Aerial Robotics Kharagpur Task-2 Documentation

Rishabh Agarwal

***Abstract*— The main objective of this task was to determine the trajectory of the billiard balls after one on of them was initially hit by a cue, and thereafter collides with the other balls and the cushions on the tables. The collisions are assumed elastic.**

I first started the task with the detection of the balls (assumed to be perfect circles) and the cue. After finding the direction of initial motion, I detected collisions between the balls and between the walls and balls. I changed the direction of motion of a ball if it hits the walls. If it strikes another ball, the struck ball becomes the active ball and the initial ball comes to a rest. I have made a simulation of the predicted trajectory using OpenCV. However the determined trajectory is a very approximate one, involving a lot of approximations.

The task gave an introduction to image processing and the basics of shape and collision detection. Image processing is an essential part of robot vision. Developing on the basics learnt in this task, we can program automated drones to detect objects and avoiding collisions.

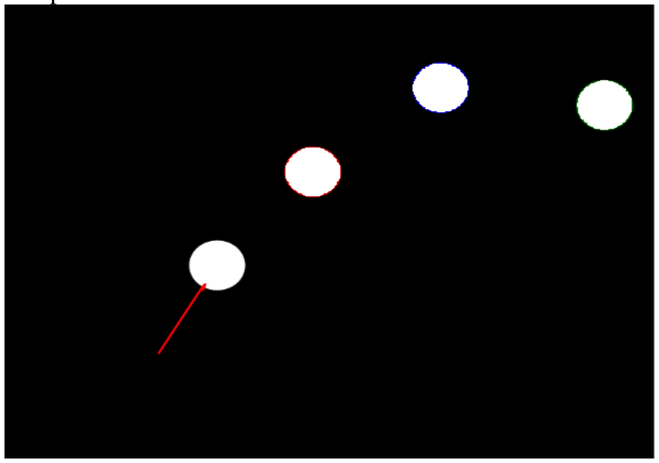
I. INTRODUCTION

In this problem statement we were given images of a pool table with the cue and ball to be hit indicated and we find pairs of collisions assuming perfectly elastic ball-ball collisions and ball-table collisions. The balls centers might not be in-line and this would have lead to two divergent ball movements but we assume that in case of ball-ball collision, the incoming ball stops and the incident-ball moves with the same velocity as the incoming ball.

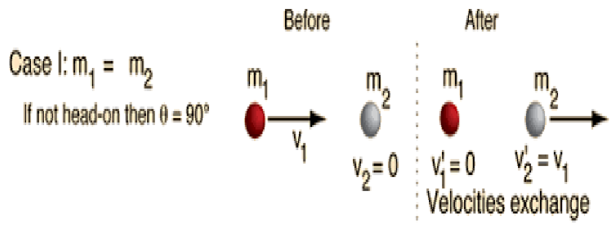
I decided to make a simulation actually showing the various collisions taking place as opposed to just calculating and displaying the coordinates of the collision. I have done this using the basic concept of elastic collisions ie the ball hit will move in the direction of the slope joining the centers of the colliding balls.

II. PROBLEM STATEMENT

In the problem statement we have been given an image representing a billiard table with a cue and different balls. We have to identify the ball being hit by the cue and predict the collisions that will take place between this ball and the other balls. In these collisions, we have assumed that the ball initially in motion comes to rest after colliding and there happens a complete transfer of velocity to the ball which was hit. In real world scenarios, this kind of collision will take place only if the collision is perfectly elastic i.e, no friction exists between the colliding balls and both kinetic energy and momentum are conserved and if the two balls have the same mass. The problem statement has been designed taking into consideration these 2 assumptions. I have taken a simulation based approach of dealing with this problem, in which the collisions are actually visible to the viewer. Below is an example of the billiard board setup we have been given in the problem statement.



The case of elastic collision we have been required to use can be illustrated by the image below:



The problem statement does not explicitly mention the number of collisions that need to be recorded or the time duration till which we have to continue the collision. Thus I have played the simulation till the user enters 'q' to quit.

III. RELATED WORK

Other approaches to solving this problem can involve heavy calculation based approaches rather than a visual approach. These will involve calculating the coordinates where collisions can occur using a reference coordinate system with the coordinates of the wall planes being stored as well as storing coordinated of non colliding balls. The challenges of visual display are omitted in this method.

