Trinity College Dublin
Coláiste na Tríonóide, Baile Átha Cliath
The University of Dublin

CSU44053 COMPUTER VISION ASSIGNMENT 1

Assignment 1

Submitted by:
Vanaj Kamboj
21355100

INTRODUCTION

This report aims to point out the sequential actions conducted for each part of the assignment using the knowledge gained in the CSU44053 Computer Vision modules coursework along with flowcharts where necessary. We will go over the drawbacks and the advantages of using various techniques in OpenCV and also emphasize the reasoning behind each selected strategy.

PART 1 – PIXEL CLASSIFICATION AND COMPARISON

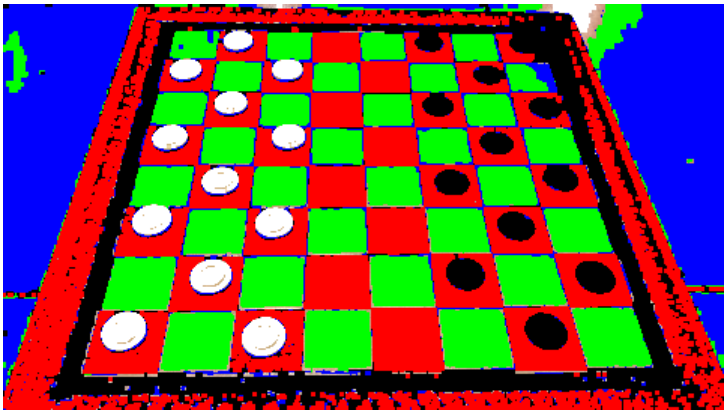
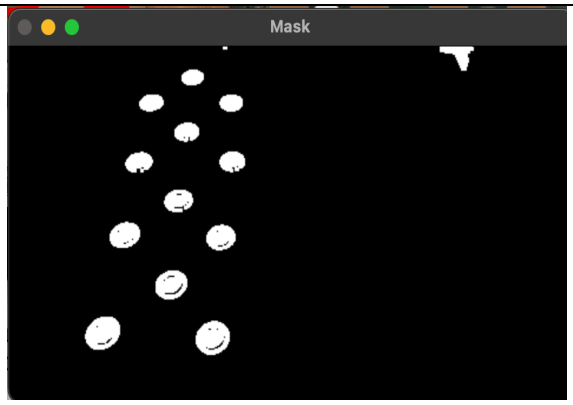
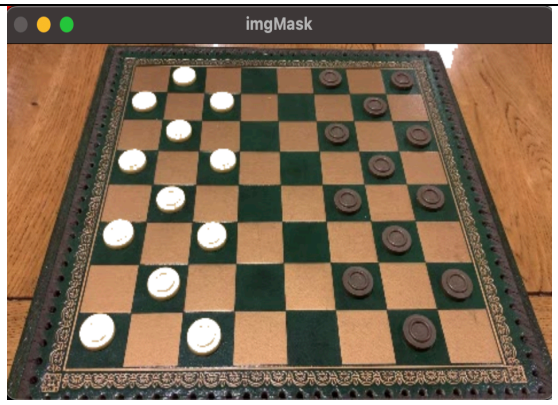


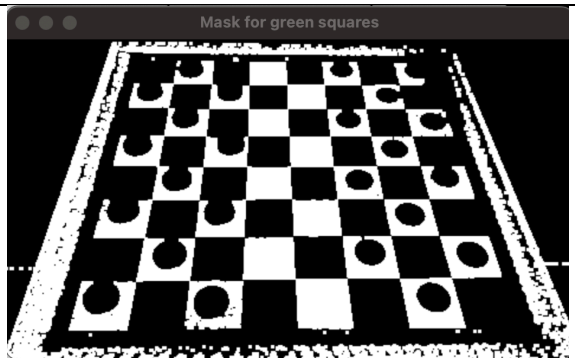
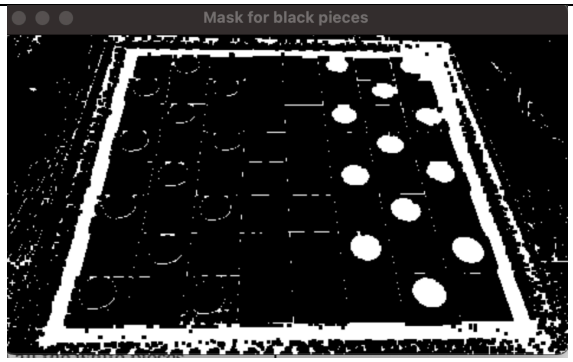
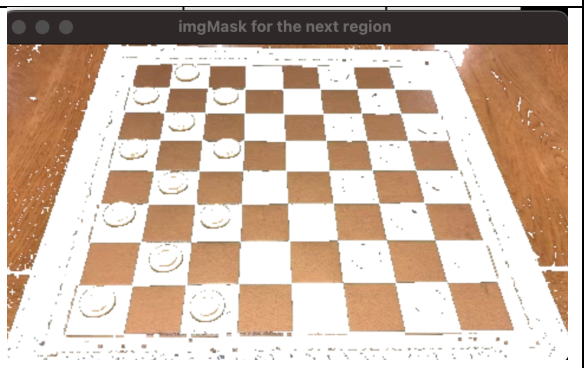
Fig 1: Output for Part 1

