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

CSU44053 Computer Vision
Assignment 1 - Draughts

Ted Johnson, TCD 19335618
8th of November, 2022

Apologies for the delayed assignment submission.
I hope I have not caused any inconvenience.

0 Introduction

This project explores a few techniques in computer vision within the context of analysing the video recording of a game of draughts. While some of my solutions for parts of the assignment build on top of previous parts, I will be presenting and discussing each part individually with only references to others where appropriate.

The source code of the project is split across several files as it was too unwieldy to place within just a single file. The required myApplication function, a main function to run the application, parts 1 through 5 and some common utilities are placed in their corresponding files. I chose to use CMake to handle the build process. As such, the following commands can be used to build and run the application:

```
cmake -S . -B ./build
cmake --build ./build
./build/draughts
```

There is a minimal UI for running and evaluating each part of the assignment. You can select menu items using your keyboard. When running the program, all resources are expected to be inside a directory `./res` - The contents of this directory has been provided along with the rest of the project within a single ZIP archive file for your convenience. Finally, all work presented here is my own and this project was completed individually.

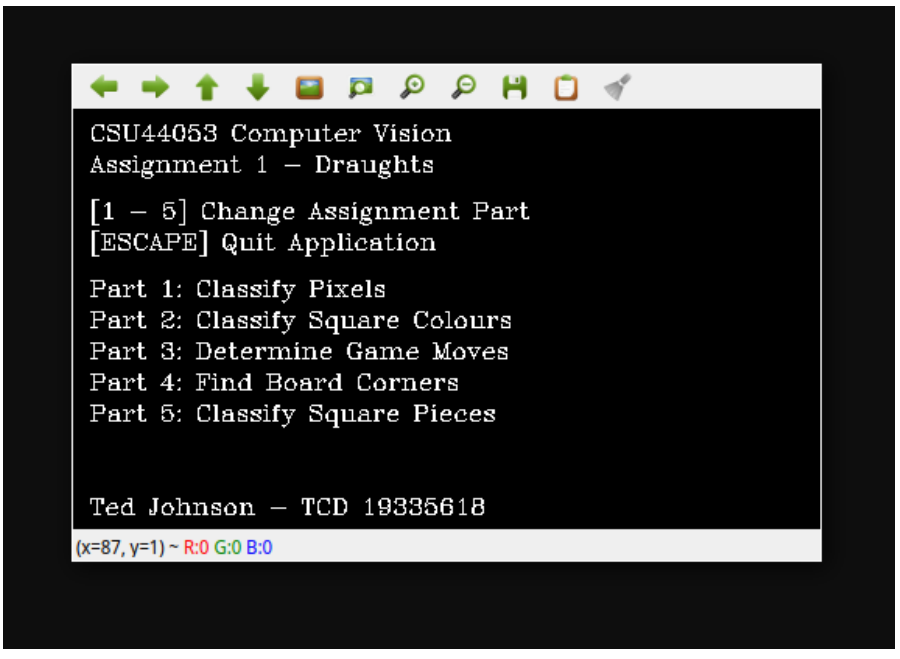

Figure 0.0 - The main menu of the application

1 Part One - Back Projection

1.1 Overview of Solution

To classify all pixels in an image, I chose to use histogram back projection. By taking the histogram of some region of interest, it is possible to label every pixel in the input image as having some probability of belonging to the region of interest. In this case, I was able to take the histogram of four sample images provided: Black pieces, white pieces, black squares and white squares. Then, for the image being classified, we perform a back projection using each of the sample image histograms. Finally, we classify each pixel in the image as belonging to the class with the highest back projection probability.

More recently, I came back to this assignment part and attempted to improve my solution. I discovered that dramatically limiting the number of histogram bins . In fact, by only allocating four bins I was able to achieve my highest performance. I believe this is due to the system trying to differentiate exactly four classes - As such, when taking the histogram of the image, the four different classes present within the image most strongly influence just one of the bins which may lead to a much clearer result.