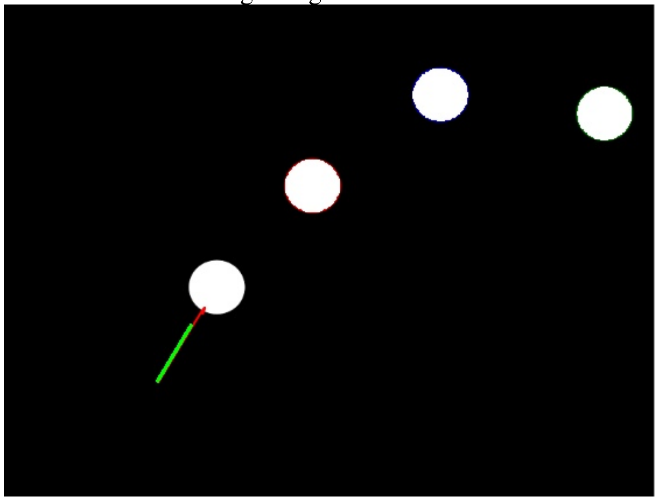
IV. INITIAL ATTEMPTS

Initially I faced a lot of issues with the Hough Lines function, as it was not detecting the cue. But adjusting the parameters I was able to get the required results. Also I was printing a black circle in place of the initial position of the circle and in the new position I was printing the circle. But in some cases the circle is taking 2 steps at a time and hence there were traces of the previous circle left, which made the output screen a bit messy. So I decided to print each frame in a new screen so that I didn't have to deal with printing black circles.

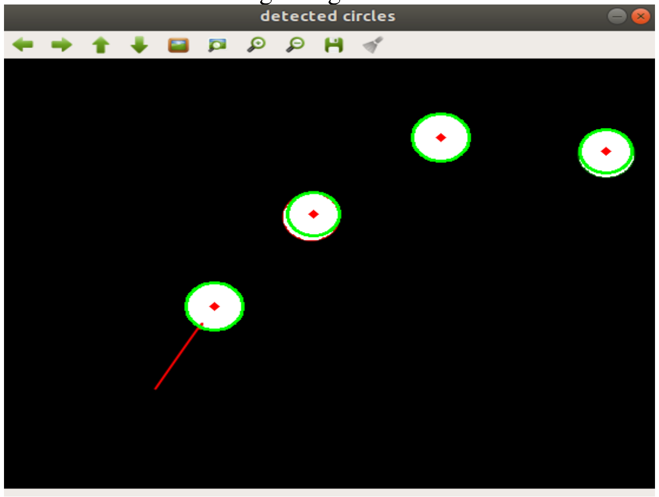
V. FINAL APPROACH

General Overview: In this task, I am displaying the path of the moving ball and the collisions that take place during its motion. My approach starts with first finding the initial slope along which the ball will move by detecting the cue using **Hough Lines**, which gives us the end coordinates of the cue and finding the ball closest to either of its ends. This ball is currently our active ball denoted by **ball_hit** which is going to move. The balls are detected using **Hough Circles** which returns their center coordinates and radii as well. The slope of the line along which it is going to move is obtained using the end coordinates of the cue. The motion of the moving ball throughout the code is shown by printing it in **new** board after moving it by the required number of pixels along x as well as y.

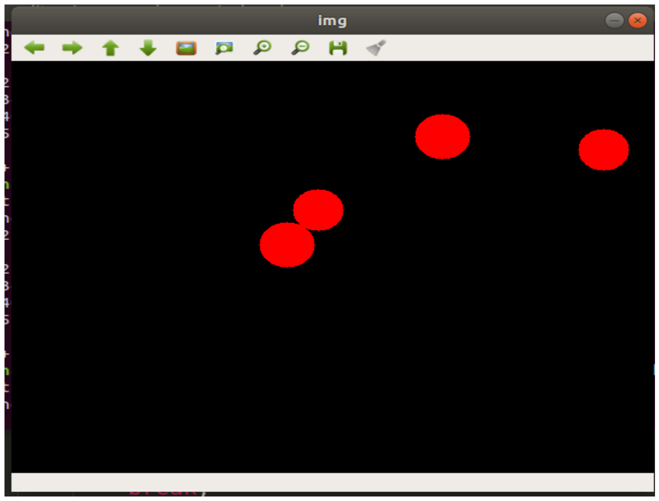
Detction of lines using Hough Lines:



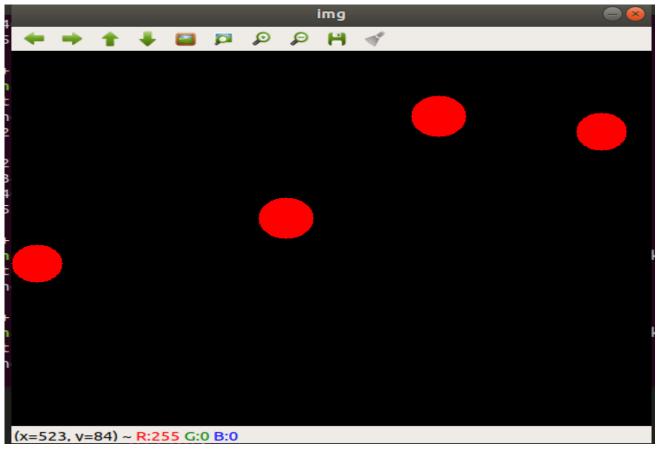
Detction of circles using Hough Circles:



Ball with ball collisions: To detect ball collisions **coll_bnb** is called. I used the geometric concept that **distance between centers of the balls is less than or equal to the sum of their radii**. However, due to the availability of only integer coordinates and non unity increasing of x and y coordinates, a compensation of approximately 1 pixel had to be made to ensure proper detection in a wide variety of cases including the ones in which a slight overlap between the two balls occurs as well. After the collision is detected, the line joining their centers is obtained and the ball which was hit is made the ball_hit which now traverses this line. After the ball_hit is updated, its made to travel 1 additional step along this new line to ensure that no recollision occurs with the previous ball.



