c_v$, $c_g$, and $c_f$ by running your code. Pause the simulation to make sure the physics isn't hogging CPU. The number of vertices is shown on the green text near the top, the total number is at the end of the line. Triangle strips are used for both the balls and the links. Pressing the `+` and `-` keys when setup 3 is visible will change the number of arms on the gyroscope, use this to vary the number of balls and links.

**Problem 2:** The instance shader code rendering links shows all links the same color.

(*a*) Modify the shaders in `hw04-shdr-links.cc` so that that the fragment shader obtains the color using a value based on `glVertex_ID` rather than reading the color itself from on of its inputs. Only the vertex shader has access to `glVertex_ID`, define shader inputs and outputs as needed to send its value to the fragment shader.

Modify other shaders in this file consistently to remove the old code passing color data through the pipeline.

*Note: The original version of the problem was to use pre-defined fragment shader variable* `glVertex_ID`. *As a student pointed out, there is no such variable in the fragment shader.*

In the unmodified code, `color` travels from the CPU, through the fragment shader, geometry shader, and finally reaching the fragment shader where it is used. It is unmodified in its journey from CPU to fragment shader, meaning that

the vertex and geometry shader pass its value unmodified. The color is type `vec4`, which is 16 bytes. In our solution, rather than passing a 16-byte color through the pipeline we'll pass a 4-byte vertex ID (sort of a link number) through the pipeline, and have the fragment shader look up the color values from an array using the vertex ID as an array index.

To implement this we need to modify both the CPU code, in `hw04.cc`, and the shader code, in `hw04-shdr-links.cc`. (In the solution those file names are `hw04-sol.cc` and `hw04-sol-shdr-links.cc`.)

In the shader code interface blocks all occurrences of `vec4 color` are replaced with `int vtx_id`. For example,

```
// Interface block for vertex shader output / geometry shader input.
out Data_to_GS {
  ivec2 indices;
  /// SOLUTION -- Problem 2
  //  Replace vec4 color with int vtx_id;
  // vec4 color;
  int vtx_id;
};
```

Similar changes would be made to the other interface blocks. The solution above shows `vec4 color` commented out. In real life we would just delete the line and rely on the repositories diff function to tell us what was there before.

In the vertex shader we need to assign `vtx_id`, and in the fragment shader we need to use it. The changed routines are shown below.

```
void vs_main() {
  indices = gl_Vertex;
  vtx_id = gl_VertexID;
}

void fs_main() {
  /// SOLUTION -- Problem 2
  //  Use vtx_id to retrieve color.
  //
  vec4 color = links_color[vtx_id];

  gl_FragColor = generic_lighting(vertex_e, color, normalize(normal_e));
  gl_FragDepth = gl_FragCoord.z;
}
```

The code above reads the color from array `links_color`, which is a buffer object. In the shader code we need to declare the array, providing the data type (`vec4`) and also a binding point (3). The interface block name, `Links_Color` is ignored in this code.

```
 /// SOLUTION -- Problem 2
layout ( binding = 3 ) buffer Links_Color { vec4 links_color[]; };
```

In the CPU code we need to create a buffer object for the colors, prepare an array of link colors, copy the array to the buffer object, and then bind the buffer object to binding point 3 (the number used in the layout declaration above). The added code is shown below with surrounding code removed. In other words the additions below are in the declaration of the `World` class and in routine `render_objects`.

```
  //  In class World declare buffer object name, and array for colors.
  //
  GLuint links_color_bo;
  pColor *links_color;
```

```
// In World::render_objects: Just once, generate a buffer object name.
glGenBuffers(1,&links_color_bo);

// In World::render_objects: Whenever more space is needed, allocate.
delete links_color;
links_color = new pColor[links_size];

// In World::render_objects: Whenever links change, update color array.
if ( link_change ) for ( LIter link(links); link; ) {
  links_indices[link].x = link->ball1->idx;
  links_indices[link].y = link->ball2->idx;
  links_color[link] = link->color;      /// SOLUTION -- Problem 2
}

// In World::render_objects: Whenever links change, update buffer object.
glBindBuffer(GL_ARRAY_BUFFER,links_color_bo);
glBufferData
  (GL_ARRAY_BUFFER,
   links.size() * sizeof(links_color[0]), // Amount of data (bytes) to copy.
   links_color,
   GL_STATIC_DRAW); // Hint about who, when, how accessed.

// In World::render_objects:
//    Just before rendering pass, bind buffer object to point 3.
glBindBufferBase(GL_SHADER_STORAGE_BUFFER,3,links_color_bo);
```

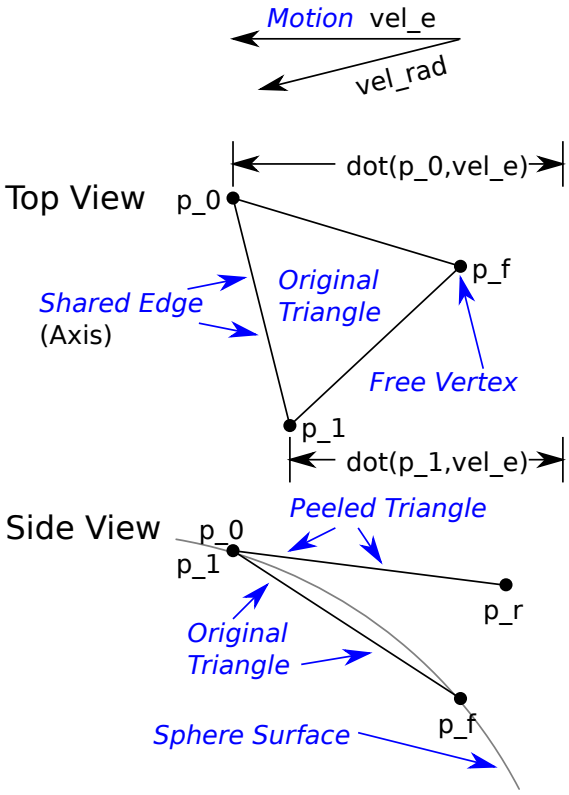
Possible test question: Describe what would happen if each change above (in the solution) were omitted. In particular indicate whether it would be a compile error, a runtime error, or just a wrong answer. For example, if the declaration `GLuint links_color_bo` were omitted there would be a compile error.

**Problem 3:** Modify `gs_main_simple` so that it can emit a second triangle that looks like it is being peeled off a moving sphere by the rushing air. The second triangle should be the same shape as the original triangle and should share one edge with the original triangle. See the illustration at the beginning of this assignment. The choice of shared edge and the angle of the triangle should be based in some way on the velocity of the ball.

There are three parts to this solution: getting velocity values to the shader, identifying the shared edge (the one that remains on the surface of the sphere, and determining a position for the free vertex.

The changes needed to get the ball velocity to the shader are similar to the changes needed to get link colors to the shader in the previous problem. About the only significant difference is that when we rendered links, each link was rendered as a point (the geometry shader converted the "point" into a cylinder), and so the vertex ID could be used to lookup the color. Spheres are rendered using an instanced draw. For an instanced draw the vertex ID indicates a point on the sphere's surface, while the primitive ID indicates which sphere is being rendered. Therefore will use the primitive ID to look up velocity.

The bulk of the solution is in routine `gs_peel`, which is called by `gs_main_simple`. Routine `gs_main_simple` calls `gs_peel` before it renders a triangle on the sphere surface. If `gs_peel` returns true it means that it has emitted a peeled triangle, in which case `gs_main_simple` omits the original triangle colored red. Otherwise, `gs_main_simple` omits the original triangle colored yellow.




Routine `gs_peel` identifies the shared edge by finding the triangle vertex that's furthest downstream of the direction of motion. That's done by taking the dot product of each vertex with the velocity vector and choosing the smallest value (see the illustration above). (To get a feel for how this works consider velocity vector $v = [-2.1, 0, 0]$. The dot product of $v$ with coordinate $[x, y, z]$ is $-2.1x$. The vertex with the largest $x$ component is the furthest downstream.) Since eye-space vertex coordinates are available, the object-space velocity vector read from the array, `vel_o`, is converted to eye space coordinates, `vel_e`. All of that is done by the simplified code below:

```
vec3 vel_o = balls_velocity[In[0].inst_id].xyz;
vec3 vel_e = gl_NormalMatrix * vel_o;

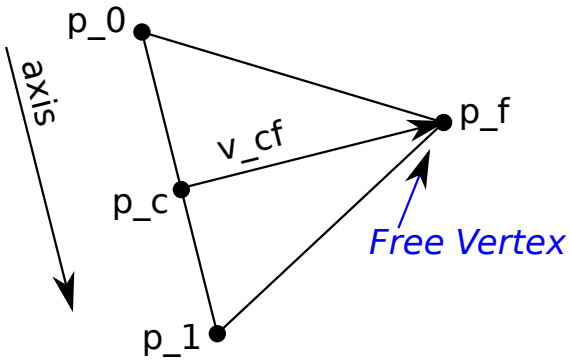
float upstr_amt[3];
for ( int i=0; i<3; i++ ) upstr_amt[i] = dot(In[i].vertex_e.xyz,vel_e);
int min01 = upstr_amt[0] < upstr_amt[1] ? 0 : 1;
int free_i = upstr_amt[min01] < upstr_amt[2] ? min01 : 2;

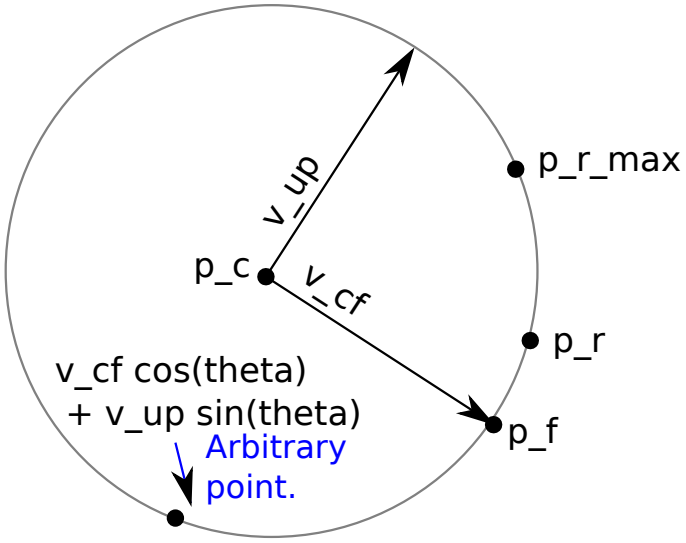
// Get the indices of the other two vertices.
int a0 = ( free_i + 1 ) % 3;
int a1 = ( free_i + 2 ) % 3;

vec3 p_0 = In[a0].vertex_e.xyz;
vec3 p_1 = In[a1].vertex_e.xyz;
vec3 p_f = In[free_i].vertex_e.xyz;
```