Additionally, I use some morphology to close the resulting image. This was able to clean up the result greatly by removing minority pixels classified differently to the majority pixels around it - Effectively removing ‘noise’ in the classification image.

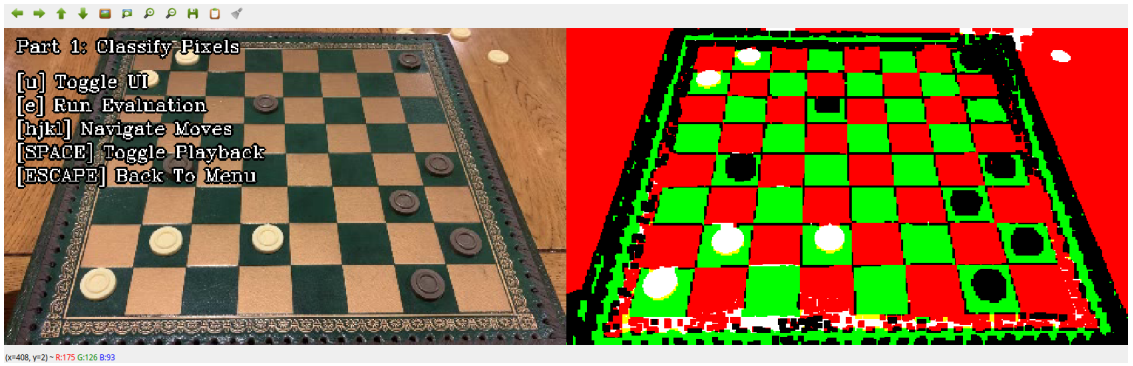

Fig 1.1 - Output of part 1

In the classification image, I have labelled white and black pieces as white and black respectively. White tiles are labelled green and black tiles are labelled red. Notice that my solution does not even attempt to label the table, hands or anything else outside of the game.

1.2 Performance

While I can visually compare the output of my program to my human understanding of the video as a sanity check, we need to be able to systematically evaluate its performance. Otherwise, without such measurement, it becomes very difficult to find problems and iteratively improve on a solution.

For this part, it's possible to use an image already labelled with the ground truth and compare each pixel in my output with the pixels of the ground truth image - That is, the absolute difference between the two images. We can then measure how many pixels my solution got right. However, this isn't very scalable as each ground truth image has to be carefully labelled. For a real-world system, we would want to be able to process many such images to be confident our solution performs correctly, which would be very time-consuming. Thankfully, Dr. Dawson-Howe has provided a hand-labelled image of the initial state of the game.

Fig 1.2 - Output of part 1 evaluation

In the image above, a black pixel is a pixel which was correctly classified. As you can see, my solution did not do a fantastic job. It was only able to classify 45.6% of the pixels correctly.

1.3 Potential Improvements

Classifying every pixel in the image is hard, especially in the general case. Without context from previous and future frames or more information on the features of what you should be classifying, I believe it's impossible to get a satisfactory result with any certainty. Clearly, in my own implementation, many of the pixels outside of the game board are completely miss-identified due to not even trying to identify them. It would be possible to designate a threshold where any pixel whose probability for all four back projections is low enough could be considered a "table" pixel. This would likely improve my evaluation score but may result in more noisy classifications.

2 Part Two - Histogram Comparison

2.1 Overview of Solution

Of all the parts of this assignment, I spent most of it on part 2. This is because several of the later parts build on top of the solution I implemented for this part. As such, I repeatedly returned to this part to redesign and improve the best I could. When developing my solution to classify all square colours on the board of an image, there were a few important observations made:

1. As the pixel corners of the board are given, it is straight-forward to use a perspective transformation to remap the image in such a way that we only consider the board itself and the pieces which sit atop it.
2. Furthermore, the perspective-fixed image could be specified in such a way that every square on the image would be the same size and distance from each other.
3. As with part 1, histograms prove very useful for comparing classes of colours. In this case, we can compare the colours of black and white pieces.
4. Each square on the board can only be in one of three states: Empty, holding a black piece or holding a white piece.

