Computer vision techniques on Draughts game

Computer vision project

M. MANSOUR Yanis
Student n°: 22327392

02/11/2022

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

Table of Contents

Introduction

This report explains the computer vision techniques I used throughout the Draughts project.

It contains 5 parts, namely:

- Classifying the pixels of draughts game images
- Detecting the presence of a piece in each square, while recognizing the color of the piece
- Detecting the moves in the draughts game video
- Determine the location of the four corners of the draughts board using different techniques
- Distinguish between normal pieces and kings

Each of these parts contains an explanation part where I address the theory of what I am doing, a result part which mainly consists of result images of my processing, and finally a robustness part, where I go through the borderline cases where my algorithm may fail.

1. Classifying the pixels in static images

1.1. Concepts

The goal of this part is to classify the pixels of the draughts image into 5 parts. The solution must cope with changes in lighting and in the appearance of the pieces and the board

1.2. Application and results

The main method I thought about to do this is histogram back projection:

Histogram back projection

This method consists of taking samples of the different parts of the board:

- Black squares
- Black pieces
- White squares
- White pieces

Once we have those samples, we form normalized histograms for each of them and back project those histograms into every single image we want to segment to get a probability image.

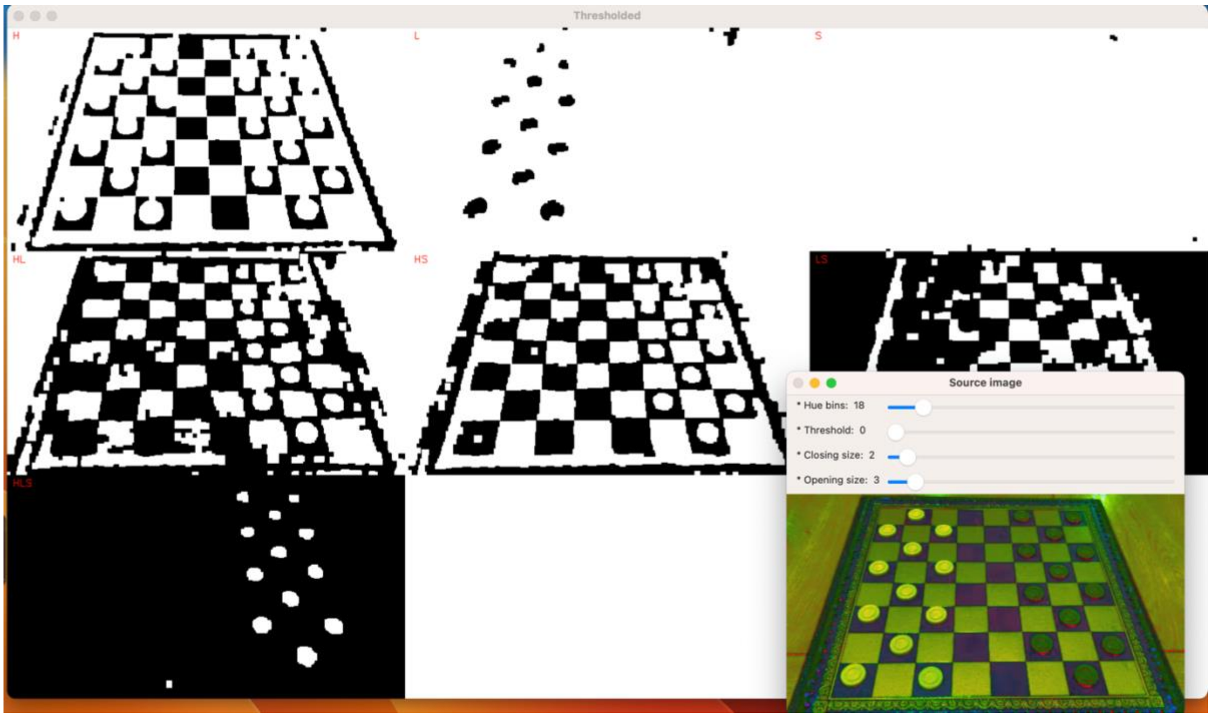
However, the problems I got with is that depending on the part of the image we want to back project, we have to use different channels to get a good result. For instance, using the hue channel only to back project white squares ends up returning the background and the pieces as well.