Figure 1 shows the final output I achieved in part 1. I managed to get this output using binary thresholding and morphological opening and closing at various stages. My approach for this part was to first, binary threshold the complete image and look for a value that creates a mask for any of the regions we wish to classify. First the threshold values for white pieces were found followed by green squares, black pieces and white squares. The table had a similar color to the white squares, and was classified by converting the image into HSV space as it was not clearly distinguished using binary thresholding.

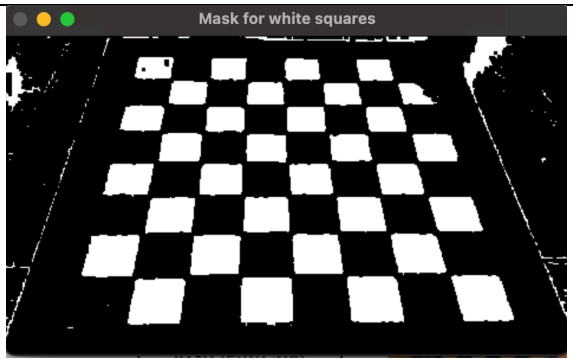
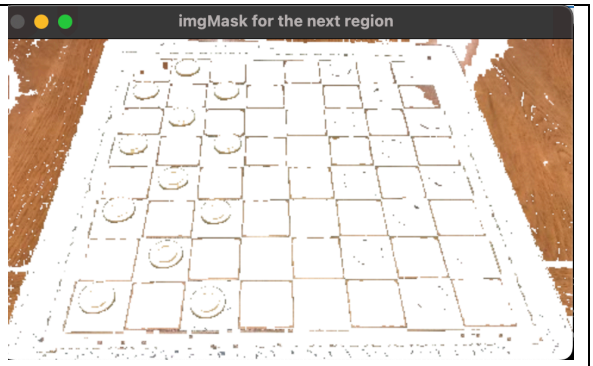
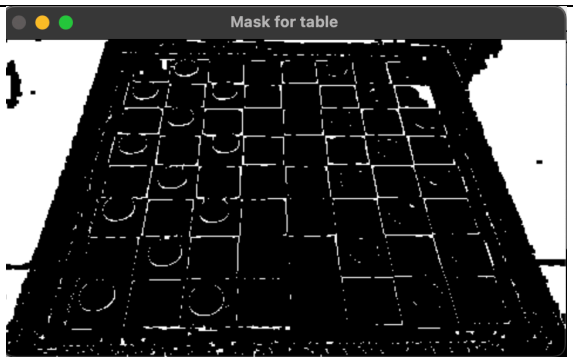
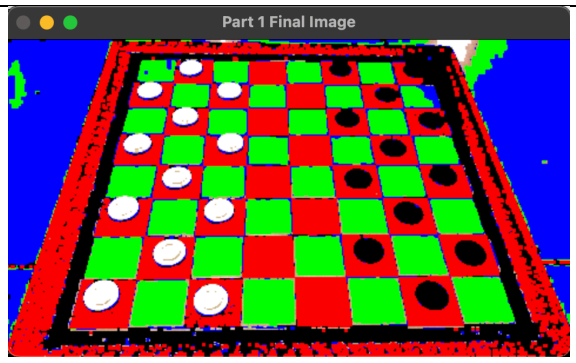
STEP BY STEP PROCESS FOR PART 1

For completing part 1 Mat img, imgGrey, mask, imgMask, imgHSV, imgFinal were used. The following table is will make us get a better understanding of the process.

1) The static image is converted to greyscale using the cvtColor() function. The grey image is then binary thresholded with a value of 186/255 along with morphological opening to get the mask for the **white pieces**. At each step the detected mask is annotated on to the **imgFinal** Mat using the **setTo()** function.		

2) Same steps are followed to detect the **green squares** by **inverse thresholding** the original grey scaled image with a value of 80/255. This is what the **mask** and **imgMask** looks like. The mask for green squares is again annotated onto the imgFinal using setTo() function.		
3) Now the step 3 is a little different. We move onto detecting the **black pieces** but instead of greyscaling the original image. We **greyscale imgMask** and **inverse threshold** it with a value of 100/255 The mask for black pieces is again annotated onto imgFinal.		

As you can see after step 3 the imgMask for next region contains white squares to be detected and the table. These two regions are very similar in greyscale and I couldn't find a threshold value to separate the two. This is why I decided convert my image to HSV space and look for values that would be good enough to detect my white squares.