With these points in mind, I designed a system which uses a perspective matrix to transform the input image into a “top-down” perspective where each square on the board is exactly 50 pixels wide. The system then iterates over each playable square on the board and takes the histogram of each. It compares these histograms to the histograms of the black pieces and white pieces sample images provided using Bhattacharyya distance, which gives every square a probability score - Positive scores for being a black piece and negative scores for being a white piece. Finally, the system classifies the colour of each square based on its score when compared to every other square's score by using k-means. Given all these steps, the system is then able to produce a drawing of the board:

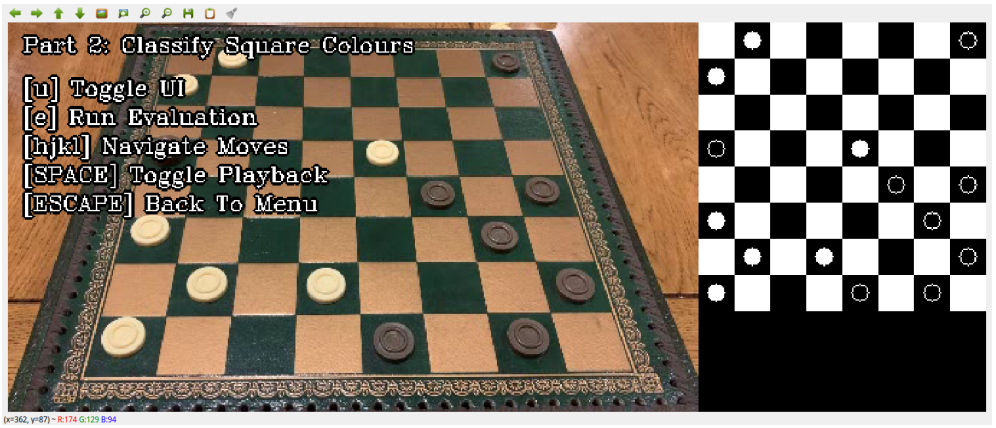

Fig 2.1.1 - Output of part 2

The application of k-means in this system is rather unusual. It is necessary to find the natural threshold between three classes. My initial approach was to consider a sorted list of the scores for a specific colour for all squares (see Fig 2.1.2 left). I would then designate the threshold for being considered that colour as wherever there was the largest gap between two sequential scores. I would do the same with the other colour. This approach actually proved quite effective.

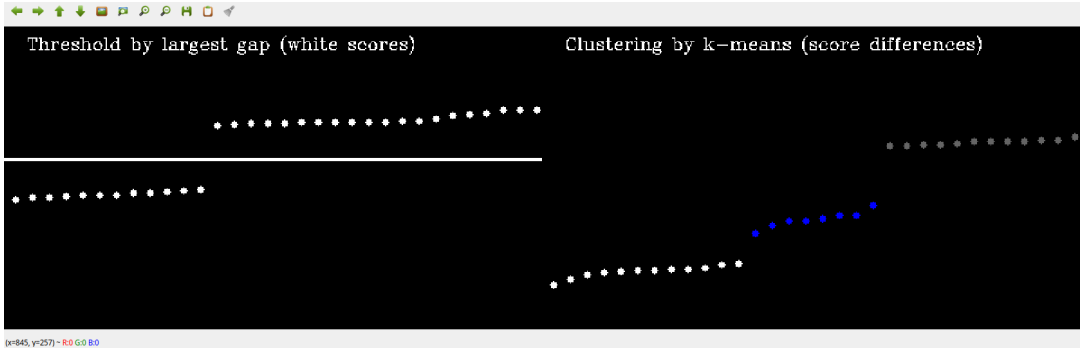

Fig 2.1.2 - Visualisation of natural thresholding methods considered

Later, I implemented my solution to instead use k-means to determine the three clusters from all the differences between scores (i.e. the score of a square is equal to its black score minus its white score). Then we just use the centres and labels computed by k-means to label every square on our board accordingly.

