Aerial Robotics Kharagpur Task Documentation (Task 2.2)

Ashutosh Ghuge, Roll No. 20CH10019

***Abstract*— The main focus of this task was using face detection in OpenCV. When tackling real world scenarios the ability to differentiate between humans and objects is vital. For example, using a drone in a rescue operation or for supplies in a disaster struck region, the drone should be able to locate humans with high precision. In many similar scenarios, the concept of face detection proves to be important.**

I. INTRODUCTION

The problem statement was to build a simple game where face detection is utilized to map the face of the player and a ball is to be prevented from falling below a certain level. There were some problems with developing the game environment initially, but were later resolved. The face detection in the game is done by using haar cascades.

II. PROBLEM STATEMENT

As mentioned in the introduction the problem in the task was pretty straightforward-building a simple game environment using OpenCV and face detection. The basic idea of the game was to have a ball whose motion is independent of gravity as is only affected by collision which are perfectly elastic. The boundaries of the frame are the 3 colliders and the face of the player is the 4th collider. In the game environment I made, the player has to move his head around the frame and try to stop the ball from falling below a certain level on the vertical axis. The ball is given some initial velocity and the player's face is mapped as a circular object. The major goal of the task was to learn the implementation of face detection in image processing.

III. INITIAL ATTEMPTS

My first attempt was to develop the basic motion of the ball. I used a simple function to calculate the position of the ball in real time using the velocity in both x and y directions.

$$pos_x + = V * v_x$$

$$pos_y + = V * v_y$$

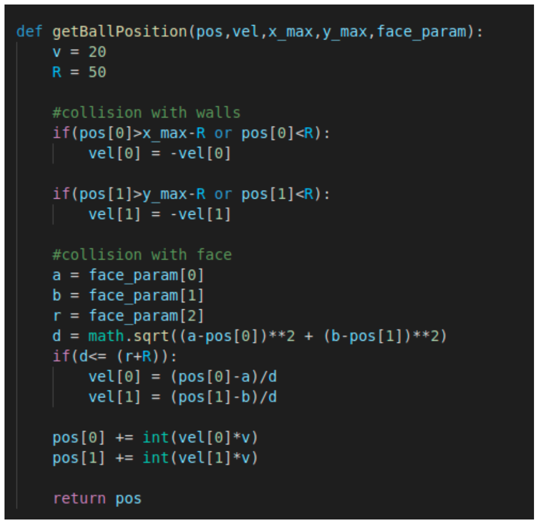
$$where v_x and v_y are the direction vectors$$

$$and V is the velocity magnitude$$

For displaying the ball I used the built-in function to draw a circle on an img in OpenCV. I struggled with the idea of how to bring the face into the game and initially was thinking about forming a separate frame for the game and taking in only the face from live video feed and superimposing it on the blank frame along with the ball. But that seemed to complicated and also had a negative point as it would take a lot of run time thus making the game slow.

IV. FINAL APPROACH

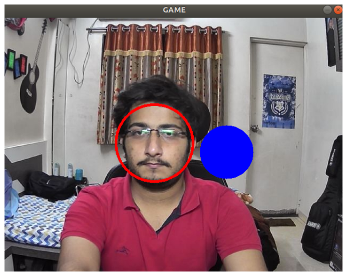
My final approach to the problem was directly capturing the live camera feed and drawing the ball on each frame in real time. For face detection I used the haar cascade which has functions that return co-ordinates of a reactangle around the detected face. Using those co-ordinates I drew a circular boundary around the face. For the motion of the ball, I used the formula mentioned in the previous section for updating the position of the ball. For detecting collision of the ball with the walls and face I used the following function.