4) Converting imgMask to imgHSV using cvtColor() function. Using inRange() function with a value of Scalar(10, 102, 130), Scalar(15, 135, 243) [(**h**min, **s**min, **v**min), (**h**max, **s**max, **v**max)] to create a mask for annotating white squares onto our imgFinal image.		
5) Now we can see from the previous step that our complete board has been annotated onto the imgFinal the only region left is the table which can be detected by **inverse thresholding** the greyscale imgMask by a value of 154/255		

RESULTS FOR PART 1

The scores achieved after performing Histogram Comparisons were fairly good enough because we weren't supposed to use techniques like edge detection, chessboard corner detection, and many more. So getting an output similar to the ground truth image is only possible by using binary thresholding and creating masks smartly at each step to detect another region of the image since many areas of the image are very similar. For example the white squares, white pieces, and the table itself.

There are 4 methods that I have used for histogram comparison and the results are shown below

```
Assignment part 1
Correlation--> Provided_Ground_Truth, My_Prediction : 1 / 0.386215
Chi-Square--> Provided_Ground_Truth, My_Prediction : 0 / 2.61337
Intersection--> Provided_Ground_Truth, My_Prediction : 2.26447 / 0.792051
Bhattacharyya Distance--> Provided_Ground_Truth, My_Prediction : 0 / 0.76234
```

Fig 2: Results for Part 1

PART 2 – RECORDING OCCURRENCES IN CONFUSION MATRIX

The process for completing the part 2 of the assignment can be shown in the following flow chart:

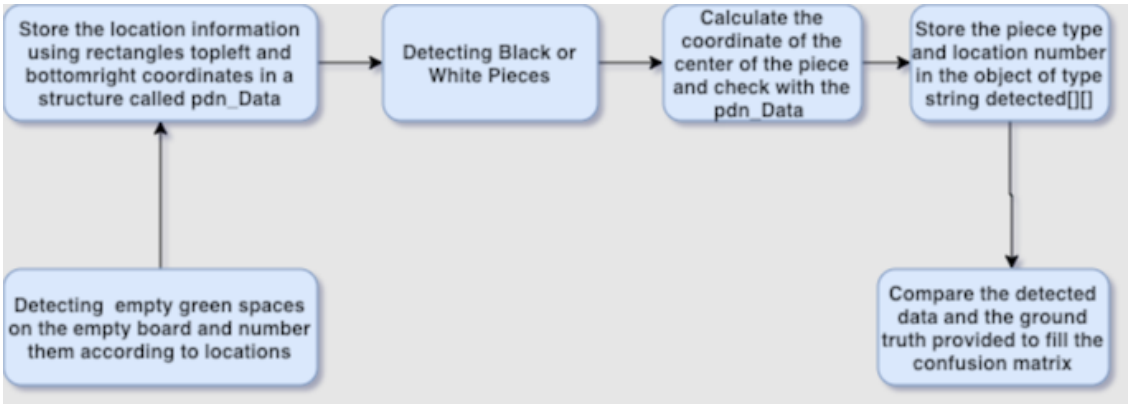


Fig 3: Flowchart for Part 2

Detecting empty white and green squares and numbering the green squares:

The first step for the part 2 of the assignment was to wrap the image according to the locations of the four corners given to us. So I made a function for this task

```
void wrapImage(Mat img){
    Point2f src[4] = { {114, 17}, {355, 20}, {53, 245}, {433, 241} };
    Point2f dst[4] = { {0.0f, 0.0f}, {w, 0.0f}, {0.0f, h}, {w, h} };
    matrix = getPerspectiveTransform(src, dst);
    warpPerspective(img, imgWrap, matrix, Point(w, h));
}
```

The next step is to convert the warped image into HSV space and look for values to detect the green spaces. I wrote a function for this task called FindHSVVals() which creates trackbars to easily detect the values required to create a mask for the region of interest.