Ball with wall collisions: To detect collisions of balls with the respective walls **coll_bnw** function is called. I used the fact that whell a ball collides with a wall, the **perpendicular distance of the center of the ball with the wall plane will be equal to the radius** of the ball. We also allow values less than the radius in case of slight overlapping due to x or y values increasing with high magnitude. After the collision, the increase or decrease of x is reversed if the collision is with the left/right wall and y values will be reversed if collision is with up/down wall. The walls are by default taken to be the edges of the image window.



Updating Coordinates: The coordinates of the **center** of ball_hit need to be updated in every iteration. But before the coordinates can be incremented/decremented, **optimize_delx_dely** is called. This function determines the change in x and y coordinates depending on the slope. It aslo rounds it off to the nearest integer, takes care of when slope tends to zero or infinity and even controls the magnitude of the increment/decrement from getting too high. We check whether the slope is greater than 1 or less than -1. If that is the case then x is increased/decreased by 1 and y by the corresponding slope. else y is increased/decreased by 1 and x by the corresponding slope.

Approximations: The first major approximation I have taken is that**slope values can only be integers (1,2,3,4...)** since when we move in the x-direction by 1, we can move in the y-direction by 1, 2, 3, 4, etc. Similarly when we move in the y-direction by 1, we can move in the x-direction by 1, 2, 3, 4, etc. Therefore, **slope values can also be $\frac{1}{4}$, $\frac{1}{3}$, $\frac{1}{2}$**. These are a result of the unavailability of floating-point numbers, and impose a lot of restrictions on the movement of the balls. Another approximation is that **slope angles greater than 86.18° are taken to be 90**$^\circ$ and hence the ball moves vertically, and **slope angles less than 3.81**$^\circ$ **are considered to be 0** and hence ball moves horizontally. High slope values lead to higher magnitude of decrement or increment of the x and y center coordinates, which results in several anomalous situations- ball moving inside another ball, ball moving inside the table cushions. To avoid this, the **magnitude of the increment or decrement is reduced**. This affects the speed of the ball and it might happen that a slow moving ball causes a ball to move very fast after collision and vice-versa. This is one of the cons of my approach but this does not affect the trajectory or the angle in which the balls move.

Errors encountered and their solution: One recurring problem faced by me was the movement of the balls into other balls or into the walls. For detecting collisions, the distance between the centre of the balls (or that between the balls and walls) has to lie in a range of values, and this is checked before the ball moves. And if the ball **intersects another ball collision might not be detected**. Also if the **increment/decrement of the struck ball is too low, successive false collisons might be detected between the same balls**. In order to overcome this I have moved the ball struck in the direction it is supposed to move before checking for collisions, so that the balls are not too close when collisions are being checked for. Also if the ball is entering the cushion, it is being shifted parallely out of the cushion. The trajectory is printed in a new window every time, as printing it in the same window was leaving behind the outer traces of the circles.

VI. RESULTS AND OBSERVATIONS

As opposed to this simulation which I've made, I could've used Hough Circles to just read the values of the coordinates of centers and radii of circles and Hough Lines for the end points of the cue and instead of displaying the movement of the ball just displayed the coordinates at which the collision would have taken place. This could have been done with a much greater accuracy as then the coordinates of the centres could be stored as floating numbers and aslo the slope could take up a larger range of values as the exact angles could be found using the trigonometric functions available in Python.

Despite some errors arising due to approximation I chose the visual approach as opposed to the approach mentioned above. This is because I found it better to visualize the the entire motion of the balls and see the collisions happening in real time. Also the errors due to approximation are well within the limits, and hence the result is fairly accurate.

VII. FUTURE WORK

The major problem in my algorithm arises due to the large number of approximations that had to be taken since the pixels are integral values. These problems can be solved if we can show the movement of the ball in an environment where the circle can be drawn in floating point coordinates as well. The current algorithm too can be improved if we decide not ot show each and every step of the simulation, but only show the final positions. This would reduce the error in the approximations. A floating-point coordinate plotting system will be ensure a close to error free visual approach to the problem statement.

CONCLUSION

The task required us to simulate the path of a billiard ball around a pool table colliding perfectly with similar balls as well as the sides of the table. I used a visual approach in which I displayed the ball as it moved across the board and as it collided with the other balls. This task enabled me to get acquainted with OpenCV and also the basics of computer vision as well as helps us analysing a simple motion and detecting collisions of circles which can be helpful in programming automated drones.

REFERENCES

[1] https://opencv-python-tutroals.readthedocs.io/en/latest/py_tutorials/py_imgproc/py_houg
[2] https://opencv-python-tutroals.readthedocs.io/en/latest/py_tutorials/py_imgproc/py_houg