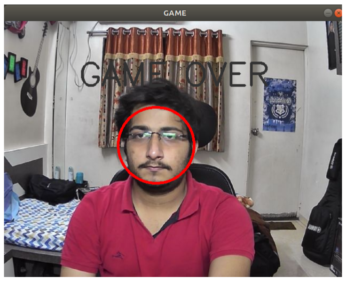


```python
def getBallPosition(pos,vel,x_max,y_max,face_param):
    v = 20
    R = 50

    #collision with walls
    if(pos[0]>x_max-R or pos[0]<R):
        vel[0] = -vel[0]

    if(pos[1]>y_max-R or pos[1]<R):
        vel[1] = -vel[1]

    #collision with face
    a = face_param[0]
    b = face_param[1]
    r = face_param[2]
    d = math.sqrt((a-pos[0])**2 + (b-pos[1])**2)
    if(d<= (r+R)):
        vel[0] = (pos[0]-a)/d
        vel[1] = (pos[1]-b)/d

    pos[0] += int(vel[0]*v)
    pos[1] += int(vel[1]*v)

    return pos
```

The array face_param contains the x,y coordinates of the center and the radius of the circle around the face.

The main function which plays the game contains a while loop which continuously reads the frames of the video feed and detects the location of the face and hence, calculates the position of the ball and draws it over there. If the ball falls below a certain height, a game over message is displayed. The images of a game play are shown below.





V. Results and Observation

The face detection algorithm used in this program is by using the Haar Cascade. It is basically a ML model that is trained by exposing it to a huge number of positive and negatives images of a face. After a lot of iterations the model detects faces with high accuracy. The accuracy can be increased by increasing the number of iterations and the variations in the dataset provided to train. Although haar cascade detects faces pretty accurately it perfoms a little poorly when it comes to time required. Applying other deep learning algorithms along with haar cascades can reduce the runtime to less than half.

VI. Future Work

As mentioned above, the problem with haar cascades is the run time. Sometimes the game lags a little due to the time taken by the algorithm to detect rapid movement of the face and also since many operations are performed on each frame. Face detection is a very interesting topic and the accuracy and runtime can be improved by researching and implementing some other deep learning models.

Conclusion

Although, the apparant aim of the problem was to make a simple game, it involved a lot of learning about face detection methods and how different deep learning models can be applied for the same. The different methods available for face detection can be compared by seeing the runtime is this game and we can come up with better and faster models, which can prove to be vital in rescue and supply purposes. Also in making human friendly drones, face detection will be of prime importance and the use of better and more efficient algorithms is always a topic of research and improvement.

References

[1] https://www.youtube.com/watch?v=oXlwWbU8l2o
[2] https://docs.opencv.org/master/d2/d99/tutorial_js_face_detection.html
[3] http://datahacker.rs/017-face-detection-algorithms-comparison/

Aerial Robotics Kharagpur Task Documentation (Task 3.1)

Ashutosh Ghuge, Roll No. 20CH10019

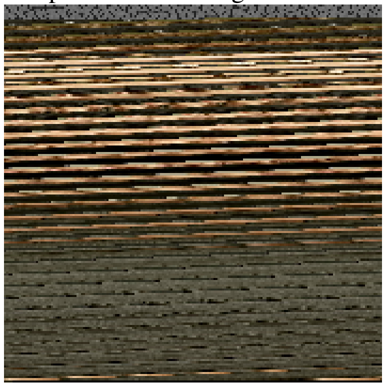
***Abstract*— The concepts of image processing used in this task have wide applications in the real world.Firstly, noise reduction is important considering the fact that most of the data obtained from sensors will be filled with noise due to environment and instrument errors. Various path finding algorithms play an important role in the traversal of the drone in real world scenarios. Conceiving real world obstacles and paths as a maze, algorithms like DFS, A*, Dijkstra,etc can be implemented to find the most efficient path.**

I. INTRODUCTION

The goal was to reach a password to a protected zip file and extract the treasure from it. The intermediate clues were to be obtained by using concepts from OpenCV and image processing. For the 1st level I used basic averaging to get the grayscale image. In the 2nd level to find the position of smaller image I used bitwise XOR to overlap 2 images and check whether they are the same. For noise reduction in level 3 I used the data obtained from the histograms of each color channel.