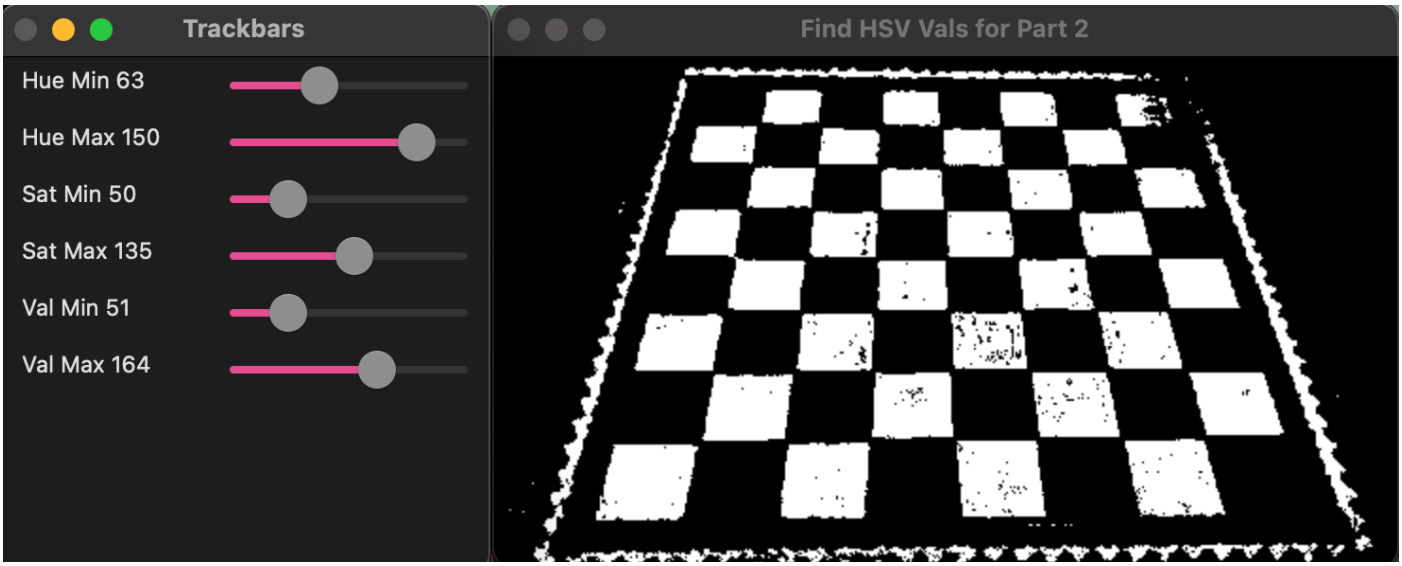


Fig 4: Use of Function FindHSVVals() to create a mask for green squares

After finding the right HSV values I used the inRange() function to create the exact mask as shown in figure 4 and pass that Mat image into the function getContours()

```
void getContours(Mat imgBinary, Mat img, string category)
```

This function takes in the mask we created as Mat imgBinary, the image to display the detected region on as Mat img, and string category is a variable I am passing on to check whether detection for green square is to be performed or white squares.

```
vector<vector<Point>> contours;
Point center;
vector<Vec4i> hierarchy;
int squares = 0;
findContours(imgBinary, contours, RETR_TREE, CHAIN_APPROX_NONE);
```

Here I have created the required objects for finding the contours (green squares) using OpenCVs function findContours(). I had to do a little bit of fine tuning as well in terms of the contours that were being detected because of area of the contours and also whether there is a piece inside that square in the future. As we do not want to detect a piece in a region classified as empty square. This was done by the following lines of code below

```
for ( int i = 0; i < contours.size(); i++ )
    {
        // For fine-tuning the image we are going to bound the counters to rectangles
        Moments M = moments(contours[i]);
        Point center(M.m10/M.m00, M.m01/M.m00);
        int inContour;
        if (contourArea(contours[i]) > 500)
        {
        Rect boundRect = boundingRect(contours[i]);
            if(boundRect.area()>500 && (boundRect.width < 70 || boundRect.height < 70))
            {
```

To check whether there is a piece inside the square I globally call a function using HoughCircles to check whether there is a piece inside a square. This was just a precautionary measure as I kept testing images and

sometimes a square was detected with a piece inside it and that really wasn't my goal. The following code took care of that.

```
// This part is to detect whether there is a piece inside the rectangle.. If so then please do
not detect this as an empty square
                // The data inside circles is globally created every time the program is run
using HoughCircles
                int struct_no = 0;
                for (int j = 0; j < circles.size(); j++)
                {
                    Vec3i c = circles[j];
                    Point center = Point(c[0],c[1]);
                    inContour = pointPolygonTest(contours[i], center, false);
                    if (inContour == 1 || inContour == 0){goto noprint;}
                }
```

No print is a goto label I have created that just skips the part of printing onto main image. This was the way I fine-tuned my detection system.

```
rectangle(img, boundRect.tl(), boundRect.br(), Scalar(255 ,0 ,0), 3);
                    if (category == "W_sq")
                    putText(img, "W", (center), FONT_HERSHEY_COMPLEX, 0.6,Scalar(20, 208, 14), 1);
                    else
                    {
                        if (category == "Empty")
                        {
                            putText(img, numbers[squares], (center), FONT_HERSHEY_COMPLEX, 0.6,Scalar(0, 0, 240), 1);
                            PDN_Data[stoi(numbers[squares])-1].tl = boundRect.tl();
                            PDN_Data[stoi(numbers[squares])-1].br = boundRect.br();
                            PDN_Data[stoi(numbers[squares])-1].sq_no = numbers[squares];
                            struct_no++;
                        }
                        else
                            putText(img, "B", (center), FONT_HERSHEY_COMPLEX, 0.6,Scalar(0, 0, 240), 1);
                    }
                noprint:;
            }
            squares++;
```

This step was just to print whether the squares are "B" or "W" onto the image.