To overcome this, depending on the part of the image, I used different channel combination. First, the black squares seem to have a hue range which is well separated from the other parts of the image. Besides of this channel giving good results, the hue values of the black squares being in a certain range allow us to have only 3 hue bins. That gives us a good separation in the histogram.

At the opposite of the black squares which have really well grouped hue values, the white squares are much more complicated, if not impossible to classify with only the hue channel. Thus, to classify the white pieces I added luminance and saturation channel in the back projection.

Similarly for the pieces, I could classify the black pieces with the 3 hue, luminance and saturation channel, where the classification of the white pieces could be done with only the hue and luminance channels.

In addition to channel configuration, I perform thresholding, followed by closing to fill holes, and opening to remove noise. To select the number of bins, which is related to the distribution of the hue values in the histogram, as well as the other parameter, I used sliders:



* Hue bins: 18
* Threshold: 0
* Closing size: 2
* Opening size: 3

Figure 1 - Sliders showing the process of selecting the different parameters

We can see here that for the black pieces, the HLS channel combination does really well, along with the used parameters.

Those are my parameters for each part of the draughts board:

Feature	*Threshold*	*Channels*	*Closing size*	*Opening size*	*Hue bins*
White squares	2.3%	HLS	3	3	2
Black squares	0%	H	1	0	5
White pieces	6%	HL	2	2	33
Black pieces	0	HLS	2	3	18

And here is my result:

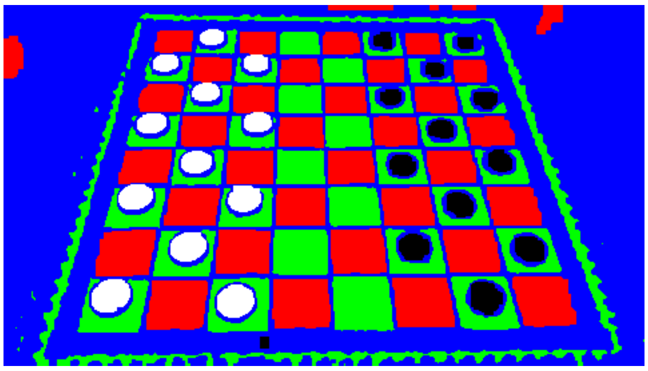


Figure 2 - Part 1 classification result with 82.5% accuracy

Before talking about the robustness of this method, I'd like to mention that I tried using K-Means, but the white squares of the draughts board get classified either with white pieces or with the background.

1.3. Robustness

Even if this method gives us pretty good results on this image, it may fail on others. We already see here that a part of the frame of the draughts game is classified with the black squares.

We also notice that the black pieces that are far from the camera are very poorly detected. A significant change in luminance can cause those pieces to disappear completely when performing the opening operation.

Moreover, the parameters I took (opening) eliminate a lot of noise in the background but may reduce the size of the detected parts (furthest black pieces for instance) because of the opening operation. We may not want this behaviour and say, "It is worse to miss parts in the draughts board than to have some noise in the background". In this case, we may accept some background noise and not risk the furthest black pieces to disappear.

2. Detect the presence of pieces in each square

2.1. Concepts

The goal of this part is to explain the methods I used to detect whether there is a piece in each square of the draughts board, after a perspective transformation that makes the draughts board flat. I also determine the colour of the piece in this part.

To evaluate how well I've done, I created a confusion matrix that compares my results with the ground truth given in the code

2.2. Application and results

First, we have been given the points of the 4 corners of the draughts board. That has been very helpful to simplify the process.

