Project 2

CS 320

due Mar. 29 at noon

In this project you will implement the first phase of a pool simulation, starting from `collision.cpp`. You have a pool table with certain attributes and pool balls with certain attributes. There is a simulation attribute or two to handle as well. Sample data is collected into the file `poolData.txt`, included in the project zip file. Your simulation program will read this file, using the data in the file to define the simulation. The data format is as follows. Notes: Lengths are in inches. Colors are rgb components on [0.0, 1.0]. Velocities are inches per second. Time is seconds. The coefficient of rolling friction (C_{rf}) has units of inches per second per inch. Mathematically:

$$V' = (1 - C_{\text{rf}} T_{\text{i}}) V$$

where V' is future velocity, T_{i} is time elapsed in seconds for the current simulation step, and V is current velocity. Ball masses are ounces. Positions and velocities are given in three dimensions (x, y, z), but you can "throw away" the z dimension data. Each data item will be separated by one or more whitespace characters. File format begins with next line.

```
number of simulation steps per render step
ll.x ll.y ur.x ur.y   // area of play boundary
color of area of play
width of fringe area surrounding area of play
color of fringe area
coefficient of restitution
coefficient of rolling friction
number of balls
mass radius color position velocity   // ball data, repeated for each ball
```

Your simulation should have the following characteristics:

1. As already mentioned, data will be read from a file. The name of the file to be read is `poolData.txt`. The simulation should exit if the data file can't be opened. Here's some sample code demonstrating file I/O in C++:

   ```
   #include <fstream>

   ...

   std::ifstream data;
   int displayThreshold;
   vec2 ll, ur;
   ```

```
...

data.open("poolData.txt");

if (!data.is_open())
{
   std::cout << "Could not open poolData.txt!" << std::endl;
   exit(1);
}

data >> displayThreshold;

data >> ll.x;
data >> ll.y;
data >> ur.x;
data >> ur.y;

data.close();
```

2. It is acceptable to use constant values to define the window size rather than compute values. I used a window of size 900 by 450. You may assume that the window will not be resized.

3. Your simulation should be physically accurate. This means you will need to record the time between simulation steps and account for it in your simulation. This also applies to the coefficient of rolling friction. These will ensure the simulation looks the same regardless of the capabilities of the machine on which it runs. The Windows function `GetTickCount()` will be useful. Note that `GetTickCount()` is not available in Linux, so the following preprocessor code should be used:

   ```
   #ifdef WIN32
   #include <Windows.h>
   #include <Winbase.h>
   #else
   #include <sys/time.h>
   int GetTickCount()
   {
      struct timeval tv;
      gettimeofday(&tv, NULL);
      return 1000 * tv.tv_sec + tv.tv_usec / 1000;
   }
   #endif
   ```

   This function returns the current number of milliseconds (0.001 seconds) since boot time.

4. Using the first data item given in the data file, your simulation should be capable of performing multiple simulation steps per single rendering step.

5. You may assume that this simulation will never involve more than 100 pool balls.

6. If designed and written properly, very few modifications need to be made to `collisionResponse()` for handling collisions between a ball and the boundary. It will be quite useful to have a wrapper function generate a virtual ball modeling the boundary and then pass the two balls to `collisionResponse()`, as discussed in class. You will need to find a way to represent the infinite mass of the virtual ball and modify `collisionResponse()` to recognize this and adjust the computation (A mass of `-1.0`, perhaps?). See items (e) and (f) on pp. 3 and 4 of the class notes on Collision Detection and Response. This is the only change needed to `collisionResponse()`.

7. The vertex shader provided in the project zip file has a uniform variable, `vColor`, for object color data. Note that this variable has type `vec3`. The function `glUniform3fv()` should be used to assign values to `vColor` from your simulation program.

Submitting Your Project

We will follow this procedure for all projects this semester. Your project files are to be emailed to me at kelliher[at]goucher.edu. All source files necessary for building your program (C++ source(s), .h files, vertex shader, and fragment shader) and any documentation files you've created should be sent as a single zip archive attachment in a single email. I will build your program from the source files provided and test the resulting program.