Because we don't want the triangle to change shape, the free vertex will be constrained to move in a circle with the center being the closest point on the shared edge to the free vertex, that point is called `p_c` in the code. The shared edge is called `axis`. We need to compute two vectors that we can use to find points on this circle, those vectors are called `v_cf` and `v_up` in the code:

```
vec3 axis = p_1 - p_0;  // Shared edge.
vec3 axis_n = normalize(axis);
vec3 v_0f = p_f - p_0;
vec3 p_c = p_0 + axis_n * dot(axis_n,v_0f);
vec3 v_cf = p_f - p_c;
float len_cf_sq = dot(v_cf,v_cf);
float len_cf = sqrt(len_cf_sq);
vec3 v_up = cross(axis_n,v_cf);
```



```
/// LSU EE 4702-1 (Fall 2015), GPU Programming
//
 /// Homework 4 -- SOLUTION
 //
 //  Use this file for your solution.

 /// Instructions
 //
 //  Read the assignment: http://www.ece.lsu.edu/koppel/gpup/2015/hw04.pdf



// Specify version of OpenGL Shading Language.
//
#version 430 compatibility

// Use this variable to debug your code. Press 'y' to toggle tryout.x
// and 'Y' to toggle debug_bool.y (between true and false).
//
layout ( location = 3 ) uniform bvec2 tryout;


// Array of ball positions.
//
layout ( binding = 1 ) buffer Balls_Pos { vec4 balls_pos_rad[]; };
layout ( binding = 2 ) buffer Balls_Color { vec4 balls_color[]; };

 /// SOLUTION -- Problem 2
layout ( binding = 3 ) buffer Links_Color { vec4 links_color[]; };

#ifdef _VERTEX_SHADER_

in ivec2 gl_Vertex;

// Interface block for vertex shader output / geometry shader input.
//
out Data_to_GS
{
  ivec2 indices;

  /// SOLUTION -- Problem 2
  //
  //  Replace vec4 color with int vtx_id;
  int vtx_id;
};


void
vs_main()
{
  // Here, the vertex shader does nothing except pass variables
  // to the geometry shader.

  indices = gl_Vertex;

  /// SOLUTION -- Problem 2
  //
  //  Transfer vertex id
  vtx_id = gl_VertexID;
}

#endif
```

```
#ifdef _GEOMETRY_SHADER_

in Data_to_GS
{
  ivec2 indices;

  /// SOLUTION -- Problem 2
  //  Replace color with vtx_id.
  int vtx_id;
} In[];

out Data_to_FS
{
  vec3 normal_e;
  vec4 vertex_e;
  /// SOLUTION -- Problem 2
  //  Replace color with vtx_id.
  flat int vtx_id;
};

// Type of primitive at geometry shader input.
//
layout ( points ) in;

// Type of primitives emitted geometry shader output.
//
layout ( triangle_strip, max_vertices = 42 ) out;

void
gs_main_links()
{
  vec4 pos_rad1 = balls_pos_rad[In[0].indices.x];
  vec4 pos_rad2 = balls_pos_rad[In[0].indices.y];
  vec3 pos1 = pos_rad1.xyz;
  vec3 pos2 = pos_rad2.xyz;
  float radius = pos_rad1.w * 0.3;
  vec3 v12 = pos2 - pos1;
  vec3 vna =
    normalize(v12.x == 0 ? vec3(0,-v12.z,v12.y) : vec3(-v12.y,v12.x,0));
  vec3 vnb = normalize(cross(v12,vna));
  mat4 to_obj;
  to_obj[0] = vec4(radius * vna, 0);
  to_obj[1] = vec4(radius * vnb, 0);
  to_obj[2] = vec4(v12, 0);
  to_obj[3] = vec4(pos1,1);
  mat4 mv = gl_ModelViewMatrix * to_obj;
  mat4 mvp = gl_ModelViewProjectionMatrix * to_obj;
  vec4 pos1_e = gl_ModelViewMatrix * vec4(pos1,1);

  const int sides = 20;
  const float M_PI = 3.1415926535f;
  const float delta_theta = 2 * M_PI / sides;

  /// SOLUTION -- Problem 2
  //  Replace color with vtx_id.
  vtx_id = In[0].vtx_id;
  for ( int i=0; i<=sides; i++ )
    {
      const float theta = delta_theta * i;
      const float cos_t = cos(theta);
      const float sin_t = sin(theta);
      for ( int j=0; j<2; j++ )
        {
          vec4 p1 = vec4( cos_t, sin_t, j, 1 );
          vertex_e = mv * p1;
```

```
        if ( j == 0 ) normal_e = vertex_e.xyz - pos1_e.xyz;
        gl_Position = mvp * p1;
        EmitVertex();
      }
    }
  EndPrimitive();
}

#endif


#ifdef _FRAGMENT_SHADER_

in Data_to_FS
{
  vec3 normal_e;
  vec4 vertex_e;
  /// SOLUTION -- Problem 2
  //  Replace color with vtx_id.
  flat int vtx_id;

};

vec4 generic_lighting(vec4 vertex_e, vec4 color, vec3 normal_e);


void
fs_main()
{
  // Perform lighting, fetch and blend texture, then emit fragment.
  //

  /// SOLUTION -- Problem 2
  //  Use vtx_id to retrieve color.
  //
  vec4 color = links_color[vtx_id];

  // Multiply filtered texel color with lighted color of fragment.
  //
  gl_FragColor = generic_lighting(vertex_e, color, normalize(normal_e));

  // Copy fragment depth unmodified.
  //
  gl_FragDepth = gl_FragCoord.z;
}


///
/// Routine used by Either Vertex or Fragment Shader
///

vec4
generic_lighting(vec4 vertex_e, vec4 color, vec3 normal_e)
{
  // Return lighted color of vertex_e.
  //
  vec4 light_pos = gl_LightSource[0].position;
  vec3 v_vtx_light = light_pos.xyz - vertex_e.xyz;
  float d_n_ve = -dot(normal_e,vertex_e.xyz);
  float d_n_vl = dot(normal_e, normalize(v_vtx_light).xyz);
  bool same_sign = ( d_n_ve > 0 ) == ( d_n_vl > 0 );
  float phase_light = same_sign ? abs(d_n_vl) : 0;

  vec3 ambient_light = gl_LightSource[0].ambient.rgb;
```

```
  vec3 diffuse_light = gl_LightSource[0].diffuse.rgb;
  float dist = length(v_vtx_light);
  float distsq = dist * dist;
  float atten_inv =
    gl_LightSource[0].constantAttenuation +
    gl_LightSource[0].linearAttenuation * dist +
    gl_LightSource[0].quadraticAttenuation * distsq;
  vec4 lighted_color;
  lighted_color.rgb =
    color.rgb * gl_LightModel.ambient.rgb
    + color.rgb * ( ambient_light + phase_light * diffuse_light ) / atten_inv;
  lighted_color.a = color.a;
  return lighted_color;
}

#endif
```

```
/// LSU EE 4702-1 (Fall 2015), GPU Programming
//
 /// Homework 4
 //
 //  Use this file for your solution.

 /// Instructions
 //
 //  Read the assignment: http://www.ece.lsu.edu/koppel/gpup/2015/hw04.pdf



// Specify version of OpenGL Shading Language.
//
#version 430 compatibility

layout ( location = 2 ) uniform int tri_cull;

// Use this variable to debug your code. Press 'y' to toggle tryout.x
// and 'Y' to toggle debug_bool.y (between true and false).
//
layout ( location = 3 ) uniform bvec2 tryout;



// Array of ball positions.
//
layout ( binding = 1 ) buffer Balls_Pos { vec4 balls_pos_rad[]; };
layout ( binding = 2 ) buffer Balls_Color { vec4 balls_color[]; };
layout ( binding = 4 ) buffer Balls_Velocity { vec4 balls_velocity[]; };



#ifdef _VERTEX_SHADER_

// Interface block for vertex shader output / geometry shader input.
//
out Data_to_GS
{
  vec3 normal_e;
  vec4 vertex_e;
  vec4 color;
  vec4 gl_Position;

  /// SOLUTION -- Problem 3
  //
  //  Pass instance ID to geometry shader.
  int inst_id;

  // Any changes here must also be made to the fragment shader input.

};


void
vs_main()
{
  // Here, the vertex shader does nothing except pass variables
  // to the geometry shader.

  gl_Position = gl_ModelViewProjectionMatrix * gl_Vertex;
  vertex_e = gl_ModelViewMatrix * gl_Vertex;
  normal_e = gl_NormalMatrix * gl_Vertex.xyz;
  color = gl_Color;
```

```
  /// SOLUTION -- Problem 3
  //
  //  Set instance ID to -1, indicating that this is not a sphere.
  inst_id = -1;
}

void
vs_main_instances()
{
  vec4 center_o_rad = balls_pos_rad[gl_InstanceID];
  float rad = center_o_rad.w;
  vec3 to_surface = rad * gl_Vertex.xyz;
  vec4 position_o = vec4(center_o_rad.xyz + to_surface, 1f);

  gl_Position = gl_ModelViewProjectionMatrix * position_o;
  vertex_e = gl_ModelViewMatrix * position_o;
  normal_e = gl_NormalMatrix * gl_Vertex.xyz;
  color = balls_color[gl_InstanceID];

  /// SOLUTION -- Problem 3
  //
  //  Pass instance ID to geometry shader. This is only done for spheres.
  //
  inst_id = gl_InstanceID;
}

#endif


#ifdef _GEOMETRY_SHADER_

in Data_to_GS
{
  vec3 normal_e;
  vec4 vertex_e;
  vec4 color;
  vec4 gl_Position;
  int inst_id;        /// SOLUTION -- Problem 3
} In[];

out Data_to_FS
{
  vec3 normal_e;
  vec4 vertex_e;
  flat vec4 color;
};

// Type of primitive at geometry shader input.
//
layout ( triangles ) in;

// Type of primitives emitted geometry shader output.
//
 /// SOLUTION -- Problem 3
//   Increase number of vertices to 6 to accommodate peeled triangle.
layout ( triangle_strip, max_vertices = 6 ) out;


 /// SOLUTION -- Problem 3
//   Entire peeled routine.
bool
peeled(vec3 gnorm_e)
{
  // Routine to be called from the geometry shader before emitting a
```

```
// triangle. This routine will, if appropriate, emit a triangle that
// looks like it's peeling off the sphere due to the force of
// rushing air, and return true. Otherwise return false.

// Return if this primitive isn't part of a sphere. (The
// vertex shader sets inst_id, and it knows.)
//
if ( In[0].inst_id < 0 ) return false;

// Since we already have eye-space vertex coordinates, convert
// the velocity to eye space. Note that vectors are transformed
// using a different transformation than coordinates.
//
vec3 vel_o = balls_velocity[In[0].inst_id].xyz;
vec3 vel_e = gl_NormalMatrix * vel_o;

// Return if air presses triangle into sphere.
//
float inward_amt = dot(vel_e,gnorm_e);
if ( inward_amt > 0 ) return false;

// Find the triangle vertex that is furthest downstream in the
// direction of motion.
//
float upstr_amt[3];
for ( int i=0; i<3; i++ ) upstr_amt[i] = dot(In[i].vertex_e.xyz,vel_e);
int min01 = upstr_amt[0] < upstr_amt[1] ? 0 : 1;
int free_i = upstr_amt[min01] < upstr_amt[2] ? min01 : 2;

// Get the indices of the other two vertices.
//
int a0 = ( free_i + 1 ) % 3;
int a1 = ( free_i + 2 ) % 3;

// If the downstream vertex (free_i) is too close to either of the
// other two vertices, return. This happens when the sphere is
// moving slowly or not at all.
//
if ( abs(upstr_amt[free_i] - upstr_amt[a0]) < .01 ) return false;
if ( abs(upstr_amt[free_i] - upstr_amt[a1]) < .01 ) return false;

// Get the vertices' coordinates.
//
vec3 p_0 = In[a0].vertex_e.xyz;
vec3 p_1 = In[a1].vertex_e.xyz;
vec3 p_f = In[free_i].vertex_e.xyz;

// Note: edge 01 (axis) doesn't move.
//
vec3 axis = p_1 - p_0;
vec3 axis_n = normalize(axis);

// Find the velocity component that's not along the axis, vel_rad.
// It is vel_rad that will cause the triangle to swing along axis.
//
float vel_axis = dot(vel_e,axis_n);
vec3 vel_rad = vel_e - vel_axis * axis_n;
float vel_rad_mag = length(vel_rad);
vec3 vel_rad_n = vel_rad/vel_rad_mag;

// If axis is fixed and the triangle can't change its shape,
// the free vertex will move in a circle. Find two vectors
// orthogonal to axis, v_cf and v_up, that can be used to find
// points on this circle. (See the line assigning p_r.)
//
```

```
  vec3 v_0f = p_f - p_0;
  vec3 p_c = p_0 + axis_n * dot(axis_n,v_0f);
  vec3 v_cf = p_f - p_c;
  float len_cf_sq = dot(v_cf,v_cf);
  float len_cf = sqrt(len_cf_sq);
  vec3 v_up = cross(axis_n,v_cf);

  // Compute the position of the free vertex that aligns the triangle
  // with the direction of motion (as though it were pushed by the
  // rushing air [ignoring the effect of the sphere shape on air
  // flow]).
  //
  vec3 p_r_max = p_c - len_cf * vel_rad_n;
  vec3 v_cr_max = p_r_max - p_c;

  // Compute the angle that triangle 01f makes with 01rmax.
  //
  float bend = dot(v_cf,v_cr_max) / len_cf_sq;
  float angle_max = acos(bend);

  // Scale that angle based on the velocity.
  //
  float stiffness = 10;
  float angle = ( 1 - stiffness/(stiffness+vel_rad_mag) ) * angle_max;

  // Compute the position of the free vertex based upon the magnitude
  // of the velocity.
  //
  vec3 p_r = p_c + v_cf * cos(angle) + v_up * sin(angle);

  // Collect the vertices in a convenient array.
  //
  vec3 vertices[3] = { p_0, p_1, p_r };

  // The triangle should look like it's peeling off the
  // sphere. Therefore, use the sphere normals for the two fixed
  // vertices (p_0 and p_1), and the normal of triangle 01r for the
  // free vertex (p_r).
  //
  vec3 normal_r_e = normalize(cross(axis,p_r-p_0));
  vec3 normals[3] = { In[a0].normal_e, In[a1].normal_e, normal_r_e };

  color = In[0].color;  // We know that all vertices are the same color.
  for ( int i=0; i<3; i++ )
    {
      normal_e = normals[i];
      vertex_e = vec4(vertices[i],1);
      gl_Position = gl_ProjectionMatrix * vertex_e;
      EmitVertex();
    }
  EndPrimitive();

  return true;
}


void
gs_main_simple()
{
  /// Problem 1 and 3 Solution Here.
  //
  //  Behavior:
  //    When try_cull = 0, emit all triangles.
  //    When try_cull = 1, only emit triangles facing user.
  //    When try_cull = 2, only emit triangles not facing user.
```

```
  /// SOLUTION Problem 1

  // Compute the normal based on the triangle coordinates (geometric
  // norm) rather than using the vertex normal.
  //
  vec3 gnorm_e = cross
    ( In[1].vertex_e.xyz - In[0].vertex_e.xyz,
      In[2].vertex_e.xyz - In[1].vertex_e.xyz );

  // Compute the vector to the user. This is easy in eye space because
  // the user is at the origin.
  //
  vec3 to_user = -In[0].vertex_e.xyz;

  // Determine if the triangle is facing the user.
  float visibility = dot( gnorm_e, to_user );
  if ( tri_cull == 1 && visibility < 0 ) return;
  if ( tri_cull == 2 && visibility > 0 ) return;

  /// SOLUTION -- Problem 3
  //
  //  Call routine peeled to emit a peeled triangle (if any). If the
  //  return value is true then a peeled triangle was emitted.
  //
  bool peeled_tri = peeled(gnorm_e);

  for ( int i=0; i<3; i++ )
    {
      normal_e = In[i].normal_e;
      vertex_e = In[i].vertex_e;

      /// SOLUTION -- Problem 3
      //  If this triangle has peeled, then use dark red for the
      //  surface of the sphere.
      color = peeled_tri ? vec4(0.5,0,0,1) : In[i].color;
      gl_Position = In[i].gl_Position;
      EmitVertex();
    }
  EndPrimitive();
}



#endif


#ifdef _FRAGMENT_SHADER_

in Data_to_FS
{
  vec3 normal_e;
  vec4 vertex_e;
  flat vec4 color;

};

vec4 generic_lighting(vec4 vertex_e, vec4 color, vec3 normal_e);


void
fs_main()
{
  // Perform lighting, fetch and blend texture, then emit fragment.
  //
```

```
  // Multiply filtered texel color with lighted color of fragment.
  //
  gl_FragColor = generic_lighting(vertex_e, color, normalize(normal_e));

  // Copy fragment depth unmodified.
  //
  gl_FragDepth = gl_FragCoord.z;
}


///
/// Routine used by Either Vertex or Fragment Shader
///

vec4
generic_lighting(vec4 vertex_e, vec4 color, vec3 normal_e)
{
  // Return lighted color of vertex_e.
  //
  vec4 light_pos = gl_LightSource[0].position;
  vec3 v_vtx_light = light_pos.xyz - vertex_e.xyz;
  float d_n_ve = -dot(normal_e,vertex_e.xyz);
  float d_n_vl = dot(normal_e, normalize(v_vtx_light).xyz);
  bool same_sign = ( d_n_ve > 0 ) == ( d_n_vl > 0 );
  float phase_light = same_sign ? abs(d_n_vl) : 0;

  vec3 ambient_light = gl_LightSource[0].ambient.rgb;
  vec3 diffuse_light = gl_LightSource[0].diffuse.rgb;
  float dist = length(v_vtx_light);
  float distsq = dist * dist;
  float atten_inv =
    gl_LightSource[0].constantAttenuation +
    gl_LightSource[0].linearAttenuation * dist +
    gl_LightSource[0].quadraticAttenuation * distsq;
  vec4 lighted_color;
  lighted_color.rgb =
    color.rgb * gl_LightModel.ambient.rgb
    + color.rgb * ( ambient_light + phase_light * diffuse_light ) / atten_inv;
  lighted_color.a = color.a;
  return lighted_color;
}

#endif
```

```
/// LSU EE 4702-1 (Fall 2015), GPU Programming
//
 /// Homework 4 -- SOLUTION
 //

 /// Instructions
 //
 //  Assignment: http://www.ece.lsu.edu/koppel/gpup/2015/hw04.pdf
 //  Solution:   http://www.ece.lsu.edu/koppel/gpup/2015/hw04_sol.pdf

/// Purpose
//
//   Demonstrate simulation of point masses connected by springs.

/// What Code Does

// Simulates balls connected by springs over a platform. Balls and
// springs can be initialized in different arrangements (called
// scenes). Currently scene 1 is a simple string of beads, and scenes
// 2, 3, and 4 are trusses. The platform consists of tiles, some are
// purple-tinted mirrors (showing a reflection of the ball), the
// others show the course syllabus.


///  Keyboard Commands
 //
 /// Object (Eye, Light, Ball) Location or Push
 //   Arrows, Page Up, Page Down
 //        Move object or push ball, depending on mode.
 //        With shift key pressed, motion is 5x faster.
 //   'e': Move eye.
 //   'l': Move light.
 //   'b': Move head (first) ball. (Change position but not velocity.)
 //   'B': Push head ball. (Add velocity.)
 //
 /// Eye Direction
 //   Home, End, Delete, Insert
 //   Turn the eye direction.
 //   Home should rotate eye direction up, End should rotate eye
 //   down, Delete should rotate eye left, Insert should rotate eye
 //   right.  The eye direction vector is displayed in the upper left.

 /// Simulation Options
 //  (Also see variables below.)
 //
 //  'v'    Cycle through display of links/balls (SKEL) and triangles (SKIN).
 //  'w'    Twirl balls around axis formed by head and tail. (Prob 2 soln).
 //  '1'    Set up scene 1.
 //  '2'    Set up scene 2.
 //  '3'    Set up scene 3.
 //  'p'    Pause simulation. (Press again to resume.)
 //  ' '    (Space bar.) Advance simulation by 1/30 second.
 //  'S- '  (Shift-space bar.) Advance simulation by one time step.
 //  'h'    Freeze position of first (head) ball. (Press again to release.)
 //  't'    Freeze position of last (tail) ball. (Press again to release.)
 //  's'    Stop balls.
 //  'g'    Turn gravity on and off.
 //  'y'    Toggle value of opt_tryout1. Intended for experiments and debugging.
 //  'Y'    Toggle value of opt_tryout2. Intended for experiments and debugging.
```

```
//  'F11'  Change size of green text (in upper left).
//  'F12'  Write screenshot to file.

/// Variables
//   Selected program variables can be modified using the keyboard.
//   Use "Tab" to cycle through the variable to be modified, the
//   name of the variable is displayed next to "VAR" on the bottom
//   line of green text.

//  'Tab' Cycle to next variable.
//  '`'   Cycle to previous variable.
//  '+'   Increase variable value.
//  '-'   Decrease variable value.
//
//  VAR Spring Constant - Set spring constant.
//  VAR Time Step Duration - Set physics time step.
//  VAR Air Resistance - Set air resistance.
//  VAR Light Intensity - The light intensity.
//  VAR Gravity - Gravitational acceleration. (Turn on/off using 'g'.)


#define GL_GLEXT_PROTOTYPES
#define GLX_GLXEXT_PROTOTYPES

#include <GL/gl.h>
#include <GL/glext.h>
#include <GL/glx.h>
#include <GL/glxext.h>
#include <GL/glu.h>
#include <GL/freeglut.h>

#include <gp/util.h>
#include <gp/glextfuncs.h>
#include <gp/coord.h>
#include <gp/shader.h>
#include <gp/pstring.h>
#include <gp/misc.h>
#include <gp/gl-buffer.h>
#include <gp/texture-util.h>
#include <gp/colors.h>

#include "util-containers.h"
#include "shapes.h"


///
/// Main Data Structures
///
//
// class World: All data about scene.


class World;


// Object Holding Ball State
//
class Ball {
public:
  Ball():velocity(pVect(0,0,0)),locked(false),
         color(color_lsu_spirit_gold),contact(false){};
  pCoor position;
  pVect velocity;
  int idx; // Position in balls_pos_rad;
```

```
  float mass;
  float mass_min; // Mass below which simulation is unstable.
  float radius;

  bool locked;

  pVect force;
  pColor color;
  bool contact;                 // When true, ball rendered in gray.
  float spring_constant_sum;    // Used to compute minimum mass.

  void push(pVect amt);
  void translate(pVect amt);
  void stop();
  void freeze();
};

class Link {
public:
  Link(Ball *b1, Ball *b2):ball1(b1),ball2(b2),
     distance_relaxed(pDistance(b1->position,b2->position)), snapped(false),
     natural_color(color_lsu_spirit_purple),color(color_lsu_spirit_purple){}
  Link(Ball *b1, Ball *b2, pColor colorp):ball1(b1),ball2(b2),
     distance_relaxed(pDistance(b1->position,b2->position)), snapped(false),
     natural_color(colorp),color(colorp){}
  Ball* const ball1;
  Ball* const ball2;
  float distance_relaxed;
  bool snapped;
  pColor natural_color;
  pColor color;
};

// Declare containers and iterators for Balls and Links.
// (See util_container.h.)
//
typedef pVectorI<Link> Links;
typedef pVectorI_Iter<Link> LIter;
typedef pVectorI<Ball> Balls;
typedef pVectorI_Iter<Ball> BIter;


//
//   Use this class to define variables and member functions.
//
class My_Piece_Of_The_World {
public:

  float link_stressed_thd;  // Amount of stretch to be considered stressed.
  float link_snap_thd;      // Amount of stretch needed to snap link.

  pColor color_stressed;
};


struct Truss_Info {

  // See make_truss for a description of what the members are for.

  // Inputs
  PStack<pCoor> base_coors;  // Coordinates of first set of balls.
  pVect unit_length;
  int num_units;
```

```
  // Output
  Balls balls;
  Links links;
};



enum Render_Option { RO_Normally, RO_Simple, RO_Shadow_Volumes };
enum Shader_Option { SO_Fixed, SO_Phong, SO_Instance, SO_ENUM_SIZE };

struct ivec2 { int x; int y; };

class World {
public:
  World(pOpenGL_Helper &fb):ogl_helper(fb){init();}
  void init();
  void init_graphics();
  static void frame_callback_w(void *moi){((World*)moi)->frame_callback();}
  void frame_callback();
  void render();
  void render_objects(Render_Option render_option);
  void render_my_piece();
  void objects_erase();
  void cb_keyboard();
  void modelview_update();

  My_Piece_Of_The_World mp;

  pOpenGL_Helper& ogl_helper;
  pVariable_Control variable_control;
  pFrame_Timer frame_timer;
  double world_time;
  double last_frame_wall_time;
  float opt_time_step_duration;
  int time_step_count;
  float opt_gravity_accel;      // Value chosen by user.
  pVect gravity_accel;          // Set to zero when opt_gravity is false;
  bool opt_gravity;
  bool opt_head_lock, opt_tail_lock;
  bool opt_ride; // When true, move eye to ball_eye.
  int opt_tri_cull;  // =0, don't cull; =1, cull back, =2, cull front.
  bool opt_tryout1, opt_tryout2;  // For ad-hoc experiments.
  int opt_sides;  // Number of arms on thing in scene 3.

  bool opt_ball_texture;

  // Tiled platform for ball.
  //
  float platform_xmin, platform_xmax, platform_zmin, platform_zmax;
  float platform_pi_xwidth_inv;
  pBuffer_Object<pVect> platform_tile_coords;
  pBuffer_Object<float> platform_tex_coords;
  pVect platform_normal;
  GLuint texid_syl;
  GLuint texid_emacs;
  bool opt_platform_texture;
  void platform_update();
  bool platform_collision_possible(pCoor pos);

  pCoor light_location;
  float opt_light_intensity;
```

```
enum { MI_Eye, MI_Light, MI_Ball, MI_Ball_V, MI_COUNT } opt_move_item;
bool opt_pause;
bool opt_single_frame;      // Simulate for one frame.
bool opt_single_time_step;  // Simulate for one time step.
int viewer_shadow_volume;

pCoor eye_location;
pVect eye_direction;
pMatrix modelview;
pMatrix transform_mirror;

pVect adj_vector;
double adj_t_prev;
double adj_t_stop;
double adj_duration;

int opt_shader;
bool opt_shadows;
pShader *sp_fixed;          // Fixed functionality.
pShader *sp_phong;          // Phong shading.
pShader *sp_instances;
pShader *sp_links;

GLuint balls_pos_rad_bo;
GLuint balls_color_bo;
GLuint links_indices_bo;

/// SOLUTION -- Problem 2
//
//  Variables used for transferring link colors to OpenGL.
GLuint links_color_bo;
pColor *links_color;

/// SOLUTION -- Problem 3
//
//  Variables for transferring ball velocity to OpenGL.
pCoor *balls_velocity;    // Note: Alignment problems with 3-elt type.
GLuint balls_velocity_bo;

pColor *balls_color;
pCoor *balls_pos_rad;
ivec2 *links_indices;
size_t balls_size;
size_t links_size;
int last_setup; // Last scene set up.
bool link_change;

void ball_setup_1();
void ball_setup_2();
void ball_setup_3();
void ball_setup_4();
void ball_setup_5();
void time_step_cpu(double);
void balls_stop();
void balls_freeze();
void balls_translate(pVect amt, int idx);
void balls_translate(pVect amt);
void balls_push(pVect amt, int idx);
void balls_push(pVect amt);
void balls_twirl();
void lock_update();
```

```
  Ball *make_marker(pCoor pos, pColor col);
  void make_truss(Truss_Info *ti);

  Ball *ball_eye, *ball_gaze, *ball_down;

  float opt_spring_constant;
  float opt_air_resistance;
  float distance_relaxed;
  int chain_length;
  Balls balls;
  Ball *head_ball, *tail_ball;
  Links links;
  Sphere sphere;
  Cylinder cyl;
};


void
World::init_graphics()
{
  ///
  /// Graphical Model Initialization
  ///

  balls_pos_rad = NULL;
  balls_pos_rad_bo = 0;
  balls_color = NULL;
  balls_color_bo = 0;
  balls_size = 0;
  links_size = 0;
  links_indices = NULL;
  links_indices_bo = 0;
  link_change = true;

  opt_platform_texture = true;
  opt_head_lock = false;
  opt_tail_lock = false;
  opt_ball_texture = false;
  opt_tryout1 = opt_tryout2 = false;

  opt_sides = 8;

  eye_location = pCoor(24.2,11.6,-38.7);
  eye_direction = pVect(-0.42,-0.09,0.9);

  platform_xmin = -40; platform_xmax = 40;
  platform_zmin = -40; platform_zmax = 40;
  texid_syl = pBuild_Texture_File("gpup.png",false,255);
  texid_emacs = pBuild_Texture_File("mult.png", false,-1);

  opt_light_intensity = 100.2;
  light_location = pCoor(platform_xmax,platform_xmax,platform_zmin);

  variable_control.insert(opt_light_intensity,"Light Intensity");

  opt_move_item = MI_Eye;
  opt_pause = false;
  opt_single_time_step = false;
  opt_single_frame = false;

  sphere.init(15);

  platform_update();
  modelview_update();
```

```
  adj_t_stop = 0;
  adj_duration = 0.25;

}


void
World::platform_update()
{
  const float tile_count = 19;
  const float ep = 1.00001;
  const float delta_x = ( platform_xmax - platform_xmin ) / tile_count * ep;
  const float zdelta = ( platform_zmax - platform_zmin ) / tile_count * ep;

  const float trmin = 0.05;
  const float trmax = 0.7;
  const float tsmin = 0;
  const float tsmax = 0.4;

  platform_normal = pVect(0,1,0);

  PStack<pVect> p_tile_coords;
  PStack<pVect> p1_tile_coords;
  PStack<float> p_tex_coords;
  bool even = true;

  for ( int i = 0; i < tile_count; i++ )
    {
      const float x0 = platform_xmin + i * delta_x;
      const float x1 = x0 + delta_x;
      const float y = 0;
      for ( float z = platform_zmin; z < platform_zmax; z += zdelta )
        {
          PStack<pVect>& t_coords = even ? p_tile_coords : p1_tile_coords;
          p_tex_coords += trmax; p_tex_coords += tsmax;
          t_coords += pVect(x0,y,z);
          p_tex_coords += trmax; p_tex_coords += tsmin;
          t_coords += pVect(x0,y,z+zdelta);
          p_tex_coords += trmin; p_tex_coords += tsmin;
          t_coords += pVect(x1,y,z+zdelta);
          p_tex_coords += trmin; p_tex_coords += tsmax;
          t_coords += pVect(x1,y,z);
          even = !even;
        }
    }

  while ( pVect* const v = p1_tile_coords.iterate() ) p_tile_coords += *v;

  platform_tile_coords.re_take(p_tile_coords);
  platform_tile_coords.to_gpu();
  platform_tex_coords.re_take(p_tex_coords);
  platform_tex_coords.to_gpu();
}

void
World::modelview_update()
{
  pMatrix_Translate center_eye(-eye_location);
  pMatrix_Rotation rotate_eye(eye_direction,pVect(0,0,-1));
  modelview = rotate_eye * center_eye;
  pMatrix reflect; reflect.set_identity(); reflect.rc(1,1) = -1;
  transform_mirror = modelview * reflect * invert(modelview);
}

void
```

```
World::render_objects(Render_Option option)
{
  const float shininess_ball = 5;
  pColor spec_color(0.2,0.2,0.2);
  const bool show_links = true;

  if ( option == RO_Shadow_Volumes )
    viewer_shadow_volume = 0;

  if ( option == RO_Normally )
    {
      glLightf(GL_LIGHT0, GL_LINEAR_ATTENUATION, 1.0);
      if ( opt_ball_texture )
        {
          glEnable(GL_TEXTURE_2D);
          glBindTexture(GL_TEXTURE_2D,texid_emacs);
        }
      else
        {
          glDisable(GL_TEXTURE_2D);
        }
      glMaterialf(GL_FRONT_AND_BACK,GL_SHININESS,shininess_ball);
      glMaterialfv(GL_FRONT_AND_BACK,GL_SPECULAR,spec_color);
      glLightModeli(GL_LIGHT_MODEL_COLOR_CONTROL, GL_SEPARATE_SPECULAR_COLOR);
      glEnable(GL_COLOR_SUM);
    }

  cyl.apex_radius = 1; cyl.set_color(color_lsu_spirit_purple);
  if ( option == RO_Shadow_Volumes )
    {
#ifdef DB_SV
      glLightf(GL_LIGHT0, GL_LINEAR_ATTENUATION, 1.0);
      glEnable(GL_COLOR_SUM);
      glColor3f(0.5,0,0);
#endif
      cyl.light_pos = light_location;
      sphere.light_pos = light_location;

      for ( BIter ball(balls); ball; )
        sphere.render_shadow_volume(ball->radius,ball->position);

      if ( show_links )
        for ( LIter link(links); link; )
          {
            if ( link->snapped ) continue;
            Ball *const ball1 = link->ball1;
            Ball *const ball2 = link->ball2;
            cyl.render_shadow_volume
              (ball1->position,0.3*ball1->radius,
               ball2->position-ball1->position);
          }
    }
  else
    {
      sphere.opt_texture = opt_ball_texture;

      if ( opt_shader != SO_Instance )
        {
          if ( opt_shader == SO_Phong )
            sp_phong->use();
          else if ( opt_shader == SO_Fixed )
            sp_fixed->use();

          if ( opt_shader != SO_Fixed )
            {
```

```
        glUniform1i(2, opt_tri_cull);
        glUniform2i(3, opt_tryout1, opt_tryout2);
      }

    for ( BIter ball(balls); ball; )
      {
        if ( ball->contact )
          sphere.color = color_gray;
        else if ( ball->mass > 0 && ball->mass < ball->mass_min )
          sphere.color = color_red;
        else if ( ball->mass > 0 && ball->locked )
          sphere.color = color_pale_green;
        else
          sphere.color = ball->color;
        sphere.render(ball->radius,ball->position);
      }
    if ( show_links )
      for ( LIter link(links); link; )
        {
          if ( link->snapped ) continue;
          Ball *const ball1 = link->ball1;
          Ball *const ball2 = link->ball2;
          cyl.set_color(link->color);
          cyl.render(ball1->position,0.3*ball1->radius,
                     ball2->position-ball1->position);
        }
    sp_fixed->use();
  }
else
  {
    sp_instances->use();

    if ( balls_pos_rad_bo == 0 )
      {
        glGenBuffers(1,&balls_pos_rad_bo);
        glGenBuffers(1,&balls_color_bo);
        glGenBuffers(1,&links_indices_bo);

        /// SOLUTION -- Problem 2
        glGenBuffers(1,&links_color_bo);

        /// SOLUTION -- Problem 3
        glGenBuffers(1,&balls_velocity_bo);
      }

    if ( balls_size < balls.size() )
      {
        delete balls_pos_rad;
        delete balls_color;
        balls_pos_rad = new pCoor[balls.size()];

        /// SOLUTION -- Problem 2
        balls_color = new pColor[balls.size()];

        /// SOLUTION -- Problem 3
        //  Note that a four-component type, pCoor, used even
        //  though velocity has three components.
        balls_velocity = new pCoor[balls.size()];

        balls_size = balls.size();
      }

    if ( links_size < links.size() )
      {
        delete links_indices;
```

```
    links_size = links.size();
    links_indices = new ivec2[links_size];

    /// SOLUTION -- Problem 2
    links_color = new pColor[links_size];
  }

int color_change = 0;

for ( BIter ball(balls); ball; )
  {
    pColor color;
    if ( ball->contact )
      color = color_gray;
    else if ( ball->mass > 0 && ball->mass < ball->mass_min )
      color = color_red;
    else if ( ball->mass > 0 && ball->locked )
      color = color_pale_green;
    else
      color = ball->color;
    if ( balls_color[ball] != color ) color_change++;
    ball->idx = ball; // Position of ball in balls.
    balls_color[ball->idx] = color;
    pCoor pos_rad = ball->position;
    pos_rad.w = ball->radius;
    balls_pos_rad[ball->idx] = pos_rad;

    /// SOLUTION -- Problem 3
    balls_velocity[ball->idx] = ball->velocity;
  }
if ( link_change )
  for ( LIter link(links); link; )
    {
      links_indices[link].x = link->ball1->idx;
      links_indices[link].y = link->ball2->idx;
      /// SOLUTION -- Problem 2
      links_color[link] = link->color;
    }

GE();
glBindBuffer(GL_SHADER_STORAGE_BUFFER,balls_pos_rad_bo);
glBufferData
  (GL_SHADER_STORAGE_BUFFER,
   // Amount of data (bytes) to copy.
   balls.size() * sizeof(balls_pos_rad[0]),
   balls_pos_rad,
   GL_DYNAMIC_DRAW); // Hint about who, when, how accessed.

/// SOLUTION -- Problem 3
glBindBuffer(GL_SHADER_STORAGE_BUFFER,balls_velocity_bo);
glBufferData
  (GL_SHADER_STORAGE_BUFFER,
   // Amount of data (bytes) to copy.
   balls.size() * sizeof(balls_velocity[0]),
   balls_velocity,
   GL_DYNAMIC_DRAW); // Hint about who, when, how accessed.

if ( link_change )
  {
    glBindBuffer(GL_ARRAY_BUFFER,links_indices_bo);
    glBufferData
      (GL_ARRAY_BUFFER,
       // Amount of data (bytes) to copy.
       links.size() * sizeof(links_indices[0]),
       links_indices,
```

```
                GL_STATIC_DRAW); // Hint about who, when, how accessed.

              /// SOLUTION -- Problem 2
              glBindBuffer(GL_ARRAY_BUFFER,links_color_bo);
              glBufferData
                (GL_ARRAY_BUFFER,
                 // Amount of data (bytes) to copy.
                 links.size() * sizeof(links_color[0]),
                 links_color,
                 GL_STATIC_DRAW); // Hint about who, when, how accessed.
            }
          if ( color_change )
            {
              glBindBuffer(GL_SHADER_STORAGE_BUFFER,balls_color_bo);
              glBufferData
                (GL_SHADER_STORAGE_BUFFER,
                 // Amount of data (bytes) to copy.
                 balls.size() * sizeof(balls_color[0]),
                 balls_color,
                 GL_STATIC_DRAW); // Hint about who, when, how accessed.
            }
          GE();

          glUniform1i(2, opt_tri_cull);
          glUniform2i(3, opt_tryout1, opt_tryout2);

          glBindBufferBase(GL_SHADER_STORAGE_BUFFER,1,balls_pos_rad_bo);
          glBindBufferBase(GL_SHADER_STORAGE_BUFFER,2,balls_color_bo);

          /// SOLUTION -- Problem 3
          glBindBufferBase(GL_SHADER_STORAGE_BUFFER,4,balls_velocity_bo);

          sphere.points_bo.bind();
          glVertexPointer(3,GL_FLOAT,0,0);
          glEnableClientState(GL_VERTEX_ARRAY);
          GE();
          glDrawArraysInstanced
            (GL_TRIANGLE_STRIP,0,sphere.points_bo.elements,balls);
          GE();

          glBindBuffer(GL_ARRAY_BUFFER,0);
          glDisableClientState(GL_VERTEX_ARRAY);
          glDisableClientState(GL_TEXTURE_COORD_ARRAY);

          /// Homework 4 Problem 2 Solution Here (and other places).
          sp_links->use();
          link_change = false;

          /// SOLUTION -- Problem 2
          glBindBufferBase(GL_SHADER_STORAGE_BUFFER,3,links_color_bo);

          glBindBuffer(GL_ARRAY_BUFFER,links_indices_bo);
          glVertexPointer(2,GL_INT,0,0);
          glEnableClientState(GL_VERTEX_ARRAY);
          glDrawArrays(GL_POINTS,0,links_size);
          glBindBuffer(GL_ARRAY_BUFFER,0);
          glDisableClientState(GL_VERTEX_ARRAY);
        }

      sp_fixed->use();
    }

  glDisable(GL_COLOR_SUM);
  glLightModeli(GL_LIGHT_MODEL_COLOR_CONTROL, GL_SINGLE_COLOR);
  glLightfv(GL_LIGHT0, GL_SPECULAR, color_black);
```

```
  //
  // Render Platform
  //
  const int half_elements = platform_tile_coords.elements >> 3 << 2;

  glEnable(GL_TEXTURE_2D);

  // Set up attribute (vertex, normal, etc.) arrays.
  //
  glBindTexture(GL_TEXTURE_2D,texid_syl);
  platform_tile_coords.bind();
  glVertexPointer(3, GL_FLOAT, sizeof(platform_tile_coords.data[0]), 0);
  glEnableClientState(GL_VERTEX_ARRAY);
  glNormal3fv(platform_normal);
  platform_tex_coords.bind();
  glTexCoordPointer(2, GL_FLOAT,2*sizeof(float), 0);
  glEnableClientState(GL_TEXTURE_COORD_ARRAY);

  // Write lighter-colored, textured tiles.
  //
  glMaterialfv(GL_FRONT_AND_BACK,GL_SPECULAR,spec_color);
  glMaterialf(GL_FRONT_AND_BACK,GL_SHININESS,2.0);
  glColor3f(0.35,0.35,0.35);
  glColor3f(0.55,0.55,0.55);
  glDrawArrays(GL_QUADS,0,half_elements+4);

  glDisableClientState(GL_VERTEX_ARRAY);
  glDisableClientState(GL_TEXTURE_COORD_ARRAY);
  glBindBuffer(GL_ARRAY_BUFFER,0);
}

void
World::render()
{
  // Get any waiting keyboard commands.
  //
  cb_keyboard();

  // Start a timer object used for tuning this code.
  //
  frame_timer.frame_start();

  /// Emit a Graphical Representation of Simulation State
  //

  // Understanding of the code below not required for introductory
  // lectures.

  // That said, much of the complexity of the code is to show
  // the ball shadow and reflection.


  const int win_width = ogl_helper.get_width();
  const int win_height = ogl_helper.get_height();
  const float aspect = float(win_width) / win_height;

  glMatrixMode(GL_MODELVIEW);
  glLoadTransposeMatrixf(modelview);

  glMatrixMode(GL_PROJECTION);
  glLoadIdentity();
  // Frustum: left, right, bottom, top, near, far
  glFrustum(-.8,.8,-.8/aspect,.8/aspect,1,5000);
```

```
  glViewport(0, 0, win_width, win_height);
  pError_Check();

  glClearColor(0,0,0,0.5);
  glClearDepth(1.0);
  glClearStencil(0);
  glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT | GL_STENCIL_BUFFER_BIT );

  glEnable(GL_DEPTH_TEST);
  glDepthFunc(GL_LESS);
  glDisable(GL_BLEND);

  glLightModeli(GL_LIGHT_MODEL_TWO_SIDE,1);
  glLightfv(GL_LIGHT0, GL_POSITION, light_location);

  glLightf(GL_LIGHT0, GL_CONSTANT_ATTENUATION, 0.5);
  glLightf(GL_LIGHT0, GL_LINEAR_ATTENUATION, 1.0);
  glLightf(GL_LIGHT0, GL_QUADRATIC_ATTENUATION, 0);

  pColor ambient_color(0x555555);

  glLightModelfv(GL_LIGHT_MODEL_AMBIENT, ambient_color);
  glLightfv(GL_LIGHT0, GL_DIFFUSE, color_white * opt_light_intensity);
  glLightfv(GL_LIGHT0, GL_AMBIENT, color_black);
  glLightfv(GL_LIGHT0, GL_SPECULAR, color_white * opt_light_intensity);

  glEnable(GL_LIGHT0);
  glEnable(GL_LIGHTING);

  glEnable(GL_COLOR_MATERIAL);
  glColorMaterial(GL_FRONT_AND_BACK,GL_AMBIENT_AND_DIFFUSE);

  glShadeModel(GL_SMOOTH);

  // Common to all textures.
  //
  glActiveTexture(GL_TEXTURE0);
  glEnable(GL_TEXTURE_2D);
  glTexParameterf(GL_TEXTURE_2D,GL_TEXTURE_MIN_FILTER,
                  GL_LINEAR_MIPMAP_LINEAR);
  glTexParameterf(GL_TEXTURE_2D,GL_TEXTURE_MAG_FILTER,GL_LINEAR);
  glTexEnvi(GL_TEXTURE_ENV,GL_TEXTURE_ENV_MODE,GL_MODULATE);
  glTexParameterf(GL_TEXTURE_2D,GL_TEXTURE_WRAP_S,GL_REPEAT);
  glTexParameterf(GL_TEXTURE_2D,GL_TEXTURE_WRAP_T,GL_REPEAT);

  glEnable(GL_RESCALE_NORMAL);
  glEnable(GL_NORMALIZE);

  const double time_now = time_wall_fp();
  const bool blink_visible = int64_t(time_now*3) & 1;
# define BLINK(txt,pad) ( blink_visible ? txt : pad )

  ogl_helper.fbprintf("%s\n",frame_timer.frame_rate_text_get());

  ogl_helper.fbprintf
    ("Code Compiled: %s\n",
#ifdef __OPTIMIZE__
     "WITH OPTIMIZATION"
#else
     BLINK("WITHOUT OPTIMIZATION","")
#endif
     );

  const int cyl_sides = 20;
  const int cyl_vtx = cyl_sides * 2 + 2;
```

```
  const int cyl_vtx_tot = cyl_vtx * links.size();
  const int ball_vtx_tot = sphere.points_bo.elements * balls.size();
  const int vtx_total = cyl_vtx_tot + ball_vtx_tot;

  ogl_helper.fbprintf
    ("Balls: %3d   Links: %3d  Vtx/Ball: %3d  Vtx/Link: %2d  Total: %d\n",
     balls.size(), links.size(),
     sphere.points_bo.elements,
     cyl_vtx,
     vtx_total);

  ogl_helper.fbprintf
    ("Time Step: %8d  World Time: %11.6f  %s\n",
     time_step_count, world_time,
     opt_pause ? BLINK("PAUSED, 'p' to unpause, SPC or S-SPC to step.","") :
     "Press 'p' to pause."
     );

  ogl_helper.fbprintf
    ("Eye location: [%5.1f, %5.1f, %5.1f]  "
     "Eye direction: [%+.2f, %+.2f, %+.2f]\n",
     eye_location.x, eye_location.y, eye_location.z,
     eye_direction.x, eye_direction.y, eye_direction.z);

  Ball& ball = *balls[0];

  if ( 0 )
  ogl_helper.fbprintf
    ("Head Ball Pos  [%5.1f,%5.1f,%5.1f] Vel [%+5.1f,%+5.1f,%+5.1f]\n",
     ball.position.x,ball.position.y,ball.position.z,
     ball.velocity.x,ball.velocity.y,ball.velocity.z );

  ogl_helper.fbprintf
    ("Shader: %s  ('v' to change)  "
     "Tryout 1: %s  ('y' to change)  Tryout 2: %s  ('Y' to change)\n",
     opt_shader == SO_Fixed ? "FIXED" :
     opt_shader == SO_Phong ? "PLAIN" :
     opt_shader == SO_Instance ? "INSTANCE" : "other",
     opt_tryout1 ? BLINK("ON","  ") : "OFF",
     opt_tryout2 ? BLINK("ON","  ") : "OFF");

  ogl_helper.fbprintf
    ("Cull %d-%s ('c' to change)  "
     "Shadows: %s ('o' to change)\n",
     opt_tri_cull,
     opt_tri_cull == 0 ? "OFF  " : opt_tri_cull == 1 ? "BACK " :
     opt_tri_cull == 2 ? "FRONT" : "???",
     opt_shadows ? "ON" : "OFF" );

  pVariable_Control_Elt* const cvar = variable_control.current;
  ogl_helper.fbprintf("VAR %s = %.5f  (TAB or '`' to change, +/- to adjust)\n",
                      cvar->name,cvar->get_val());

  const int half_elements = platform_tile_coords.elements >> 3 << 2;

  //
  // Render ball reflection.  (Will be blended with dark tiles.)
  //

  // Write stencil at location of dark (mirrored) tiles.
  //
  glDisable(GL_LIGHTING);
  glEnable(GL_STENCIL_TEST);
  glStencilFunc(GL_NEVER,2,2);
  glStencilOp(GL_REPLACE,GL_KEEP,GL_KEEP);
```

```
  platform_tile_coords.bind();
  glVertexPointer(3, GL_FLOAT, sizeof(platform_tile_coords.data[0]), 0);
  glEnableClientState(GL_VERTEX_ARRAY);
  glDrawArrays(GL_QUADS,half_elements+4,half_elements-4);
  glEnable(GL_LIGHTING);
  glDisableClientState(GL_VERTEX_ARRAY);
  glBindBuffer(GL_ARRAY_BUFFER,0);

  // Prepare to write only stenciled locations.
  //
  glStencilFunc(GL_EQUAL,2,2);
  glStencilOp(GL_KEEP,GL_KEEP,GL_KEEP);

  // Use a transform that reflects objects to other side of platform.
  //
  glMatrixMode(GL_PROJECTION);
  glPushMatrix();
  glMultTransposeMatrixf(transform_mirror);

  // Reflected front face should still be treated as the front face.
  //
  glFrontFace(GL_CW);

  render_objects(RO_Normally);

  glFrontFace(GL_CCW);
  glMatrixMode(GL_PROJECTION);
  glPopMatrix();
  glDisable(GL_STENCIL_TEST);


  // Setup texture for platform.
  //
  glBindTexture(GL_TEXTURE_2D,texid_syl);

  // Blend dark tiles with existing ball reflection.
  //
  glEnable(GL_STENCIL_TEST);
  glBlendEquation(GL_FUNC_ADD);
  glBlendFunc(GL_CONSTANT_ALPHA,GL_ONE_MINUS_CONSTANT_ALPHA); // src, dst
  glBlendColor(0,0,0,0.5);

  glDepthFunc(GL_ALWAYS);

  if ( opt_platform_texture )
    {
      glEnable(GL_TEXTURE_2D);
      platform_tex_coords.bind();
      glTexCoordPointer(2, GL_FLOAT,2*sizeof(float), 0);
      glEnableClientState(GL_TEXTURE_COORD_ARRAY);
    }

  platform_tile_coords.bind();
  glVertexPointer
    (3, GL_FLOAT,sizeof(platform_tile_coords.data[0]), 0);
  glEnableClientState(GL_VERTEX_ARRAY);
  glNormal3fv(platform_normal);

  if ( opt_platform_texture ) glEnable(GL_TEXTURE_2D);

  // Write lighter-colored, textured tiles.
  //
  glMaterialfv(GL_FRONT_AND_BACK,GL_SPECULAR,color_gray);
  glMaterialf(GL_FRONT_AND_BACK,GL_SHININESS,2.0);
  glColor3f(0.35,0.35,0.35);
```

```
  glDrawArrays(GL_QUADS,0,half_elements+4);

  // Write darker-colored, untextured, mirror tiles.
  //
  glEnable(GL_BLEND);
  glMaterialfv(GL_FRONT_AND_BACK,GL_SPECULAR,color_white);
  glMaterialf(GL_FRONT_AND_BACK,GL_SHININESS,20);
  glDisable(GL_TEXTURE_2D);
  glColor3fv(color_lsu_spirit_purple);
  glDrawArrays(GL_QUADS,half_elements+4,half_elements-4);
  glDisable(GL_BLEND);

  glDisableClientState(GL_TEXTURE_COORD_ARRAY);
  glDisableClientState(GL_VERTEX_ARRAY);
  glBindBuffer(GL_ARRAY_BUFFER,0);
  glDisable(GL_STENCIL_TEST);
  glDepthFunc(GL_LESS);

  glLightModelfv(GL_LIGHT_MODEL_AMBIENT, ambient_color);

  if ( !opt_shadows )
    {
      // Render.
      //
      render_objects(RO_Normally);
    }
  else
    {
      //
      // First pass, render using only ambient light.
      //
      glDisable(GL_LIGHT0);

      // Send balls, tiles, and platform to opengl.
      // Do occlusion test too.
      //
      render_objects(RO_Normally);

      //
      // Second pass, add on light0.
      //

      // Turn off ambient light, turn on light 0.
      //
      glLightModelfv(GL_LIGHT_MODEL_AMBIENT, color_black);
      glEnable(GL_LIGHT0);


      glClear(GL_STENCIL_BUFFER_BIT);

      // Make sure that only stencil buffer written.
      //
#ifndef DB_SV
      glColorMask(false,false,false,false);
      glDepthMask(false);

      // Don't waste time computing lighting.
      //
      glDisable(GL_LIGHTING);
#endif
      glDisable(GL_TEXTURE_2D);

      // Set up stencil test to count shadow volume surfaces: plus 1 for
      // entering the shadow volume, minus 1 for leaving the shadow
      // volume.
```

```
      //
      glEnable(GL_STENCIL_TEST);
      // sfail, dfail, dpass
      glStencilOpSeparate(GL_FRONT,GL_KEEP,GL_KEEP,GL_INCR_WRAP);
      glStencilOpSeparate(GL_BACK,GL_KEEP,GL_KEEP,GL_DECR_WRAP);
      glStencilFuncSeparate(GL_FRONT_AND_BACK,GL_ALWAYS,1,-1); // ref, mask

      // Write stencil with shadow locations based on shadow volumes
      // cast by light0 (light_location).  Shadowed locations will
      // have a positive stencil value.  Routine will set viewer_shadow_volume
      // to the number of view volumes containing the eye.
      //
      render_objects(RO_Shadow_Volumes);

      glEnable(GL_LIGHTING);
      glColorMask(true,true,true,true);
      glDepthMask(true);

      // Use stencil test to prevent writes to shadowed areas.
      //
      glStencilOp(GL_KEEP,GL_KEEP,GL_KEEP);
      glStencilFunc(GL_EQUAL,viewer_shadow_volume,-1); // ref, mask

      // Allow pixels to be re-written.
      //
      glDepthFunc(GL_LEQUAL);
      glEnable(GL_BLEND);
      glBlendEquation(GL_FUNC_ADD);
      glBlendFunc(GL_ONE,GL_ONE);

      // Render.
      //
      render_objects(RO_Normally);

      glDisable(GL_BLEND);
      glDisable(GL_STENCIL_TEST);

    }


  // Render Marker for Light Source
  //
  insert_tetrahedron(light_location,0.5);

  pError_Check();

  glColor3f(0.5,1,0.5);

  frame_timer.frame_end();

  ogl_helper.user_text_reprint();

  glutSwapBuffers();
}


void
World::cb_keyboard()
{
  if ( !ogl_helper.keyboard_key ) return;
  pVect adjustment(0,0,0);
  pVect user_rot_axis(0,0,0);
  const bool shift = ogl_helper.keyboard_shift;
  const float move_amt = shift ? 2.0 : 0.4;
  const int opt_sides_before = opt_sides;
```

```
switch ( ogl_helper.keyboard_key ) {
case FB_KEY_LEFT: adjustment.x = -move_amt; break;
case FB_KEY_RIGHT: adjustment.x = move_amt; break;
case FB_KEY_PAGE_UP: adjustment.y = move_amt; break;
case FB_KEY_PAGE_DOWN: adjustment.y = -move_amt; break;
case FB_KEY_DOWN: adjustment.z = move_amt; break;
case FB_KEY_UP: adjustment.z = -move_amt; break;
case FB_KEY_DELETE: user_rot_axis.y = 1; break;
case FB_KEY_INSERT: user_rot_axis.y =  -1; break;
case FB_KEY_HOME: user_rot_axis.x = 1; break;
case FB_KEY_END: user_rot_axis.x = -1; break;
case '1': ball_setup_1(); break;
case '2': ball_setup_2(); break;
case '3': ball_setup_3(); break;
case '4': ball_setup_4(); break;
case '5': ball_setup_5(); break;
case 'b': opt_move_item = MI_Ball; break;
case 'B': opt_move_item = MI_Ball_V; break;
case 'c': case 'C':
  opt_tri_cull++; if ( opt_tri_cull == 3 ) opt_tri_cull = 0;
  break;
case 'e': case 'E': opt_move_item = MI_Eye; break;
case 'g': case 'G': opt_gravity = !opt_gravity; break;
case 'h': case 'H': opt_head_lock = !opt_head_lock; break;
case 't': case 'T': opt_tail_lock = !opt_tail_lock; break;
case 'l': case 'L': opt_move_item = MI_Light; break;
case 'n': case 'N': opt_platform_texture = !opt_platform_texture; break;
case 'o': case 'O': opt_shadows = !opt_shadows; break;
case 'p': case 'P': opt_pause = !opt_pause; break;
case 'r': case 'R': opt_ride = !opt_ride; break;
case 's': case 'S': balls_stop(); break;
case 'v': case 'V': opt_shader++; if ( opt_shader > 2 ) opt_shader = 0; break;
case 'w': case 'W': balls_twirl(); break;
case 'y': opt_tryout1 = !opt_tryout1; break;
case 'Y': opt_tryout2 = !opt_tryout2; break;
case ' ':
  if ( shift ) opt_single_time_step = true; else opt_single_frame = true;
  opt_pause = true;
  break;
case 9: variable_control.switch_var_right(); break;
case 96: variable_control.switch_var_left(); break; // `, until S-TAB works.
case '-':case '_': variable_control.adjust_lower(); break;
case '+':case '=': variable_control.adjust_higher(); break;
default: printf("Unknown key, %d\n",ogl_helper.keyboard_key); break;
}

if ( opt_sides_before != opt_sides && last_setup == 3 )
  ball_setup_3();

gravity_accel.y = opt_gravity ? -opt_gravity_accel : 0;

lock_update();

// Update eye_direction based on keyboard command.
//
if ( user_rot_axis.x || user_rot_axis.y )
  {
    pMatrix_Rotation rotall(eye_direction,pVect(0,0,-1));
    user_rot_axis *= invert(rotall);
    eye_direction *= pMatrix_Rotation(user_rot_axis, M_PI * 0.03);
    modelview_update();
  }

// Update eye_location based on keyboard command.
```

```
  //
  if ( adjustment.x || adjustment.y || adjustment.z )
    {
      const double angle =
        fabs(eye_direction.y) > 0.99
        ? 0 : atan2(eye_direction.x,-eye_direction.z);
      pMatrix_Rotation rotall(pVect(0,1,0),-angle);
      adjustment *= rotall;

      switch ( opt_move_item ){
      case MI_Ball:
        adj_vector = adjustment;
        if ( adj_t_stop == 0 )
          adj_t_prev = adj_t_stop = world_time;
        adj_t_stop += adj_duration;
        break;
      case MI_Ball_V: balls_push(adjustment,0); break;
      case MI_Light: light_location += adjustment; break;
      case MI_Eye: eye_location += adjustment; break;
      default: break;
      }
      modelview_update();
    }

}




void
World::init()
{
  mp.color_stressed = pColor(1,0,0); // Red, what else?
  mp.link_stressed_thd = 1.01;
  mp.link_snap_thd = 1.5;

  chain_length = 14;

  opt_sides = 8;
  variable_control.insert(opt_sides,"Scene 3 Object Number of Sides",1,3);

  opt_time_step_duration = 0.0003;
  variable_control.insert(opt_time_step_duration,"Time Step Duration");

  distance_relaxed = 15.0 / chain_length;
  opt_spring_constant = 15000;
  variable_control.insert(opt_spring_constant,"Spring Constant");

  opt_gravity_accel = 9.8;
  opt_gravity = true;
  gravity_accel = pVect(0,-opt_gravity_accel,0);
  variable_control.insert(opt_gravity_accel,"Gravity");

  opt_air_resistance = 0.04;
  variable_control.insert(opt_air_resistance,"Air Resistance");

  world_time = 0;
  time_step_count = 0;
  last_frame_wall_time = time_wall_fp();
  frame_timer.work_unit_set("Steps / s");

  opt_shadows = false;
  opt_shader = SO_Instance;

  ball_eye = NULL;
```

```
  opt_ride = false;

  opt_tri_cull = 0;

  init_graphics();

  // Declared like a programmable shader, but used for fixed-functionality.
  //
  sp_fixed = new pShader();

  sp_phong = new pShader
    ("hw04-sol-shdr.cc",// File holding shader program.
     "vs_main(); ",     // Name of vertex shader main routine.
     "gs_main_simple();",
     "fs_main();"       // Name of fragment shader main routine.
     );
  sp_instances = new pShader
    ("hw04-sol-shdr.cc",// File holding shader program.
     "vs_main_instances(); ",
     "gs_main_simple();",
     "fs_main();"
     );

  sp_links = new pShader
    ("hw04-sol-shdr-links.cc",// File holding shader program.
     "vs_main(); ",     // Name of vertex shader main routine.
     "gs_main_links();",
     "fs_main();"       // Name of fragment shader main routine.
     );

  ball_setup_3();
  lock_update();
}

Ball*
World::make_marker(pCoor position, pColor color)
{
  Ball* const ball = new Ball;
  ball->position = position;
  ball->locked = true;
  ball->velocity = pVect(0,0,0);
  ball->radius = 0.2;
  ball->mass = 0;
  ball->contact = false;
  ball->color = color;
  return ball;
}

void
World::lock_update()
{
  // This routine called when options like opt_head_lock might have
  // changed.

  // Update locked status.
  //
  if ( head_ball ) head_ball->locked = opt_head_lock;
  if ( tail_ball ) tail_ball->locked = opt_tail_lock;

  // Re-compute minimum mass needed for stability.
  //
  for ( BIter ball(balls); ball; ) ball->spring_constant_sum = 0;
  const double dtis = pow( opt_time_step_duration, 2 );
  for ( LIter link(links); link; )
    {
```

```
      Ball* const b1 = link->ball1;
      Ball* const b2 = link->ball2;
      b1->spring_constant_sum += opt_spring_constant;
      b2->spring_constant_sum += opt_spring_constant;
    }
  for ( BIter ball(balls); ball; )
    ball->mass_min = ball->spring_constant_sum * dtis;
}

void
World::balls_twirl()
{
  if ( !head_ball || !tail_ball ) return;

  pNorm axis(head_ball->position, tail_ball->position);

  for ( BIter ball(balls); ball; )
    {
      pVect b_to_top(ball->position,head_ball->position);
      const float dist_along_axis = dot(b_to_top,axis);
      const float lsq = b_to_top.mag_sq() - dist_along_axis * dist_along_axis;
      if ( lsq <= 1e-5 ) { ball->velocity = pVect(0,0,0); continue; }
      const float dist_to_axis = sqrt(lsq);
      pNorm rot_dir = cross(b_to_top,axis);
      ball->velocity += 2 * dist_to_axis * rot_dir;
    }
}

void
World::objects_erase()
{
  ball_eye = NULL;
  link_change = true;
  balls.erase();
  links.erase();
}

///
/// Physical Simulation Code
///

 /// Initialize Simulation
//

void
World::ball_setup_1()
{
  // Arrange and size balls to form a pendulum.
  last_setup = 1;

  pCoor first_pos(13.4,17.8,-9.2);
  pVect delta_pos = pVect(distance_relaxed,0,0);

  // Remove objects from the simulated objects lists, balls and links.
  // The delete operator is used on objects in the lists.
  //
  objects_erase();

  for ( int i=0; i<chain_length; i++ )
    {
      // Construct a new ball and add it to the simulated objects list (balls).
      //
      Ball* const ball = balls += new Ball;
```

```
      // Initialize position and other information.
      //
      ball->position = first_pos + i * delta_pos;
      ball->locked = false;
      ball->velocity = pVect(0,0,0);
      ball->radius = 0.3;
      ball->mass = 4/3.0 * M_PI * pow(ball->radius,3);
      ball->contact = false;

      // If it's not the first ball link it to the previous ball.
      if ( i > 0 ) links += new Link( ball, balls[i-1] );
    }

  // The balls pointed to by head_ball and tail_ball can be manipulated
  // using the user interface (by pressing 'h' or 't', for example).
  // Set these variables.
  //
  head_ball = balls[0];
  tail_ball = balls[balls-1];

  opt_head_lock = true;    // Head ball will be frozen in space.
  opt_tail_lock = false;   // Tail ball can move freely.
}


void
World::make_truss(Truss_Info *truss_info)
{
  /// Construct a truss based on members of truss_info.
  //

  //            <---- num_units (=9) ----------->
  //  j
  //  0         O---O---O---O---O---O---O---O---O     ^
  //            |   |   |   |   |   |   |   |   |     |
  //  1         O---O---O---O---O---O---O---O---O   num_sides (=3)
  //            |   |   |   |   |   |   |   |   |     |
  //  2         O---O---O---O---O---O---O---O---O     v
  //
  //      i ->  0   1   2   3   4   5   6   7   8
  //
  //  Note: Not all links are shown in the diagram above.

  /// Truss_Info Inputs
  //
  //  truss_info->num_units
  //    The number of sections in the truss (see diagram above).
  //
  //  truss_info->base_coors
  //    A list containing num_sides coordinates, the coordinates of
  //    balls at i=0 (see diagram above). (num_sides is the number of
  //    elements in this list.)  These coordinates should all be
  //    in the same plane.
  //
  //  truss_info->unit_length
  //    A vector pointing from the ball at (i,j) to the ball at (i+1,j).
  //
  /// Truss_Info Outputs
  //
  //  truss_info->balls
  //    A list that should be filled with the balls making up the truss.
  //
  //  truss_info->links
  //    A list that should be filled with the links making up the truss.
```

```
  const int num_sides = truss_info->base_coors.occ();
  const int num_units = truss_info->num_units;

  // Lists to hold balls and links created for truss.
  //
  Balls& bprep = truss_info->balls;
  Links& lprep = truss_info->links;

  // Create balls for truss.
  //
  for ( int i=0; i<num_units; i++ )
    for ( pCoor bcoor; truss_info->base_coors.iterate(bcoor); )
      {
        Ball* const ball = bprep += new Ball;
        ball->position = bcoor + i * truss_info->unit_length;
        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;
      }

  // Create links.
  //
  for ( int i=0; i<num_units; i++ )
    for ( int j=0; j<num_sides; j++ )
      {
        const int idx = j + num_sides * i;

        // Retrieve the ball corresponding to (i,j).
        //
        Ball* const ball = bprep[idx];

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

        // Insert link to neighbor ball with name i.
        //
        lprep += new Link( ball, bprep[pn_idx], color_gray );

        // Insert links to balls with same i that are not neighbors.
        //
        if ( j == i % num_sides )
          for ( int k = 2; k < num_sides-1; k++ )
            lprep += new Link
              ( ball, bprep[ idx + (k+j)%num_sides - j ], color_white );

        if ( i == 0 ) continue;

        // Insert link to ball at (i-1,j).
        //
        lprep +=
          new Link( ball, bprep[idx-num_sides], color_lsu_official_purple );

        // Insert link to ball at (i-1, j-1 mod num_sides ).
        //
        lprep += new Link( ball, bprep[pn_idx-num_sides], color_green );
      }
}

void
World::ball_setup_2()
{
```

```
  last_setup = 2;

  pCoor first_pos(13.4,17.8,-9.2);
  const float spacing = distance_relaxed;
  pVect delta_pos = pVect(spacing*0.05,-spacing,0);
  pNorm loc_y = delta_pos;
  pNorm loc_x = pVect(0,0,1);
  pNorm loc_z = cross(loc_y,loc_x);

  // Erase the existing balls and links.
  //
  objects_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) * loc_x
        + 0.5 * spacing * sin(angle) * loc_z;

      truss_info.base_coors += chain_first_pos;
    }

  make_truss(&truss_info);

  // 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;
}
```

```
void
World::ball_setup_3()
{
  last_setup = 3;

  pCoor first_pos(13.4,17.8,-9.2);
  const int factor = 2;
  const float spacing = distance_relaxed / factor;
  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.
  //
  objects_erase();

  Truss_Info truss_info;

  const int length = chain_length * factor;

  truss_info.num_units = length;
  truss_info.unit_length = delta_pos;

  const int sides = opt_sides;

  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);

  const int idx_center = length / 2 * sides;

  for ( int i=0; i<sides; i++ )
    {
      Truss_Info ti;
      ti.num_units = length / 2;
      const int idx_1 = idx_center + ( i == 0 ? sides - 1 : i - 1 );
      const int idx_2 = idx_center + i;

      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];
      ti.base_coors += b0->position;
      ti.base_coors += b1->position;
      ti.base_coors += b2->position;
      ti.base_coors += b3->position;

      pNorm v_head = cross(b1->position,b2->position,b3->position);
      ti.unit_length = delta_dir.magnitude * v_head;
      make_truss(&ti);
      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);
      links += ti.links;
```

```
      balls += ti.balls;

      int tsz = ti.balls.size();

      if ( i == 2 )
        {
          ball_eye = ti.balls[tsz-2];
          ball_down = ti.balls[tsz-1];
        }
      else if ( i == 1 )
        {
          ball_gaze = ti.balls[tsz/2];
        }

    }

  // 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 + 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 = true;
}


void
World::ball_setup_4()
{
  //  last_setup = 4;
}

void
World::ball_setup_5()
{
  //  last_setup = 5;
}


 /// Advance Simulation State by delta_t Seconds
//
void
```

```
World::time_step_cpu(double delta_t)
{
  time_step_count++;

  // Smoothly move ball in response to user input.
  //
  if ( adj_t_stop )
    {
      const double dt = min(world_time,adj_t_stop) - adj_t_prev;
      pVect adj = dt/adj_duration * adj_vector;
      balls_translate(adj,0);
      adj_t_prev = world_time;
      if ( world_time >= adj_t_stop ) adj_t_stop = 0;
    }

  for ( BIter ball(balls); ball; )
    ball->force = ball->mass * gravity_accel;

  for ( LIter link(links); link; )
    {
      if ( link->snapped ) continue;

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

      // Construct a normalized (Unit) Vector from ball to neighbor.
      //
      pNorm ball_to_neighbor(ball1->position,ball2->position);

      const float distance_between_balls = ball_to_neighbor.magnitude;

      // Compute the speed of ball towards neighbor_ball.
      //
      pVect delta_v = ball2->velocity - ball1->velocity;
      float delta_s = dot( delta_v, ball_to_neighbor );

      // Compute by how much the spring is stretched (positive value)
      // or compressed (negative value).
      //
      const float spring_stretch =
        distance_between_balls - link->distance_relaxed;

      if ( link->distance_relaxed > 0.001
           && distance_between_balls
           > link->distance_relaxed * mp.link_stressed_thd )
        {
          if ( distance_between_balls
               > link->distance_relaxed * mp.link_snap_thd )
            link->snapped = true;
          const float factor =
            ( distance_between_balls / link->distance_relaxed
              - mp.link_stressed_thd )
            / ( mp.link_snap_thd - 1 );
          link->color = max(0.0f,1-factor) * link->natural_color
            + factor * mp.color_stressed;
        }
      else
        {
          link->color = link->natural_color;
        }

      // Determine whether spring is gaining energy (whether its length
      // is getting further from its relaxed length).
```

```
    //
    const bool gaining_e = ( delta_s > 0.0 ) == ( spring_stretch > 0 );

    // Use a smaller spring constant when spring is loosing energy,
    // a quick and dirty way of simulating energy loss due to spring
    // friction.
    //
    const float spring_constant =
      gaining_e ? opt_spring_constant : opt_spring_constant * 0.7;

    ball1->force += spring_constant * spring_stretch * ball_to_neighbor;
    ball2->force -= spring_constant * spring_stretch * ball_to_neighbor;
  }

///
/// Update Position of Each Ball
///

for ( BIter ball(balls); ball; )
  {
    if ( ball->locked )
      {
        ball->velocity = pVect(0,0,0);
        continue;
      }

    // Update Velocity
    //
    // This code assumes that force on ball is constant over time
    // step. This is clearly wrong when balls are moving with
    // respect to each other because the springs are changing
    // length. This inaccuracy will make the simulation unstable
    // when spring constant is large for the time step.
    //
    const float mass = max( ball->mass, ball->mass_min );

    pVect delta_v = ( ball->force / mass ) * delta_t;

    if ( platform_collision_possible(ball->position) && ball->position.y < 0 )
      {
        const float spring_constant_plat =
          ball->velocity.y < 0 ? 100000 : 50000;
        const float fric_coefficient = 0.1;
        const float force_up = -ball->position.y * spring_constant_plat;
        const float delta_v_up = force_up / mass * delta_t;
        const float fric_force_mag = fric_coefficient * force_up;
        pNorm surface_v(ball->velocity.x,0,ball->velocity.z);
        const float delta_v_surf = fric_force_mag / mass * delta_t;

        if ( delta_v_surf > surface_v.magnitude )
          {
            // Ignoring other forces?
            delta_v =
              pVect(-ball->velocity.x,delta_v.y,-ball->velocity.z);
          }
        else
          {
            delta_v -= delta_v_surf * surface_v;
          }
        delta_v.y += delta_v_up;
      }


    ball->velocity += delta_v;
```

```
      // Air Resistance
      //
      const double fs = pow(1+opt_air_resistance,-delta_t);
      ball->velocity *= fs;

      // Update Position
      //
      // Assume that velocity is constant.
      //
      ball->position += ball->velocity * delta_t;
    }
}

bool
World::platform_collision_possible(pCoor pos)
{
  // Assuming no motion in x or z axes.
  //
  return pos.x >= platform_xmin && pos.x <= platform_xmax
    && pos.z >= platform_zmin && pos.z <= platform_zmax;
}

 /// External Modifications to State
//
//   These allow the user to play with state while simulation
//   running.

// Move the ball.
//
void Ball::translate(pVect amt) {position += amt;}

// Add velocity to the ball.
//
void Ball::push(pVect amt) {velocity += amt;}

// Set the velocity to zero.
//
void Ball::stop() {velocity = pVect(0,0,0); }

// Set the velocity and rotation (not yet supported) to zero.
//
void Ball::freeze() {velocity = pVect(0,0,0); }



void World::balls_translate(pVect amt,int b){head_ball->translate(amt);}
void World::balls_push(pVect amt,int b){head_ball->push(amt);}
void World::balls_translate(pVect amt)
{ for ( BIter ball(balls); ball; ) ball->translate(amt); }
void World::balls_push(pVect amt)
{ for ( BIter ball(balls); ball; ) ball->push(amt); }
void World::balls_stop()
{ for ( BIter ball(balls); ball; ) ball->stop(); }
void World::balls_freeze(){balls_stop();}


void
World::render_my_piece()
{
}


void
```

```
World::frame_callback()
{
  // This routine called whenever window needs to be updated.

  frame_timer.phys_start();
  const double time_now = time_wall_fp();

  if ( !opt_pause || opt_single_frame || opt_single_time_step )
    {
      /// Advance simulation state.

      // Amount of time since the user saw the last frame.
      //
      const double wall_delta_t = time_now - last_frame_wall_time;

      // Compute amount by which to advance simulation state for this frame.
      //
      const double duration =
        opt_single_time_step ? opt_time_step_duration :
        opt_single_frame ? 1/30.0 :
        wall_delta_t;

      const double world_time_target = world_time + duration;
      const double wall_time_limit = time_now + 0.05;

      while ( world_time < world_time_target )
        {
          time_step_cpu(opt_time_step_duration);
          world_time += opt_time_step_duration;
          const double time_right_now = time_wall_fp();
          if ( time_right_now > wall_time_limit ) break;
        }

      // Reset these, just in case they were set.
      //
      opt_single_frame = opt_single_time_step = false;
    }

  last_frame_wall_time = time_now;

  if ( opt_ride && ball_eye )
    {
      pNorm b_eye_down(ball_eye->position,ball_down->position);
      pVect b_eye_up = -b_eye_down;
      pCoor eye_pos = ball_eye->position + 2.2 * ball_eye->radius * b_eye_up;
      pNorm b_eye_direction(eye_pos,ball_gaze->position);

      pNorm b_eye_left = cross(b_eye_direction,b_eye_up);
      pMatrix_Translate center_eye(-eye_pos);
      pMatrix rotate; rotate.set_identity();
      for ( int i=0; i<3; i++ ) rotate.rc(0,i) = b_eye_left.elt(i);
      for ( int i=0; i<3; i++ ) rotate.rc(1,i) = b_eye_up.elt(i);
      for ( int i=0; i<3; i++ ) rotate.rc(2,i) = -b_eye_direction.elt(i);
      modelview = rotate * center_eye;
      pMatrix reflect; reflect.set_identity(); reflect.rc(1,1) = -1;
      transform_mirror = modelview * reflect * invert(modelview);
    }

  render();
}

int
main(int argv, char **argc)
{
  pOpenGL_Helper popengl_helper(argv,argc);
```