The **Category = "Empty"** is the main part of the code that stores the location numbers into my pdn_Data structure. As we can see it holds the square number, top left, and bottom right coordinates of the green square. Later the pdn_Data is going to be used during our piece detection system for storage of detected_data[][].

Here is the output for detecting black and white empty squares along with the numbering of empty board.

Fig 5: Regions of Empty Black and White Squares along with empty green square locations

As of now we can differentiate between black and white squares. We also have locations and square numbers of green squares. All that is left to do is detect black and white pieces and store the data into detected[][] by checking the square number using pdn_Data we just created.

Detecting white and black pieces and storing their locations in PDN format:

The next step is to detect white and black pieces. My approach for detecting the pieces was similar to how I detected the green and white squares. I converted the warped image into HSV and found the appropriate values for the pieces using FindHSVVals() function and after creating a mask for the pieces I called the function getPieces().

```
getPieces(Mat imgBinary, Mat img, string category, string path, int itr_no)
```

This function takes the Mat mask of the pieces, Mat background to display the detected pieces, category of the mask (either black or white piece), path that is currently being processed (to store in detected PDN format), and iteration number to keep track of which move is currently active.

```
vector<vector<Point>> contours;
    vector<Vec4i> hierarchy;
    int index;
    findContours(imgBinary, contours, RETR_TREE, CHAIN_APPROX_NONE);
    vector<vector<Point> > contours_poly( contours.size() );
    vector<Point2f>centers( contours.size() );
    vector<float>radius( contours.size() );
    for( size_t i = 0; i < contours.size(); i++ )
    {
        approxPolyDP( contours[i], contours_poly[i], 3, true );
        minEnclosingCircle( contours_poly[i], centers[i], radius[i] );
    }
```

Inside the getPieces() function we find the contours and iterate for every contour to find the apporxPolyDP(). Then we use the minEnclosingCircle() function to store the centers and radius.

```
for ( int i = 0; i < contours.size(); i++ ){
        Moments M = moments(contours[i]);
        Point center(M.m10/M.m00, M.m01/M.m00);
        if (contourArea(contours[i]) > 310){
            if (category == "W")
```

```
            {
                circle(img, centers[i], (int)radius[i], Scalar(255, 0, 255), 2 );
                putText(img, "Wp", (center), FONT_HERSHEY_COMPLEX, 0.4,Scalar(0, 0, 0), 1);
                index = check_piece(center);
                if (index != -1){
                    detected[itr_no][0] = path;
                    detected[itr_no][1] += to_string(index+1);
                    detected[itr_no][1] += ",";
                }
            }
```

Here we are fine tuning the results a bit using area. This code snippet shows how white pieces are detected as the category here is labelled “W”. If these conditions pass then we move onto drawing the circle and displaying the text onto the image.

```
index = check_piece(center);
```

This function shown in the above code checks if this piece belongs in any of the squares in the pdn_Data structure that we created while numbering the green squares. If it does exist then that data is appended to detected[][] database;

Here is the code for checking whether the piece exists in a square: It is a simple logic to check whether the center of the piece is located inside the coordinates of the square.

```
int check_piece(Point center){
    for( int i = 0 ; i < 32 ; i++ )
    {
        if( int(PDN_Data[i].tl.x) < int(center.x) && int(center.x) < int(PDN_Data[i].br.x) &&
int(PDN_Data[i].br.y) > int(center.y) && int(center.y) > int(PDN_Data[i].tl.y) ){
            return i;
        }
    }
    return -1;
}
```

Finally if all goes well. We get the following output:

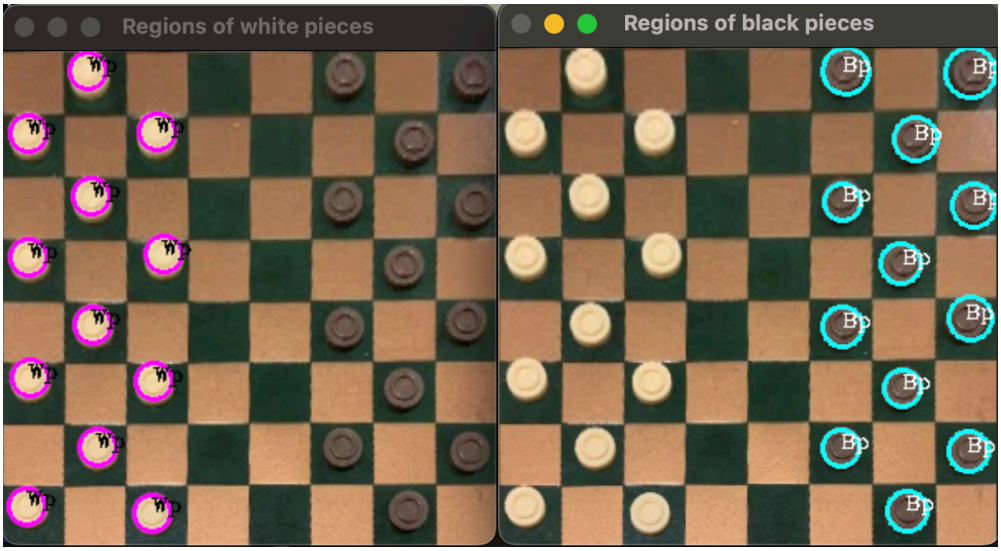


Fig 6: Detected white and black pieces

Here is the data stored in the PDN format for the detected values:

```
DraughtsGames1Moves/DraughtsGame1Move0.jpg 12,4,8,11,3,7,2,10,6,1,9,5, 28,32,24,27,23,31,26,30,22,25,29,21,
DraughtsGames1Moves/DraughtsGame1Move1.jpg 12,4,8,11,3,7,2,10,6,1,13,5, 28,32,24,27,23,31,26,30,22,25,29,21,
DraughtsGames1Moves/DraughtsGame1Move2.jpg 12,4,8,11,3,7,2,10,6,1,13,5, 20,28,32,27,23,31,26,30,22,25,29,21,
DraughtsGames1Moves/DraughtsGame1Move3.jpg 12,4,8,11,3,7,2,10,9,1,13,5, 20,28,32,27,23,31,26,30,22,25,29,21,
DraughtsGames1Moves/DraughtsGame1Move4.jpg 12,4,8,11,3,7,2,10,9,1,13,5, 20,28,32,27,23,31,26,30,17,25,29,21,
DraughtsGames1Moves/DraughtsGame1Move5.jpg 12,4,8,11,3,7,2,10,22,9,1,5, 20,28,32,27,23,31,26,30,25,21,29,
DraughtsGames1Moves/DraughtsGame1Move6.jpg 12,4,8,11,3,7,2,10,9,1,5, 20,28,32,27,23,31,30,17,25,29,21,
DraughtsGames1Moves/DraughtsGame1Move7.jpg 12,4,8,11,3,7,2,10,1,13,5, 20,28,32,27,23,31,30,17,25,29,21,
DraughtsGames1Moves/DraughtsGame1Move8.jpg 12,4,8,11,3,7,2,10,1,13,5, 20,28,32,27,23,31,26,17,25,29,21,
DraughtsGames1Moves/DraughtsGame1Move9.jpg 12,4,8,11,3,7,2,10,22,1,5, 20,28,32,27,23,31,26,25,29,21,
DraughtsGames1Moves/DraughtsGame1Move10.jpg 12,4,8,11,3,7,2,10,1,5, 20,28,32,27,23,31,26,18,29,21,
DraughtsGames1Moves/DraughtsGame1Move11.jpg 4,16,8,11,3,7,2,10,1,5, 20,28,32,27,31,26,29,21,
```

Evaluating the confusion matrix:

This is the final section of part 2 of the assignment. Here I just created a simple pdn_parser that takes data one by one as a unit of a vector and checks if that number exists inside the provided ground truth for the assignment using the compute_matrix() function. The final score I got for the assignment was not that bad considering I had to do a lot of do-overs for making my system more robust to detect pieces.

```
Part 2 - Confusion Matrix
        1269    0       8
        0       411     0
        0       0       520
```

Fig 7: Confusion Matrix for Part 2

PART 3 – PROCESS THE GAME VIDEO AND RECORD MOVES

The part 3 of the assignment requires us to process the draughts game video frame by frame to select appropriate frames that do not include a person's hand and record the moves made in a PDN format.

Finding perfect frames:

The first thing I decided to do was to process the video frame by frame and look for ways to distinguish between the frames with hands and without hands. The main function that comes into play with this task is the createBackgroundSubtractorMOG2(). With the help of this function I was able to create a foreground mask to count the number of white pixels in an image. If the number of white pixel suddenly increases in a frame, that means there is a hand in the frame. To complete this part of the assignment I made a function called findFrames():

```
void findFrames()
{
    string video = "DraughtsGame1.avi";
    VideoCapture cap(video);
    Mat frame, videoMask, difference;
    Ptr<BackgroundSubtractor> pBackSub = createBackgroundSubtractorMOG2();
```

```
unsigned long counter = 0;
int saveCount = 21;
int framecount = 0;
int flag = 0;
Mat previous;
while(cap.read(frame))
{
    string framename = "part3/DraughtsGame1Move";
    if (counter == 0)
    {
        frame.copyTo(videoMask);
        counter++;
        continue;
    }
    pBackSub->apply(frame, videoMask);
    saveCount++;
    int TotalNumberOfPixels = videoMask.rows * videoMask.cols;
    int ZeroPixels = TotalNumberOfPixels - countNonZero(videoMask);
    if (ZeroPixels > 126670)
    {
        if (framecount != 0)
        {
            Mat scoreImg,framewrap, previouswrap;
            double maxScore;
            Point2f source[4] = { {114, 17}, {355, 20}, {53, 245}, {433, 241} };
            Point2f destination[4] = { {0.0f, 0.0f}, {w, 0.0f}, {0.0f, h}, {w, h} };
            matrix = getPerspectiveTransform(source, destination);
            warpPerspective(previous, previouswrap, matrix, Point(w, h));
            warpPerspective(frame, framewrap, matrix, Point(w, h));

            matchTemplate(previouswrap, framewrap, scoreImg, TM_CCOEFF_NORMED);
            minMaxLoc(scoreImg, 0, &maxScore);
            if(maxScore > 0.993)
                continue;
        }
        framename += to_string(framecount);
        framename += ".jpg";
        framecount++;
        saveCount = 0;
        frame.copyTo(previous);
        imshow("Frame", frame);
        imwrite(framename, frame);
    }
```