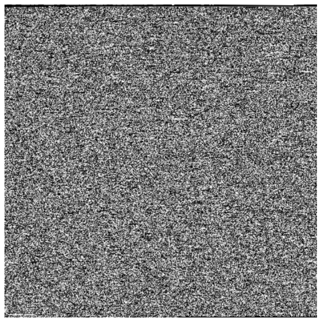
II. PROBLEM STATEMENT

As mentioned earlier, the problem statement was a puzzle with three levels where on completion of each level there were clues to the next level. For the 1st level an RGB image(shown below) was given which was to be converted into grayscale and get some instructions by decoding the initial pixel values using ASCII code.



The next level contained a bigger image and a smaller image formed by rearranging the rgb pixels of the level1 image was to be found in the bigger image. Its x co-ordinate gave the color of a monochrome picture of a maze which was the third level. Using this color value and noise reduction techniques, a clear picture of the maze was to be extracted from a given noisy image(shown below)

After solving this maze by using any maze solving algorithms, a password could be seen in the path. This was the password to a zip file containing another image, made of black and white pixels.(shown below)



The matrix formed by its pixel values was to be used and converted into an mp3 file. Thus, coming to the end of the puzzle.

III. INITIAL ATTEMPTS

For the first level, the initial attempts were successful in converting the image into grayscale by averaging the RGB pixel intensities as:

$$grayImg[i][j] = (img[i][j][0]+img[i][j][1]+img[i][j][2])/3$$

And using the ASCII code, a paragraph regarding the further instructions was printed out which indicated to form a (200,150,3) image by using the pixel values in the given image. While forming the given image my initial code was miscounting the pixels thus giving a rather odd image to find in the larger image.(as shown below)

But on correcting the increment of the matrix indices, I arrived at the correct image. (A picture of Mark Zuckerberg). For the next part of the puzzle, my few initial attempts failed at precisely identifying the co-ordinates of the cropped picture from the bigger image. But soon, I figured out a way using Bitwise XOR to compare two images and check whether they were same or not.

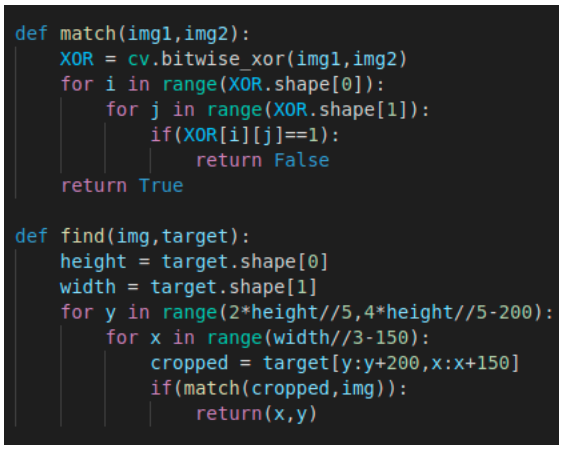
Now, for the noise reduction part,I tried using some blurring techniques to see if the noise could be reduced. But none of them seemed to work. But then having a look at the histograms of the individual color channels['b','g','r'] I figured out the colors causing the noise and their respective intensities. A simple subtraction of that intensity value from each pixel worked and gave a much better result.

To solve the maze, I struggled with implementing DFS initially as the image of the maze was a large one and considering each pixel as one block in a grid was too tedious of an approach. But later a simple grouping of 3x3 pixels into one cell worked and gave the result.

IV. Final Approach

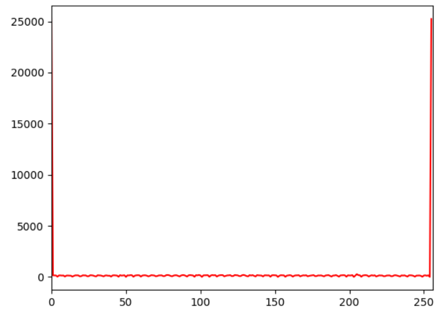
I used averaging to get the grayscale values from the first image and then simply printed out the characters corresponding to the ASCII value taken from the pixel value. After some trial and error, I figured out the exact amount of pixels used for the text and then rearranged the following pixels to get the following image as a result for level 1.