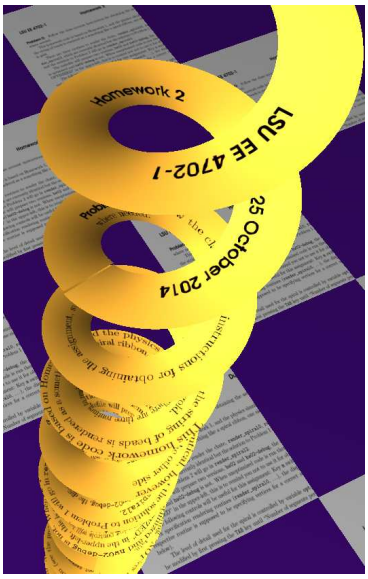
```
  World world(popengl_helper);

  glDisable(GL_DEBUG_OUTPUT);
  glDebugMessageControl(GL_DONT_CARE,GL_DONT_CARE,
                        GL_DEBUG_SEVERITY_NOTIFICATION,0,NULL,false);

  popengl_helper.rate_set(30);
  popengl_helper.display_cb_set(world.frame_callback_w,&world);
}
```

# 28 Fall 2014 Solutions

# LSU EE 4702-1 Homework 2/3 Solution Due: 25 October 2014




**Problem 0:** Follow the class account instructions for obtaining the assignment, substitute hw02 where needed.

This homework code is based on Homework 1, and the physics simulation is identical, however the string of beads is rendered as a something like a spiral ribbon, one side is purple, the other side is gold.

