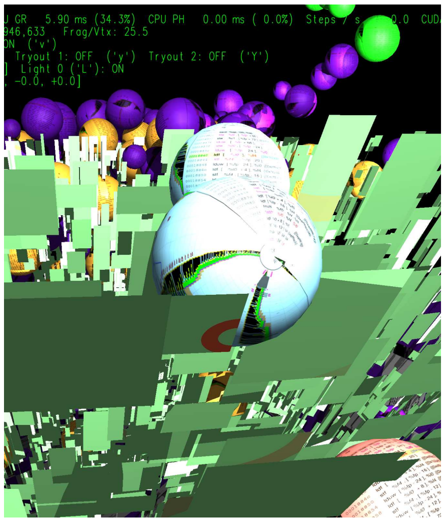
Problem 0: Follow the instruction on the `http://www.ece.lsu.edu/koppel/gpup/proc.html` page for account setup and programming homework work flow. Compile and run the homework code unmodified. It should initially show Scene 1, a cloud of green rectangles, *tiles*, with balls dropping on them. (The familiar water wheel is in scene 2.)



```
J GR   5.90 ms (34.3%)  CPU PH   0.00 ms ( 0.0%)  Steps / s   0.0  CUD
946,633   Frag/Vtx: 25.5
ON  ('v')
  Tryout 1: OFF  ('y')  Tryout 2: OFF  ('Y')
]  Light 0 ('L'): ON
, -0.0, +0.0]
```

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.