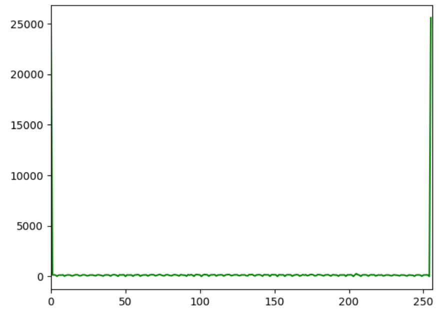
I used the following code to find the position from where this image was taken and got the coordinates as (230,460)

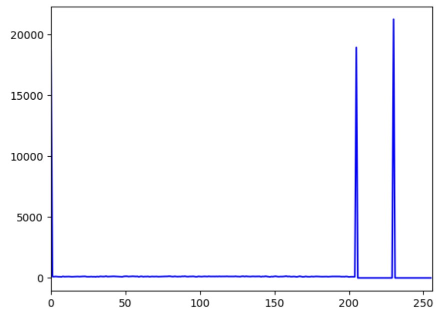


```
def match(img1,img2):
    XOR = cv.bitwise_xor(img1,img2)
    for i in range(XOR.shape[0]):
        for j in range(XOR.shape[1]):
            if(XOR[i][j]==1):
                return False
    return True

def find(img,target):
    height = target.shape[0]
    width = target.shape[1]
    for y in range(2*height//5,4*height//5-200):
        for x in range(width//3-150):
            cropped = target[y:y+200,x:x+150]
            if(match(cropped,img)):
                return(x,y)
```

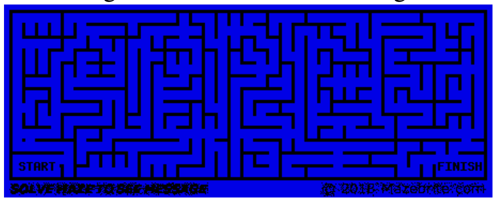
For denoising the image of the maze, I used the histograms to get the intensities which showed spikes meaning that those were responsible for the noise in the given monochrome image. Following you can see the histograms of the three color channels.



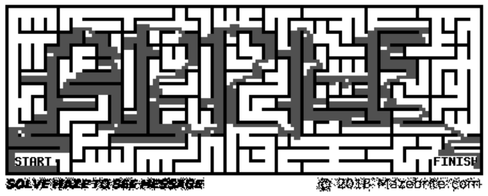




As you can see the spikes in red and green channels around the high intensity values were mostly responsible for the noise. After subtracting these intensities from the R and G channels of each pixel a blue monochrome image with lesser noise was obtained. Then binarising that image gave a much cleaner image which I used for solving the maze.



For solving the maze, I converted the image into a black and white binary image for easier handling of pixel values. I implemented DFS to solve the maze. It didn't give the shortest path, but gave one of the many paths possible. I explored other algorithms like A* and Dijkstra but couldn't find the time to implement those. The result I achieved from DFS is as follows.



The path mapped out read APPLE which was the password to the zip file which gave another b/w image. I used the python library ”pydub” for converting the image matrix into an mp3 file. As an mp3 file stores data in the form of a matrix of bytes, converting the image matrix into bytes and passing other parameters like frameRate and bitRate, the mp3 file was generated. The audio clip turned out to be the music of a RickRoll!

V. Results and Observation

For the maze solving part many better algorithms could be used as DFS is a very primary maze solving algorithm and doesn't give the shortest path but the first found possible path. A* visits less number of nodes than DFS as it uses a heuristic function to get the priority of a node while DFS randomly goes to a neighbour. But in cases where heuristics may fail or get complex, DFS or BFS can be a faster option.

VI. Future Work

The A* algorithms should be implemented while tackling this problem in a real world scenario as finding the shortest path can prove vital in terms of real life applications.

Conclusion

Overall the problem was quite challenging and a very interesting one. I faced a few difficulties with some applications of image processing in the intermediate steps, but a little thinking over the problem gave solutions. The concepts utilised in this problem are of vital use in the real life problems ARK tackles. Majorly, noise reduction and path finding algorithms are of heavy use.