Notice that with this solution, we never once have to specify a threshold for a square to be classified as either black or white. Instead, by comparing the histograms of each piece colour, we can dynamically determine whether a square belongs to one colour or the other or neither. Altogether, this is an effective strategy for dealing with different lighting conditions and appearance of the pieces and the boards - As long as appropriate sample images are provide, of course

2.2 Performance

To evaluate this solution, we need images which have the colour of each piece on every square provided with them. For this assignment, we were provided with 69 images with the current PDN of the draughts game in play. Within the project, I have a function which converts the given PDN into the same representation of the board that my solution outputs. Now we can compare the ground truth of these 69 images with the results that my system outputs.

To do this comparison, we use a confusion matrix to hold the detected empty, black and white squares versus the actual empty black and white squares. The score of my solution is then the sum of the positive diagonal. This information is outputted into the terminal during the evaluation as seen below.

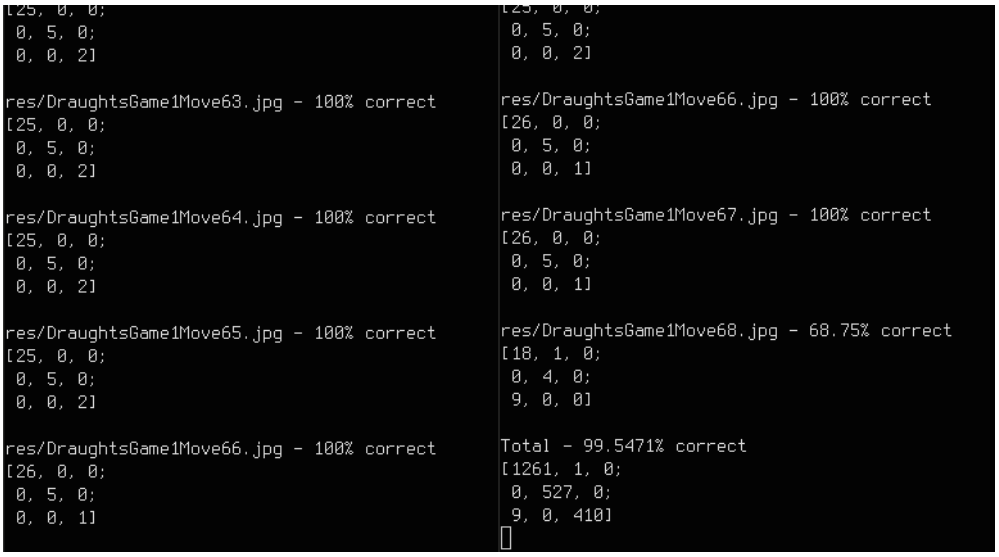

Fig 2.3 - Output of part 2 evaluation using confusion matrices

Here, we see my solution does very well, getting 99.54% of squares correctly identified. However, in the final test image where it falls apart completely. It is able to correctly label every single image otherwise.

2.3 Potential Improvements

The primary issue with this system is it completely fails once one of the pieces is missing entirely - This is apparent at the end of the video once the final move has been played and all of whites pieces have been removed. A possible solution to this would be to specify a minimum threshold for both black and white pieces that, if never exceeded once when analysing each square, would be used to determine if only one player remains. This could augment processing in such a way to only classify one colour.