Another challenge I was able to overcome using this function was the task to handle duplicate frames. Initially I was storing more than **200 frames** in my **part3/** subdirectory. Then I made my code more efficient by using template matching between the current frame and the previous frame. If were different that is only when I used to store them. This was done using the matchTemplate() function of OpenCV and minMaxLoc() function which returns the matching score. With the help of this I was able to store the exact amount of frames as our professor provided in the TIPS media folder. The next step was to call my same assignPart2() function to go through all the frames in the **part3/** sub directory and store the moves in detected[][].

Recording the moves made:

After I had got my new detected[][] data for part 3. I just had to write a simple logic similar to the one we had for confusion matrix but this time it was to check for changes made between moves and store it in a pdn format. This is the result I got for part 3 of the assignment with just a few errors in between.

```
Part 3
 W:9-13   B:24-20 W:6-9   B:22-17   W:22-13       B:17-18 W:22-16   B:18-14   W:6-9  W:17-15 B:22-18 W:15-6 B:18-2 W:9-18
B:26-22 W:18-9   B:22-18 W:11-15    W:9-11    W:3-7   B:20-11     W:6-9  W:9-14 B:27-10 W:14-17   B:7-3 W:6-9     W:9-19
   W:15-18  W:11-15 B:7-10 W:18-22   B:10-14  B:3-7 W:15-18 B:14-17  B:7-10 W:21-25   B:17-26 W:17-21  W:25-30 B:14-17 W:3
0-23 B:26-27  B:27-26 W:25-23   B:31-15   W:23-26 B:27-18 W:19-23    W:23-31   B:15-27 W:26-24   B:32-27 W:31-9 B:27-19
       B:25-22 W:14-18   B:22-15 W:4-8   B:11-4
```

Ground Truth – "1. 9-13 24-20 2. 6-9 22-17 3. 13-22 26-17 4. 9-13 30-26 5. 13-22 25-18 6. 12-16 18-14 7. 10-17 21-14 8. 2-6 26-22 9. 6-9 22-18 10. 11-15 18-2 11. 9-18 23-14 12. 3-7 20-11 13. 7-16 2-7 14. 8-11 27-24 15. 1-6 7-2 16. 6-9 14-10 17. 9-14 10-7 18. 14-17 7-3 19. 11-15 24-20 20. 16-19 3-7 21. 15-18 7-10 22. 18-22 10-14 23. 17-21 14-17 24. 21-25 17-26 25. 25-30 31-27 26. 30-23 27-18 27. 19-23 18-15 28. 23-26 15-11 29. 26-31 32-27 30. 31-24 28-19 31. 5-9 29-25 32. 9-14 25-22 33. 14-18 22-15 34. 4-8 11-4"

Fig 8: Result for Part 3

PART 4 – DETERMINE FOUR CORNERS OF THE BOARD

Using Hough Lines:

For this part of the Assignment I chose the static image for move 0 and converted it to greyscale. After that I Gaussian Blurred the image and used Canny() function to get sharp edges. I used the HoughlinesP() to calculate all the possible lines that exist inside the image and fine-tuned it using a simple logic.