With those points, I could obtain a flat draughts board:

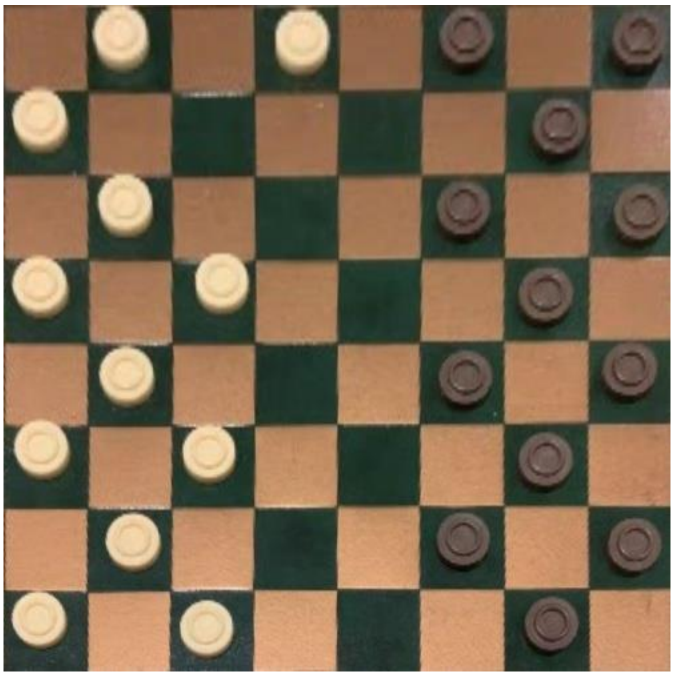


Figure 3 - Flat draughts board after perspective transformation

Now, we just have to divide the image into 8 columns and 8 lines. Then, we have to perform 32 operations (on black squares only). Black squares are those in which this condition is satisfied: $row \,\%\, 2 \neq column \,\%\, 2$

First, I passed the image to the classifier developed in part 1 to get binary images of black and white pieces. Then, I go through the 32 black squares, and crop both the binary images to only get the region corresponding to the current square. With a fixed threshold, I decide whether a square contains enough "white piece pixels" or "black piece pixels".

Here is an example of what we obtain when we pass the perspective transformed image to the classifier:

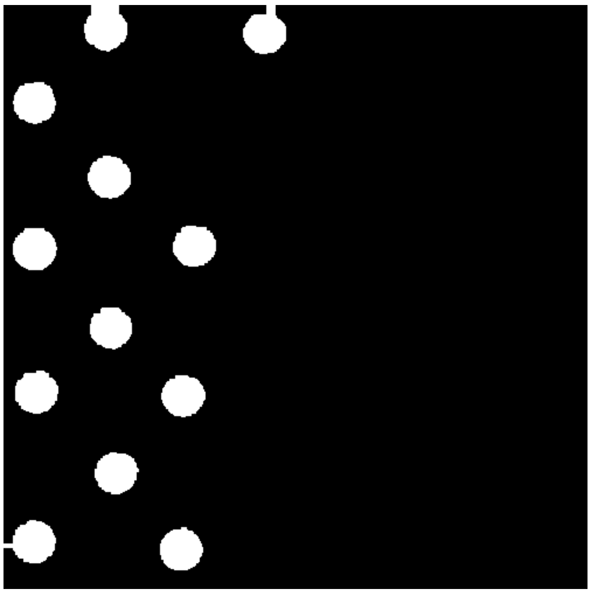


Figure 4 - Binary white pieces

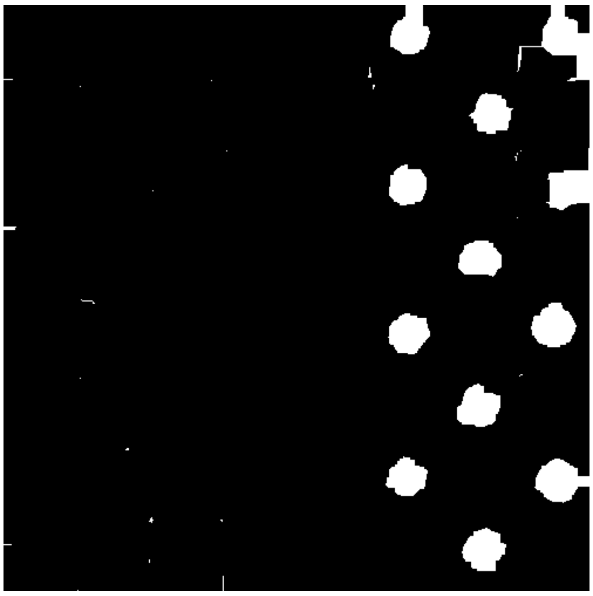
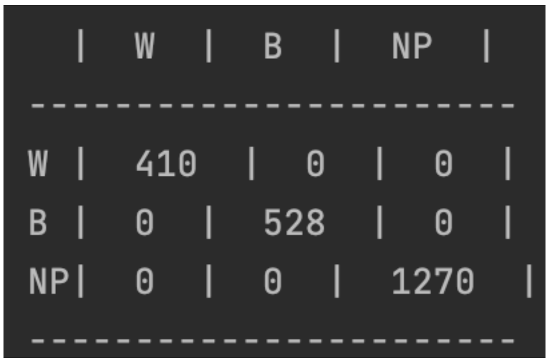