Fig 2.4 - Output of part 2 during the final move

3 Part Three - Motion Detection

3.1 Overview of Solution

This system needs to process the entire video of the draughts game and determine what moves are made and when. There were a number of approaches that could have been taken here but I chose to use a simple enough one: We detect movement across the board which we use to segregate the video into different moves. Each of these periods are then processed with my solution from part 2 to tell us what colour is on what square. Finally, we examine how the board changes between moves by detecting that a square used to be empty but now contains some colour. All we have to do then is to find the square which is now empty which used to contain that colour. We must do our check in this order as many of one colour may disappear from the board (the pieces were taken) while the other colour will always move from exactly one square to another.

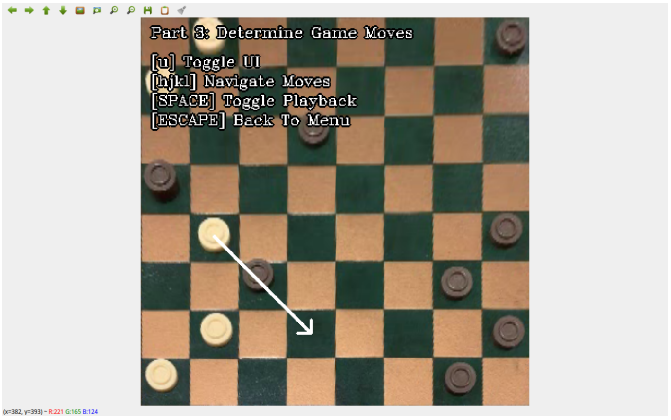

Fig 3.1.1 - Output of part 3

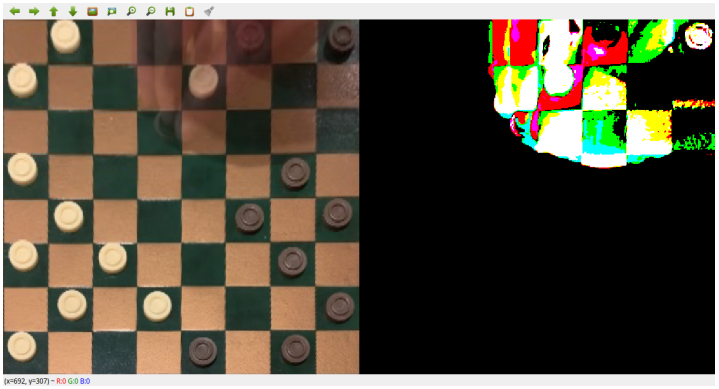

Fig 3.1.2 - Visualisation of motion detection algorithm within part 3

The motion detection algorithm I made uses the Gaussian Mixture Model from OpenCV to produce the median background image. One of the issues faced with this is trying to find out how many frames to include in the median background. It's a balance between including too many and subsequently missing a turn due to the median background not clearing before another motion is detected and including too few so that frames with

hands still in them are accepted as moves - This, of course, completely breaks my part 2 solution! In the end, I settled on using just two frames from the past.

3.2 Performance

This system is competent but not perfect. It will detect the same move a number of times but is able to successfully identify unique moves as discussed. It was actually quite difficult to evaluate the output of this system against some ground truth. It's trivial to tell if it worked - Are all the moves correct or not? It's more of a problem to rank *how* well the system did if it didn't get everything right. That's because if it misses or adds a move, it is now off by a move when compared to the ground truth. In my evaluation, my system missed one of the later moves and as such missed every subsequent move.

3.3 Potential Improvements

This system cannot handle a game using a different frame rate very well, as the number of frames considered is always three. It may be possible to dynamically decide how many frames to use in the median background somehow.

Another interesting improvement I thought of, unrelated to computer vision, is to consider the context of the game when trying to determine what move has been made. That is, only consider valid moves in the game and discard possible moves which are invalid. Unfortunately, this wouldn't solve how my solution skipped a move - Instead it would completely stall the system while it waited for a valid move to be played. It would take a bit more work, but it could be possible to generate a number of potential series of moves from the motion detection system ranked by likelihood of being the true series of moves. Then, each series of moves are evaluated in order and the first one which contains only valid moves is accepted as the truth.

