James Sunseri

Yuri Kolomensky

Physics 77

5/17/19

Capstone Final Project Write-Up

The goals of this project were to successfully create an accurate two-dimensional graphical simulation of collisions between circular objects utilizing the python language and skills we developed throughout the course of the semester. Since this project was modeling colliding bodies, we figured the most exciting project for our purposes would to be to just build a billiards game from scratch which allows us to model elastic collisions and friction.

The game of billiards has been around at least since the beginning of the 20th century, and it has been a staple of bar game culture ever since. This game as it stands is quite simple, one takes a stick and tries to strategically knock all of their balls into their desired pockets before the opposing player gets the chance to, the player to get all their balls sunk wins the game unless they sink the infamous solid-eight ball before it is the last ball for that player. This game has really interesting physics because it allows us to simulate macroscopic nearly totally elastic collisions between balls which is not commonly observed in nature unless on a microscopic scale. We can take the collisions as approximately elastic because all though they are rotating slightly extended bodies (circles or spheres), they are incredibly smooth and the collisions with other balls only occur for such a small time that the frictional forces from the spin and other factors of them being spheres are so minute that it does not impact the way the collisions happen by a noticeable amount. They behave as if they were just disks on a frictional plane (similar to hockey pucks) colliding.

The thought process behind our code was fairly straight forward. We knew we had to create three major things for this game to work: the physics engine, the user input and interface, and the game logic. I worked on the physics engine, Frenly worked on the user input, and Darby worked on the game logic. We worked together on a few things to stitch the final project together. The way PyGame works is that we have to draw a surface in a window, and anything we want to make visible to the user has to be drawn to that surface. Just like any other video game in order to make things move on this surface we have to update the surface and redraw the translated images with each new update. The best way to do this is to have a main game loop (while loop) function which takes care of this process, and every function which we write will be nested in this while loop so that each time the while loop is iterated the game mechanics, graphics, sounds, etc.… will be updated.

The physics engine which I wrote utilizes a Vector class which we made through the concepts and tools of object-oriented programming. This vector class makes the code shorter, slightly more optimized, easier to read, and easier to understand. We also defined a Ball class (each ball has its own mass, radius, and color), QueStick class, and Pocket class. The physics engine consists three main functions: BallsCollide, Move, and CollisionDetect. The Move function iterates through all the balls and applies friction to the opposing direction of motion by calculating the speed of the ball (length of velocity vector) and dividing the fixed friction factor (friction * secs) by that speed, that creates a scaled small vector which we just subtract from the current velocity of the ball to slow down the ball by friction. If the magnitude of the Velocity vector is less than the friction factor, then we just say that the velocity vector of the ball is a 0 vector. It is also important to note the Move function takes “secs” as an argument, “secs” comes from the Draw function, it is how much time it takes the draw function to run. The reason for all of this is to fix the way the positions update: instead of pixels per frame, we are now pixels per second which is smoother to the eye. The CollisionDetect iterates through all the balls in a standard for-loop and a nested for-loop, and determines whether a ball has collided with a ball or a wall, if a ball-wall collision occurs then simply flip the velocity component that was opposing the wall (if ball hits right wall, flip x component of vector), if it collides with a ball we call the BallsCollide function. The BallsCollide function is the heart of the collisions. Since we are working in vectors the function is fairly short, we find the center between the two balls by averaging the position, find the vector between the two ball’s centers by vector subtraction then normalizing that directional vector. Using this we create a new set of axes to treat it as a simpler 1D collision. The effective speed in this collision for our equations is that which is projected onto the difference axis between the two balls. Then we just use this equation from Wikipedia and the 5A Textbook (angle brackets mean dot product).

