Task 2: Face Detection Game

Ashwani Kumar Kamal

***Abstract*— The purpose of this document is to demonstrate how a simple arcade game can be developed in python. Also how we can make it more interesting using face detection using OpenCV library**

I. INTRODUCTION

A game environment was to be created similar to the animation provided. The main focus was on two elements of the game: Dynamics of the ball and mapping Face detection movement onto the game window. I used a blank window and added all the elements of the game into it. Added a score counter and a game over screen.

II. PROBLEM STATEMENT

Have a look at the animation provided here. You have to make a game environment where you can play a similar game. Your face will replace Tom's face. For that you have to detect and track your face, assume your face to be a perfect circle. You'll have to keep a track of the ball as well. The ball should follow the laws of physics and the collisions can be assumed to be perfectly elastic. The game ends if the ball drops to the floor. You do not have to consider gravity, i.e., the ball comes down only after hitting the upper wall

III. RELATED WORK

Got a basic understanding of how face detection works in general.

IV. INITIAL ATTEMPTS

The first thought was to make a simple ball bouncing simulation and then step by step, add constraints of collision, add a big ball (referred to as player throughout this documentation) constrained to move horizontally in the bottom corridor of the window, then adding collision constraints for the two balls. Ball movement to be tracked using Cartesian system, however the motion to be tracked using relative polar system (an angle variable and a step variable)

V. FINAL APPROACH

A. Setting up playable environment

The first step was to realise game potential by making a screen and the most basic elements namely the ball and the player

B. Animating components

There doesn't seem to exist any animation creating methods built in OpenCV. Given the problem statement explicitly mentions to make a simple UI using only OpenCV, I knew what I had to do- back to the basics of animation. The idea is to show something using predefined methods like cv2.circle() or cv2.line, make a replica of the same object but displaced by some defined distance, then blackening out the original object. The movement of player is fairly simple and basic- displacing the x coordinate by some definite amount and blackening the original. However for the ball, not only movement magnitude but the direction also matters. So an $angle$ variable has been used to keep track of the current angle of the ball velocity vector with the x axis of game window. Then having the polar system at our disposal we can simply add $r\cos(angle)$ to x coordinate and $r\sin(angle)$ to the y coordinate to realise a movement in the desired direction. r being the step movement magnitude.

C. Adding constraints to movement and initial conditions

A Class has been implemented named $Controller()$. Creating an instance using the x and y coordinate ($ball\_pos = Controller(initial\_x, initial\_y)$) which has the $\_\_add\_\_$ magic method so that we can easily add the displaced coordinates to realise movement.
The initial condition for $angle$ must be random each time the game starts or the game will become monotonous. So starting the game window invokes a random call to the $random.randint()$ function with the required scaling to get an angle in some range not having 0°or 90°as they can force the game in an infinite loop (ball bouncing off right and left directions with no vertical velocity component)
Constraints were to be added to right and left walls:
if($ball\_pos.x \geq$ width - 50 or $ball\_pos.x \leq$ 50):

$$angle = 180 - angle$$

where $ball\_pos$ is an instance of $Controller()$ class, $width$ being the width of game window
Similar conditions have been imposed to control the flow of game
if($ball\_pos.y \leq 50$):

$$angle = -angle$$

if($ball\_pos.y \geq height - 50$):

$$return gameOver$$

if($distance \leq 100$ and $distance \geq 20$):

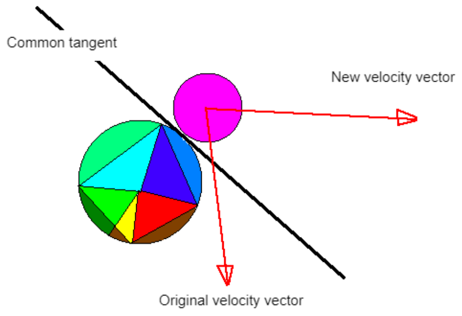
$$\theta = \tan^{-1}\left(\frac{ball\_pos.y - pos.y}{ball\_pos.x - pos.x}\right)$$

$$\theta = \theta * \frac{180}{\pi}$$

$$angle = \mid 2 * theta - angle - 180 \mid \%360$$

$$if(2 * \theta - angle - 180 \geq 0) : angle* = -1$$

$distance$ variable contains the current distance between center of ball and player's center (to realise the player collision).

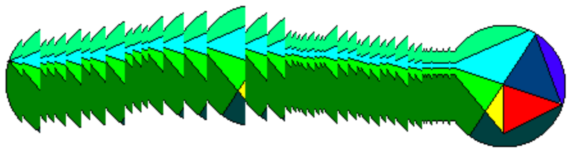


Visual representation of last condition (ball collides with player): Angle of reflection = angle of incidence at common tangent

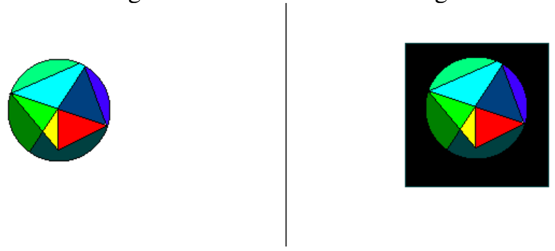
All these calculatons can be worked out very easily using the aid of geometry. I used the modulo 360 because after a few bounces the angle becomes very large (as a consequnce of adding $2 * \theta$ hence, $\sin$ and $\cos$ function start returning erroneous values; making the ball go out of window suddenly. So we scale the new angle so that it remains in the range [0, 360)). The absolute value function was added because the modulo operator with a negative number results into something which we don't want. For example let the value inside absolute function be 11 and say we're doing modulo 4, 11%4 = 3 but (-11)%4 = 1. To overcome this anomaly add an absolute function before modulo and negate the result if it was negative earlier.