References

[1] https://medium.com/image-vision/noise-in-digital-image-processing-55357c9fab71
[2] https://medium.com/image-vision/noise-filtering-in-digital-image-processing-d12b5266847c
[3] https://www.geeksforgeeks.org/depth-first-search-or-dfs-for-a-graph/
[4] https://www.youtube.com/watch?v=oXlwWbU8l2o

Aerial Robotics Kharagpur Task Documentation (Task 4.2 Part 1)

Ashutosh Ghuge, Roll No. 20CH10019

***Abstract*— One of the most popular and important algorithms when it comes to game theory is the MiniMax algorithm. This algorithm is basically a backtracking algorithm which searches through all possibilites of the game after the computer makes its move and returns the best move. It is applicable for two player games where every player takes alternate turns.**

I. INTRODUCTION

The problem statement for the Part 1 of this task was to build a 2D tic-tac-toe game, where the minimax algorithm is to be implemented.

II. PROBLEM STATEMENT

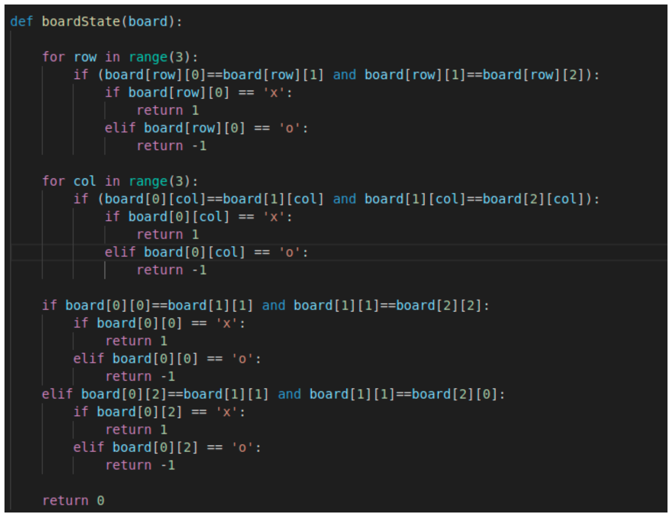
Almost any two-player game where each player makes a move alternately can be made by implementation of the minimax algorithm. A game like tic tac toe is a very basic application of the minimax algorithm. In a 2D tic tac toe game, the winning conditions are few and thus the sample space of the move tree is limited and hence the computer can pretty much win in every game. This idea once applied on a simple game, can be extended to other advanced games like chess, where the moves are infinitely many and hence it is difficult to apply a straightforward minimax algorithm.

III. INITIAL ATTEMPTS

I made a simple game environment initially, where a simple 3x3 matrix was used as the board. I tried applying the minimax algorithm but I made some errors in the recursive part of the algorithm and returning the best move. I got a properly working program the first time but it was a bit messy.

IV. FINAL APPROACH

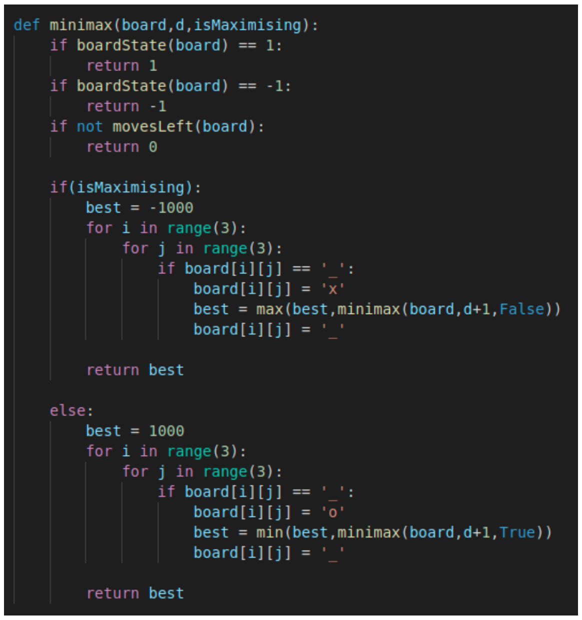
In the final approach I better applied the same algorithm in a more systematic manner. First there was need of a end state function which checks the board and returns a value based on whether someone has won or there is a tie. The number of end cases are very few and could be checked easily in the function, which is as follows.



```python
def boardState(board):

    for row in range(3):
        if (board[row][0]==board[row][1] and board[row][1]==board[row][2]):
            if board[row][0] == 'x':
                return 1
            elif board[row][0] == 'o':
                return -1

    for col in range(3):
        if (board[0][col]==board[1][col] and board[1][col]==board[2][col]):
            if board[0][col] == 'x':
                return 1
            elif board[0][col] == 'o':
                return -1

    if board[0][0]==board[1][1] and board[1][1]==board[2][2]:
        if board[0][0] == 'x':
            return 1
        elif board[0][0] == 'o':
            return -1
    elif board[0][2]==board[1][1] and board[1][1]==board[2][0]:
        if board[0][2] == 'x':
            return 1
        elif board[0][2] == 'o':
            return -1

    return 0
```