Figure 5 - binary black pieces

Here, we face 2 major issues. The first one is the fact that the classifier may return some noise on the empty black squares. To overcome this, we can set a threshold ofa minimum “piece pixels” we want in order to decide that there is a piece in this square. However, this leads us to the second problem: some pieces may overlap between two squares. Even if the second square is white and we never consider it, a high enough threshold may lead the system to fail when there are pieces that have a big enough part on a neighbouring white square.

In the given dataset, I managed to find a threshold that is big enough to avoid the noise, and low enough to include pieces that are slightly overlapping with white squares.

With a threshold of 3% of square space, I get this confusion matrix:



```
   |  W   |  B   |  NP   |
 ------------------------
W |  410  |  0  |  0  |
B |  0   |  528  |  0  |
NP|  0   |  0   |  1270  |
 ------------------------
```

Figure 6 - Confusion matrix of board piece detection

This confusion matrix is obtained by comparing my results with those given in the ground truth. Since the ground truth includes kings, I eliminate them before comparing the results.

2.3. Robustness

As stated in the section before, this method fails if a piece overlaps significantly with a white square. To overcome this, we might consider taking a slightly bigger area for each square.

Second, our classifier is way better at detecting white pieces than black pieces (noise). If the lighting conditions change significantly, it may lead the system to detect enough noise in some regions, and thus exceed the 3% fixed threshold

3. Tracking the moves in a draughts board video

3.1. Concepts

This part consists of tracking the moves of pieces in the video using motion tracking techniques with a background model. The moves are then captured and stores with PDN notation. Finally, the results of my technique will be compared to the ground truth to assess how well I've done.

3.2. Application and results

To track the moves, I had to select an adequate background model that could answer the problems in this section.

It is worth noting that I've tried a static background model, where I update the background every time the video stabilizes. To track moves, I used an additive model that returned the sum of all the pixels that moved in the past 3 frames. Though, this model seems to be highly sensitive to shadows outside of the draughts board and may sometimes stabilizes when the player puts his hand on the draughts board and takes a bit of time to think.

Therefore, I had to abandon this model and go for the gaussian model that could perform way better.

GMM (Gaussian Mixture Model)

In this method, I used the Gaussian Mixture Model as a background subtractor. This allowed me to accurately detect when there are movements in the videos and when the video stabilizes.

Compared to the static background model that returned the sum of the pixels that moved in the 3 last frames, the GMM model seems to deal better with shadows thanks to its approach of selecting foreground pixels, which relies on the fact that a pixel is foreground if its current value did not appear very often in the history (the past images).

However, detecting the moving pixels doesn't necessarily indicate which pieces have been moved. It is worth noting that to detect the pieces, I rely on a state variable that can take 2 states: MOVING and STABLE.