D. Adding Face detection

After having a workable game environment it was time to add the final dish into the plate- face detection. I made a separate program this time. The motive was to bridge the two programs and hence get the final product. After implementing the face detection algorithm on the main feed, I took the roi (region of interest) and did an overlay of it on a black image, not to mention that the overlay was 'mapped' onto the bottom corridor (by only allowing x coordinate to change and fixing y to some definite value). However there was one problem- if we move our head object the previous instance of the head remains persisting on the screen and the new one is added too. This was happening because I was not blackening out the previous instance. But since it's an overlay blackening out previous instance was not a simple task. Hence to work around the problem I added another overlay (black in color referred to as 'mask' in this documentation) which would surround the roi. The following image explains it better.



A tail of images is created on moving head object



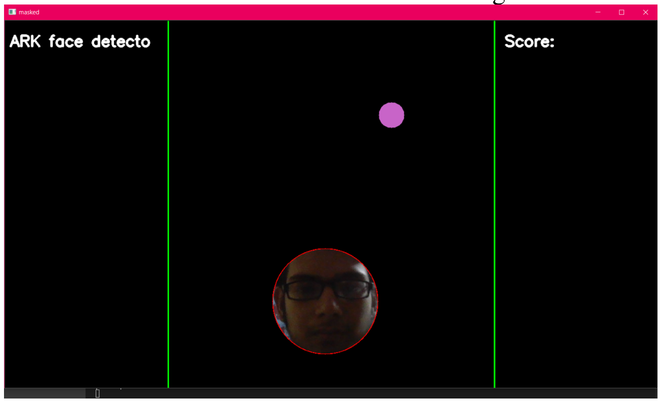
A black mask overrides the tail with black color thereby making the whole motion smooth

E. Video-feed width and final implementation

The width of video feed less than our game window, also I wanted some space on my screen to display score and title of game, so I trimmed the playable area to that of the area of video feed (specifically width) and changed some variables from original game. Added a score function. Rest all things are same except minor changes in code- like instead of fixed $distance$ threshold, now I'm taking ball's radius + player's radius

VI. Results and Observation

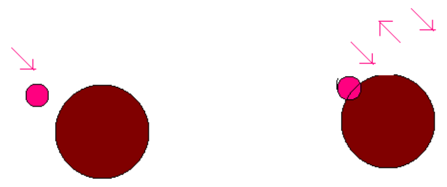
The final window looks something like this:



The overall game performance is good but only if the following conditions are fulfilled:

- There's only one face in frame
- Face is upright, detectable and within the frame width
- Face is moved gently and not very fast

A big bug in game- The whole game runs under single thread and movement is controlled by pixel step size. Imagine a situation in which the player moves a bit fast and if it gets to a position overlapping with the ball, then a series of bounces happens each contributing to score. The reason can be explained but first have a visual representation the whole thing:



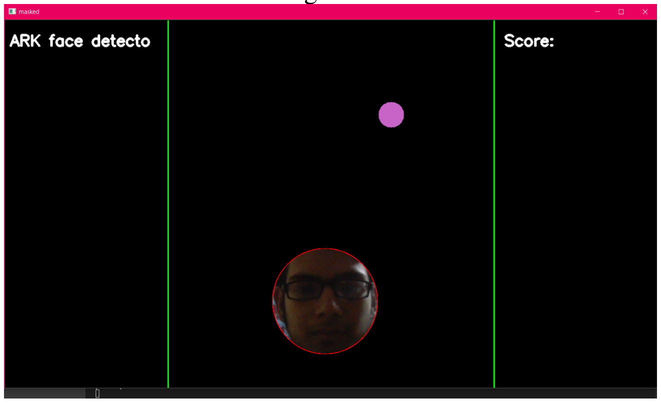
Now if the player ball is not withdrawn, then a potential infinite loop can be invoked which changes the angle vector continuously without letting the ball get out of player's ball area.
Another potential bug is that when face gets even a teensy bit outside frame, the overlay raises Value Error. Currently I'm taking care of it by showing a warning screen for the player.

VII. Future Work

Multi-threading to optimise performance and adding more interactive elements

Conclusion

The final game window is:



Score updates each time ball bounces from player's
Important fact being that face detection is very dynamic and the data changes every millisecond, the game is not very robust. The prototype is however performing very good.
Play the game in well lit environment, single face, atleast 50 cm away from the screen (the far the better, as it reduces the chances of running into Value Error described above.)

References

[1] OpenCV face detection documentation OpenCV: Cascade Classifier