There are three routines to render the chute, `render_spiral0`, `render_spiral1`, and `render_spiral2`, which are currently identical but the solution to Problem 1 will go in `render_spiral1` and the solution to Problem 2 will go in `render_spiral2`.

The makefile will prepare two versions, `hw02` and `hw02-debug`, the difference being that `hw02` is optimized and `hw02-debug` is not. When unoptimized code is run there will be a flashing "NOT OPTIMIZED" in the upper-left, this is to remind you not to use it for obtaining performance data.

The following controls will be useful for this assignment: Key `m` switches between the different vertex specification rendering routines (`render_spiral0`, ...), the display will indicate how the respective routine is *supposed to be* specifying vertices for a correct solution (see the problems below).

The level of detail used for the spiral is controlled by variable `opt_segments`. Its value can be modified by first pressing the `TAB` key until "Number of segments per spiral" appears, and then pressing `+` and `-` to adjust its value.

Check out and compile the code, and make sure that it runs correctly. The image on the upper right shows the appearance of the spiral after a correct solution to Problem 2.

The source code for the solution is in the repo under the name `hw02-sol.cc` and at `http://www.ece.lsu.edu/koppel/gpup/2014/hw02-sol.cc.html`

**Problem 1:** Examine the code in `render_spiral1`. There you will find that individual triangles are used to render the chute.

(*a*) Modify the code so that it uses triangle strips. Be sure to do this correctly, the ribbon should be gold on one side and purple on the other and fewer vertices should be sent to the GPU.

In the original version of the code each iteration of the `t` loop rendered two triangles by sending six vertices. In the solution a triangle strip is used for the whole `t` loop, with each iteration providing two vertices, the new inner and outer vertices computed in the iteration.

A common error was to put the `glBegin/glEnd` inside the `t` loop (not around it). The original code put the `glBegin/glEnd` pair around just two triangles. The solution puts these around the entire segment (`t`) loop. Thus a triangle strip has $2s$ triangles specified using $2s+2$ vertices. Putting `glBegin/glEnd` inside the `t` loop wastes most of the potential of the triangle strip. If that were done the $2s$ triangles would be rendered using $4s$ vertices, almost twice as much.

Another common error was to retain the `if ( t > 0 )` which prevented vertices being emitted on the first iteration. Since we are using triangle strips we need those first two vertices.

(*b*) Measure how much of a performance difference triangle strips make. (Do this by comparing to `render_spiral0` using the `m` key.)

The measurement below is based on the optimized code (the one without `-debug` in the file name. It ran on an NVIDIA Quadro K2100M. The number of segments was increased to 100 to make it easier to measure the graphics load. For individual triangles the reported CPU and GPU times were $5.4\,\text{ms}$ and $1.35\,\text{ms}$. For strips they were $2.5\,\text{ms}$ and $0.77\,\text{ms}$. So, by switching to triangle strips the GPU performance was $2.16\times$ better and the CPU time was $1.75\times$ better.

($c$) Estimate the amount of data sent from CPU to GPU for the two versions of the code.

Let $s$ denote the number of segments (value of `opt_segments`) and let $n$ denote the number of balls (the value of `chain_length`). Each spiral segment (the part that's computed by an iteration of the `t` loop) consists of two triangles. The total number of triangles is therefore $2sn$.

For each vertex the following information is sent: a texture coordinate, a normal, and the vertex coordinate. The vertex coordinate and normal each have three components, the texture coordinate has two components, for a total of 8 components, each of which is a four-byte float. So for each vertex $8 \times 4\,\text{B} = 32\,\text{B}$ are sent.

When specifying vertices using individual triangles, three vertices are sent for each triangle. The total data is $3 \times 2sn \times 32\,\text{B} = 192sn\,\text{B}$ per frame. For triangle strips, the number of vertices sent is two plus the number of triangles, or $2 + 2sn$. The total amount of data is $(2 + 2sn)32\,\text{B} \approx 64sn\,\text{B}$ per frame, a bit more than a third of the amount sent when using individual triangles.

($d$) Performance should be better with the triangle strips. How much was the improvement from `render_spiral0` to `render_spiral1` affected by:

- reduction in computation by the CPU and GPU? *Note: The original question did not include the phrase "by the CPU and GPU".*

A lot for the GPU. That is, the vertex shader (the fixed function vertex shader for this homework) is run three times more frequently when using individual triangles. The CPU does about the same amount of computation in both cases. For example, it computes `norm_inner` the same number of times in `render_spiral0` and `render_spiral1`.

- reduction in data sent to the GPU?

A lot. One third as much data is sent.

- reduction in work performed by the fragment shader (or fixed function equivalent)?

None. The number of triangles hasn't changed, and therefore the number of fragments has not changed either, and so the fragment shader does the same amount of work in both cases.

**Problem 2:** The code in `render_spiral0` and `render_spiral1` apply a texture to the spiral. The texture is a draft of this homework assignment. The code applies the texture so that the top and bottom of the page is mapped to the inner- and outer- part of the spiral. Modify `render_spiral1` so that the width of the page is one over the chain length and as one goes down the spiral one goes down the page. (As though the page were cut into horizontal strips and those strips were connected to form a spiral.) See the illustration above.

Based on the description above, the width of the page should span the `t` loop. That is, for `t=0` the texture coordinate for the left-hand side of the page should be used, for `t=0.5` the texture coordinate for the middle of the page should be used, etc. This matches the coordinate space for textures, so the $s$ component of the texture (akin to the $x$ axis) can be set to `t`. The $t$ component (akin to the $y$ axis) of the texture (which has nothing to do with variable `t`) should be based on variable `i`. For `i=1` we want to be at the top of the page, meaning $t$ is zero, for `i=chain_length` we want $t$ to be 1.

*The problem below was originally assigned as Homework 2 Problem 3 but was later reassigned as Homework 3 Problem 1.*

**Problem 1:** In this problem use buffer objects to store vertices for one spiral section and transformation matrices to render all of them.

(*a*) Modify `render_spiral2` so that it uses buffer objects. (See the code in `demo-7-vtx-arrays.cc` for example of how to do that.)

A spiral section is a spiral going from one ball to the next, it is what is constructed inside of the `t` loop. Use buffer objects to store just one spiral section.

Then use transformation matrices to transform the spiral into the buffer object into the one that you need. For your convenience a routine `make_transform(f1,f2,t1,t2)` is available which will transform a spiral between points `f1` and `f2` to a spiral between points `t1` and `t2`.

(*b*) Measure the performance improvement over the two methods used in Homework 2.

```
/// LSU EE 4702-1 (Fall 2014), GPU Programming
//
 /// Homework 2/3 -- SOLUTION
 //
 /// Your Name: David M. Koppelman

 /// Note:
 //
 //  Homework 2 Problem 3 was not graded. Instead, it was
 //  reassigned as the sole problem in Homework 3.

 //  Assignment: http://www.ece.lsu.edu/koppel/gpup/2014/hw02.pdf
 //  Solution: http://www.ece.lsu.edu/koppel/gpup/2014/hw02_sol.pdf
```

## /// What Code Does

```
// Simulates a string of beads over a platform. The string is modeled
// as point masses connected by springs with a long relaxed
// length. The platform consists of tiles, some are purple-tinted
// mirrors (showing a reflection of the ball), the others show the
// course syllabus.
```

## /// Keyboard Commands

```
 //
 /// Object (Eye, Light, Ball) Location or Push
 //   Arrows, Page Up, Page Down
 //        Move object or push ball, depending on mode.
 //        With shift key pressed, motion is 5x faster.
 //   'e': Move eye.
 //   'l': Move light.
 //   'b': Move head (first) ball. (Change position but not velocity.)
 //   'B': Push head ball. (Add velocity.)
 //
 /// Eye Direction
 //   Home, End, Delete, Insert
 //   Turn the eye direction.
 //   Home should rotate eye direction up, End should rotate eye
 //   down, Delete should rotate eye left, Insert should rotate eye
 //   right.  The eye direction vector is displayed in the upper left.

 /// Simulation Options
 //  (Also see variables below.)
 //
 //  'w'    Turn twirling on and off. (Problem 2 solution.)
 //  '1'    Set up scene 1.
 //  '2'    Set up scene 2.
 //  '3'    Set up scene 3.
 //  'p'    Pause simulation. (Press again to resume.)
 //  ' '    (Space bar.) Advance simulation by 1/30 second.
 //  'S- '  (Shift-space bar.) Advance simulation by one time step.
 //  'h'    Freeze position of first (head) ball. (Press again to release.)
 //  't'    Freeze position of last (tail) ball. (Press again to release.)
 //  's'    Stop balls.
 //  'g'    Turn gravity on and off.
 //  'F12'  Write screenshot to file.

 /// Variables
 //   Selected program variables can be modified using the keyboard.
 //   Use "Tab" to cycle through the variable to be modified, the
 //   name of the variable is displayed next to "VAR" on the bottom
 //   line of green text.
```

```
 //  'Tab' Cycle to next variable.
 //  '`'   Cycle to previous variable.
 //  '+'   Increase variable value.
 //  '-'   Decrease variable value.
 //
 //  VAR Spring Constant - Set spring constant.
 //  VAR Air Resistance - Set air resistance.
 //  VAR Light Intensity - The light intensity.
 //  VAR Gravity - Gravitational acceleration. (Turn on/off using 'g'.)


#define GL_GLEXT_PROTOTYPES
#define GLX_GLXEXT_PROTOTYPES

#include <GL/gl.h>
#include <GL/glext.h>
#include <GL/glx.h>
#include <GL/glxext.h>
#include <GL/glu.h>
#include <GL/freeglut.h>

#include <gp/util.h>
#include <gp/glextfuncs.h>
#include <gp/coord.h>
#include <gp/shader.h>
#include <gp/pstring.h>
#include <gp/misc.h>
#include <gp/gl-buffer.h>
#include <gp/texture-util.h>

#include "shapes.h"


///
/// Main Data Structures
///
//
// class World: All data about scene.


class World;


// Object Holding Ball State
//
class Ball {
public:
  pCoor position;
  pVect velocity;

  float mass;
  float radius;

  bool contact;                 // Can be used for special effects.

  void push(pVect amt);
  void translate(pVect amt);
  void stop();
  void freeze();
};

enum Render_Option { RO_Normally, RO_Simple, RO_Shadow_Volumes };
```

```
class World {
public:
  World(pOpenGL_Helper &fb):ogl_helper(fb){init();}
  void init();
  void init_graphics();
  static void frame_callback_w(void *moi){((World*)moi)->frame_callback();}
  void frame_callback();
  void render();
  void render_spiral0();
  void render_spiral1();
  void render_spiral2();
  void render_objects(Render_Option render_option);
  void cb_keyboard();
  void modelview_update();

  pOpenGL_Helper& ogl_helper;
  pVariable_Control variable_control;
  pFrame_Timer frame_timer;
  double world_time;
  double last_frame_wall_time;
  int time_step_count;
  float opt_gravity_accel;      // Value chosen by user.
  pVect gravity_accel;          // Set to zero when opt_gravity is false;
  bool opt_gravity;
  bool opt_head_lock, opt_tail_lock;
  bool opt_time_step_alt;
  int opt_vtx_method;
  int opt_segments;

  // Tiled platform for ball.
  //
  float platform_xmin, platform_xmax, platform_zmin, platform_zmax;
  float platform_pi_xwidth_inv;
  pBuffer_Object<pVect> platform_tile_coords;
  pBuffer_Object<float> platform_tex_coords;
  pVect platform_normal;
  GLuint texid_syl;
  GLuint texid_emacs;
  bool opt_platform_texture;
  void platform_update();
  bool platform_collision_possible(pCoor pos);

  pCoor light_location;
  float opt_light_intensity;
  enum { MI_Eye, MI_Light, MI_Ball, MI_Ball_V, MI_COUNT } opt_move_item;
  bool opt_pause;
  bool opt_single_frame;      // Simulate for one frame.
  bool opt_single_time_step;  // Simulate for one time step.
  int viewer_shadow_volume;

  pCoor eye_location;
  pVect eye_direction;
  pMatrix modelview;
  pMatrix transform_mirror;

  void ball_setup_1();
  void ball_setup_2();
  void ball_setup_3();
  void ball_setup_4();
  void ball_setup_5();
  void time_step_cpu(double);
```

```
  void balls_stop();
  void balls_freeze();
  void balls_translate(pVect amt, int idx);
  void balls_translate(pVect amt);
  void balls_push(pVect amt, int idx);
  void balls_push(pVect amt);

  float opt_spring_constant;
  float opt_air_resistance;
  float distance_relaxed;
  int chain_length;
  Ball *balls;
  Sphere sphere;

  /// Problem 3: Items can be added here.

  /// SOLUTION

  GLuint spiral_norm_buffer, spiral_vtx_buffer;
  int bo_vertices, bo_segments;
  bool bo_stale;
  pCoor bo_pos1, bo_pos2;
  pNorm bo_v12;

};



#include "hw02-graphics.cc"


void
World::init()
{
  chain_length = 18;
  balls = new Ball[chain_length];

  opt_segments = 50;
  variable_control.insert(opt_segments,"Number of segments per spiral.");

  distance_relaxed = 15.0 / chain_length;
  opt_spring_constant = 1000;
  variable_control.insert(opt_spring_constant,"Spring Constant");

  opt_gravity_accel = 9.8;
  opt_gravity = true;
  gravity_accel = pVect(0,-opt_gravity_accel,0);
  variable_control.insert(opt_gravity_accel,"Gravity");

  opt_air_resistance = 0.001;

  world_time = 0;
  time_step_count = 0;
  last_frame_wall_time = time_wall_fp();
  frame_timer.work_unit_set("Steps / s");

  eye_location = pCoor(24.2,11.6,-38.7);
  eye_location = pCoor(18.9,11.6,-22.6);
  eye_direction = pVect(-0.42,-0.09,0.9);

  init_graphics();

  ball_setup_1();
```

```
  opt_vtx_method = 1;

  /// SOLUTION

  bo_stale = true;
  spiral_norm_buffer = spiral_vtx_buffer = 0;
}

///
/// Physical Simulation Code
///

 /// Initialize Simulation
//

void
World::ball_setup_1()
{
  // Arrange and size balls to form a pendulum.
  //
  pCoor next_pos(5.2,15.8,-2.9);

  for ( int i=0; i<chain_length; i++ )
    {
      Ball* const ball = &balls[i];
      ball->position = next_pos;
      ball->velocity = pVect(0,0,0);
      ball->radius = 0.5 * distance_relaxed;
      ball->mass = 4/3.0 * M_PI * pow(ball->radius,3);
      ball->contact = false;
      next_pos += pVect(distance_relaxed,0,0);
    }

  opt_tail_lock = false;
  opt_head_lock = true;
}


void
World::ball_setup_2()
{
  pCoor first_ball(20,11,-9.2);
  pCoor ref_pos(8,11,-9.2);

  const float radius = 0.3 * distance_relaxed;

  pNorm direction(first_ball,ref_pos);
  pVect delta_v = distance_relaxed * direction;

  for ( int i=0; i<chain_length; i++ )
    {
      Ball* const ball = &balls[i];
      ball->position = first_ball + i * delta_v;
      ball->velocity = pVect(0,0,0);
      ball->radius = radius;
      ball->mass = 4/3.0 * M_PI * pow(ball->radius,3);
      ball->contact = false;
    }

  opt_head_lock = true;
  opt_tail_lock = true;
}

void
```

```
World::ball_setup_3()
{
}

void
World::ball_setup_4()
{
}

void
World::ball_setup_5()
{
}


 /// Advance Simulation State by delta_t Seconds
//
void
World::time_step_cpu(double delta_t)
{
  time_step_count++;

  //
  /// Compute force and update velocity of each ball.
  //
  for ( int i=0; i<chain_length; i++ )
    {
      Ball* const ball = &balls[i];

      // Skip locked balls.
      //
      if ( opt_head_lock && i == 0 || opt_tail_lock && i == chain_length - 1 )
        {
          ball->velocity = pVect(0,0,0);
          continue;
        }

      pVect force(0,0,0);

      // Gravitational Force
      //
      force += ball->mass * gravity_accel;

      // Spring Force from Neighbor Balls
      //
      for ( int direction = -1; direction <= +1; direction += 2 )
        {
          const int n_idx = i + direction;  // Compute neighbor index.
          if ( n_idx < 0 ) continue;
          if ( n_idx == chain_length ) break;

          Ball* const neighbor_ball = &balls[n_idx];

          // Construct a normalized (Unit) Vector from ball to neighbor.
          //
          pNorm ball_to_neighbor(ball->position,neighbor_ball->position);

          const float distance_between_balls = ball_to_neighbor.magnitude;

          // Compute the speed of ball towards neighbor_ball.
          //
          pVect delta_v = neighbor_ball->velocity - ball->velocity;
          float delta_s = dot( delta_v, ball_to_neighbor );
```

```
          // Compute by how much the spring is stretched (positive value)
          // or compressed (negative value).
          //
          const float spring_stretch =
            distance_between_balls - distance_relaxed;

          // Determine whether spring is gaining energy (whether its length
          // is getting further from its relaxed length).
          //
          const bool gaining_e = ( delta_s > 0.0 ) == ( spring_stretch > 0 );

          // Use a smaller spring constant when spring is loosing energy,
          // a quick and dirty way of simulating energy loss due to spring
          // friction.
          //
          const float spring_constant =
            gaining_e ? opt_spring_constant : opt_spring_constant * 0.7;

          force += spring_constant * spring_stretch * ball_to_neighbor;
        }

      // Update Velocity
      //
      // This code assumes that force on ball is constant over time
      // step. This is clearly wrong when balls are moving with
      // respect to each other because the springs are changing
      // length. This inaccuracy will make the simulation unstable
      // when spring constant is large for the time step.
      //
      ball->velocity += ( force / ball->mass ) * delta_t;

      // Air Resistance
      //
      const double fs = pow(1+opt_air_resistance,-delta_t);
      ball->velocity *= fs;
    }

  ///
  /// Update Position of Each Ball
  ///

  for ( int i=0; i<chain_length; i++ )
    {
      Ball* const ball = &balls[i];

      // Update Position
      //
      // Assume that velocity is constant.
      //
      ball->position += ball->velocity * delta_t;

      // Possible Collision with Platform
      //

      // Skip if collision impossible.
      //
      if ( !platform_collision_possible(ball->position) ) continue;
      if ( ball->position.y >= 0 ) continue;

      // Snap ball position to surface.
      //
      ball->position.y = 0;

      // Reflect y (vertical) component of velocity, with a reduction
      // due to energy lost in the collision.
```

```
      //
      if ( ball->velocity.y < 0 )
        ball->velocity.y = - 0.9 * ball->velocity.y;
    }
}

bool
World::platform_collision_possible(pCoor pos)
{
  // Assuming no motion in x or z axes.
  //
  return pos.x >= platform_xmin && pos.x <= platform_xmax
    && pos.z >= platform_zmin && pos.z <= platform_zmax;
}

pMatrix
make_transform(pCoor from_p1, pCoor from_p2, pCoor to_p1, pCoor to_p2)
{
  pNorm from_v12(from_p1,from_p2);
  pNorm to_v12(to_p1,to_p2);
  float scale_factor = to_v12.magnitude / from_v12.magnitude;
  pMatrix bo_to_orig = pMatrix_Translate(-pVect(from_p1));
  pMatrix scale = pMatrix_Scale(scale_factor);
  pMatrix rot = pMatrix_Rotation_Shortest(from_v12,to_v12);
  pMatrix orig_to_p1 = pMatrix_Translate(pVect(to_p1));
  pMatrix m = orig_to_p1 * rot * scale * bo_to_orig;
  return m;
}


void
World::render_spiral1()
{
  /// Modify for problem 1 and 2.

  /// SOLUTION

  glDisable(GL_COLOR_MATERIAL);
  glMaterialfv(GL_FRONT,GL_AMBIENT_AND_DIFFUSE,color_lsu_spirit_gold);
  glMaterialfv(GL_BACK,GL_AMBIENT_AND_DIFFUSE,color_lsu_spirit_purple);

  glEnable(GL_TEXTURE_2D);
  glActiveTexture(GL_TEXTURE0);
  glBindTexture(GL_TEXTURE_2D,texid_syl);

  float theta_0 = 0;

  for ( int i=1; i<chain_length; i++ )
    {
      Ball *const ball1 = &balls[i-1];
      Ball *const ball2 = &balls[i];
      pCoor pos1 = ball1->position;
      pCoor pos2 = ball2->position;

      //
      // Render a spiral slide from position of ball1 to ball2.
      //
      const float omega = 10;

      pNorm v12(pos1,pos2);

      // Find a vector that's orthogonal to v12.
      //
      pNorm van =
```

```
        v12.x == 0 ? pVect(0,v12.z,-v12.y) : pVect(v12.y,-v12.x,0);

      pNorm vbn = cross(v12,van);

      pVect va = van * ball1->radius;
      pVect vb = vbn * ball1->radius;

      const float v12_dist = v12.magnitude;

      /// SOLUTION
      //
      //  The prev variables below are not needed for the
      //  triangle strip and so have been removed.
      //    pCoor p_prev, p_outer_prev, p_inner_prev;
      //    pVect norm_prev, norm_inner_prev;

      const float delta_t = 1.0 / opt_segments;

      /// SOLUTION
      //  Start triangle strip here.

      glBegin(GL_TRIANGLE_STRIP);

      for ( float t=0; t<1.001; t += delta_t )
        {
          const bool last_iter = t + delta_t > 1.001;
          const float dist = t * v12_dist;
          pCoor p = ball1->position + dist * v12;
          const float theta = theta_0 + dist * omega;
          if ( last_iter ) theta_0 = theta;
          pCoor p_outer = p + va * cos(theta) + vb * sin(theta);
          const float inner_frac = 0.5;
          pCoor p_inner = p
            + inner_frac * va * cos(theta) + inner_frac * vb * sin(theta);
          pVect tang = v12
            - v12_dist * omega * va * sin(theta)
            + v12_dist * omega * vb * cos(theta);
          pVect tang_inner = v12
            - inner_frac * v12_dist * omega * va * sin(theta)
            + inner_frac * v12_dist * omega * vb * cos(theta);
          pNorm norm = cross(tang,pVect(p_outer,p));
          pNorm norm_inner = cross(tang_inner,pVect(p_outer,p));

          const float du = 1.0 / chain_length;
          const float u = float(i) * du;

          /// SOLUTION
          //
          //  Just send the two "new" vertices to the GPU.

          glTexCoord2f(t,u+du);
          glNormal3fv(norm_inner);
          glVertex3fv(p_inner);

          glTexCoord2f(t,u);
          glNormal3fv(norm);
          glVertex3fv(p_outer);
        }

      glEnd();

    }

  glEnable(GL_COLOR_MATERIAL);
}
```

```
void
World::render_spiral2()
{
  /// Modify for Homework 3  (Neé Homework 2 Problem 3)

  /// SOLUTION

  glDisable(GL_COLOR_MATERIAL);
  glMaterialfv(GL_FRONT,GL_AMBIENT_AND_DIFFUSE,color_lsu_spirit_gold);
  glMaterialfv(GL_BACK,GL_AMBIENT_AND_DIFFUSE,color_lsu_spirit_purple);

  glEnable(GL_TEXTURE_2D);
  glActiveTexture(GL_TEXTURE0);
  glBindTexture(GL_TEXTURE_2D,texid_syl);

  float theta_0 = 0;

  for ( int i=1; i<chain_length; i++ )
    {
      Ball *const ball1 = &balls[i-1];
      Ball *const ball2 = &balls[i];
      pCoor pos1 = ball1->position;
      pCoor pos2 = ball2->position;

      //
      // Render a spiral slide from position of ball1 to ball2.
      //
      const float omega = 10;

      pNorm v12(pos1,pos2);

      /// SOLUTION
      //
      //  Store information for just 1 spiral segment in a set of
      //  buffer objects (one buffer object for the vertices and one
      //  for the normals).
      //
      //  This set of buffer objects describes a segment in one particular
      //  orientation (the values of pos1 and pos2 when they were loaded),
      //  but by using a transformation we can use it to render a
      //  segment in any position.

      bo_stale = bo_segments != opt_segments;

      if ( bo_stale )
        {
          bo_stale = false;
          bo_segments = opt_segments;
          bo_pos1 = pos1;  bo_pos2 = pos2;  bo_v12 = v12;

          // Find a vector that's orthogonal to v12.
          //
          pNorm van =
            v12.x == 0 ? pVect(0,v12.z,-v12.y) : pVect(v12.y,-v12.x,0);

          pNorm vbn = cross(v12,van);

          pVect va = van * ball1->radius;
          pVect vb = vbn * ball1->radius;

          const float v12_dist = v12.magnitude;

          const float delta_t = 1.0 / opt_segments;
```

```
    const int num_vertices = 2 * ( opt_segments + 1 );
    pVect* norm_array = new pVect[num_vertices];
    pCoor* vtx_array = new pCoor[num_vertices];
    int ni=0, vi=0;

    for ( float t=0; t<1.001; t += delta_t )
      {
        const bool last_iter = t + delta_t > 1.001;
        const float dist = t * v12_dist;
        pCoor p = ball1->position + dist * v12;
        const float theta = theta_0 + dist * omega;
        if ( last_iter ) theta_0 = theta;
        pCoor p_outer = p + va * cos(theta) + vb * sin(theta);
        const float inner_frac = 0.5;
        pCoor p_inner = p
          + inner_frac * va * cos(theta) + inner_frac * vb * sin(theta);
        pVect tang = v12
          - v12_dist * omega * va * sin(theta)
          + v12_dist * omega * vb * cos(theta);
        pVect tang_inner = v12
          - inner_frac * v12_dist * omega * va * sin(theta)
          + inner_frac * v12_dist * omega * vb * cos(theta);
        pNorm norm = cross(tang,pVect(p_outer,p));
        pNorm norm_inner = cross(tang_inner,pVect(p_outer,p));
        {
          norm_array[ni++] = norm_inner;
          norm_array[ni++] = norm;
          vtx_array[vi++] = p_inner;
          vtx_array[vi++] = p_outer;
        }
      }

    ASSERTS( ni == num_vertices );
    ASSERTS( vi == num_vertices );
    bo_vertices = num_vertices;

    if ( !spiral_norm_buffer ){
      glGenBuffers(1,&spiral_norm_buffer);
      glGenBuffers(1,&spiral_vtx_buffer);
    }

    glBindBuffer(GL_ARRAY_BUFFER,spiral_norm_buffer);
    glBufferData(GL_ARRAY_BUFFER, sizeof(norm_array[0])*num_vertices,
                 norm_array, GL_STATIC_DRAW);
    glBindBuffer(GL_ARRAY_BUFFER,spiral_vtx_buffer);
    glBufferData(GL_ARRAY_BUFFER, sizeof(vtx_array[0])*num_vertices,
                 vtx_array, GL_STATIC_DRAW);

    delete norm_array;
    delete vtx_array;

  }

glMatrixMode(GL_MODELVIEW);
glPushMatrix();

pMatrix m = make_transform(bo_pos1,bo_pos2,pos1,pos2);

glMultTransposeMatrixf(m);

glBindBuffer(GL_ARRAY_BUFFER,spiral_norm_buffer);
glNormalPointer(GL_FLOAT,0,NULL);

glBindBuffer(GL_ARRAY_BUFFER,spiral_vtx_buffer);
glVertexPointer( 4, GL_FLOAT, 4*sizeof(float), NULL);
```

```
      glBindBuffer(GL_ARRAY_BUFFER,0);

      glEnableClientState(GL_VERTEX_ARRAY);
      glEnableClientState(GL_NORMAL_ARRAY);
      glDrawArrays(GL_TRIANGLE_STRIP,0,bo_vertices);
      glDisableClientState(GL_NORMAL_ARRAY);
      glDisableClientState(GL_VERTEX_ARRAY);

      glPopMatrix();

    }

  glEnable(GL_COLOR_MATERIAL);
}


 /// External Modifications to State
//
//   These allow the user to play with state while simulation
//   running.

// Move the ball.
//
void Ball::translate(pVect amt) {position += amt;}

// Add velocity to the ball.
//
void Ball::push(pVect amt) {velocity += amt;}

// Set the velocity to zero.
//
void Ball::stop() {velocity = pVect(0,0,0); }

// Set the velocity and rotation (not yet supported) to zero.
//
void Ball::freeze() {velocity = pVect(0,0,0); }



void World::balls_translate(pVect amt,int b){balls[b].translate(amt);}
void World::balls_push(pVect amt,int b){balls[b].push(amt);}
void World::balls_translate(pVect amt)
{ for(int i=0;i<chain_length;i++)balls[i].translate(amt);}
void World::balls_push(pVect amt)
{ for(int i=0;i<chain_length;i++)balls[i].push(amt);}
void World::balls_stop()
{ for(int i=0;i<chain_length;i++)balls[i].stop();}
void World::balls_freeze(){balls_stop();}


void
World::frame_callback()
{
  // This routine called whenever window needs to be updated.

  const double time_now = time_wall_fp();

  if ( !opt_pause || opt_single_frame || opt_single_time_step )
    {
      /// Advance simulation state.

      // Amount of time since the user saw the last frame.
```

```
      //
      const double wall_delta_t = time_now - last_frame_wall_time;

      const double time_step_duration = 0.0001;

      // Compute amount by which to advance simulation state for this frame.
      //
      const double duration =
        opt_single_time_step ? time_step_duration :
        opt_single_frame ? 1/30.0 :
        wall_delta_t;

      const double world_time_target = world_time + duration;

      while ( world_time < world_time_target )
        {
          time_step_cpu(time_step_duration);
          world_time += time_step_duration;
        }

      // Reset these, just in case they were set.
      //
      opt_single_frame = opt_single_time_step = false;
    }

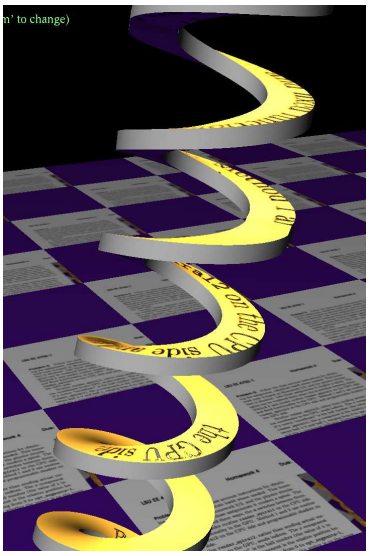
  last_frame_wall_time = time_now;
  render();
}

int
main(int argv, char **argc)
{
  pOpenGL_Helper popengl_helper(argv,argc);
  World world(popengl_helper);

  popengl_helper.rate_set(30);
  popengl_helper.display_cb_set(world.frame_callback_w,&world);
}
```

# LSU EE 4702-1 Homework 4 Due: 7 November 2014

*The solution code is in the repo in file name* `hw04-shdr-sol.cc`. *If you pull and* `make` *you can run the solution by executing* `hw04sol`. *An html version is available at* `https://www.ece.lsu.edu/koppel/gpup/2014/hw04-shdr-sol.cc.html`. *Note that the git version is always the most up to date.*

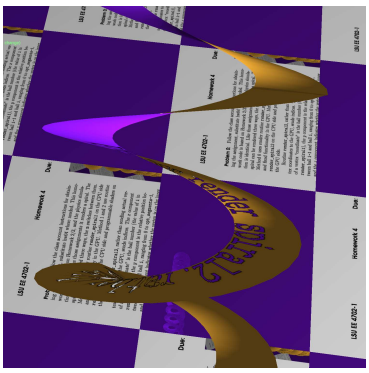


**Problem 0:** Follow the class account instructions for obtaining the assignment, substitute hw04 where needed. This homework code is based on Homework 2/3, and the physics simulation is identical. Like those assignments it renders a spiral. The spiral can be rendered three ways, the `m` switches between them. Method 0 uses render routine `render_spiral1` on the CPU side and fixed functionality in the GPU. Method 1 and 2 use routine `render_spiral2` on the CPU side and programmable shaders on the GPU side.

