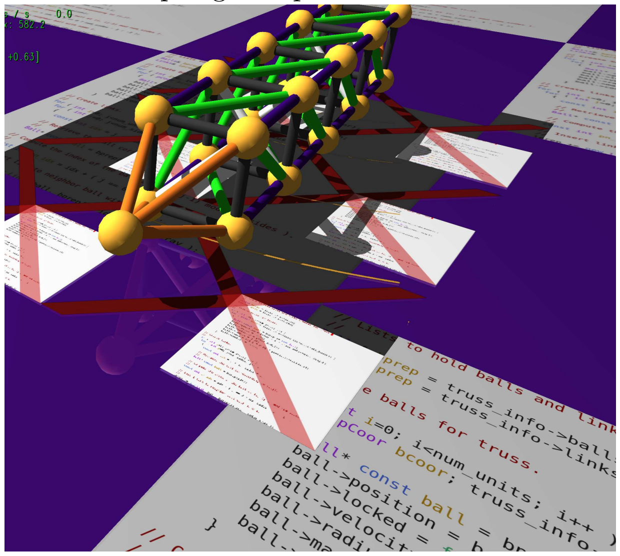
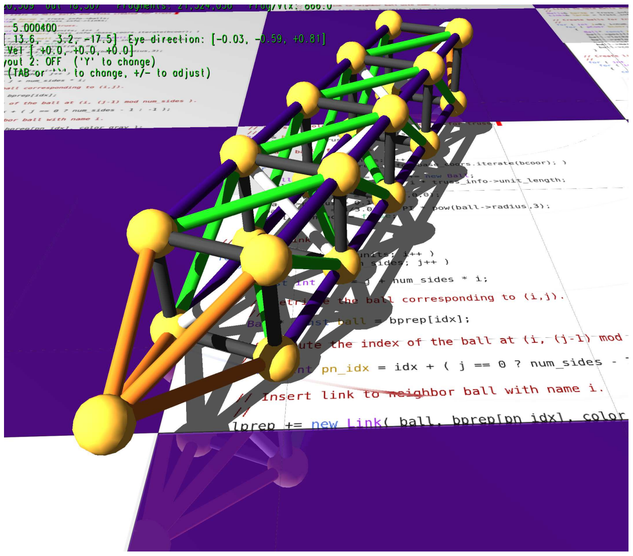
LSU EE 4702-1 **Homework 3** **Due: X3X23 October 2017**

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.

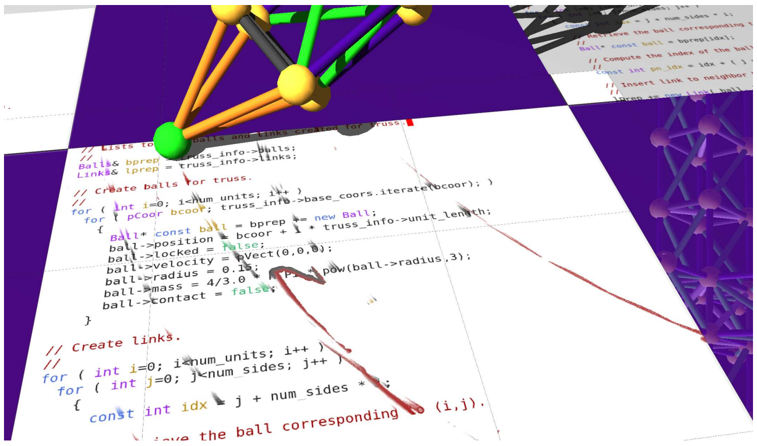
Look at the comments in the file `hw03.cc` for documentation on other keys.

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.



```
// Lists to [illegible] 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;
      [illegible]ve the ball corresponding to (i,j).
```

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.