CSCI 3010U: Simulation and Modelling

Billiard Simulation Project

Analysis, Verification, and Validation

Group Members: Collin Stubbs & Andrew Gulla

Professor: Randy Fortier

Analysis, Verification, and Validation

For our topic we simulated and modelled a single shot in a game of billiards. There is a simulation of 1 ball (the cue ball) that is on the opposite end of a billiards table from 15 other balls. Each ball has been wrapped with an image of a billiards ball, the table siding has a wooden material applied to it, and the playing field has been wrapped with a cloth like material. The balls (sphere objects) have been placed with their lowest y point as the same level as the playing table. The size of the table is 24.3 by 12.63 which was chosen because the standard size of a pool table is 9.32 by 4.65. The standard size did not have a good sizing for the model we were using so we scaled it up by 2.5 times and added 1 to account for the size of the siding. The size of the balls are realistic as well, being converted from inches to ft (2.25 in -> 0.1875 ft), scaled up by 2.5 (0.1875 -> 0.46875), and then halved to create the radius (0.46875 -> 0.234375). All of these aesthetic changes have been done to give the simulation as much of a realistic feel as possible. We have chosen not to include the animation of the force application from the cue to the cue ball as it does not add the simulation in a technical way, all we need is the angle of the shot and the force applied. We have also chosen not to simulate rotation of the balls as we do not have the capabilities to rotate the object in visualpython.

When you run our application a GUI is opened (using Tkinter) that has 5 textviews that the user can give input into. The first is the mass of the ball, for realistic purposes we recommend using the standard weight of a billiards ball which is 0.107. The second is the simulated force applied to the cue ball upon initialization; recommended force input is 108 which is the estimated average shot force. The third is the force of gravity which is generally around 9.81 N, depending of course where the simulation is supposed to be located. Fourth we have the coefficient of friction for the ball during movement; realistic values are 0.15-0.4 and typically it would be 0.2. The last input value is the angle that the cue ball initially moves. Due to visual python constraints and the way we applied the physics the angle is as if you were looking up under the table at the cue ball (ex. 90 degrees would be straight down at the other balls and 270 would be up directly opposite of the balls). When the “start simulation” button is pressed the startSimulation function is called. Within this function we apply the inputted values to globally declared variables and we call two other function: drawTable() and animate(). drawTable() sets up all of the objects (balls and table) as well as initializes the graph that is produced and animate() is where all of the simulation happens.

Animate() is the main function of the simulation, this is where the time-steps are taken and all the relevant equations and data are applied. The first thing that happens is the shot angle is applied to the cue ball, this is necessary for the second step which calls the calculateVelocity() method on the cue ball. calculateVelocity() first finds the magnitude of the velocity (no direction attached) and stores it in v2. Magnitude is calculated using:

$$v2 = \frac{F_i \Delta T + m v_1}{m}$$

Where F_i is the initial force, ΔT is the time step size, m is the mass of the ball, and v_1 is the velocity of the ball in the previous time-step. After calculating the magnitude the velocity vector is created using calXComponent() and calcZComponent(). Both of these functions take the angle of the ball and the magnitude and apply trigonometric functions to calculate the components velocity. The last thing this function does is decide whether or not the velocity vector should have negative components based on the shot angle. After the cue ball is set up all of the collision array balls (every other ball) are initialized. It is now that the simulation steps into a time incrementing loop. First, every collision[] ball is checked for a collision with the cue ball. The checkCollision() method decides if the addition of both balls radiuses is greater than the distance between the two balls (calculated with Pythagoreans theorem on points) than it returns true. If true then the velocities of the two balls are recalculated using:

$$\vec{n} = \frac{\vec{r}_1 - \vec{r}_2}{|\vec{r}_1 - \vec{r}_2|}$$

, to calculate the normal to the collision plane,

$$\vec{v}_{n_1} = [\vec{v}_1 \cdot (-\vec{n})](-\vec{n})$$
$$\vec{v}_{n_2} = [\vec{v}_2 \cdot \vec{n}]\vec{n}$$

, to calculate the normal component of the velocity vector,

$$\vec{v}_{t_1} = \vec{v}_{n_1} - \vec{v}_1$$
$$\vec{v}_{t_2} = \vec{v}_{n_2} - \vec{v}_2$$

, to calculate the tangential components, and finally,

$$\vec{v}_1' = \vec{v}_{t_1} + \vec{v}_{n_2}$$
$$\vec{v}_2' = \vec{v}_{t_2} + \vec{v}_{n_1}$$

, to calculate the full velocities of each ball.

If the collision[] ball has not yet had movement then it is given “checks” which just tell the system if the vectors components should be negative or not and it is decided what its initial angle will be. The velocity magnitudes are then calculated which uses Pythagorean Theorem and the velocity components. The angle of the balls movement is important for updating the velocity in the next step so it is calculated using trig functions.

After checking whether or not the cue ball had collided with any other balls it is checked whether any of the collision[] balls collided with any other collision[] balls also using checkCollision(). If true, then the same steps as above were taken for the collision[] balls. Next all of the balls have their velocity updated using:

$$v2_2 = \frac{-\mu g m \Delta t + m v 2_1}{m}$$

Where mu is the coefficient of friction, g is the user inputted gravity, m is the user inputted mass, and t is the time step. The velocity vectors are created and the components are turned negative if appropriate. Each ball is then put through the hitWall() function which checks if the ball has hit a wall, moves the ball to the edge of the wall, turns the appropriate velocity component negative, and then updates the checks. Lastly each ball has their position updated. The simulation ends when the time runs out or when each ball has a velocity of 0. All of this is why the data can be trusted; we put lots of effort into ensuring as realistic a simulation as possible.