$$\begin{aligned}
\mathbf{v}'_1 &= \mathbf{v}_1 - \frac{2m_2}{m_1+m_2}\,\frac{\langle \mathbf{v}_1-\mathbf{v}_2,\,\mathbf{x}_1-\mathbf{x}_2\rangle}{\|\mathbf{x}_1-\mathbf{x}_2\|^2}\,(\mathbf{x}_1-\mathbf{x}_2),\\
\mathbf{v}'_2 &= \mathbf{v}_2 - \frac{2m_1}{m_1+m_2}\,\frac{\langle \mathbf{v}_2-\mathbf{v}_1,\,\mathbf{x}_2-\mathbf{x}_1\rangle}{\|\mathbf{x}_2-\mathbf{x}_1\|^2}\,(\mathbf{x}_2-\mathbf{x}_1)
\end{aligned}$$

Since, we are working in vector form this becomes a much easier equation to code, rather than dealing with several angles and several if and elif statements for each type of possible collision. To add some “realness” to the collisions I added an “Elasticity” factor (slightly less than 1) which allows us to better simulate loss of energy due to heat, friction, and sound.

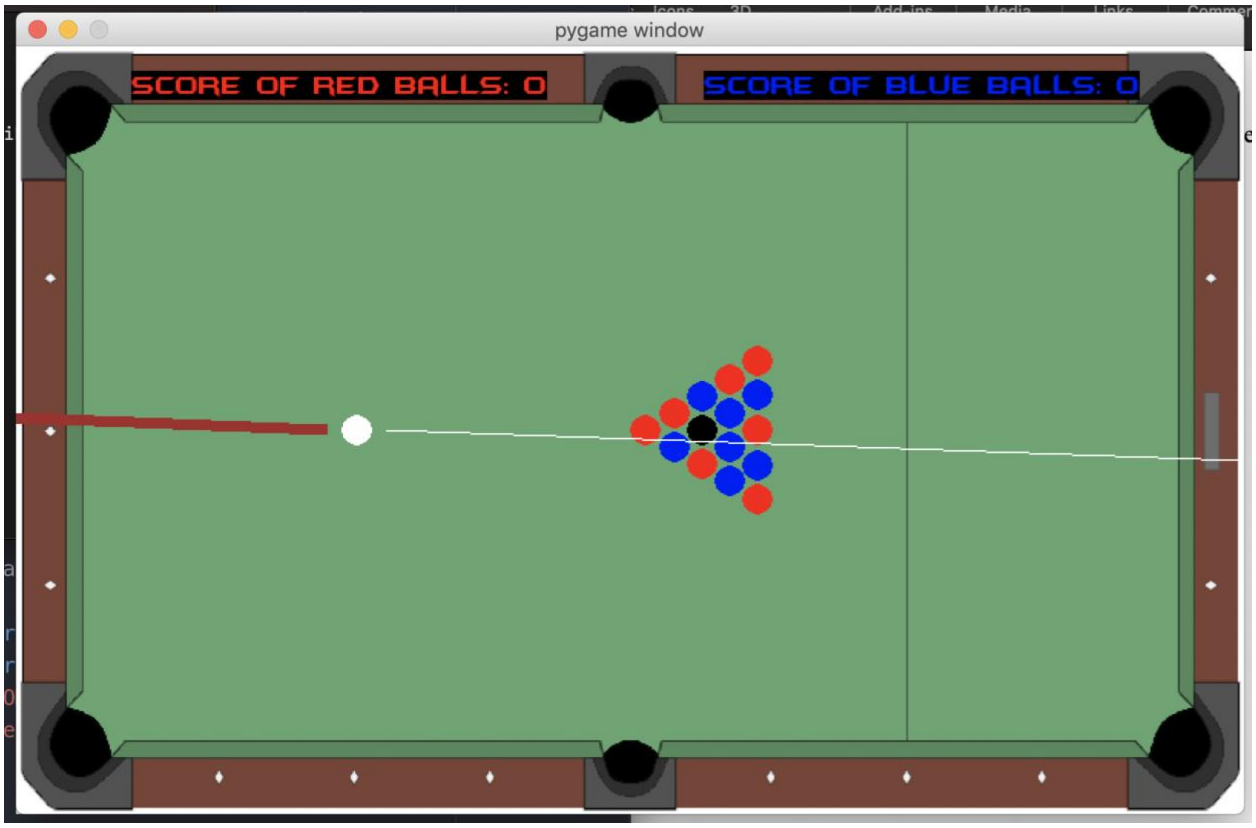
The way which Frenly went about creating the user input mechanics was fairly simple as well. He utilized several PyGame functions which allow him to set an (x, y) coordinate for the mouse at any given time, he then used the que ball's position as another point. With these two points he was able to create a triangle by finding the Δx and Δy and use trigonometry to find the angle. He then drew lines of fixed length and thickness using this new found angle which is our effective que stick and guider line which can rotate around the screen. In order to only draw the que stick when the user is ready to play their turn Frenly made sure that the magnitude of the velocity vector had to be 0 in order for the que stick to be drawn on the screen so the user isn't trying to hit a moving que ball. The way he went about creating a system for the power of a given shot was to record the time at which the spacebar is pressed down and the time at which the spacebar was released, subtract the two times and scale it by a power factor. This allows the user to have full control of direction and power of their shot (should hold down for 1-4 seconds).