First, I keep the first frame of the video in a variable named `background`. Then, at each frame, I compute the number of moving pixels. If this number exceeds a certain fixed threshold (0.5% of the total number of pixels), I set the state to MOVING. Similarly, when the state variable has the MOVING state and the number of moving pixels drops below another fixed threshold (0.2% of the frame), I set the state to STABLE, and compare the current frame to the image I kept in the `background` to get a difference image representing the moved pieces:

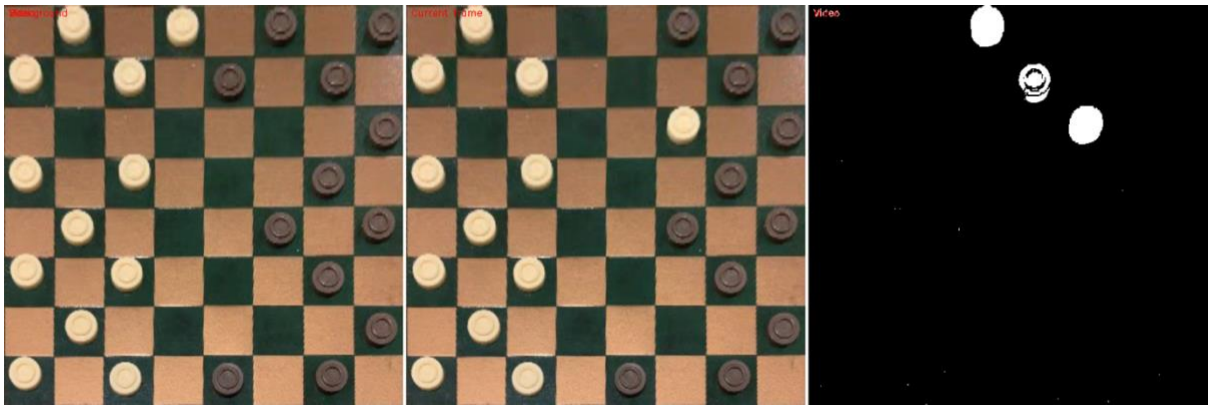


Figure 7 - Last background (Left), Current frame (middle), Absolute difference (right)

From this difference image and depending on the current player (black or white), I can easily look for the move of the current player. With the intersection between the binary image and the positions in the last image to detect the start move, and with the intersection between the binary image and the positions in the current frame, I can get the end position of the move.

The final step is updating the background variable to put the current "stable" frame in it

Though, this model requires parameters to work. The first one is the length of the history I talked about earlier. If the history is too long, moved pieces may take too long to become part of the background, and we thus miss getting to the STABLE status before another move is started (hand entering the draughts board). As opposed to this, a history that is too short may result on the hand fading to the background very quickly. In cases where the player puts their hand into the draughts board and starts thinking before performing the move (stable hand), we may switch to a STABLE state too often.

Another parameter of the model is the var threshold. For each single pixel, GMM creates a set of gaussian distributions that it fits into the pixel values histogram:

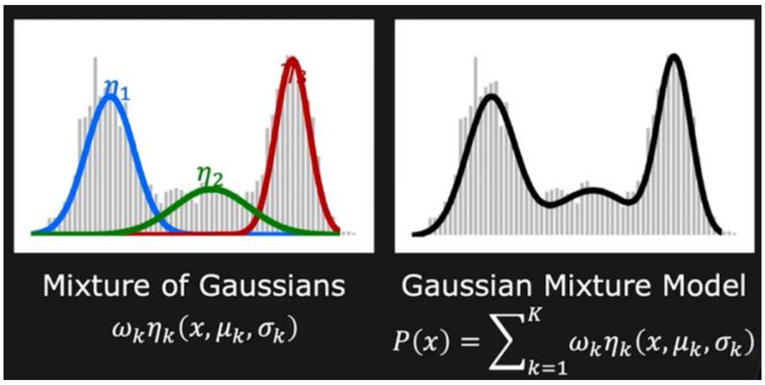


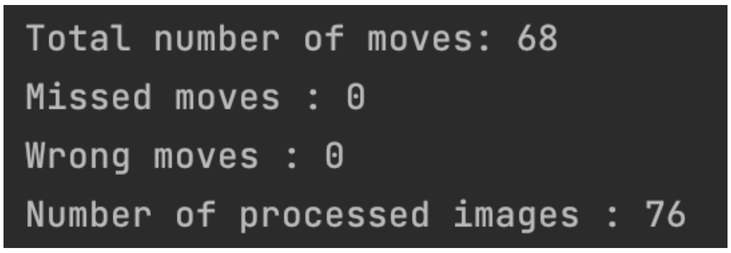
Figure 8 - Gaussian mixture model | Source : https://www.youtube.com/watch?v=0nz8JMyFF14

Var threshold is applied to the distance between each new pixel and the different gaussians to see if it is well described by the model. If not, a new gaussian is created.