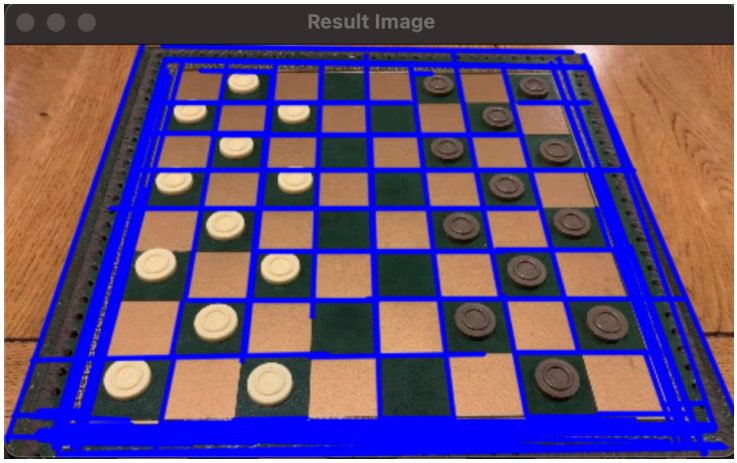


Fig 9: HoughLines for Detecting Corners of the Board

As I can see from the static image that the lines for the top left corner must exist in a space between (30,215) and (100,251) [These are any approximate values]. I can then just display these lines as opposed to the ones that don't lie in that area. I can perform a similar logic for bottom right. This is the following output.

Fig 10: After fine tuning the HoughLines image using approximation

Drawbacks of using this method:

The main drawback of using this method is that I already have information about the image as it is a static image that I am processing. If the camera angles were a little different then this method would fall short of evaluating the four corners of the board.

Using Contours:

Now we move onto evaluating the board using contours. For this method we convert the static image to greyscale, blur the image, detect the edges using canny edge detection, and dilate the image. Finally we find the contours using the findContours() function to get the following output.

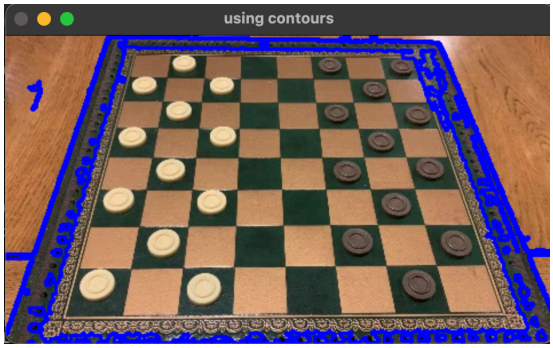


Fig 11: Output for part 4 using contours

We can fine-tune this image further on the bases of area. Then we get the following output:

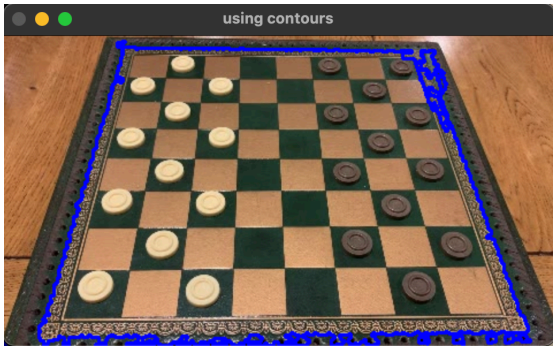


Fig 12: Output after fine-tuning the image

Drawbacks:

The main drawback of this method is that the detection can go bad if the lighting conditions change or we place a piece in the far corners where there is a lot of data to be processed. For example the top right corner.

PART 5 – CODE TO DISTINGUISH BETWEEN KING AND NORMAL MEN

For part 5 of the assignment I wrote the code for detecting the pieces that reach to the opponents edge and keep track of those pieces in the detected[][] database. Every time a piece turns into a king, it is tracked throughout all its moves and calculated into a new extended confusion matrix, the logic for which is written into the function compute_ext_matrix().

This is the result I achieved after the implementation which isn't perfect but was the only way I could detect the kings as the detection of kings was very hard using the HSV space and contour detection.

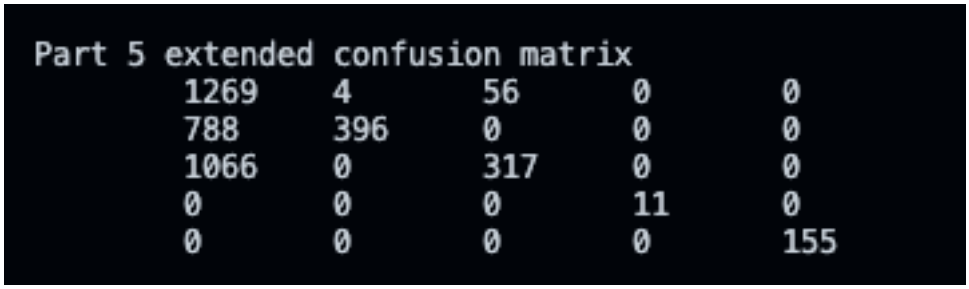


Fig 13: Result for part 5

CONCLUSION

From assignment 1 of the module CSU44053. I gained knowledge about the fundamentals of computer vision using OpenCV. This assignment not only helped me make my understanding for the subject more robust but also made me understand how to implement the knowledge gained in my class in code.

Across all the parts of this assignment I have tried various implementations for the same goal. For example, during part 2 I tried to detect pieces using binary thresholding and hough circles but halfway through my testing I realised that it wasn't a robust way to do the task and switched to using HSV space with contours as contours allows me to find-tune the detection process further.