Math 198 Project Cover Sheet

- Name: Jeffrey Zhou
- Netid: zhou30
- Name: Curtis Woodruff
- Netid: woodruf1
- Date: December 18, 2009
- Project title: Bouncing Around
- Name of RTICA: FinalB.c
- Programming language: C
- Graphics system: OpenGL

The joint project `FinalB.c` is a simple bouncing simulation. Using kinematic equations for projectile motion and conservation of momentum and energy for elastic collisions, an adequate representation of two balls bouncing inside a box was created. While heavily influenced by other programs, `FinalB.c` was essentially built from scratch. As a C and OpenGL graphics program, writing the code was difficult since neither contributers Curtis Woodruff nor Jeffrey Zhou had experiance in the C language before this project and no prior exposure to programming before Math198, so this was an enlightening learning experiance.

Basically, the program is two balls in a box. The balls themselves move according to basic kinematic equations. The most useful one is $x_f = x_i + vt + \frac{1}{2}at^2$ which describes the projectile motion of the ball as it is affected by gravity. Gravity is denoted as y acceleration and air resistance and friction are of course ignored. This calculation is performed for each axis. Each frame recalculates both from the change in time dt which is drawn from `sys\timeb.h` and prevents lagging by being dependent on epoch time. Essentially, program speed will be the same no matter how powerful the computer is, although the framerate may become choppy on slower computers.

There are three different collisions utilized in the program. The first type is ball-wall collision. The collision occurs if the distance between the wall plane and center of the sphere is equal to or less than the radius. Upon detection, the final velocities of the previous kinematic path are calculated, and set as the initial velocities of the new path. The velocity of the axis that tripped the collision detection is then reversed. When dampening is activated, this new velocity is multiplied by the dampening number which should be between $0 - 1$ to decrease the magnitude of the post-collision velocity. The initial position of the sphere is then set to the last position it had before the collision. At the conclusion of these calculations, the running timer is reset for the new kinematic path .

Ball-ball collisions work much the same way. The distance between the balls is $\sqrt{x^2 + y^2 + z^2}$, and if this is less than or equal to the sum of the radii of the two balls, a collision occurs. Once again, the position is resets to the last precollision spot, but the velocity vectors are determined with the equations $\vec{v}_1 = \vec{u}_1 - \frac{a}{m_1}\vec{k}$ and $\vec{v}_2 = \vec{u}_2 - \frac{a}{m_2}\vec{k}$ which are derived from the following conservation of energy and momentum equations.

m =mass
$\vec{x}$ =position vector
$\vec{u}$ =velocity vector before collision
$\vec{v}$ =velocity vector after collision
$\vec{k} = \frac{\vec{x}_1 - \vec{x}_2}{|\vec{x}_1 - \vec{x}_2|}$ =unit vector of collision
a =impulse magic thing

$$\text{momentum}= mv$$
$$m_1\vec{u}_1 + m_2\vec{u}_2 = m_1\vec{v}_1 + m_2\vec{v}_2$$
$$m_1(\vec{u}_1 - \vec{v}_1) = -m_2(\vec{u}_2 - \vec{v}_2)$$
$$\text{change in momentum=impulse=}ak$$
$$\vec{v}_1 = \vec{u}_1 - \frac{a}{m_1}\vec{k} \qquad \vec{v}_2 = \vec{u}_2 - \frac{a}{m_2}\vec{k}$$

$$\text{kinetic energy} = \tfrac{1}{2}mv^2$$
$$\tfrac{1}{2}m_1(\vec{u}_1 \cdot \vec{u}_1) + \tfrac{1}{2}m_2(\vec{u}_2 \cdot \vec{u}_2) = \tfrac{1}{2}m_1(\vec{v}_1 \cdot \vec{v}_1) + \tfrac{1}{2}m_2(\vec{v}_2 \cdot \vec{v}_2)$$
$$m_1(\vec{u}_1 \cdot \vec{u}_1 - \vec{v}_1 \cdot \vec{v}_1) = -m_2(\vec{u}_2 \cdot \vec{u}_2 - \vec{v}_2 \cdot \vec{v}_2)$$
$$m_1(\vec{u}_1 - \vec{v}_1) \cdot (\vec{u}_1 + \vec{v}_1) = -m_2(\vec{u}_2 - \vec{v}_2) \cdot (\vec{u}_2 + \vec{v}_2)$$
$$\cancel{m_1(\vec{u}_1 - \vec{v}_1)} \cdot (\vec{u}_1 + \vec{v}_1) = \cancel{-m_2(\vec{u}_2 - \vec{v}_2)} \cdot (\vec{u}_2 + \vec{v}_2)$$
$$\vec{k} \cdot (\vec{u}_1 + \vec{v}_1) = \vec{k} \cdot (\vec{u}_2 + \vec{v}_2)$$