Routine `render_spiral2`, rather than sending actual vertex coordinates to the GPU, sends indices. The $x$ component of a vertex "coordinate" is the ball number (the value of `i` in `render_spiral1`), the $y$ component is the relative position between ball `i-1` and ball `i`, ranging from 0 to `opt_segments-1`, and the $z$ component indicates whether the vertex is on the inner or outer edge of the spiral. The code in vertex shader `vs_main` in file `hw04-shdr.cc` uses these values to compute the actual vertex coordinate and normal, and to compute the texture coordinates.

Notice that the indices sent by `render_spiral2` are not affected by the position of the balls. Therefore indices can be put in a buffer object and sent just once. (They do need to be re-sent if `opt_segments` changes.) The ball positions do need to be sent each frame, but that's much less data. The code in `render_spiral2` always uses vertex shader `vs_main` and fragment shader `fs_main`, but can activate either of two geometry shaders, `gs_main_simple` (for Method 1) and `gs_main_solution` (for Method 2). Geometry shader `gs_main_simple` makes no changes to the triangles it receives, and it should not be modified. Geometry shader `gs_main_solution` is initially the same as `gs_main_simple` but should be modified in one of the problems below.

The level of detail used for the spiral is controlled by variable `opt_segments`. Its value can be modified by first pressing the `TAB` key until "Number of segments per spiral" appears, and then pressing `+` and `-` to adjust its value. Check out and compile the code, and make sure that it runs correctly. **For this assignment** only modify file `hw04-shdr.cc`.



**Problem 1:** Notice that with Method 1 and 2 pale blue is used for both sides of the spiral. In this problem modify the fragment shader, `fs_main`, so the spiral appears like the one on the image to the right. Details are given in the subproblems. Changes to the fragment shader affect Methods 1 and 2, but not zero.

(*a*) Modify the fragment shader so that the front surface is gold and has the texture applied and so that the back surface is purple. Do this *without* using the color attribute (fragment shader input), instead rely on the material properties that are set in `render_spiral2`. Note that these material properties are uniform variables.

For the names of the predefined material property uniforms available in the fragment shader see Section 7.4.1 in the OpenGL Shading Language spec version 4.5. To determine whether a fragment is from the front or the back of a primitive see Section 7.1 in the spec.

(*b*) The surface of the spiral shows a draft this homework assignment handout, using black for the text and yellow for the background. Modify the fragment shader so that there is no spiral wherever the texture is black (or a dark color). In other words, modify it so that you can see through the spiral wherever the texture is black.

To determine if a texel is dark add up the red, green, and blue components and check if the sum is less than .05. If a texel is dark, then prevent the corresponding fragment from being written. To see how to prevent a fragment from being written see Section 6.4 in the OpenGL Shading Language spec version 4.5. (This section describes jump-like statements in the shading language.)

**Problem 2:** The spiral's appearance is a little unnatural because it is two-dimensional. Modify the shaders (except `gs_main_simple`) and surrounding code so that when Method 2 is active the spiral has some thickness. (See the illustration at the beginning of this assignment.)

Use geometry shader `gs_main_solution` to render multiple triangles as follows. Let $V_1$, $V_2$, and $V_3$ denote the vertices of a triangle at the input to the geometry shader. The geometry shader should emit $V_1V_2V_3$ (as it already does) and also $U_1U_2U_3$ where $U_i = V_i + \vec{d}$ and $\vec{d}$ is a vector of length 0.1 pointing from ball $i-1$ to $i$ (the balls that the spiral segment is between). The geometry shader should also emit triangles for an inner and outer edge which joins the two spirals. The edge should be the gray color set in `render_spiral2`. See the illustration.

The solution to this problem will require modifying all three shaders, and modifying the shaders' interface blocks (such as `Data_to_GS`). **No changes** should be necessary in file `hw04.cc`. It's okay to modify this file to help in debugging your code, but the solution itself should only be in `hw04-shdr.cc`. If you believe otherwise, please contact the instructor. Divide work appropriately between the vertex shader and geometry shader.

Here are some things to watch out for:

- Be sure to adjust the maximum number of vertices specified for the geometry shader output. If this number is too low execution will end with an error.
- When you modify an interface block (such as `Data_to_GS`) be sure to modify the other interface block with the same name (one is a shader output, the other is a shader input).

Here are some debugging tips:

- Check for shader code compilation errors when your program starts. The errors are sent to `stdout` and should appear in the shell or in gdb, depending on how you started the program.
- If execution ends with an OpenGL error, you can get a more detailed error message by turning on OpenGL debugging. To do this change `false` to `true` in `popengl_helper.ogl_debug_set(false);` near the end of `hw04.cc`.
- Three UI-controlled variables are available for debugging. They are `debug_bool.x`, `debug_bool.y`, and `debug_float`. The Booleans can be toggled with `d` and `D`. The float can be adjusted by pressing `Tab` until "Debug Float" appears and then press `+` and `-` to adjust the value.

```
/// LSU EE 4702-1 (Fall 2014), GPU Programming
//
 /// Homework 4
 //
 ///  SOLUTION

 /// Updated for Vulkan Fall 2023

 /// Instructions
 //
 //  Read the assignment: http://www.ece.lsu.edu/koppel/gpup/2014/hw04.pdf


// Specify version of OpenGL Shading Language.
//
#version 460

#extension GL_GOOGLE_include_directive : enable
#include <light.h>
#include <transform.h>
#include "shader-common-generic-lighting.h"
#include "shdr-common.h"

layout ( binding = BIND_UNI_COMMON ) uniform UC
{
  Shdr_Uni_Common uc;
};

// Use this variable to debug your code. Press 'd' to toggle
// debug_bool.x and 'D' to toggle debug_bool.y (between true and
// false).
//
bvec2 debug_bool = bvec2( bool(uc.debug_bool.x), bool(uc.debug_bool.y) );

// Use this to debug your code. Press TAB until "debug_float"
// appears, then press +/- to adjust its value.
//
float debug_float = uc.debug_float;

// Array of ball positions.
//
layout ( binding = BIND_BALLS_POS ) buffer Balls_Pos { vec4 balls_pos[]; };


#ifdef _VERTEX_SHADER_

// Redefine this vertex shader input to be an integer vector.
//
layout ( location = LOC_IN_INT4 ) in ivec4 in_indices;

// Interface block for vertex shader output / geometry shader input.
//
layout ( location = 0 ) out Data_to_GS
```

```
{
  vec4 vertex_c;
  vec3 normal_e;
  vec4 vertex_e;
  vec2 tcoor;

  // Any changes here must also be made to the fragment shader input.

  /// SOLUTION - Problem 2
  //
  vec4 vertex_e_upper;
  vec4 vertex_c_upper;
  vec3 radial_e;
  ivec3 indices;
} Out;


void
vs_main()
{
  const float spiral_radius = 0.5;
  const float omega = 10;

  const int bidx = in_indices.x;
  const int ti = in_indices.y;
  const bool inner = in_indices.z == 1;

  const int radial_idx = bidx * uc.opt_segments + ti;
  const float delta_t = 1.0 / uc.opt_segments;
  const float t = float(ti) * delta_t;
  const float theta = delta_t * radial_idx * omega;

  vec3 pos1 = balls_pos[bidx-1].xyz;
  vec3 pos2 = balls_pos[bidx].xyz;

  vec3 v12 = pos2.xyz - pos1.xyz;

  // Find a vector that's orthogonal to v12.
  //
  vec3 ax =
    normalize(v12.x == 0 ? vec3(0,v12.z,-v12.y) : vec3(v12.y,-v12.x,0));

  // Find a vector that's orthogonal to v12 and ax.
  //
  vec3 ay = normalize(cross(v12,ax));

  vec3 vx = spiral_radius * ax;
  vec3 vy = spiral_radius * ay;

  // Point on line between ball1 and ball2.
  //
  vec3 p = pos1 + t * v12;

  // Vector from p to spiral outer edge.
```

```
//
vec3 radial = cos(theta) * vx + sin(theta) * vy;
vec3 p_outer = p + radial;

const float inner_frac = 0.5;
vec3 p_inner = p + inner_frac * radial;

// Compute surface normals.
//
vec3 tangial = -omega * sin(theta) * vx + omega * cos(theta) * vy;
vec3 tang = v12 + tangial;
vec3 tang_inner = v12 + inner_frac * tangial;
vec3 norm = normalize(cross(radial,tang));
vec3 norm_inner = normalize(cross(radial,tang_inner));

// The code above computed both the inner and outer spiral
// points. But, we only need one of them. Tsk, tsk, that's wasteful!
//
vec4 vertex_o = vec4( inner ? p_inner : p_outer, 1 );
vec3 normal_o = inner ? norm_inner : norm;

// Write position and normal to shader output variables. Position
// is written in both eye and clip space.
//
Out.vertex_c = ut.clip_from_object * vertex_o;
Out.normal_e = mat3(ut.eye_from_object) * normal_o;
Out.vertex_e = ut.eye_from_object * vertex_o;

// Amount by which to zoom the texture.
//
float tex_zoom = 0.5;

// Uncomment the line below to use "debug_float" to zoom text.
// tex_zoom /= debug_float;

const float du = 0.5 * tex_zoom / uc.chain_length;
const float u = float(bidx) * du;

Out.tcoor.x = tex_zoom * t;
Out.tcoor.y = 0.18 + u + (inner ? du : 0 );

/// SOLUTION - Problem 2
//
//  Compute the position and radial of a point on the
//  second spiral and write them to new vertex shader outputs.
//  The radial is used as the surface normal for the edge triangles.

vec3 v12n = normalize(v12);
vec3 depth_vector = 0.1f * v12n;
vec4 vertex_o_upper = vertex_o + vec4(depth_vector,0);
Out.vertex_c_upper = ut.clip_from_object * vertex_o_upper;
Out.vertex_e_upper = ut.eye_from_object * vertex_o_upper;
Out.indices = in_indices.xyz;
Out.radial_e = mat3(ut.eye_from_object) * radial;
```

```
}

#endif



#ifdef _GEOMETRY_SHADER_

layout ( location = 0 ) in Data_to_GS
{
  vec4 vertex_c;
  vec3 normal_e;
  vec4 vertex_e;
  vec2 tcoor;

  /// SOLUTION - Problem 2
  //
  vec4 vertex_e_upper;
  vec4 vertex_c_upper;
  vec3 radial_e;
  ivec3 indices;

} In[];

layout ( location = 0 ) out Data_to_FS
{
  vec3 normal_e;
  vec4 vertex_e;
  vec2 tcoor;

  /// SOLUTION - Problem 2
  //
  flat int is_edge;  // True if primitive is an inner or outer edge.
} Out;

// Type of primitive at geometry shader input.
//
layout ( triangles ) in;

 /// SOLUTION - Problem 2
// layout ( triangle_strip, max_vertices = 3 ) out;
layout ( triangle_strip, max_vertices = 10 ) out;

void
gs_main_simple()
{
  // DO NOT MODIFY THIS ROUTINE.
  // This shader used with Method 1.

  // Pass the triangle unchanged.

  for ( int i=0; i<3; i++ )
    {
      Out.normal_e = In[i].normal_e;
```

```
      Out.vertex_e = In[i].vertex_e;
      gl_Position = In[i].vertex_c;
      Out.tcoor = In[i].tcoor;
      EmitVertex();
    }
  EndPrimitive();
  // DO NOT MODIFY THIS ROUTINE.
}


void
gs_main_solution()
{
  // PROBLEM 2 Solution Goes Here

  /// SOLUTION - Problem 2

  // Emit the triangles on the upper and lower spirals.
  //
  for ( int level=0; level<2; level++ )
    {
      const bool upper = level == 1;

      for ( int i=0; i<3; i++ )
        {
          Out.normal_e = In[i].normal_e;
          Out.vertex_e = upper ? In[i].vertex_e_upper : In[i].vertex_e;
          gl_Position = upper ? In[i].vertex_c_upper : In[i].vertex_c;
          Out.tcoor = In[i].tcoor;
          Out.is_edge = 0;
          EmitVertex();
        }
      EndPrimitive();
    }

  //
  // Emit the triangles on the edge.
  //

  // First, find two vertices that are both on the outer edge or both
  // on the inner edge.
  //
  int idx[2];
  if ( In[0].indices.z == In[1].indices.z )       { idx[0] = 0;  idx[1] = 1; }
  else if ( In[0].indices.z == In[2].indices.z )  { idx[0] = 0;  idx[1] = 2; }
  else                                            { idx[0] = 1;  idx[1] = 2; }

  bool is_inner = In[idx[0]].indices.z == 1;

  // Emit the edge triangles.
  //
  for ( int i=0; i<2; i++ )
    {
      const int v = idx[i];
```

```
      Out.vertex_e = In[v].vertex_e;
      gl_Position = In[v].vertex_c;
      Out.normal_e = is_inner ? -In[v].radial_e : In[v].radial_e;
      Out.is_edge = 1;
      EmitVertex();
      Out.vertex_e = In[v].vertex_e_upper;
      gl_Position = In[v].vertex_c_upper;
      Out.normal_e = is_inner ? -In[v].radial_e : In[v].radial_e;
      Out.is_edge = 1;
      EmitVertex();
    }
  EndPrimitive();
}

#endif


#ifdef _FRAGMENT_SHADER_

layout ( location = 0 ) in Data_to_FS
{
  vec3 normal_e;
  vec4 vertex_e;
  vec2 tcoor;

  /// SOLUTION - Problem 2
  //
  flat int is_edge;
} In;

layout ( binding = BIND_TEXUNIT ) uniform sampler2D tex_unit_0;
layout ( location = 0 ) out vec4 out_frag_color;


void
fs_main()
{
  /// SOLUTION - Problem 1 (mostly).

  /// SOLUTION
  //
  vec4 color =
    bool(In.is_edge) ? uc.color_edge :
    gl_FrontFacing ? uc.color_front : uc.color_back;

  // Get filtered texel, unless the fragment belongs to an edge primitive.
  //
  vec4 texel = bool(In.is_edge) ? vec4(1,1,1,1) : texture(tex_unit_0,In.tcoor);

  // If texel is too dark don't write fragment.
  //
  bool hole = texel.r + texel.g + texel.b < 0.05;
  if ( hole ) discard;
```

```
  // Multiply filtered texel color with lighted color of fragment.
  //
  out_frag_color =
    texel * generic_lighting(In.vertex_e, color, normalize(In.normal_e));
}

void
fs_main_orig()
{
  vec4 color = gl_FrontFacing ? uc.color_front : uc.color_back;

  // Get filtered texel.
  //
  vec4 texel = texture( tex_unit_0, In.tcoor );

  // Multiply filtered texel color with lighted color of fragment.
  //
  out_frag_color =
    texel * generic_lighting(In.vertex_e, color, normalize(In.normal_e));
}

#endif
```

# LSU EE 4702-1 Homework 5 Solution Due: ≈ 12 November 2014

The following assignment is based on the code package for Homework 4. The questions are asking about the code provided for the assignment, not about code written as part of the solution. In other words, you can answer these questions without having yet solved Homework 4.

The solution discussion refers to Homework 4 code. Parts are of the code are reproduced at the end of the solution. For HTML versions visit `https://www.ece.lsu.edu/koppel/gpup/2014/hw04.cc.html` for the CPU code and `https://www.ece.lsu.edu/koppel/gpup/2014/hw04-shdr.cc.html` for the shader code.

**Problem 1:** Compare the amount of data sent from CPU to GPU in Methods 0 (triangle strip, vertex contains coordinates) and 1 (triangle strip, simple geometry shader, vertex contains indices) from the Homework 4 code. (Note that a solution to Homework 4 does not change the amount of data sent from CPU to GPU.) *Note: An earlier version of the problem asked about Methods 1 and 2. The amount of data sent by 1 and 2 would be the same.*

Let $n$ denote the number of balls (`chain_length`) and $s$ the number of segments (`opt_segments`). Determine the amount of data, in bytes, sent to the GPU per frame for Method 0 (routine `render_spiral1`) and Method 1 (`render_spiral2`).

In the discussion below we'll ignore data sent for uniform variables and OpenGL commands. Uniform data includes explicitly declared uniforms, such as `opt_segments`, and pre-defined uniforms such as `gl_ModelViewMatrix`. Command data (which was not talked about much in class) includes whatever needs to be sent from CPU to GPU to set up a rendering pass, such as indicating the number vertices to expect. The amount of uniform and command data can be ignored because there is not much of it (though misusing uniforms and commands can slow things down). We'll also ignore texture data, since that's just sent once.

Method 0 renders using `render_spiral1`. By inspection of the code in `render_spiral1` we learn that with each vertex we are sending a texture coordinate, `glTexCoord2f`, a normal, `glNormal3fv`, and the vertex coordinate, `glVertex3fv`. Each of these is a vector of floats, a two-element vector for texture coordinates and three-element vectors for the normal and vertex coordinate. The total size per vertex is $2+3+3=8$ floats or $8\times 4 = 32\,\mathrm{B}$.

The `i` loop iterates `chain_length-1` or $n-1$ times, the `t` loop iterates `opt_segments` or $s$ times (see the code), and a `t` loop iteration sends 2 vertices, and so the total number of vertices sent per frame is $(n-1)s2$. The amount of data is therefore $\boxed{(n-1)s2\times 32\,\mathrm{B} = 64(n-1)s\,\mathrm{B}}$.

Method 1 renders using `render_spiral2` and the programmable shaders `vs_main`, `gs_main_simple`, and `fs_main`. The vertices are specified using `glDrawArrays` rather than `glBegin`. To find the amount of data, we need to tally the amount of data sent directly into the rendering pipeline (as we did for `render_spiral1`) and the amount of non-input data read by the shaders. The non-input data is from buffer object `balls_bo` (CPU name), which is bound to `balls_pos` in the shader code.

To find the data sent through the rendering pipeline we need to look at the client arrays that are enabled. Client arrays are enabled, of course, using `glEnableClientState`. For `render_spiral2` only one array is enabled, `GL_VERTEX_ARRAY`. This array is bound to buffer object `indices_bo`. The `glVertexPointer` command tells us that the size of a vertex coordinate is `3*sizeof(int)` or 12 bytes. The number of vertices sent is, as before, $(n-1)s2$.

**But** these vertices are coming from a buffer object, and inspection of the code indicates that the buffer object is only updated when `opt_segments` changes. We expect that buffer object data is sent from the CPU to GPU each time it is updated on the CPU. Since `opt_segments` only changes in response to user input we can assume that the buffer object is rarely updated (at least rare with respect to frame update). The size of the indices buffer object is $(n-1)s2\times 12\,\mathrm{B}$, but because it is rarely transferred we **will not include it** in our tally of CPU to GPU data.

As mentioned earlier, the vertex shader is reading the `balls_pos` buffer object. As we can see from the code, that buffer object is updated each time we call `render_spiral2`. (It's updated because the balls move.) The buffer object is an array of `pCoor` variables (which are arranged in memory identically to `vec4` variables). The size of an element is $4 \times 4\,\mathrm{B} = 16\,\mathrm{B}$ and the total size of the buffer object is $16n\,\mathrm{B}$.

The total amount of data sent from CPU to GPU for `render_spiral2` is therefore only $\boxed{16n\,\mathrm{B}\text{ per frame}}$. (Remember that we are ignoring uniform data, which for large $n$ would not make much of a difference.)

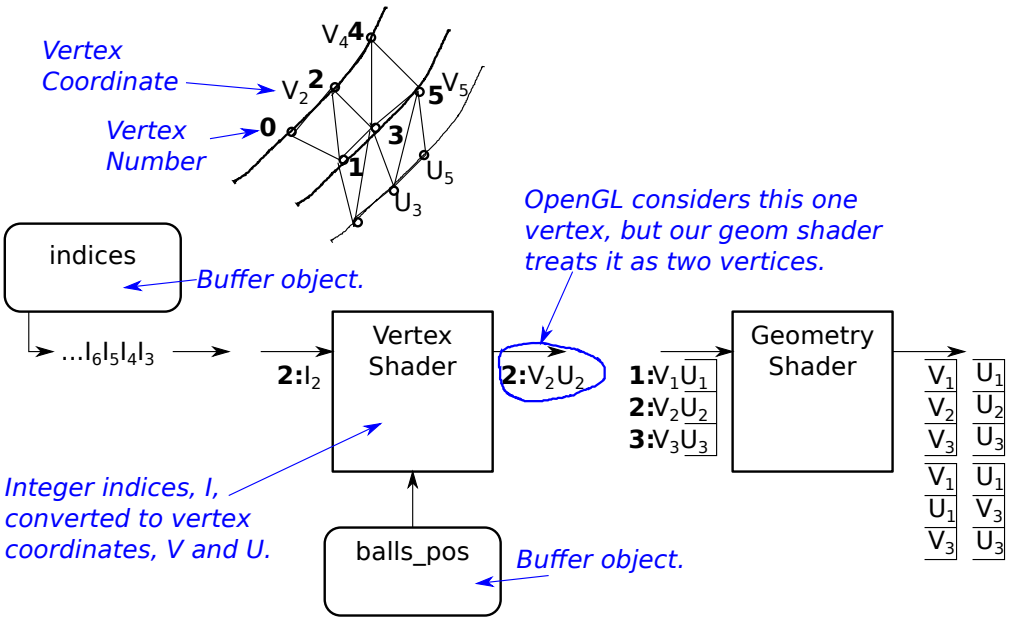
**Problem 2:** Consider the amount work to compute vertex coordinates and normals done by `render_spiral1` and by the vertex shader `vs_main`. The vertex shader is actually computing things multiple times that the code in `render_spiral1` computes just once. Identify such redundant computation. *Hint: It happens in two different ways, one way results in things computed twice, another way results in things computed $s$ times.*

For the factor of two duplication by `vs_main` look at the `t` loop in `render_spiral1`. Notice that it emits two vertices. Both of these vertices are computed from the same value of `radial` and the normal for these vertices are computed from the same value of `tangial`. In contrast, the vertex shader `vs_main` is called once for each vertex. Each time it is computing `radial` and `tangial`, so it is computing those values twice as many times as `render_spiral1`.

For the factor of $s$ duplication look at the code at the top of the `i` loop body in `render_spiral1`. Notice that values for `va` and `vb` are computed by `render_spiral1` once and then used in each of the $s$ iterations of the `t` loop. But in the `render_spiral2` version the vertex shader computes `va` and `vb` for each vertex. So `vs_main` is computing these values $2s$ times for frequently than it needs to.

**Problem 3:** Explain how some of the problems above could be avoided if the CPU specified vertices using line strips and the geometry shader used lines as inputs. Of course, the geometry shader's output would be triangles. Each point on the line would correspond to a `p` in the `t` loop in the `render_spiral1` routine.

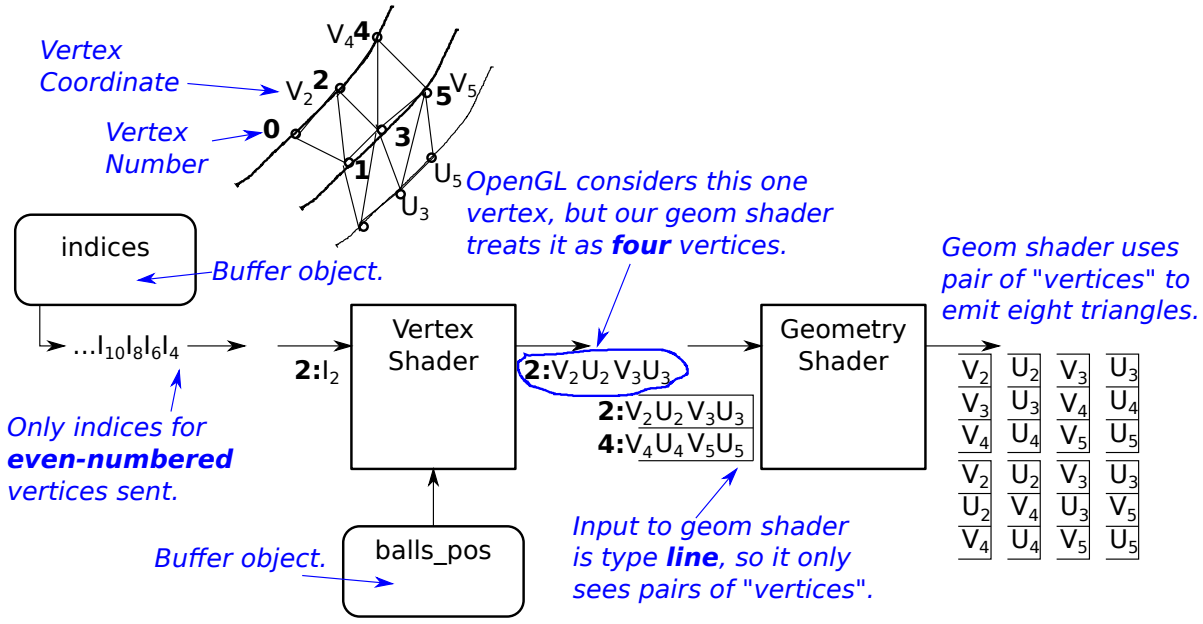
First, consider the operation of `render_spiral2`:




The buffer object is supplying indices for all of the vertices on one spiral, including the inner edge (even-numbered in illustration) and outer edge (odd numbered). The vertex shader computes the corresponding upper spiral coordinate, $U$, but as far as OpenGL is concerned the output of the vertex shader is still just one vertex, to be collected in to triangles

and passed to the geometry shader, which emits the upper and lower triangles, and the edge triangles. That's Homework 4.

In the solution to this problem we will only send the inner-rim vertices (even-numbered) to the vertex shader, organized as line strips:




The vertex shader will generate coordinates for the outer-rim vertices along with the corresponding upper vertices. So here the vertex shader output is four vertices (but still considered one vertex by OpenGL). The duplication of computation is avoided by having the vertex shader compute the inner and outer vertex coordinates together.

Our geometry shader just needs two of these 4-vertex vertices to construct two triangles each on the upper and lower spiral and two triangles each on the inner and outer edge. Line strips rather than triangle strips are being used because the geometry shader only needs two of our vertices.