4 Part Four - Edge Detection

4.1 Method Comparison

I have attempted to use the three methods to identify the draughts board corners and was only able to successfully do so using the findChessboardCorners OpenCV function. I was unable to use just the Hough line transform or contour following with straight line segmentation to identify board corners.

The Hough line transform method seemed useful but I wasn't able to distinguish the edges of the board from the edges of the squares. A potential solution to this would be to find the four lines closest to the edge of the image and use these to guess the board corners. However, another issue was that the Hough line transform will try to find lines crossing the entire image rather than the line segments of the board edges. This was fine for finding the bottom and side edges as they almost reach the image edges but the top edge only crosses about two-thirds of the image.

Similarly, the contour following method provided some information on where the corners might be but, by itself, it wasn't able to distinguish them from everything else it found. Using straight line segmentation looks promising as the corners are successfully picked out as straight line segment contours.

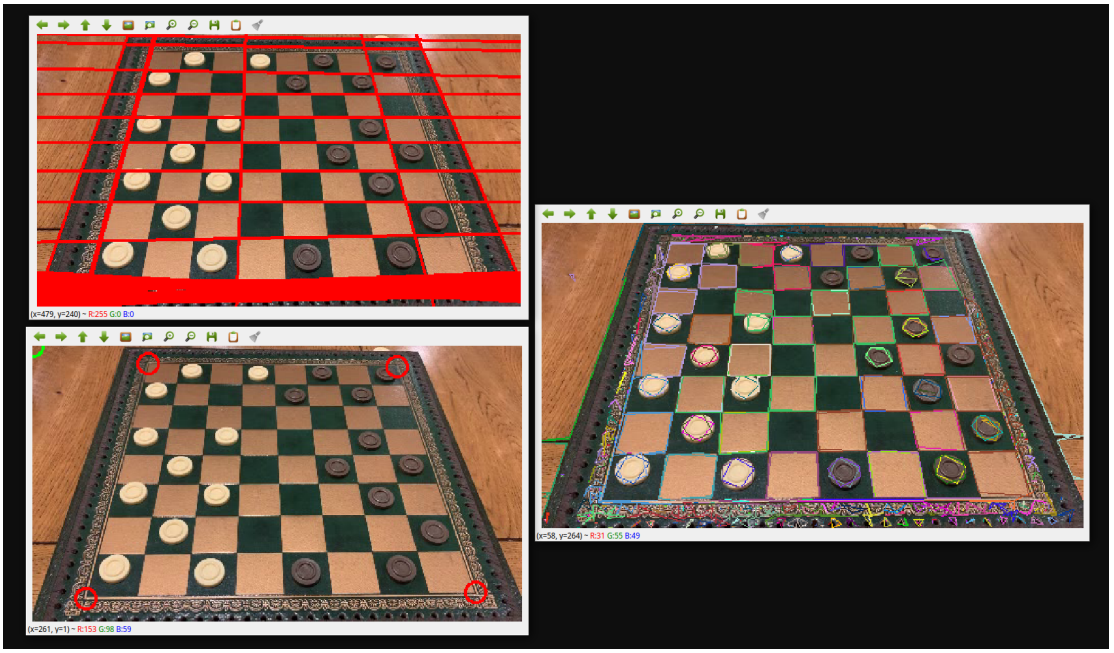

Fig 4.1 - Clockwise from top-left: Hough line transform, contour following with straight line segmentation and findChessboardCorners methods

I imagine OpenCVs findChessboardCorners might make use of both of the above methods together. It's also likely that it uses some form of pattern-matching to match against a binary image of the squares in the image as you must specify the layout of the board. Overall, it does a very good job at finding corners.