$$\vec{k} \cdot (\vec{u}_1 + \vec{u}_1 - \tfrac{a}{m_1}\vec{k}) = \vec{k} \cdot (\vec{u}_2 + \vec{u}_2 - \tfrac{a}{m_2}\vec{k})$$
$$2\vec{k} \cdot (\vec{u}_1 - \vec{u}_2) = a(\tfrac{a}{m_1} + \tfrac{a}{m_2})$$
$$a = 2\vec{k} \cdot (\vec{u}_1 - \vec{u}_2)/(\tfrac{1}{m_1} + \tfrac{1}{m_2})$$
$$m_1(\vec{u}_1 - \vec{v}_1) = -m_2(\vec{u}_2 - \vec{v}_2) = a\vec{k}$$

Ball-curved floor collisions use similar methods as ball-ball collisions, although detection is much more difficult. By running an r-theta loop, we plot a spiral of points beneath the profile of the sphere starting at the center to the x-z plane. The y values are then calculated from the floor equation. Then, the 3-d distance formula checks for collision with the sphere (distance should be less than or equal to the radius). The same math as the ball collision is used for the reflection off the floor, however mass is eliminated from the equation. The normal vector is the vector from the center of the sphere to the collision point on the floor.

While the mathmatics provide the principle behind the program, actually graphically demonstrating the physics is entirely different. The choice of C and OpenGL made it a little difficult to find good examples, which is the main reason that the program was built from scratch. Fortunately, two programs, `bnc.c` and `dinoshade` helped immensely for laying out the framework. The camera controls were heaviliy influenced by dinoshade which utilized mouse click-and-drag to change the camera angles. This project's controls however give both control of position of the camera as well as the focus point which most other programs encountered did not. The movements were simply a matter of keeping the camera view vector parrallel while moving which involved simply shifting both the view and camera positions equally the same way by adding a constant. The angle of the camera was a bit more difficult, and drew more heavily from `dinoshade`. Looking in the verticle direction simply meant moving the focus point up and down since the max look above and below the horizon was limited to prevent flipping and such. Along the horizontal plane, the view point had to follow a circular path since 360° turns would have to be possible.

The box and floor is just extensive use of `GL_LINES`. The ground plane based off of $y = x \times 1.05^{-x^2-z^2}$ was shifted over to center around $(q/2, q/2, q/2)$ and the lines were staggered diagonally instead to make the slopes more obvious. It was suggested to draw loxodromic spheres instead of using the solid sphere from the glut library. This was an attempt to make the CUBE version of the program more functional since the solid spheres show depth poorly in the CUBE. The math required to draw these spheres is below. A `GL_LINE_STRIP` was used to connect the points. θ and τ are incremented simultaneously across the specified ranges to produce the loxodromic sphere, which has a coiled appearance.

$$10^\circ \leq \theta \leq 350^\circ \times 10$$
$$-90^\circ \leq \tau \leq 90^\circ$$
$$x = \cos(\theta)\cos(\tau) \times radius$$
$$y = \sin(\theta)cos(\tau) \times radius$$
$$z = \sin(\tau) \times radius$$

The keyboard functions are mostly for camera controlling as they determine the value of `SPEED` which essentially controls the camera position and angle. For this reason, it is impossible to give multiple simultaneous camera commands and `SPEED` can only have one value at a time and affect one function at a time. Other than that, the "esc" button exits the program.

Broken down, these three parts, path, collisions, and graphics, work together to create a fairly accurate physics simulation of two balls bouncing inside a box. Further information can be found on our sites which can be found on http://new.math.uiuc.edu/math198/MA198-2009/. From starting with no knowledge of programming to a buggy but still operational project, we have succeeded and becoming familiar with computer coding. This may come in handy in the future, and later students may look to this to improve upon it.