On the CPU `render_spiral3` would be very similar to `render_spiral2` except for the following changes: The number of vertices sent, `num_elts`, would be half the amount. That is, rather than sending a vertex for the inner and outer part of the spiral, a vertex would only be sent for the inner part of the spiral.

```
//    CPU RENDER SPIRAL ROUTINES, FROM FILE hw04.cc
void
World::render_spiral1()
{
  glDisable(GL_COLOR_MATERIAL);
  glMaterialfv(GL_FRONT,GL_AMBIENT_AND_DIFFUSE,color_lsu_spirit_purple);
  glMaterialfv(GL_BACK,GL_AMBIENT_AND_DIFFUSE,color_lsu_spirit_gold);

  glActiveTexture(GL_TEXTURE0);
  glEnable(GL_TEXTURE_2D);
  glBindTexture(GL_TEXTURE_2D,texid_syl);

  float theta = 0;
  const float spiral_radius = 0.5;
  const float delta_t = 1.0 / opt_segments;
  const float omega = 10;

  glBegin(GL_TRIANGLE_STRIP);

  for ( int i=1; i<chain_length; i++ )
    {
      Ball *const ball1 = &balls[i-1];
      Ball *const ball2 = &balls[i];
      pCoor pos1 = ball1->position;
      pCoor pos2 = ball2->position;

      //
      // Render a spiral slide from position of ball1 to ball2.
      //

      pVect v12 = pos2 - pos1;

      // Find a vector that's orthogonal to v12.
      //
      pNorm ax = v12.x == 0 ? pVect(0,v12.z,-v12.y) : pVect(v12.y,-v12.x,0);
      pNorm ay = cross(v12,ax);

      pVect vx = ax * spiral_radius;
      pVect vy = ay * spiral_radius;

      for ( float t=0; t<0.999; t += delta_t )
        {
          pCoor p = pos1 + t * v12;
          theta += delta_t * omega;

          // Compute a vector from the spiral axis to a point on the spiral.
          //
          pVect radial = vx * cos(theta) + vy * sin(theta);

          // Find a vector in the "direction" of the spiral motion as
          // if t were time.
```

```
            //
            pVect tangial = -omega * vx * sin(theta) + omega * vy * cos(theta);
            //
            // Note: expression above is derivative of radial w.r.t. t.

            pCoor p_outer = p + radial;
            const float inner_frac = 0.5;
            pCoor p_inner = p + inner_frac * radial;
            pVect tang = v12 + tangial;
            pVect tang_inner = v12 + inner_frac * tangial;

            // Compute a vector pointing up.
            //
            pNorm norm = cross(radial,tang);
            pNorm norm_inner = cross(radial,tang_inner);

            const float du = 0.5 / chain_length;
            const float u = float(i) * du;

            glTexCoord2f(t,u+du);
            glNormal3fv(norm_inner);
            glVertex3fv(p_inner);

            glTexCoord2f(t,u);
            glNormal3fv(norm);
            glVertex3fv(p_outer);
          }
      }

  glEnd();

  vs_fixed->use();

  glEnable(GL_COLOR_MATERIAL);
}

void
World::render_spiral2()
{
  glDisable(GL_COLOR_MATERIAL);
  glMaterialfv(GL_FRONT,GL_DIFFUSE,color_lsu_spirit_gold);
  glMaterialfv(GL_FRONT,GL_AMBIENT,color_gray);
  glMaterialfv(GL_BACK,GL_DIFFUSE,color_lsu_spirit_purple);

  glActiveTexture(GL_TEXTURE0);
  glEnable(GL_TEXTURE_2D);
  glBindTexture(GL_TEXTURE_2D,texid_syl);

  if ( !balls_bo ) glGenBuffers(1, &balls_bo);
  glBindBuffer(GL_ARRAY_BUFFER, balls_bo);
  for ( int i=0; i<chain_length; i++ ) balls_pos[i] = balls[i].position;
```

```
  glBufferData
    (GL_ARRAY_BUFFER,chain_length*sizeof(balls_pos[0]),
     balls_pos, GL_DYNAMIC_DRAW);
  glBindBuffer(GL_ARRAY_BUFFER, 0);
  const int num_elts = ( chain_length - 1 ) * opt_segments * 2;

  if ( !indices_bo ) glGenBuffers(1, &indices_bo );

  if ( indices_segments != opt_segments )
    {
      indices_segments = opt_segments;
      struct ivec3 { int i, j, k; ivec3(){}; ivec3(int ip, int jp, int kp):i(ip),j(jp),k(kp){}};
      ivec3* const indices = new ivec3[ num_elts ];
      ivec3 *ip = indices;
      for ( int i=1; i<chain_length; i++ )
        for ( int t=0; t<opt_segments; t++ )
          for ( int inner=0; inner<2; inner++ )
            *ip++ = ivec3(i,t,inner);
      glBindBuffer(GL_ARRAY_BUFFER, indices_bo);
      glBufferData
        (GL_ARRAY_BUFFER,
         num_elts * sizeof(indices[0]),
         indices,
         GL_STATIC_DRAW);
      delete indices;
    }

  if ( opt_vtx_method == 1 )
    vs_geo_simple->use();
  else
    vs_geo_sol->use();

  glColor3fv(color_powder_blue);

  glBindBufferBase(GL_SHADER_STORAGE_BUFFER,1,balls_bo);
  glUniform1i(1,opt_segments);
  glUniform1i(2,chain_length);
  glUniform2i(3,opt_debug_x,opt_debug_y);
  glUniform1f(4,opt_debug_float);

  glBindBuffer(GL_ARRAY_BUFFER, indices_bo);
  glVertexPointer(3,GL_INT,3*sizeof(int),0);
  glEnableClientState(GL_VERTEX_ARRAY);
  glBindBuffer(GL_ARRAY_BUFFER, 0);

  glDrawArrays(GL_TRIANGLE_STRIP, 0, num_elts );

  glDisableClientState(GL_VERTEX_ARRAY);
  vs_fixed->use();
  glEnable(GL_COLOR_MATERIAL);
}
```

# LSU EE 4702-1 Homework 6 Solution Due: 24 November 2014

**Problem 0:** Follow the class account instructions for obtaining the assignment, substitute hw06 where needed. This homework code is based on Homework 4/3, and the physics simulation is identical. Like those assignments it renders a spiral. The spiral can be rendered using two routines, `render_spiral2` and `render_spiral3`; the `m` key switches between them. Routine `render_spiral2` is based on the Homework 4 solution, and initially `render_spiral3` is identical to `render_spiral2`, it will be modified for this assignment.

Routine `render_spiral2`, rather than sending actual vertex coordinates to the GPU, sends indices. The $x$ component of a vertex "coordinate" is the ball number (the value of `i` in `render_spiral1`), the $y$ component is the relative position between ball `i-1` and ball `i`, ranging from 0 to `opt_segments-1`, and the $z$ component indicates whether the vertex is on the inner or outer edge of the spiral. The code in vertex shader `vs_main` in file `hw06-shdr-triangles.cc` uses these values to compute the actual vertex coordinate and normal, and to compute the texture coordinates.

The level of detail used for the spiral is controlled by variable `opt_segments`. Its value can be modified by first pressing the `TAB` key until "Number of segments per spiral" appears, and then pressing `+` and `-` to adjust its value. Check out and compile the code, and make sure that it runs correctly.

**For this assignment** modify files `hw06.cc` and `hw06-shdr-lines.cc`. Solutions to the non-code questions can be provided in any of the following forms:

- As comments in one of the files above.
- As plain text in a separate file named `hw06.txt` in the `hw06` directory.
- In portable document format in a separate file named `hw06.pdf` in the `hw06` directory.
- Submitted on paper.
- E-mailed. If E-mailing, plain text and PDF are preferred.

**Problem 1:** The code in `render_spiral2` is based on the solution to Homework 4. As discussed in Homework 5 Problem 3, the vertex shader used for `render_spiral2` in Homework 4 wastes computation because it computes information both for inner and outer vertices but only sends one of these to the geometry shader. The problem suggests using line strips to solve that problem. The solution to Homework 5 presented some details on how this would be done, but it did not show any code. For this problem, modify `render_spiral3` and the shaders so that it avoids the waste.

For this problem you must modify code both in `render_spiral3` in file `hw06.cc` and the code in `hw06-shdr-lines.cc`, both will be collected by the TA-bot. Initially the code in `render_spiral3` is nearly identical to `render_spiral2` and `hw06-shdr-lines.cc`, except that it invokes shaders from file `hw06-shdr-lines.cc` (which you should modify). Routine `render_spiral2` invokes shader code from `hw06-shdr-triangles.cc`, which you should not modify.

Here are some things to watch out for:

- Be sure to adjust the maximum number of vertices specified for the geometry shader output. If this number is too low execution will end with an error.
- When you modify an interface block (such ad `Data_to_GS`) be sure to modify the other interface block with the same name (one is a shader output, the other is a shader input).

Here are some debugging tips:

- Check for shader code compilation errors when your program starts. The errors are sent to `stdout` and should appear in the shell or in gdb, depending on how you started the program.
- If execution ends with an OpenGL error, you can get a more detailed error message by turning on OpenGL debugging. To do this change `false` to `true` in `popengl_helper.ogl_debug_set(false);` near the end of `hw04.cc`.
- Three UI-controlled variables are available for debugging. They are `debug_bool.x`, `debug_bool.y`, and `debug_float`. The Booleans can be toggled with `d` and `D`. The float can be adjusted by pressing `Tab` until "Debug Float" appears and then press `+` and `-` to adjust the value.

The solution has been checked into the repo, look at files `hw06sol.cc` and `hw06-shdr-lines-sol.cc`. Web versions can be found at `https://www.ece.lsu.edu/koppel/gpup/2014/hw06sol.cc.html` and `https://www.ece.lsu.edu/koppel/gpup/2014/hw06-shdr-lines-sol.cc`, the repo versions will be most up to date.

Before reading this discussion be sure to review the solution to Homework 5, `https://www.ece.lsu.edu/koppel/gpup/2014/hw05_sol.pdf`, which describes how the code operates. Highlights are provided here.

Routine `render_spiral3` must be changed to send the vertices as a line strip instead of a triangle strip. Rather than sending an inner and an outer vertex, only one vertex will be sent. The $z$ component of a vertex was used to indicate whether it was an inner our outer vertex. That's no longer needed, so the vertex structure now only has two elements. Notice in the code below that the `struct` has two members, rather than 3.

```
struct ivec2 { int i, j; ivec2(){}; ivec2(int ip, int jp):i(ip),j(jp){}};
ivec2* const indices = new ivec2[ num_elts ];
ivec2 *ip = indices;
for ( int i=1; i<chain_length; i++ )
  for ( int t=0; t<opt_segments; t++ ) *ip++ = ivec2(i,t);
glBindBuffer(GL_ARRAY_BUFFER, indices_bo);
glBufferData
  (GL_ARRAY_BUFFER, num_elts * sizeof(indices[0]), indices, GL_STATIC_DRAW);
delete indices;
```

Other changed code in `render_spiral3`:

```
/// SOLUTION
//  Change number of components from 3 to 2.
glVertexPointer(2,GL_INT,2*sizeof(int),0);
glEnableClientState(GL_VERTEX_ARRAY);
glBindBuffer(GL_ARRAY_BUFFER, 0);

/// SOLUTION
//  Change primitive from triangle strip to line strip.
glDrawArrays(GL_LINE_STRIP, 0, num_elts );
```

The vertex shader was modified so that it sends four rather than two sets of coordinates to the geometry shader. These were declared using 2D arrays to simplify the code in the geometry shader. Here is the interface block between the two shaders:

```
in Data_to_GS {
  vec4 vertex_e[2][2];  // Vertex coordinates in eye space.
```

```
  vec4 position[2][2];  // Vertex coordinates in clip space.
  vec2 texCoord[2];
  vec3 normal_e[2];
  vec3 radial_e;  // Normal for edge primitives.
} In[];
```

Only two sets of texture coordinates and two sets of normals are needed (not four) because the upper and lower spiral use the same texture coordinates and normals. The variable `radial_e` stores the normal used for the edge (wall), that is the same in all four locations (except for a sign flip).

The vertex shader code assigns these elements, taking advantage of the 2D array to avoid code duplication when computing eye- and clip-space coordinates:

```
  // Compute the four vertex coordinates in object space.
  //
  vec3 pos_o[2][2];
  pos_o[0][0] = p_inner;
  pos_o[0][1] = p_outer;
  pos_o[1][0] = p_inner + depth_vector;
  pos_o[1][1] = p_outer + depth_vector;

  // Transform the object-space coordinates to clip and eye space.
  //
  for ( int l=0; l<2; l++ )
    for ( int r=0; r<2; r++ )
      {
        vec4 position_o = vec4( pos_o[l][r], 1 );
        position[l][r] = gl_ModelViewProjectionMatrix * position_o;
        vertex_e[l][r] = gl_ModelViewMatrix * position_o;
      }
```

Because of the way the coordinates are organized the geometry shader can emit the primitives using just two loop nests:

```
  /// Emit the spiral triangles.
  //
  for ( int level=0; level<2; level++ )         // Upper / Lower
    {
      for ( int theta=0; theta<2; theta++ )
        {
          for ( int r=0; r<2; r++ )             // Inner / outer
            {
              normal_e       = In[theta].normal_e[r];
              vertex_e       = In[theta].vertex_e[level][r];
              gl_Position    = In[theta].position[level][r];
              gl_TexCoord[0] = In[theta].texCoord[r];
              is_edge = false;
              EmitVertex();
            }
        }
      EndPrimitive();           // This completes a strip of two triangles.
    }

  /// Emit the edge (wall) triangles.
```

```
//
for ( int r=0; r<2; r++ )                    // Inner / outer
  {
    for ( int theta=0; theta<2; theta++ )
      {
        for ( int level=0; level<2; level++ ) // Upper / Lower
          {
            normal_e    = r==0 ? -In[theta].radial_e : In[theta].radial_e;
            vertex_e    = In[theta].vertex_e[level][r];
            gl_Position = In[theta].position[level][r];
            is_edge = true;
            EmitVertex();
          }
      }
    EndPrimitive();           // This completes a strip of two triangles.
  }
```

Only the highlights are shown above.

Note: Everything done in the vertex shader could have been done in the geometry shader. But doing all the work in the geometry shader is inefficient because each "vertex" emitted by the vertex shader (which actually stores four vertices) is used by two invocations of the geometry shader. If the geometry shader computed things like `vertex_e` then it would be doing twice as much work.

**Problem 2:** Compare the performance of `render_spiral3` to `render_spiral2`. To see any difference at all the value of `opt_segments` will need to be increased, so do this if necessary.

(*a*) Show a table comparing the performance of the two methods versus `opt_segments`. In the table show CPU and GPU OpenGL time. Also indicate which GPU you are using.

The table should show that it takes large values of `opt_segments` before `render_spiral3` shows any benefit.

(*b*) Explain how varying `opt_segments` changes the proportion of the work performed by the three shader stages (vertex, geometry, and fragment).

Increasing `opt_segments` increases the number of triangles, but those triangles are smaller. The total area of the triangles does not change.

The amount of work performed by the vertex shader (actually the number of invocations) is proportional to the number of vertices, and that increases with `opt_segments`. For our code the amount of work performed by the geometry shader also increases with `opt_segments`. However, the amount of work performed by the fragment shader is proportional to the number of fragments, and that is determined by area of the triangles when projected on the screen (frame buffer). That does not change with `opt_segments`. When `opt_segments` increases there are more triangles, but each triangle has fewer fragments, and so the amount of work does not change.

(*c*) In a solution to an exam problem, the computation of `sin(theta)` and `cos(theta)` was moved out of the vertex shader. Instead the pre-computed sine and cosine values would be read from a buffer object. Would this increase or decrease the relative benefit of `render_spiral3`?

Short answer: It would decrease the relative benefit of `render_sprial3` because vertex shader execution time itself would be less important.

Long Answer: Moving the sine and cosine operations out of the vertex shader significantly reduces the amount of execution time spent in the vertex shader. Perhaps `render_spiral2` vertex shader execution would drop from 20% of total rendering time to 3%. (Note: those numbers are totally made up.) If we switch from `render_spiral2` to `render_spiral3` with the trig operations, we drop from 20% to 11.1% of total time, a respectable improvement. But if using `render_spiral2` we are only consuming 3% of execution time then switching to `render_spiral3` will only drop that to 1.52% of execution time, a much smaller improvement.

**Problem 3:** When the Homework 6 code starts it prints information about the available GPUs, using a CUDA API. CUDA isn't used for anything other than printing this information. Here is some sample output:

```
GPU 0: Tesla K20c @ 0.71 GHz WITH 5119 MiB GLOBAL MEM
GPU 0: CC: 3.5  MP: 13  CC/MP: 192  TH/BL: 1024
GPU 0: SHARED: 49152 B  CONST: 65536 B  # REGS: 65536
GPU 0: L2: 1280 kiB   MEM to L2: 208.0 GB/s  SP 1760.9 GFLOPS  OP/ELT 33.86
GPU 1: GeForce GTX 780 @ 1.02 GHz WITH 3071 MiB GLOBAL MEM
GPU 1: CC: 3.5  MP: 12  CC/MP: 192  TH/BL: 1024
GPU 1: SHARED: 49152 B  CONST: 65536 B  # REGS: 65536
GPU 1: L2: 1536 kiB   MEM to L2: 288.4 GB/s  SP 2348.9 GFLOPS  OP/ELT 32.58
```

In the example above information on two GPUs is given. (My machine and all of the graphic lab computers have two GPUs.) The first GPU is a Tesla K20c and has Compute Capability 3.5. The number of streaming multiprocessors (abbreviated `MP` in the output) is 13. The maximum number of threads per block is 1024.

Suppose the time step routine (a version of `time_step_cpu`) were going to be run on the GPU.

(*a*) Based on the program's default parameters and a GPU on one of the lab machines, how many blocks should be launched and how many threads per block? (The number of blocks and number of threads per block are collectively called the *launch configuration*.) *Please* indicate which GPU and its relevant statistics when answering this question.

This answer is for the K20c shown above. WARNING: If you read the answer to this subproblem you MUST read the answer to the next one. If you do not agree to do this, STOP READING NOW.

Looking in the time step routine we see two outer loops iterating over `chain_length` elements. An obvious thing to consider is to assign each iteration to a thread. The default value of `chain_length` is only 18. The number of blocks should be a multiple of the number of multiprocessors. There are 13 MPs so there should be 13 blocks. The 18 threads should be divided between these blocks, so we need two threads per block (in eight blocks only one thread would be active).

(*b*) Do you expect the code to run efficiently with this launch configuration? Explain.

No. No! NO! Efficiency increases with the number of threads. Because of the way threads are scheduled by the hardware, efficiency is better when the number of threads per block is a multiple of 32. Though the number of threads needed to realize efficiency depends on the type of GPU and especially the characteristics of the code, a good ballpark minimum is 256 threads per block.

Therefore two threads per block will yield really bad performance. The CPU would execute the code much faster.

```
/// LSU EE 4702-1 (Fall 2014), GPU Programming
//
 /// Homework 6
 //
 /// Ported to Vulkan Fall 2021

 /// SOLUTION

 //  Assignment: http://www.ece.lsu.edu/koppel/gpup/2014/hw06.pdf
 //  Solution Discussion: http://www.ece.lsu.edu/koppel/gpup/2014/hw06_sol.pdf


// Specify version of OpenGL Shading Language.
//
#version 460

#extension GL_GOOGLE_include_directive : enable
#include <light.h>
#include <transform.h>
#include "shdr-common.h"
#include "shdr-generic-lighting.h"

layout ( binding = BIND_UNI_COMMON ) uniform UC
{
  Shdr_Uni_Common uc;
};

// Use this variable to debug your code. Press 'd' to toggle
// debug_bool.x and 'D' to toggle debug_bool.y (between true and
// false).
//
bvec2 debug_bool = bvec2( bool(uc.debug_bool.x), bool(uc.debug_bool.y) );

// Use this to debug your code. Press TAB until "debug_float"
// appears, then press +/- to adjust its value.
//
float debug_float = uc.debug_float;

// Array of ball positions.
//
layout ( binding = BIND_BALLS_POS ) buffer Balls_Pos { vec4 balls_pos[]; };


#ifdef _VERTEX_SHADER_

// Redefine this vertex shader input to be an integer vector.
//
 /// SOLUTION
//
//   Reduce in_indices from four to two components since the third
//   component was used to indicate whether to compute an inner or outer
//   vertex, and now we always do both.
layout ( location = LOC_IN_INT2 ) in ivec2 in_indices;
```

```
// Interface block for vertex shader output / geometry shader input.
//
layout ( location = 0 ) out Data_to_GS
{
  /// SOLUTION
  //
  //  Change vertex_e from a vec4 to a 2-D array of vec4's.  The
  //    four values are for inner/outer, and upper/lower vertices.
  //    Indexing:  [level (upper/lower)] [ radius (inner/outer) ]
  //
  //  Also declare four position vectors (and don't bother using gl_Position).
  //
  vec4 vertex_e[2][2];  // Vertex coordinates in eye space.
  vec4 vertex_c[2][2];  // Vertex coordinates in clip space.

  //  Only two texCoord and normal_e vectors are declared because the
  //  upper and lower spirals have the same texture coordinates and normals.
  //
  vec2 texCoord[2];
  vec3 normal_e[2];

  //  All four vertices have the same radial vector.
  //
  vec3 radial_e;  // Normal for edge primitives.
};


void
vs_main_lines()
{
  const float spiral_radius = 0.5;
  const float omega = 10;

  const int bidx = in_indices.x;  // i in the CPU code, the ball number.
  const int ti = in_indices.y;    // Integer version of t from CPU code.

  // Position relative to top of whole spiral. Used only to compute angle.
  const int radial_idx = bidx * uc.opt_segments + ti;

  const float delta_t = 1.0 / uc.opt_segments;
  // Position relative to top of segment: 0 top, 1 bottom, etc.
  const float t = float(ti) * delta_t;
  const float theta = delta_t * radial_idx * omega;

  vec3 pos1 = balls_pos[bidx-1].xyz;
  vec3 pos2 = balls_pos[bidx].xyz;

  vec3 v12 = pos2.xyz - pos1.xyz;

  // Find a vector that's orthogonal to v12.
  //
  vec3 ax =
    normalize(v12.x == 0 ? vec3(0,v12.z,-v12.y) : vec3(v12.y,-v12.x,0));
```

```
// Find a vector that's orthogonal to v12 and ax.
//
vec3 ay = normalize(cross(v12,ax));

vec3 vx = ax * spiral_radius;
vec3 vy = ay * spiral_radius;

// Point on line between ball1 and ball2.
//
vec3 p = pos1 + t * v12;

// Vector from p to spiral outer edge.
//
vec3 radial = vx * cos(theta) + vy * sin(theta);
vec3 p_outer = p + radial;

const float inner_frac = 0.5;
vec3 p_inner = p + inner_frac * radial;

// Compute surface normals.
//
vec3 tangial = -omega * vx * sin(theta) + omega * vy * cos(theta);
vec3 tang = v12 + tangial;
vec3 tang_inner = v12 + inner_frac * tangial;
vec3 norm = normalize(cross(radial,tang));
vec3 norm_inner = normalize(cross(radial,tang_inner));

/// SOLUTION
//
//  Compute information for the four vertices.
//
//  Indexing:  [level (upper/lower)] [ radius (inner/outer) ]

vec3 v12n = normalize(v12);
vec3 depth_vector = 0.1f * v12n;

// Compute the four vertex coordinates in object space.
//
vec3 pos_o[2][2];
pos_o[0][0] = p_inner;
pos_o[0][1] = p_outer;
pos_o[1][0] = p_inner + depth_vector;
pos_o[1][1] = p_outer + depth_vector;

// Transform the object-space coordinates to clip and eye space.
//
for ( int l=0; l<2; l++ )
  for ( int r=0; r<2; r++ )
    {
      vec4 position_o = vec4( pos_o[l][r], 1 );
      vertex_c[l][r] = gl_ModelViewProjectionMatrix * position_o;
      vertex_e[l][r] = gl_ModelViewMatrix * position_o;
    }
```

```
  normal_e[0] = gl_NormalMatrix * norm_inner;
  normal_e[1] = gl_NormalMatrix * norm;

  // Amount by which to zoom the texture.
  //
  float tex_zoom = 0.5;

  // Uncomment the line below to use "debug_float" to zoom text.
  // tex_zoom /= debug_float;

  const float du = 0.5 * tex_zoom / uc.chain_length;
  const float u = float(bidx) * du;

  texCoord[0].x = texCoord[1].x = tex_zoom * t;
  texCoord[0].y = 0.18 + u;
  texCoord[1].y = 0.18 + u + du;

  radial_e = gl_NormalMatrix * radial;
}

#endif



#ifdef _GEOMETRY_SHADER_

layout ( location = 0 ) in Data_to_GS
{
  /// SOLUTION
  //
  //  See "out Data_to_GS" for a detailed discussion.

  vec4 vertex_e[2][2];  // Vertex coordinates in eye space.
  vec4 vertex_c[2][2];  // Vertex coordinates in clip space.
  vec2 texCoord[2];

  vec3 normal_e[2];

  vec3 radial_e;  // Normal for edge primitives.
} In[];

layout ( location = 0 ) out Data_to_FS
{
  vec3 normal_e;
  vec4 vertex_e;
  vec2 texCoord;
  flat int is_edge;  // True if primitive an inner or outer edge.
};

 /// SOLUTION
//
//   Change input primitive type to lines.  The reason for using lines,
//   remember, is because that's the most efficient way of passing
```

```
//   the needed vertices to the geometry shader.
//
layout ( lines ) in;

 /// SOLUTION
//
//   Set max_vertices.
layout ( triangle_strip, max_vertices = 16 ) out;

void
gs_main_lines()
{
  /// SOLUTION
  //
  // Use the two sets of four vertices to construct the upper and
  // lower spiral and edges. The vertices are organized into a
  // two-dimensional array in such a way that the spiral and edge
  // triangles could each be rendered with a single loop nest.

  /// Emit the spiral triangles.
  //
  for ( int level=0; level<2; level++ )         // Upper / Lower
    {
      for ( int theta=0; theta<2; theta++ )
        {
          for ( int r=0; r<2; r++ )             // Inner / outer
            {
              normal_e       = In[theta].normal_e[r];
              vertex_e       = In[theta].vertex_e[level][r];
              gl_Position    = In[theta].vertex_c[level][r];
              texCoord       = In[theta].texCoord[r];
              is_edge = 0;
              EmitVertex();
            }
        }
      EndPrimitive();           // This completes a strip of two triangles.
    }

  /// Emit the edge (wall) triangles.
  //
  // Because we know that In[0] is in back of In[1] there is no need
  // for the vertex sorting code used in the Homework 4 solution. Just
  // one simple loop nest.
  //
  for ( int r=0; r<2; r++ )                     // Inner / outer
    {
      for ( int theta=0; theta<2; theta++ )
        {
          for ( int level=0; level<2; level++ ) // Upper / Lower
            {
              normal_e    = r==0 ? -In[theta].radial_e : In[theta].radial_e;
              vertex_e    = In[theta].vertex_e[level][r];
              gl_Position = In[theta].vertex_c[level][r];
              is_edge = 1;
```

```
              EmitVertex();
            }
          }
        EndPrimitive();           // This completes a strip of two triangles.
      }
}

#endif


#ifdef _FRAGMENT_SHADER_

layout ( location = 0 ) in Data_to_FS
{
  vec3 normal_e;
  vec4 vertex_e;
  vec2 texCoord;
  flat int is_edge;
};

layout ( binding = BIND_TEXUNIT ) uniform sampler2D tex_unit_0;
layout ( location = 0 ) out vec4 frag_color;

void
fs_main()
{
  // Choose color based on which side of spiral we are rendering.
  //
  vec4 color =
    bool(is_edge) ? uc.color_edge :
    gl_FrontFacing ? uc.color_front : uc.color_back;

  // Get filtered texel, unless the fragment belongs to an edge primitive.
  //
  vec4 texel = bool(is_edge) ? vec4(1,1,1,1) : texture(tex_unit_0,texCoord);

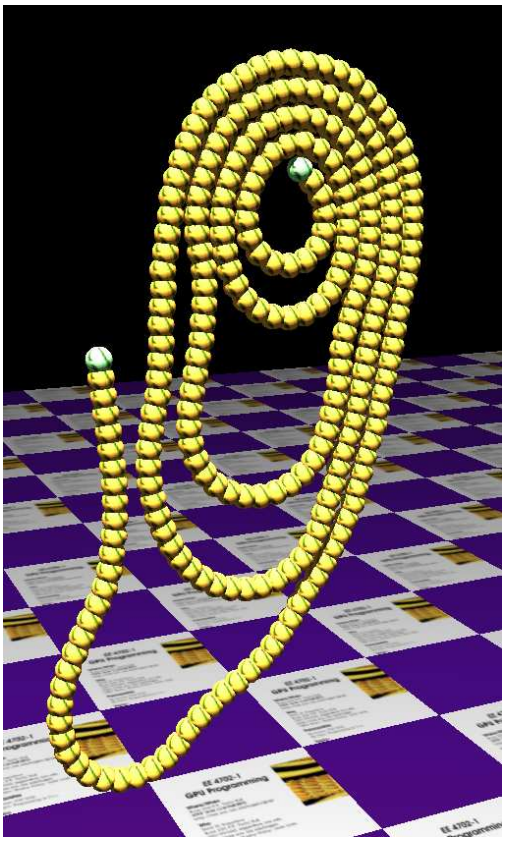
  // If texel is too dark don't write fragment, leaving a hole.
  //
  bool hole = texel.r + texel.g + texel.b < 0.05;
  if ( hole ) discard;

  // Multiply filtered texel color with lighted color of fragment.
  //
  frag_color = texel * generic_lighting(vertex_e, color, normalize(normal_e));
}

#endif
```

# 29 Fall 2013 Solutions

# LSU EE 4702-1 Homework 3 Solution Due: 20 November 2013



The solution has been checked into the repository, use `svn update` to retrieve it. Running `make` will build two executables, `hw3`, the unsolved version, and `hw3-sol` which includes the solution. The solution itself is in files `hw3-sol-cuda-sol.cu` and `hw3-sol.cc` (there is no `hw3-sol.cuh`).