The way Darby went about making the game logic was to incorporate a score mechanism, a scratch mechanism, and a win/lose mechanism. She first defined a pocket_collision function which detects whether a ball has collided with a pocket. The way to do this is to find the distance between the centers of the two circles and if that is less than the combined radii of the two circles it is a collision, this is the same way that the CollisionDetect function determines whether a ball-ball collision has occurred as well. In order to understand the logic, we have to know that all the balls are created using the Ball class and appended to the Balls list, the first ball being the que ball and the last ball being the 8 Ball. If a ball collides with a pocket, if it is not the first or last ball, then delete the ball from the Balls list. She also created two other lists called red_balls and blue_balls, the length always starts at 7 for each list and if a ball is removed then that list length gets smaller we subtract the new length from the old to get a score that increases every time that given color's team sinks their ball. If the que ball is sunk then it is just put back where it was originally and if the 8ball is not the last ball in the list and it is sunk then the player loses, if it is the last ball then the player wins the game.

At the very end we put a little artistic flare into the game and added a custom font for the score counters and a pool table image as the background surface along with a list of sounds of pool balls colliding and background music to give the game more of a fun vibe.

We were certainly able to meet our goals with our pool game, and we were quite amazed of what we were able to accomplish. The collisions agree with theory pretty well, the only deviations are that they are not rotating rigid bodies but instead simple 2D Elastic collisions, but after discussing with the GSI we felt that if we were to try and incorporate spin that would be out of the scope of the class. We learned that because PyGame was optimized for windows systems it runs slightly differently between mac and windows and it has way better performance for windows than Mac. Because of this the mac version experiences much more lag than the windows version and the friction value in the move function has to change. For Mac the value has to be 1200, and for windows it should be 120. I would also like to note that the collisions do get inaccurate with mac at high speeds because of this problem. The inaccuracy comes solely from the framerate not being able to be updated fast enough to accurately run the collision detection function in time for the correct calculation to occur. It works absolutely flawlessly on windows though. I am running a 2017 MacBook Pro and this lag occurs on Mac while no lag at all for a windows 2014 computer. This is the only real source of error found in our computer and it only stems from the library which we are using.

Going forward, a way to improve the code would be to calculate the collisions before they occur by predicting where the balls will collide in the future, then picking the most likely collision and storing it so that the there is no delay in the collision calculations, this is how true physics engines go about accurately simulating collisions, but this would've been far more complicated. Final note: in order to run this game, you have to install PyGame.



Sources:

https://en.wikipedia.org/wiki/Elastic_collision#Two-dimensional

http://programarcadegames.com/index.php

https://stackoverflow.com/questions/35211114/2d-elastic-ball-collision-physics

https://stackoverflow.com/questions/345838/ball-to-ball-collision-detection-and-handling

https://gamedevelopment.tutsplus.com/tutorials/when-worlds-collide-simulating-circle-circle-collisions--gamedev-769

https://www.pygame.org/docs/

Note: I did ask a couple questions to a CS friend of mine named Dhruba Ghosh and he suggested the idea of using a Vector class to make the code easier to write.