This function returns 0 if there is a tie, 1 if the computer wins and -1 if the player wins.

The next part was to implement the minimax algorithm. The algorithm takes the board as input in its current state and loops through every possible move of the computer and searches the further possibilites for every move returning the first move found which ends in a win or a tie (worst case). Thus it creates a branched tree and goes down a every branch until an end point is reached.(similar to dfs). One very important part in the algorithm is the determination of the maximising and minimising agents. The computer is the maximising agent and the opponent is the minimising agent. If it is the maximising agents turn it searches for the branch having the highest possible value(i.e. a win) while if it is the minimising agents turn it searches for the branch having lowest value(i.e. a loss which implies a win for the maximising agent). The implementation of the algorithm is done as follows.



```python
def minimax(board,d,isMaximising):
    if boardState(board) == 1:
        return 1
    if boardState(board) == -1:
        return -1
    if not movesLeft(board):
        return 0

    if(isMaximising):
        best = -1000
        for i in range(3):
            for j in range(3):
                if board[i][j] == '_':
                    board[i][j] = 'x'
                    best = max(best,minimax(board,d+1,False))
                    board[i][j] = '_'

        return best

    else:
        best = 1000
        for i in range(3):
            for j in range(3):
                if board[i][j] == '_':
                    board[i][j] = 'o'
                    best = min(best,minimax(board,d+1,True))
                    board[i][j] = '_'

        return best
```

V. Results and Observation

The minimax algorithm works easily work simple games having smaller game trees. If we look at the algorithm above, we can see that it returns the first best move it finds which in every case may not be the shortest move. To get the shortest move we can simply make use of the depth variable we pass in the algorithm. Instead of returning the best value we return the (best value)*depth and make the move having the smallest value. As the best value stays the same we prioritise the move which has less depth i.e. takes fewer steps to reach a finished state.

VI. Future Work

A 3D tic-tac-toe game and games like chess won't work that efficiently if we simply apply the minimax algorithm solely. Since these games have a much larger set of possibilities it would be very difficult and time consuming to check every possible end state as we did in 2D tic-tac-toe. But we can apply deep learning algorithms which work along with minimax in better refining the set of possibilities and searching only in the region where there is higher probability of winning. So instead of reaching the end state and returning the value all the way back to the top, the program simply checks till a limited depth further than the current depth and returns the probability of winning instead of a definite result.

Conclusion

As we can see through this example, minimax algorithm has a wide range of applications in designing AI agents to play any kind of two-player game. The deep learning models and other minor changes will vary for each game but a smart agent can be designed who has a much higher probability of winning or worst case ending the match in a tie.

References

[1] https://www.geeksforgeeks.org/minimax-algorithm-in-game-theory-set-1-introduction/

[2] https://www.geeksforgeeks.org/minimax-algorithm-in-game-theory-set-2-evaluation-function/?ref=rp

[3] https://www.geeksforgeeks.org/minimax-algorithm-in-game-theory-set-3-tic-tac-toe-ai-finding-optimal-move/?ref=rp