In the solution pressing `x` will turn contact pair computation on and off, the results should be dramatic. Pressing `X` will turn the use of structures partially on and off. The effect should be small. See the solution for details.

The solution that follows includes excerpts from the code in the repository, sometimes shortened for clarity. The files can be accessed directly from the repository, preferably using an svn client, at `https://svn.ece.lsu.edu/svn/gp/hw/hw3` or after the semester ends at `https://svn.ece.lsu.edu/svn/gp/hw-archive/gpup-2013/hw3`.

HTML versions of the solution code, which may not be updated, are at `http://www.ece.lsu.edu/koppel/gpup/2013/hw3-sol-cuda-sol.cu` and `http://www.ece.lsu.edu/koppel/gpup/2013/hw3-sol.cc`.

**Problem 0:** Follow the instruction on the `http://www.ece.lsu.edu/koppel/gpup/proc.html` page for programming homework work flow, substituting hw3 where appropriate. Be sure to update the include directory in addition to checking out the homework.

Compile and run the homework code unmodified. It should initially show a string of balls in the shape of a spiral with the first and last ball fixed in space, the spiral will unravel, see the screenshot to the right.

Unless the code is run on a surprisingly fast system, the frame rate should be low, resulting in jerky movement. The physics is computed on the GPU using CUDA, the goal of this assignment is to improve the performance of the code so that animation is smooth.

For this problem, familiarize yourself with the operation of the simulation (that is, play a little). To move around the scene press `e` and then the arrow keys, PageUp, and PageDown. Press `1` to set up scene 1 (vertical string), press `2` to set up scene 2 (pendulum), press `3` to set up scene 3 (the unraveling spiral). 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. Pressing the space bar will pause and then single-step the simulation. Press `p` to resume (or to pause again).

To facilitate experimentation two Boolean variables have been provided, on CPU code `c.opt_test1` and `c.opt_test2`, on CUDA code `dc.opt_test1` and `dc.opt_test2`. Their values are shown on the second line of the display after opt_testX. To change their values press `x` or `X`.

Pressing `a` will switch the method used to run the physics (time step) routine, the method appears near the upper left (near Physics On:). Initially it will show GPU Sol, indicating the code that should be modified for this problem. The other two values are CPU and GPU Base. The GPU Base code is a copy of the code to be modified. As you work on this assignment you can compare the performance of your code, GPU Sol, to the original code, GPU Base, by switching between the two. If the CPU code was compiled without optimization then unoptimized will flash when running the CPU code. Since we are not comparing to the CPU in this problem, the CPU code can be left unoptimized.

A part of the solution involves choosing good values for the grid size (number of blocks) and block size (number of threads per block). These can be modified from the UI by pressing the `TAB` until Grid Size or

Block Size appears, and then by using the + and - keys to adjust them. Gravity can also be adjusted, setting it to a low value provides something like slow motion.

The performance of the physics code is shown on the second line to the left of Time Per Time Step, it is the amount of time needed to compute a single simulation time step. The goal, of course, is to minimize this value. The first line shows other performance numbers. FPS shows the frame rate in frames per second. XF indicates the rate at which frames are updated by the code relative to how fast the device displays frames. A value of 1 is optimal. Under some circumstances the class library code may not be able to synchronize with the hardware, in which case it will update at 30 FPS. GPU.GL shows the amount of time spent on the GPU running OpenGL code per frame, it is shown in milliseconds and as a percent of time. What we do in this assignment should not directly affect this number. GPU.CU shows the amount of time running CUDA code per frame (and as a percent of total time). **Do not** use this number to compare the performance of your code because its value depends on the number of time steps performed. CPU shows the amount of time per frame spent by the CPU on rendering. It does not include physics. Steps / s shows the number of simulation time steps performed per second. This number should be about 10000, if it's lower then the physics code is too slow to keep up. This will initially be the case for the GPU sol code, but when the assignment is finished it should be up to around 10000. When Steps / s is less than 10000 the simulation will appear to be running slowly (like slow motion) and motion may be jerky. Physics shows the amount of time spent computing physics, whether on the CPU or GPU.

When solving this assignment focus on Time Per Time Step, it can always be used to compare the performance of the different code versions (CPU, GPU Base, and GPU Sol).

The first place to look for CUDA documentation is the CUDA C Programming Guide, `http://docs.nvidia.com/cuda/cuda-c-programming-guide/`, start with Chapter 2, Programming Model, it provides a quick overview of CUDA concepts. See also Section 3.2. CUDA C is summarized in Appendix B. The link above takes you to a place providing documentation for the entire CUDA Toolkit. A complete list of CUDA runtime functions (the ones we call from CPU code) can be found in the CUDA Runtime API manual. The Programming Guide describes CUDA C, but a complete list of library functions callable from CUDA C can be found in the CUDA Math API manual.

For this assignment the only CUDA runtime calls needed are for allocating memory, and examples of those can be found in the assignment code.

In this assignment the GPU Sol kernel and related code will be modified to improve performance. Here is a summary of how the code works.

The CUDA code is located in file `hw3-sol-cuda.cu`. It consists of four kernels. At the beginning of every frame kernel `data_pack` is called. It initially does nothing but should be modified to *pack* some of the ball data. Kernel `time_step_gpu_sol_part_1` updates the balls' velocities, it is launched once per time step. (There are many time steps per frame.) Kernel `time_step_gpu_sol_part_2` updates positions (and also velocities for platform collision), it is also called once per time step. Kernel `data_unpack` is called once per frame, after the other kernels, it initially does nothing but should be modified to *unpack* data back into the `Ball` structure. These kernels are launched from the routine `frame_callback` in file `hw3.cc`.

In kernels `time_step_gpu_sol_part_1` and `time_step_gpu_sol_part_2` each thread handles one ball. There are `c.chain_length` balls, which is set to 256. As we should know, 256 is not a large number of threads for a GPU, and so we should expect the GPUs hardware to be underutilized.

The block and grid sizes are initially set to arbitrary values and these can be adjusted by using the UI, initial values can be changed by modifying routine `World::init` in `hw3.cc`.

Several variables are used by both GPU and CPU code. For convenience, they have been placed in structure `CPU_GPU_Common` (see file `hw3.cuh`), this structure is instantiated in `World`, as `c`, and is a CUDA constant, named `dc`. One member is `chain_length`, which is the number of balls. It can be accessed on the CPU as `c.chain_length` and on the GPU as `dc.chain_length`. Members may be added to this structure. The structure is copied to the GPU by routine `data_cpu_to_gpu_constants`, which is called at initialization and whenever a key is pressed.

CUDA runtime API functions (the cuda functions called by CPU code) are wrapped by a macro named `CE`. This checks the return code of the CUDA function for errors. If there is an error it will end execution with a message.

The `Makefile` is set to compile the code without CPU optimization but with CUDA optimization. If

you would like to use the cuda debugger (`cuda-gdb`), you will need to modify `Makefile`. Add a `-G` to the `OPT_FLAGS` variable. (You can uncomment a line with this flag added.) Then re-build (press `F9` or type make from the command line). The code will run excruciatingly slowly, so make sure your program reaches the problematic part quickly.

Adding `-O3` to `OPT_FLAGS` will turn on CPU optimization.

In the wee hours of the 21st tnphe TA-bot will copy the following files: `hw3-sol-cuda.cu`, `hw3.cc`, and `hw3.cuh`. Limit changes to these files. If code in another file needs to be modified, contact Dr. Koppelman.

You can place text answers to questions in file `hw3-sol-cuda.cu`, either in comments or defined out. You can also E-mail or turn in paper copies.

**Problem 1:** Using the UI modify the grid and block sizes and observe the impact on performance.

(*a*) Run the GPU Base code and manually set the grid and block sizes to what you would expect to be good choices. Note the performance. Try varying these to get worse or better performance. Note which values are better, and examples of ones which are worse. Note how sensitive the program is. Does it do just as well even with fewer blocks?

For the code to run correctly it needs at least 256 threads. Reducing the block and grid sizes so that their product is less than 256 will result in incorrect execution.

Generally, the grid size should be a multiple of the number of multiprocessors and the block size should be as large as possible, at least 256. Looking at the beginning of the `part_1` and `part_2` kernels we can see that no matter how many threads are launched only 256 (or chain length) will be active. Therefore changing the total number of threads will have no effect.

Changing the grid size, however, will affect where the threads run. For example, if the grid size is 1 and the block size is 256 all the threads will run on one multiprocessor. If the grid size is 2 and the block size is 128, the threads will be spread across two multiprocessors. But, if the grid size is 2 and the block size is 256 the threads will be on one multiprocessor (because of the way the code is written, see the code fragment below). Therefore, to have any kind of an impact when adjusting the grid size the block size would have to be chosen so that the product is 256. (Any lower and the code would not run correctly, any higher and the threads doing work would not use all of the multiprocessors.)

Still, choosing grid and block sizes this way results in little change in performance. That is because 256 threads is barely enough to utilize the resources on one multiprocessor, so splitting the threads to multiple multiprocessors does not help much.

```
__global__ void
time_step_gpu_base_part_1(float delta_t)
{
  const int tid = threadIdx.x + blockIdx.x * blockDim.x;
  const int bi = tid; // Ball Index

  if ( bi >= dc.chain_length ) return;
```

(*b*) Modify the code (in routine `World::init`) so that it starts off with a good configuration. Look for comments "Helpful information" and "Put Solution Here" in this routine. (The solution to this part will have to be changed to reflect modifications made due to the solution to Problem 3. If Problem 3 isn't solved this part can still get full credit. If you prefer don't work on this part until after Problem 3.)

The solution to this part is based on the solution to Problem 3. The `part_1` kernel for the solution to Problem 3 will keep as many threads as busy as possible, up to the maximum of $256^2 = 65536$ for the default chain length of 256. Since it is possible to keep that many threads busy, here we will launch as many threads as possible in each block, and set the grid size (the number of blocks) equal to the number of processors.

Looking at the code towards the end of `World::init` we see a variable that holds the number of multiprocessors with a comment encouraging us to use it:

```
  // Helpful information for solution.
  //
  const int number_of_multiprocessors = cuda_prop.multiProcessorCount;

  Kernel_Info *ki_part_1 = NULL;
  for ( int i=0; i<gpu_info.num_kernels; i++ )
    if ( pString(gpu_info.ki[i].name) == "time_step_gpu_sol_part_1" )
```

```
    ki_part_1 = &gpu_info.ki[i];

const int max_threads_part_1 =
  ki_part_1 ? ki_part_1->cfa.maxThreadsPerBlock : 0;
```

Notice that there is also a variable that gives the maximum number of threads per block possible when launching the `part_1` kernel. We'll use that to set the block size:

```
/// SOLUTION

// Set the grid size to the number of multiprocessors.
//
opt_grid_size = number_of_multiprocessors;

// Set the block size to the maximum number of threads possible for
// the part_1 kernel.
//
opt_block_size = max_threads_part_1;
```

The block and grid size apply to all kernels that are launched but we chose the block and grid size just for the `part_1` kernel that we wrote for Problem 3. That means the block and grid size for the other kernels are larger than they have to be. Because those other kernels only need 256 threads, it doesn't matter if they use one multiprocessor or several (as we saw in part (a)). So rather than have separate block and grid size variables for the `part_2`, `pack`, and `unpack` kernels, we'll just launch them with non-optimal sizes that anyway won't make a difference.

**Problem 2:** The CUDA code accesses the same `Ball` structure as the CPU code, which, though convenient, is not GPU-friendly because memory requests will only partially be used. Fix it. *Spoiler: It won't help performance.*

(*a*) Modify the code to use a GPU-friendly organization for `Ball::position`.

For your convenience, a variable is already present for the new storage for ball positions, `c.d_pos` (CPU name) and `dc.d_pos` (GPU name).

- You'll need to allocate device storage for the positions, do that in routine `World::init`. Device storage for `balls` is already allocated in that routine, use that code as an example.
- Modify kernel `data_pack` to reorganize the data for the GPU (reading from the `Ball` structure and writing to your new array).
- Modify `time_step_gpu_sol_part_1` and `time_step_gpu_sol_part_2` so that they read and write positions from and to your new array.
- Modify `data_unpack` so that it puts data back into the `Ball` structures.
- Optional: include the ball radius in the $w$ component of position and use it where needed.

The line below in `hw3.cc` allocates the storage:

```
// Allocate storage for the "flat" position array.
//
CE( cudaMalloc( &c.d_pos, c.chain_length*sizeof(c.d_pos[0])) );
```

The pack and unpack routines contain the following lines:

```
// data_pack added lines:
Ball* const ball = &dc.d_balls[bi];
dc.d_pos[bi] = ball->position;
dc.d_pos[bi].w = ball->radius;

// data_unpack added lines:
```

```
Ball* const ball = &dc.d_balls[bi];
ball->position = dc.d_pos[bi];
ball->position.w = 1;
```

Code that had been reading `ball->position` must now be changed to read `d_pos`. Here are some sample changed lines:

```
// In part_1
// Original code:
const float4 ball_position = ball->position;
// Modified code:
const float4 ball_position = dc.d_pos[bi];

// In part_2
// Original code:
ball->position += ball->velocity * delta_t;
// Modified code:
dc.d_pos[bi] += ball->velocity * delta_t;
```

(*b*) Measure the performance change by comparing GPU Sol (the code you modified) to GPU base. Explain why the change should not have made much of a difference in this code. (This can change depending on how the next problem is solved.) To answer this question look at the part of the code that's consuming the most execution time. It's only a few lines.

In the solution checked in to the repository a user can switch between the two methods of accessing positions by pressing `X` (upper-case). If the green text to the right of `opt_testX` shows `01` then the contact pair (penetration) code accesses the position from the structure, if the text shows `00` then its using the array `d_pos`. (The lower-case `x` turns the contact code on and off.)

The difference in performance is very small. On a Kepler K20c the time per time step is $30.7\ \mu$s when accessing the array and $31.4\ \mu$s when accessing the structure.

The explanation for this small performance difference can be seen in the part of the code consuming most of the execution time, the contact pair loop. Here is an excerpt of the code in the contact pair loop (the original one):

```
for ( int j=0; j<dc.chain_length; j++ ) if ( abs(j-bi)>1 ) {
      Ball* const ball2 = &dc.d_balls[j];
      pNorm ball_to_2 = mn(ball_position,ball2->position);
      const float rsum = ball_radius + ball2->radius;
      if ( rsum * rsum > ball_to_2.mag_sq )
          force -= ( rsum - ball_to_2.magnitude) * 1000 * ball_to_2;
    }
```

In the original code each thread executes that loop for 256 iterations, and one can assume that it is responsible for most of the execution time. So why doesn't replacing `ball2->position` with `dc.d_pos[j]` have much of an impact?

**The short answer is:** because all threads in a warp are accessing the same element, `j`, and so it doesn't matter if adjacent elements are adjacent in memory.

Here is the more detailed explanation. Global memory is accessed by forming requests. (In class a request was illustrated using a guy with a tray.) A *request* has a starting address and a size. On current generation Kepler systems the request size can be 32, 64, or 128 bytes. On the older Fermi systems (which we have in the lab) the request size normally is just 128 bytes. A request is formed on behalf of the threads in a warp (or half warp on some systems). If all 32 threads are accessing the same location then there will just be a single request. If the 32 threads are each accessing one byte and the memory locations are consecutive (say, they are accessing `char_array[threadIdx.x]`), then there will be one 32-byte request. But, if each thread is accessing one byte but the addresses are very different, (say, they are accessing `char_array[1024*threadIdx.x]`), then 32 requests will be needed. Since the minimum request size is 32 bytes, the total amount of memory accessed will be $32 \times 32 = 1024$ bytes. For Fermi systems the waste is greater, $32 \times 128 = 4096$ bytes.

Remember that within a warp all threads are at the same place in the program and so in the code above when the threads in a warp access `ball2->position` they will all have the same value of `j` and so will be accessing the same location. The amount of data needed will be 16 bytes (four floats which are four bytes each). This can be satisfied with a single request. The request size is exactly the same when the array is used. Because the request sizes (and number of requests) are the same in both cases we don't expect the performance of the contact pair loop to be affected.

Elsewhere in the code the use of an array for positions does have an advantage. Consider the code below from the top of the `part_1` routine:

```
const int bi = threadIdx.x + blockIdx.x * blockDim.x;
Ball* const ball = &dc.d_balls[bi];
const float4 ball_position = ball->position;  // Structure access.
const float4 ball_position = dc.d_pos[bi];    // Array access.
```

Consider consecutive threads in a warp. (Consecutive threads have consecutive values of `threadIdx.x` [say, 0, 1, 2, ...] and the same value of `blockIdx.x` [say, all are equal to 12].) First, consider access to `d_pos`. We are accessing 32 consecutive elements for a total size of $32 \times 16 = 512$ bytes. That can be serviced with four 128-byte requests, and every byte will be used. (Ignoring the `w` component.)

Next, consider the access to `ball->position`. The size of the `Ball` structure is 48 bytes, but the line of code we are interested in only accesses 16 of those bytes. To to retrieve `ball->position` for a warp 32 requests of size 32 will be used, for a total size of 1024 bytes. This will consume twice as much bandwidth for the same amount of data, slowing execution. The waste is even larger when accessing scalar values such as `mass_inv`, there only 4 bytes of each 32-byte request is used.

This waste has a small impact because execution time is dominated by the contact-pair loop in which, as we have seen, structure access is not a problem.

The answer to this question can change depending on how Problem 3 was solved. For the posted solution the answer **does not** change. Consecutive threads still access the same element.

**Problem 3:** This is the problem where performance will be improved by a substantial amount. The code uses a brute-force $O(n^2)$ method (where $n$ is the number of balls) for resolving interpenetration—that we won't change. The GPU Base and Sol kernels use one thread per ball, so each of these $n$ threads has to check almost $n$ other balls for collisions. We know that $n = 256$ (the number of balls the code uses) threads is not enough to get good utilization on current NVIDIA GPUs.

Modify the code so that the `time_step_gpu_sol_part_1` kernel can use more than $n$ threads. For example, if it were launched with $2n$ threads, a pair of threads would handle one ball and each one would only need to check about $n/2$ other balls for contact. Try to write your code to use an arbitrary number of threads per ball, this can be based on a hard-coded constant (`const int thds_per_ball=4;`) or the code might compute this based on the total number of threads (this is what the spring demo does).

Be sure to update the launch configuration so that it chooses a good block and grid size for your new kernel. (If you like, you can provide a separate configuration for `time_step_gpu_sol_part_1` and the other kernels.)

The spring demo code solves a similar problem, but **unlike the spring demo** the `time_step_gpu_sol_part_1` kernel must compute both near neighbor forces and resolve penetration.

The first step is to assign balls to threads. A thread has is assigned a ball for which it is responsible for computing forces, the ball number is given by `bi` in both the original code and the solution. In the solution we want multiple threads to operate on the same ball and we would like all threads operating on the same ball to be in the same block. To realize that we first compute how many balls will be handled by each block (based on the grid size) and then how many threads would be assigned to each ball (based on the block size):

```
const int balls_per_block = ( gridDim.x - 1 + dc.chain_length ) / gridDim.x;
const int thds_per_ball = blockDim.x / balls_per_block;
```

Based on the number of balls per block and the block index we can find the smallest numbered ball handled by any thread in a block:

```
const int bi_block = blockIdx.x * balls_per_block;
```

Based on the code so far, a block will handle balls numbered `bi_block` to `bi_block+balls_per_block-1`. There are two reasonable ways to assign these to threads. The solution below assigns consecutive threads to consecutive balls:

```
const int b_local = threadIdx.x % balls_per_block;
const int bi = bi_block + b_local;
```

An alternative would assign consecutive threads to the same ball, that would be achieved by setting `b_local=threadIdx.x/thds_per_ball;`.

The reason for assigning consecutive threads to consecutive balls is to make the contact pair loop more efficient. That is, if the threads in a warp are all responsible for computing forces for different balls then they can each access *the same "ball2"* for comparison.

The line below computes the index of the first ball accessed in the contact pair loop:

```
const int ai_first = threadIdx.x / balls_per_block;
```

Since there are several threads assigned to a ball we need to pick one to handle gravitational force, in the solution that would be the thread that accesses ball 0 in the contact pair loop:

```
if ( ai_first == 0 ) force += ball->mass * dc.gravity_accel;
```

(If all threads executed the line above it would be as though gravity were `thds_per_ball` times higher. That was a common mistake in submitted solutions.)

This problem specifically mentioned the contact pair loop as a place to use the extra threads but the neighbor loop too can benefit. (Points were not deducted if the neighbor loop was not handled by multiple threads.)

We'll divide the iterations of the neighbor loop amongst the threads using the same variable, `ai_first`, that we use for the contact pair loop. Rather than start at `-3` the contact pair loop starts at `ai_first-3` and rather than incrementing `j` by 1 we increment by `thds_per_ball`:

```
for ( int j = ai_first - 3;  j < 4;  j += thds_per_ball )
```

When the neighbor loop finishes, the forces for each ball will be split across seven (really six) threads. We need to move those forces to the thread responsible for updating velocity, the one with `ai_first==0`.

We do that by first writing the force into a shared array, and then have the `ai_first==0` thread add up those forces one by one. The method below should only be used when the number of items being added are small, in this case 6.

```
__shared__ float3 forces[1024];
forces[threadIdx.x] = force;
__syncthreads();
if ( ai_first == 0 ) {
    const int iters = min(7,thds_per_ball);
    for ( int i=1; i<iters; i++ )
       force += forces[threadIdx.x + i * balls_per_block];
    forces[threadIdx.x] = force;
 }
```

Next is the contact pair loop. In the solution this loop is skipped if the `opt_test1` variable is true, that's to demonstrate how long its taking.

The contact pair loop has two changes. The original loop iterated from `j=0` to the chain length (256). Now we have to adjust the loop so each thread handling a ball accesses a different set of "ball2" balls. The starting point is `ai_first`. Since there are `thds_per_ball` threads handling each ball (each with a different value of `ai_first`), the increment must be `thds_per_ball`.

```
if ( !dc.opt_test1 )
  for ( int j=ai_first; j<dc.chain_length; j += thds_per_ball )
    if ( abs(j-bi)>1 )
```

The other change to the contact pair loop is the method used to update the force variable. In the neighbor loop each thread updated its own copy of `force` and these were combined after the loop completed. For the contact pair loop we'll use a different approach. Rather than combine the forces after the contact loop exits, we will have each thread for a ball add the force to the same location. Variable `b_local` is the same for all threads handling the same ball (within a block). So that will be used as the index into the `forces` shared array.

We can't just use the code `forces[b_local]+=sep_f` because multiple threads with the same value of `b_local` may execute that line at the same time (which would result in an incorrect sum). An easy but costly way to avoid the problem is by using an atomic add. Remember that an atomic add will correctly add something to a memory location (shared or global) by first locking the location, performing the operation, and then unlocking the location. The locking and unlocking slows things down, especially when multiple threads are trying to access the same location.

The reason that an atomic add is acceptable here is because we expect only a few balls to be in contact, and so the atomic adds will be reached for only a small fraction of the iterations. Here is the simplified code:

```
const float rsum = ball_radius + dc.d_pos[j].w );
if ( rsum * rsum > ball_to_2.mag_sq ) {
    float3 sep_f = ( rsum - ball_to_2.magnitude) * 1000 * ball_to_2;
    atomicAdd(&forces[b_local].x,sep_f.x);
    atomicAdd(&forces[b_local].y,sep_f.y);
    atomicAdd(&forces[b_local].z,sep_f.z);
  }
```

Finally, the code needs to use the computed force to update the velocity. Since only one thread per ball updates the velocity the others can return. But first, a `syncthreads` is executed so that the thread assigned to update the velocity does not start working until the other threads have finished updating the force.

A common mistake was to allow multiple threads to update the same ball's velocity. This would result in incorrect velocities. The solution avoids it by having each ball's velocity written by just one thread, the one with `ai_first==0`.

```
__syncthreads();
if ( ai_first ) return;
float3 velocity = ball_velocity;
velocity += delta_t * ball->mass_inv * forces[b_local];
const double fs = powf(1+dc.opt_air_resistance,-delta_t);
velocity *= fs;
ball->velocity = velocity;
```

# 30 Fall 2012 Solutions

# LSU EE 4702-1 Homework 1 Solution Due: 12 September 2012

*Most 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 1:** 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 line of balls drop to the platform. *Promptly report any problems.*

To check out the assignment use the command

Use the arrow keys, PageUp, and PageDown to move around the scene. Press lower-case 'b' and then use the same keys to move the fixed ball around. Press 'l' to move the light around and 'e' to move the eye. Look at the comments in the file `hw1.cc` for documentation on other keys. Try turning gravity on and off and changing the spring constant. *Note: There is nothing to turn in for this first problem.*

**Problem 2:** In the hw1.cc program the balls interpenetrate (overlap in space). Modify the program so that the spring forces prevent (or nearly prevent) interpenetration. Do this by making the relaxed distance of a spring connecting two balls equal to the sum of their radii. Also, the spring constant used for repulsion should be ten times that used for attraction.

- Only code in the `time_step_cpu` routine needs to be modified for this problem.
- The radius of a ball is in the `radius` member of the `Ball` class. For example, `balls[i].radius` is the radius of ball `i`.

The complete solution is in the repository in a file named hw1-sol.cc. An htmlized version is at `http://www.ece.lsu.edu/koppel/gpup/2012/hw1-sol.cc.html`. In the excerpt shown below the relaxed distance is set based on the neighboring balls radii, and the spring constant is set to one of two possible values, depending on whether the balls are overlapping.

```
// SOLUTION
const float distance_relaxed =
  ball->radius + neighbor_ball->radius;

// Compute how much the spring is stretched.
//
const float spring_stretch =
  ball_to_neighbor.magnitude - distance_relaxed;

// SOLUTION
const float spring_constant =
  spring_stretch < 0 ? 10 * opt_spring_constant : opt_spring_constant;
```

# LSU EE 4702-1 Homework 2 Solution Due: 10 October 2012

*Follow the* Programming Homework Work Flow *instructions on the procedures page, substituting* `hw2` *for* `hw1`.

The `hw2` code, based on `demo-8-texture`, draws a helix. Due to missing normals, the surface of the helix will not be shaded properly (see Problem 1).

The comments at the top of the file describe controls for the simulation. For this assignment controls have been added to change the amount of detail in the helix (search for "Big Circle" and "Small Circle"), to change how vertices are ordered (look at the `i` and `m` options), and to force computation and communication (look at `d` and `r`).

For the solution code look for file hw2-sol.cc in the repository.

**Problem 1:** Run the unmodified `hw2` code and pay attention to the way the helix is lit. Notice that the shading of the helix is wrong. That's because normals have not been specified. Modify the code so that normals are correctly specified. To do this new arrays and a new buffer object will have to be added.

The normal itself can easily be found by constructing a vector from the helix point (n0) and the wire surface point (s0). That vector is constructed in part of the expression used to compute `s0` in the first place, see the code below under `Before`. In the solution, shown below under `Solution`, the vector is assigned to a separate variable, `p0s0`, and that is used both to compute `s0` and to compute the normal. Because of the way the helix is constructed the length of `p0s0` is `small_r`, so to compute the normalized normal just divide each element of `p0s0` by `small_r`. The solution actually multiplies by the reciprocal of `small_r`.

```
// Before: Computation of wire surface point:
pCoor s0 = p0 + cos(theta) * n0 + sin(theta) * b;

// Solution:
pVect p0s0 = cos(theta) * n0 + sin(theta) * b;
pCoor s0 = p0 + p0s0;
pVect norm0 = small_r_inv * p0s0;
```

**Problem 2:** The use of triangle strips reduces vertex redundancy compared to individual triangles. However for the `hw2` program there is still redundancy. In particular, point `s1` (search for "Construct a Helix" and scroll down to the `j` loop) is the same as `s0` in a prior `i` iteration.

By using `glDrawElements` instead of `glDrawArrays` one can avoid this redundancy. That is, the command sends each point to the GPU once and a separate index array specifies which of those points are used, a point can be used multiple times.

(*a*) Modify the code to use `glDrawArrays` when variable `opt_use_elements` is true. (Use key `m` to change the value of `opt_use_elements`.) Some code is already in place, including code declaring and initializing the index array and the call to `glDrawElements`. However, the existing code still adds the same point to the `helix_coords` list multiple times. The index array (`indices`) is set to 0, 1, 2, ..., which makes `glDrawElements` equivalent to `glDrawArrays`.