However, it doesn't do all the work - I first needed to threshold the image and use morphology in such a way as to remove the pieces from the board. Then, if findChessboardCorners is able to find the chessboard corners, we need to extrapolate from the inner corners it finds to get the locations of the four outer corners. Initially, I extrapolated by simply dividing the distance between the outermost inner corners by 6 and multiplying by 8. This was an okay solution and gave me an approximate answer.

The better method I came up with was to use an inverse perspective matrix generated from the corners given by findChessboardCorners to map a perspective-fixed board back to the original image. As I knew where the four corners were in respect to the perspective fixed board, I could map these points back to the actual coordinates in the real image.

4.2 Performance

For this assignment, we were given the actual board corner coordinates. So by using each method on the 69 test images provided, we can compare their results with the ground truth. Neither the Hough line transform or contour following method ended up producing any results, but I was able to measure the accuracy of the findChessboardCorners method along with my inverse perspective transformation solution. I score its result by finding the absolute difference the four corners it gave and the actual four corners.

```
contours: 1217.04 off
findChessboardCorners: 6.0099 off

res/DraughtsGame1Move67.jpg
hough: 1217.04 off
contours: 1217.04 off
findChessboardCorners: 5.50649 off

res/DraughtsGame1Move68.jpg
hough: 1217.04 off
contours: 1217.04 off
findChessboardCorners: 6.6021 off

Averages
hough: 1217.04 off
contours: 1217.04 off
findChessboardCorners: 23.654 off
```

Fig 4.3 - Output of part 4 evaluation

As you can see, it does pretty well - Generally it’s no more than 8 pixels off in total between the four corners. However, notice that its average is significantly worse - This is because it was unable to find the corners in move 46. In that case, it returns (0,0) for all four corners which of course is nowhere near the ground truth, ruining its otherwise great average.

5 Part 5 - Recognition

5.1 Overview of Solution

My solution to part 5, classifying the piece colour and type on each square, directly builds on top of my solution to part 2. As such, this solution requires sample images to get of the colours of the pieces and the location of the corners of the board - I appreciate that I could have also used my solution to part 4 to get the board corners. With this information, we are able to get whether there is a piece on each square on the board and what its colour is.

To recognise what type of piece is on a square - a man or a king - I chose a fairly simple approach: I threshold each square with a fixed minimum and maximum value depending on the piece colour to isolate just the dark patch corresponding to the side of the piece on the board. Then I simply check if the number of remaining pixels exceeds a specified amount. I also use some other simple techniques (fixed perspective, image erosion, only considering the bottom half of the square image) to better isolate the larger dark patch of a king from the smaller dark patch of a man.

Of course, this solution does not account for changes in lighting, pieces or the board between games. As such, such a system can only really work on this exact scenario. It would be better to use shape recognition to accurately determine whether two pieces had been placed on top of each other.