Modify the code so that each point is added only once, making appropriate changes to `prep_indices` (which is used to set `indices`) so that the helix is drawn correctly. See Section 10.5 of OpenGL version 4.3 for use `glDrawArrays`.

In the original code the inner loop added two coordinates to `prep_coords` and the corresponding indices to `prep_indices`. See the code below under `before`. In the solution, see below under `Solution`, only one coordinate, `s0`, is added to the list. Two indices are added, `idx` for `s0`, the other for a coordinate that will be added later, `idx + seg_per_small_circle`.

```
// Before:
              // Compute point on surface of wire.
              pCoor s0 = p0 + cos(theta) * n0 + sin(theta) * b;
              prep_coords += s0;
              prep_indices += idx; // Index of point just added.

              pCoor s1 = p1 + cos(theta) * n1 + sin(theta) * b;
              prep_coords += s1;
              prep_indices += idx + 1;  // Index of point just added.

// Solution
              pCoor s0 = p0 + p0s0;
              prep_coords += s0;
              prep_indices += idx; // This vertex.
              prep_indices += idx + seg_per_small_circle;
```

The solution initially prepares the arrays assuming that `opt_use_elements` is true. If it's false then it will construct a new list of coordinates using the indices. See the checked in code.

(*b*) The code has a variable `opt_use_strips`. Modify the code so that when it's true the helix will be rendered using triangle strips and when it's false it will be rendered using individual triangles. *Note: This part did not appear in the original assignment.*

When `opt_use_strips` is false the helix will be rendered using individual triangles. For these there must be three indices per triangle. So rather than inserting two indices per iteration, six indices are inserted. See the excerpt below:

```
// Solution
              if ( opt_use_strips ) {
                  prep_indices += idx; // This vertex.
                  prep_indices += idx + seg_per_small_circle;
                } else {
                  prep_indices += idx; // This vertex.
                  prep_indices += idx + seg_per_small_circle;
                  prep_indices += idx + seg_per_small_circle + 1;

                  prep_indices += idx; // This vertex.
                  prep_indices += idx + seg_per_small_circle + 1;
                  prep_indices += idx + 1;
                }
```

**Problem 3:** In this problem try to measure the impact of using `glDrawElements` to reduce the redundancy.

(*a*) Compute the amount of data sent from the CPU to the GPU in terms of the number of triangles used in the helix with and without `glDrawElements`. Use $T$ to denote the number of triangles.

Each index (element of `indices`) is 4 bytes, each vertex coordinate (three floats) is 12 bytes, and each normal is 12 bytes. (The color and other attributes are the same for all vertices, so they do not significantly affect the data size.) In the solution below *vertex* refers to both the vertex coordinate and the normal.

For `glDrawArrays` with triangle strips: A total of $T$ indices and $T$ vertices are sent, the data size is therefore $4T + 24T = 28T$ bytes.

For `glDrawElements` with triangle strips: A total of $T$ indices are sent, but only $T/2$ vertices are sent. The total amount of data is $4T + 24T/2 = 16T$ bytes.

For `glDrawArrays` with individual triangles: A total of $3T$ indices and $3T$ vertices are sent, the data size is therefore $3 \times 4T + 3 \times 24T = 84T$ bytes.

For `glDrawElements` with individual triangles: A total of $3T$ indices and $T/2$ vertices are sent, the data size is therefore $3 \times 4T + 24T/2 = 24T$ bytes.

(*b*) Measure the impact on performance of this difference in data size. Use the various program options to make this performance difference large. Explain which options were used and show the results.

*Background:* There are two impacts for data size, communication between the CPU and GPU, which has already been discussed in class, and GPU memory and the GPU itself, which has not been discussed much. The solution will discuss both impacts, however the assignments were graded only on consideration of CPU/GPU communication.

For the homework code, communication between the CPU and GPU is controlled by the "Always Send" option. When this option is "no" indices and vertices are only sent when some helix parameter, such as "Small Circle" changes. This data will be sent once per change and so its impact on performance will be very brief. In other words, when "Always Send" is "no" data communication between the CPU and GPU is small, consisting of commands. When "Always Send" is "yes" the impact of data size on performance will be proportional to the array sizes (see previous problem).

In contrast, the sending of the indices and vertices from GPU memory to the GPU must occur every frame. The rendering pipeline is implemented in the GPU itself, the data that it reads (vertices, etc) initially is in GPU memory. The *vertex puller* will read the appropriate vertices from memory and send them to the first stage of the rendering pipeline, the vertex shader. The `glDrawArrays` is used the vertex puller reads vertices sequentially. When `glDrawElements` is used the puller will read the indices sequentially but fetch vertices corresponding to the indices, which can be in any order, and the same vertex can be pulled multiple times.

For `glDrawArrays` the amount of data read is equal to the size of the index and vertex arrays. (The `glDrawArrays` command does not use the indices, but our vertex shader uses them.) For `glDrawElements` the amount of data read from memory depends upon how clever the system is. Consider two cases, *simple* and *"efficient"*. In the simple case, one vertex will be read for each index. The total amount of data read will be the number of elements in the index array times the sum of the sizes of an index and a vertex. For the code in this problem using triangle strips, there will be $T$ index elements, an index is 4 bytes and a vertex coordinate/normal pair is 24 bytes, so the data read from memory will be $28T$ bytes (this is the number under the $M->S1$ column in the table below). Notice that this is more than the $16T$ bytes send from CPU to GPU, that's because each vertex is read twice in the simple case for triangle strips. For individual triangles the difference between CPU to GPU transfer and GPU memory to shader transfer is even greater. In this case there are $3 \times 4T$ bytes read from GPU memory for indices and $3 \times 24T$ bytes for vertices, for a total of $84T$ bytes. This difference in the amount of data read should be reflected in GPU execution time.

For the "efficient" case each vertex will be read from GPU memory once, even if its index appears multiple times. In that case the amount of data read from GPU memory is the same as the amount of data sent from CPU to GPU. For triangle strips that's $16T$ bytes, compared to $28T$ for the simple method. There are quotes around "efficient" to hint that the reduction in data volume over the simple method might come at some cost. That is the cost of holding on to the fetched vertices (before or after vertex processing) until their last use.

*Solution:* To determine the impact on performance with data size the code was executed for two helix sizes, both have more triangles than the default. The primitive type (triangle strip or individual triangles), draw command (`glDrawArrays` or `glDrawElements`) was varied, and data-resend options were varied. For reference, the data transfer sizes also appear in the table. Column `M->S1` shows the amount of data read from GPU memory to the GPU for the simple case, where the number of vertices read is equal to the number of indices. The data under `M->S2` shows the amount of data read for the "efficient" case, where the number of vertices read is equal to the number of *distinct* indices (which is equal to the number of vertices in the array).

```
Key: Strip, Triangle Strips;  Array, glDrawArrays
     Indiv, Individual Tri;   Elmnt, glDrawElements
     0: Only re-send data for changes.   1: Always re-send data.
     C->G:  Amount of data sent from CPU to GPU.
     M->S1: Amount of data sent from GPU memory to shader, assumption 1.
     M->S2: Amount of data sent from GPU memory to shader, assumption 2.

Small Circle: 40  Big Circle: 70 - Only recompute on change
 Config             GPU  CPU  Data  Data  Data
                              C->G  M->S1 M->S2
 Strip Array 1      2.6  0.9  28T   28T   28T
 Indiv Array 1      4.5  1.9  84T   84T   84T
 Strip Elmnt 1      2.3  0.7  16T   28T   16T
 Indiv Elmnt 1      2.5  0.8  24T   84T   24T
 Strip Array 0      1.9  0.4  0     28T   28T
 Indiv Array 0      2.5  0.4  0     84T   84T
 Strip Elmnt 0      1.9  0.4  0     28T   16T
 Indiv Elmnt 0      1.9  0.4  0     84T   24T

Small Circle: 100  Big Circle: 100 - Only recompute on change.
 Config             GPU  CPU  Data  Data  Data    # of Vtx
                              C->G  M->S1 M->S2   Shader Runs
 Strip Array 0      2.8  0.4  0     28T   28T     T      T
 Indiv Array 0      5.0  0.4  0     84T   84T     3T     3T
 Strip Elmnt 0      2.8  0.4  0     28T   16T     T      0.5T
 Indiv Elmnt 0      2.9  0.4  0     84T   24T     3T     0.5T
```

Look first at the "Always Send" entries, those with a 1 in the config column. Both the CPU and GPU times are roughly proportional to the amount of data sent. The worst time is for individual triangles with `glDrawArrays`, the time for the other options are much better.

When "Always Send" is "no" CPU time does not vary with the other options, as expected. What's interesting is the GPU time. It's only large for individual triangles with arrays. The effect is also seen in the second table, which has data on a helix with a larger number of triangles. The GPU time is consistent with the "efficient" case.

(*c*) Try to determine if there is any difference in performance due to computation. There would be less computation using `glDrawElements` if the vertex shader were run on each vertex only once. But does that actually happen? Explain which options were used and show the results.

The second table above also includes columns showing the number of times the vertex shader is run per triangle, based on different assumptions. The first `Vtx` column shows the number of vertex shader runs assuming that the vertex shader is run once per index, the second column is the number of runs under the assumption that it is run once per vertex. The last two rows provide the best basis for comparison. The slightly lower performance of the individual triangle row might be due to the $3T$ shader runs (versus $T$ for the strips) but it also might just be due to the amount of memory read.

# LSU EE 4702-1 Homework 3 Solution Due: 16 November 2012

*Follow the* Programming Homework Work Flow *instructions on the procedures page, substituting* `hw3` *for* `hw1`*. Also, update the include directory.*
*Use* `http://www.ece.lsu.edu/gp/refs/GLSLangSpec.4.30.6.pdf`*, the OpenGL 4.3 Shading Language spec, as a reference. The code in this assignment requires OpenGL 4.3, which should be installed on* `snow`*,* `ice`*, and* `frost`*.*

*The solution has been checked into the repository as new files hw3-sol.cc and hw3-shdr-sol.cc. The solution is also in the Web at* `http://www.ece.lsu.edu/koppel/gpup/2012/hw3-shdr-sol.cc.html` *and* `http://www.ece.lsu.edu/koppel/gpup/2012/hw3-sol.cc.html`*.*

**Problem 0:** Compile and run the code unmodified, a yellow helix should be displayed. The arrow keys initially move the eye position, familiarize yourself with these keys, the others in the comments of `hw3.cc`, and the keys described in this problem.

The code is set up to run two different shader programs, called "SP Geo Shade1" and "SP Geo Shade2." The name of the active shader program is displayed on the lowest green line, pressing "s" will switch from one to the other. The two shader programs use the same vertex and fragment shader, but different geometry shaders. The problems indicate which geometry shader to modify.

For this assignment there are three user-controlled variables of interest. (See the Variables in the Keyboard Commands section of the comments in hw3.cc.) *Segs Per Helix Rev* controls the number of segments in one revolution of the helix. Larger numbers mean smaller triangles. *Segs Per Wire Rev* controls the number of segments in one revolution of the wire. Larger numbers mean smaller triangles. *Door Angle* See Problem 3.

**Problem 1:** The unmodified version of the code does everything needed to display a texture on the helix, except for sending texture coordinates. (The texture is this assignment.) Rather than generating texture coordinates on the CPU and sending them to the GPU, the goal here is to compute the texture coordinates on the GPU using a vertex shader. The vertex shader in `hw3-shder.cc` currently just copies the texture coordinates from the CPU unmodified. Since the CPU never sends texture coordinates these will hold some initial value, perhaps $(0, 0)$.

Modify the vertex shader in `hw3-shdr.cc` so that it computes texture coordinates, instead of reading them from `gl_MultiTexCoord0`.

- Display the texture so that it appears right-side up, undistorted, and is recognizable.
- Optional: The size and position of the texture should not change when number of segments per helix or wire are changes.

Note: This problem is easy. Texture coordinates can be computed in terms of the values in `helix_index`. For the optional part new uniform variables need to be defined.

As the statement problem came close to explaining, the texture coordinates can be found by multiplying the helix indices by some constants. In the solution these are computed on the CPU and put in a two-element uniform vector, `texture_scale`:

```
const float scale_x = - 20.0 / seg_per_helix_revolution;
const float scale_y = 1.0f / seg_per_wire_revolution;
glUniform2f(4,scale_x,scale_y); GE();
```

In the vertex shader these are used to compute the texture coordinates:

```
gl_TexCoord[0] = texture_scale * helix_index;
```

See the code for details.

**Problem 2:** Both geometry shaders assign a variable named `type_a`, but the variable is not used. Modify `gs_main_helix` and other code so that textures are not displayed for triangles for which `type_a` is true. Do this by declaring new variables for communicating the value of `type_a` from the geometry shader to the fragment shader. Use this variable in the fragment shader to avoid doing a texture lookup.

- Declare a new variable for communicating between the geometry shader and the fragment shader. **Do not** use an existing variable, tempting though it may be.
- Set the appropriate interpolation for the new variable.
- Use the variable in the fragment shader to control whether textures are used.

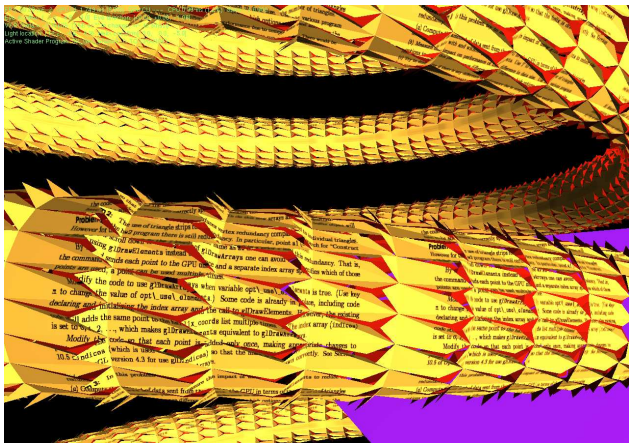
Note: The fragment shader already avoids texture lookup for the back side of primitives, it also applies a reddish tint to such primitives. Make sure that reddish tint is not applied to the front of primitives for which texture lookup has been suppressed.

See Section 4.3.4 and 4.3.6 of the OpenGL 4.3 specification for details on how to declare shader input and output variables.

This problem is also fairly easy, but it's not a one-liner.

The solution is to add a new variable to the output of the geometry shader and the input to the fragment shader. The variable is type bool and is given the interpolation qualifier, `flat`. Some sample changes are shown below, see the code for details.

```
#ifdef _GEOMETRY_SHADER_
out Data
{
  vec3 normal_e;
  vec4 vertex_e;
  flat bool no_texture;  /// SOLUTION - Problem 2
};
#endif
#ifdef _FRAGMENT_SHADER_
in Data
{
  vec3 normal_e;
  vec4 vertex_e;
  flat bool no_texture;    /// SOLUTION - Problem 2
};
#endif
```



**Problem 3:** Modify geometry shader `gs_main_helix2` so that each `type_a` helix triangle is split into three triangles under control of CPU variable `opt_door_angle`, call the value of that variable $\theta$. Each new triangle consists of two vertices from the original triangle, and (initially) the original triangle center. These new triangles will rotate, the amount based on $\theta$, around an axis formed by the original edge. Though the triangles rotate, their shape should not change. The appearance might be of doors opening up, see the screen shot to the right.

More precisely, let $p_0$, $p_1$, and $p_2$ denote the vertices of the original triangle, and let $c = (p_0 + p_1 + p_2)/3$ denote the centroid, $\overrightarrow{n} = \overrightarrow{p_0p_1} \times \overrightarrow{p_0p_2}$ denote the triangle normal, and $\hat{n}$ be the normalized normal. The original triangle, $p_0p_1p_2$, is to be split into new triangles $p_0p_1c_2$, $p_0c_1p_2$, and $c_0p_1p_2$ where $c_0$ is given by

$$c_0 = q_0 + \overrightarrow{q_0c} \cos\theta + l_0\hat{n} \sin\theta,$$

where $q_0$ is the point on line $p_1p_2$ closest to $c$ and $l_0$ is the distance from $q_0$ to $c$. Points $c_1$ and $c_2$ are defined similarly.

Geometry shader `gs_main_helix2` should emit these three triangles.

- Using an appropriate method, provide a value of CPU variable `opt_door_angle` to the shader.
- Calculation of triangle coordinates should not be performed on clip-space coordinates.
- The `normal_e` output of the geometry shader should be for the respective triangle, not the wire.
- Note that the output of the geometry shader should be both clip-space coordinates (for the fixed-functionality) and eye-space coordinates (for our fragment shader).

See the OpenGL Shading Language 4.3 specification for information on the shader language, including functions for dot product, cross product, and normalization.

Here are some tips that apply to all problems:

- Shader program error messages appear when the main program starts, not when code is compiled.
- The error message line number might be off by one.
- The following shader code error message can be ignored, (but don't ignore other messages that occur with it): "warning C7547: extension GL EXT geometry shader4 not supported in profile gp5fp."

An excerpt from the geometry shader code appears below. See the code for additional changes.

```
void
gs_main_helix2()
```

```
{
  /// SOLUTION - Problem 3

  // Select which triangles to split into "trap doors".
  //
  const bool trap_door = In[0].hidx.x >= In[2].hidx.x;

  // Set door angle (0 is fully closed, pi/2 is straight out, etc.
  //
  float th = trap_door ? theta : 0;

  // Find center of triangle (in eye-space coordinates).
  //
  vec3 pctr = (In[0].vertex_e + In[1].vertex_e
               + In[2].vertex_e ).xyz * 0.333333f;

  // Find texture coordinate of center of triangle.
  //
  vec4 tctr = ( gl_TexCoordIn[0][0] + gl_TexCoordIn[1][0]
                + gl_TexCoordIn[2][0] ) * 0.333333f;

  // Generate the three "trap door" triangles.
  //
  for ( int tri=0; tri<3; tri++ )
    {
      // Note that calculations performed in eye-space coordinates. If
      // the calculations were performed in clip-space coordinates
      // (for example, using gl_PositionIn) results would be distorted
      // due to the perspective projection.

      // Select two "fixed" vertices.
      //
      vec3 p0 = In[(0+tri)%3].vertex_e.xyz;
      vec3 p1 = In[(1+tri)%3].vertex_e.xyz;

      // Select the "swinging" vertex.
      //
      int p2_idx = (2+tri)%3;
      vec3 p2 = pctr;

      // Compute the two axes defining the plane that point new_p2
      // (below) will move along.

      // Start by finding the point between p0 and p1 that's closest to p2.
      //
      vec3 v01_norm = normalize( p0 - p1 );
      vec3 v02 = p2 - p0;
      vec3 p01_nearest = p0 + v01_norm * dot(v01_norm,v02);

      // Axis 2 is from p2 to the p0-p1 line.
      //
```

```
      vec3 axis2 = p2 - p01_nearest;
      float axis2_len = length(axis2);
      vec3 axis2_norm = axis2/axis2_len;

      // Axis 1 is a vector orthogonal to both axis 2 and the p0-p1 line.
      //
      vec3 axis1_norm = cross(v01_norm,axis2_norm);
      vec3 axis1 = axis2_len * axis1_norm;

      // Use a simple circle formula to find new_p2.
      //
      vec4 new_p2 = vec4(p01_nearest + cos(th) * axis2 + sin(th) * axis1,1);

      // Finish by computing the clip-space coordinate of new_p2_c and
      // an eye-space normal for the new triangle.
      //
      vec4 new_p2_c = gl_ProjectionMatrix * new_p2;
      vec3 new_norm = normalize(cross(v01_norm,new_p2.xyz-p0));
      normal_e = -new_norm;

      // Emit the three vertices of this triangle.
      //
      for ( int i=0; i<3; i++ )
        {
          gl_FrontColor = gl_FrontColorIn[i];
          gl_Position = i == p2_idx ? new_p2_c : gl_PositionIn[i];
          gl_TexCoord[0] = i == p2_idx ? tctr : gl_TexCoordIn[i][0];
          vertex_e = i == p2_idx ? new_p2 : In[i].vertex_e;
          no_texture = false;
          EmitVertex();
        }
      EndPrimitive();
    }
}
```

# LSU EE 4702-1 Homework 4 Solution Due: 20 November 2012

*For the problems below refer to Chapter 2, Programming Model, and Chapter 4, Hardware Implementation, in the NVIDIA CUDA Compute Unified Device Architecture Programming Guide available locally via*
`http://www.ece.lsu.edu/gp/refs/CUDA_C_Programming_Guide.pdf`.

**Problem 1:** The kernel below is launched in a configuration of grid size of $(1024, 1, 1)$ and block size of $(256, 1, 1)$. (The components are $(x, y, z)$.)

```
__global__ void dots_iterate1() {
  int thread_count = /* Omitted so things aren't too obvious. */;
  int idx_start = threadIdx.x + blockIdx.x * blockDim.x;
  for ( int idx = idx_start; idx < array_size; idx += thread_count )
    b[idx] = v0 + v1 * a[idx]; }
```

($a$) How many threads are there? How many blocks? How many warps?

The grid size is the same as the number of blocks $1024 \times 1 \times 1 = 1024$.

The number of threads is the product of the block size and grid size: $1024 \times 256 = 2^{10}2^8 = 2^{18} = 262144$.

The number of warps in a block is $\lceil B/32 \rceil$, where $B$ is the block size. The number total number of warps is the blocks per warp times the number of warps. For this example that's $2^{18}/2^5 = 2^{13} = 8192$.

($b$) Suppose the array size is $5 \times 10^5 = 500\,000_{10} = 7\,a120_{16}$ elements. How many threads will perform two iterations? One iteration? Zero iterations?

Two iterations: 237856, one iteration 262144. Zero: none.

($c$) A GPU has four multiprocessors. Explain why launch configurations of four and eight blocks are each better than a launch configuration of six blocks. Use the code above as an example.

Each block is assigned to one multiprocessor (in other words a single block can't run on two multiprocessors at the same time). If the number of blocks is not a multiple of the number of multiprocessors then some multiprocessors will be assigned more blocks than others. If each block takes the same amount of time to execute, an uneven distribution will result in execution time being longer than it has to be. The smaller the number of blocks the worse the problem is.

For example, suppose the kernel above was launched in two configurations, both having 256-thread blocks. One configuration has six blocks each taking 80 ms (or about 325 iterations per thread) or one with eight blocks each taking 60 ms (or about 244 iterations per thread). For the six-block launch two multiprocessors will be assigned two blocks and two will be assigned one block. The kernel will finish in $2 \times 80\,\text{ms} = 160\,\text{ms}$. In the eight-block launch each multiprocessor will be two blocks, the kernel will finish in $2 \times 60\,\text{ms} = 120\,\text{ms}$.

Note that the launch configuration given in this problem, 1024 blocks, is large enough so that load imbalance is not much of a problem with current GPU sizes, where the larger ones have about 16 multiprocessors.

**Problem 2:** Consider the kernel code below. It is launched with a block size of 512 threads and a grid size of 64 blocks. The array has $2^{20} = 1\,048\,576$ elements.

```
__global__ void dots_iterate15()  {
  int thread_count = /* Omitted so things aren't too obvious. */;
  int tid = threadIdx.x + blockIdx.x * blockDim.x;  // Not used in code.
  int idx_start = blockIdx.x + threadIdx.x * gridDim.x;

  for ( int idx = idx_start; idx < array_size; idx += thread_count )
    b[idx] = v0 + v1 * a[idx];
}
```

(*a*) The table below shows various information about selected threads in the launch described above. The first three columns should be self-explanatory. The three columns headed `idx` show the element number (value of `idx` in the code above) accessed by the respective thread in the first, second, and third iteration of the `for` loop. Each row has at least one column filled. Fill the remaining columns.

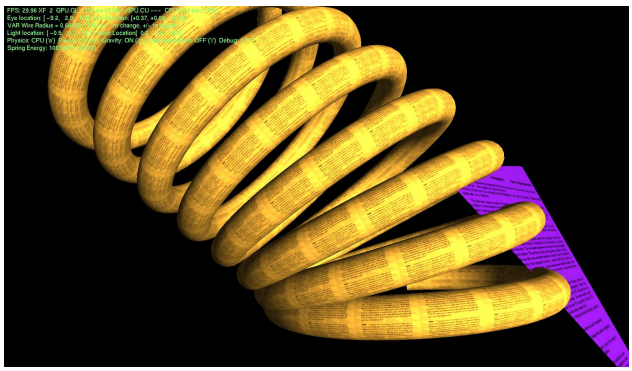
Solution appears below. Notice that consecutive thread ids do **not** access consecutive elements. In fact, the situation is far worse. Adjacent arrray elements are accessed by **different** blocks. For example, element 0 is accessed by block 0 and element 1 by block 1. These may be on different multiprocessors. Since the memory system brings in data in 128-byte units, much of that data will go unused.

| tid | blockIdx.x | threadIdx.x | -- idx --- First Iter | -- idx --- Second Iter | -- idx --- Third Iter |
|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 32768 | 65536 |
| 1 | 0 | 1 | 64 | 32832 | 65600 |
| 2 | 0 | 2 | 128 | 32896 | 65664 |
| | | | | | |
| 0 | 0 | 0 | 0 | 32768 | 65536 |
| 512 | 1 | 0 | 1 | 32769 | 65537 |
| 1024 | 2 | 0 | 2 | 32770 | 65538 |
| 513 | 1 | 1 | 65 | 32833 | 65601 |

# LSU EE 4702-1 Homework 5 Solution Due: 30 November 2012

*Follow the* Programming Homework Work Flow *instructions on the procedures page, substituting* `hw5` *for* `hw1`*. Also, update the include directory.*
*Use* `http://www.ece.lsu.edu/gp/refs/CUDA_C_Programming_Guide.pdf`*, the NVIDIA CUDA Compute Unified Device Architecture Programming Guide, as a reference. The code in this assignment requires OpenGL 4.3 and CUDA, which should be installed on* `snow`*,* `ice`*, and* `frost`*.*

See `hw5-sol-cuda.cu` in the repository for the solution, search for "Problem 1" or "Problem 2". The code is also available at `http://www.ece.lsu.edu/koppel/gpup/2012/hw5-sol-cuda.cu.html`



**Problem 0:** Follow the class procedures for homework but substitute hw05 for the assignment. Compile and run the code. It should display our familiar helix, but this time it's a simulated spring, and it should be drooping under gravity, see the illustration to the right. The physics can be performed by both CPU and GPU, by default it runs on the CPU, pressing 'a' will toggle it.

The CPU code can prevent interpenetration, this option is initially off, it can be toggled by pressing 'i'. The option was turned off when the image above was captured, notice that the bottom loops of the spring interpenetrate. The code for interpenetration can be found in `time_step_cpu`, it uses a brute-force approach, comparing every segment of the helix to every other, making it very slow. The goal of this assignment is **not** to use a clever broad-phase collision detection scheme to avoid $O(n^2)$ comparisons, but instead use the GPU to do the calculations faster. (In "real life" one might use both broad-scale collision detection and a GPU to do calculations.)

**Problem 1:** The routine `time_step_cpu` contains the CPU interpenetration code, clearly identified by comments. For this problem put the interpenetration code in CUDA kernel `time_step`, in file `hw5-cuda.cu`. The `time_step` kernel is launched in a configuration with one thread per helix segment. Each thread should test "its" segment against other segments. A correct solution will be faster than the CPU, but should still be slow.

See `hw5-sol-cuda.cu` in the repository for the solution, search for "Problem 1".

**Problem 2:** By GPU parallelism standards, the number of threads launched for the `time_step` routine is not large. In the default configuration there are 160 segments, just five warps. That will underutilize any GPU.

For this problem, add a new kernel to perform the interpenetration test. The new kernel should be launched with more threads than segments, taking advantage of more parallelism. Modify the existing `time_step` kernel so that it does not modify `helix_position` (but it still updates `helix_velocity` and the other two state variables). The new kernel should compute forces based on interpenetration tests, use that to update velocity, and then update the position.

A good solution of this problem requires the use of shared memory, atomic operations, and synchronization. That will be covered next week.

See `hw5-sol-cuda.cu` in the repository for the solution, search for "Problem 2".

A new kernel `time_step_intersect` is added. This kernel is launched with one *block* per helix segment (the `time_step` kernel is launched with one thread per helix segment). Each block compares one segment, call it the $a$

segment, against most of the others, call them the $b$ segments. The index of the $a$ segment is set to `blockIdx.x` and the index of the first $b$ segment to look at is set to `threadIdx.x`. Each block uses a shared variable to hold the force, `threadIdx.x` is responsible for initializing it. When a thread detects an intersection it updates `force` using an atomic add, which is convenient but slow. This is acceptable because even when the helix is compressed each segment will be in contact with about four others, much less than the $80 \times 6 = 480$ segments that need to be examined (excluding near-neighbors).

After the loop `threadIdx.x` 0 will use the force to update velocity.

Two small changes were made to "help" the compiler optimize. The distance between the segments in the CPU code was found using a pNorm object, which needs to compute a square root in order to normalize. This object also provides the distance between the objects squared. It is the distance squared which is used to detect intersection, and if there is no intersection there is no need to perform a square root (which is used for finding the force). The NVIDIA compiler seemed to be computing the square root before the branch which usually makes sense for a GPU since that would help if even only one thread per warp would need it. But in our case the square root was only rarely needed. So, the tweak is to compute the distance squared separately.

The second tweak was moving the neighbor branch, the one that checks `min_idx_dist`, next to the distance check. That forces the GPU to load `helix_position` even when its not needed (which is rarely). Perhaps because there is no gap in the pattern of accesses performance is better.