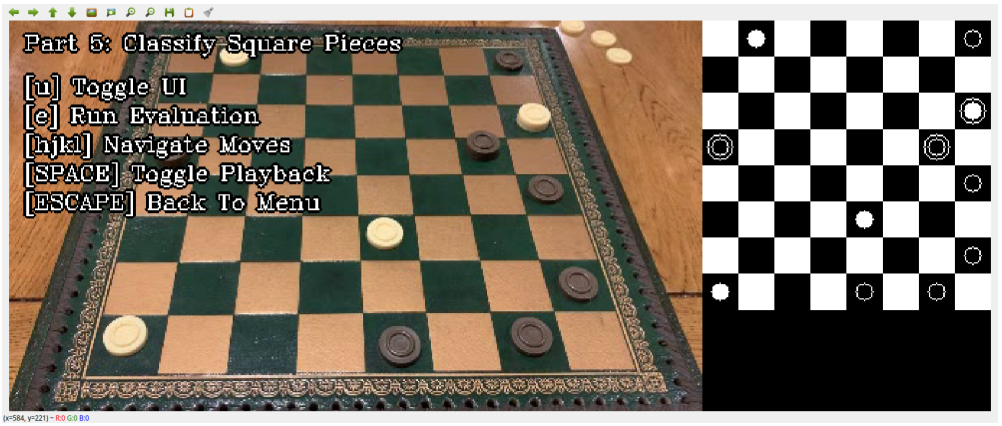

Fig 5.1 - Output of part 5

5.2 Performance

The performance evaluation of this part is the same as part 2 except this time it takes piece type into account. In fact, the solution for part 2 was already getting 96% of the squares in the test image right, including kings! This is because only a handful of the pieces ever get promoted to kings in the game. That probably indicates our test images don't cover enough cases or provide enough assurance that our system is actually accurate.

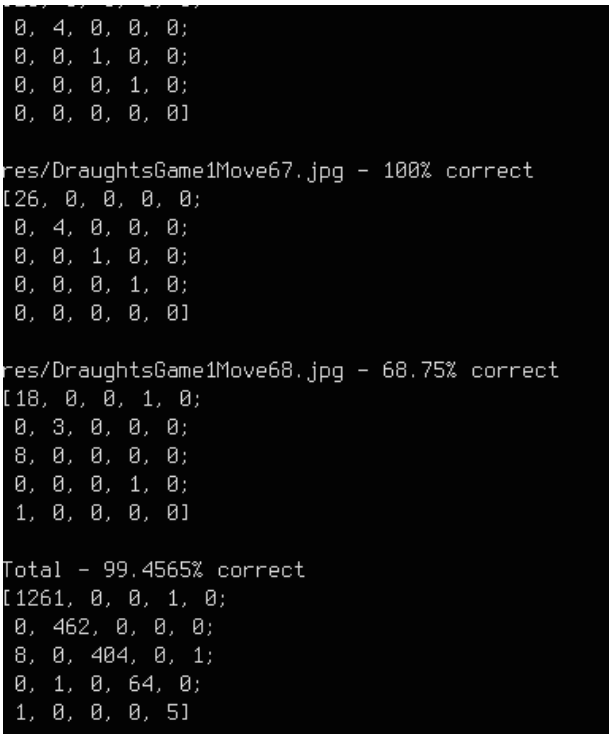

Fig 5.2 - Output of part 5 evaluation

As you can see in the terminal output, my solution for part 5 is able to correctly identify 99.45% of the squares in the test images. Generally, the system performs well in most of the cases, despite its simplistic algorithm. However, much like in part 2, the system fails to understand the last move and only receives a score of 68%. Even with this decent score, I don't believe this is a very confident algorithm as I can see the system occasionally mistakes a white man for a king when analysing the full video footage.

5.3 Potential Improvements

One immediate improvement for this system would be to fix the issues with my solution to part 2 - In particular, handle the end game where one colour has been completely removed. Additionally, as mentioned before, it would be worthwhile to invest some time into using actual image recognition techniques to identify men from kings. I wonder if SPR would be able to distinguish the elongated-ness difference between a man and king? I believe other techniques need training to work effectively - It would be interesting to see if I could train a Haar Cascades model to differentiate men from kings given just the test images.

6 Conclusion

Overall, I found this a very enjoyable and satisfying project. I was particularly happy with myself for developing and iterating on my solution for part 2. There were a number of technical problems I faced while implementing my solutions in C++ which I would not have thought of if I had only studied computer vision from a book. I was also under serious time pressure when implementing part 5 so I was not able to work as hard as I had liked there.

I spent about 60 hours in total on this project, with much of my time spent researching and comparing different computer vision techniques. It didn't take long to actually implement these techniques, primarily due to how many of the techniques are written with just a few lines of C++ when using OpenCV. However, I'm ashamed of how many hours I spent on the overall project structure and tinkering with the irrelevant user interface. I must have refactored the entire project four times as it grew over the course of the semester - I will have to re-evaluate how I approach developing medium-sized object-oriented C++ applications in the future.