For this model, those are the parameters I ended up with:

Length of the history	Var threshold
80	16

Those are the results I obtained with those parameters:



```
Total number of moves: 68
Missed moves : 0
Wrong moves : 0
Number of processed images : 76
```

Figure 9 - Move tracking results with GMM model

3.3. Robustness of the solution

This solution, although very efficient in our case, may fail under some conditions:

- A hand that moves a piece very fast (The model won't create a gaussian distribution for the set of the quickly moving pixels as they are too insignificant, and thus, won't consider them as a foreground)
- Two consecutive moves that are performed too fast => The moved pieces may not have enough time to fade into the background.
- A player who puts his hand on the draughts board and thinks for too long may cause the model to fail as well, as the hand will fade to the background. A way of improving this would be a back projection on the background to see if there are skin pixels. If so, do not update this background when computing the absolute difference with the next stable frame

4. Detecting the corners of the draughts board

4.1. Concepts

The goal of this part is to recognize the 4 corners of the draughts board from an image which has not undergone any perspective transformation

4.2. Application and results

In the first two methods (Hough lines and contour following), I used Canny edge detection. Before diving into the methods, I will first explain how I chose its parameters, especially the high and low threshold.

So, the edges detected with Canny method are selected depending on their gradient's magnitude value (that is computed with the first derivative). Basically, any pixel that has a gradient magnitude above the high threshold is definitely an edge, and any pixel with a gradient magnitude below the low threshold is definitely not a pixel. Any pixel with a magnitude value between the high and low thresholds is an edge only if it is linked with another pixel which is already an edge:

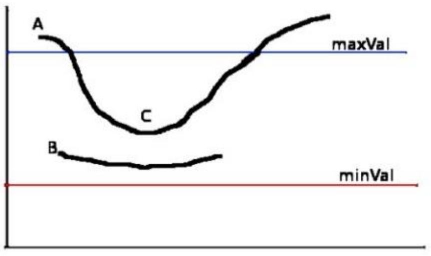


Figure 10- High threshold and low threshold illustration | Source : https://docs.opencv.org/4.x/da/d22/tutorial_py_canny.html

With this said, I selected the high and low threshold in a manner that allowed me to get the edges of the inner squares, without sacrificing a lot of the edges that are far from the camera (The top-right corner in the image below for instance):

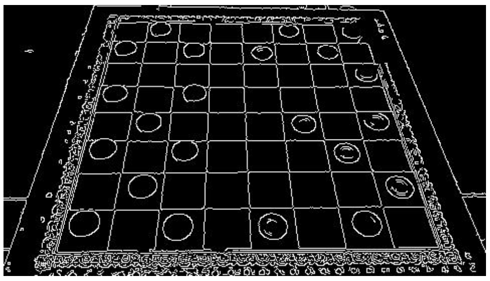


Figure 11 - Canny edge detection with high threshold = 350 and low threshold = 150

Hough transform

Hough transform for line detection is a technique that uses polar coordinates to represent lines: $r = i\,.\cos(\theta) + j\,.\sin(\theta)$

It looks for all possible lines by varying the two parameters r and θ with a resolution for each of those parameters ($\theta_{resolution} = \theta_i - \theta_{i-1}$) and picks the lines that intersect with the biggest number of points. All the examples below are produced with OpenCV, with $\theta_{resolution} = \frac{\pi}{200}$ and $r_{resolution} = 1$.

From a Canny-produced binary edge image, we can apply Hough lines, that gets us a very big number of vertical and horizontal lines. In order to continue our processing we have to separate them. To do this, I set a range of angles (θ) where the vertical lines should stand ($< 20°$ and $> 160°$), and another range of angles (θ) where the horizontal lines should stand ($\theta = 90°$). Now, we have to eliminate lines that are too close together. To keep a line, it should be at least $\epsilon = 20\ pixels$ far from the last line:

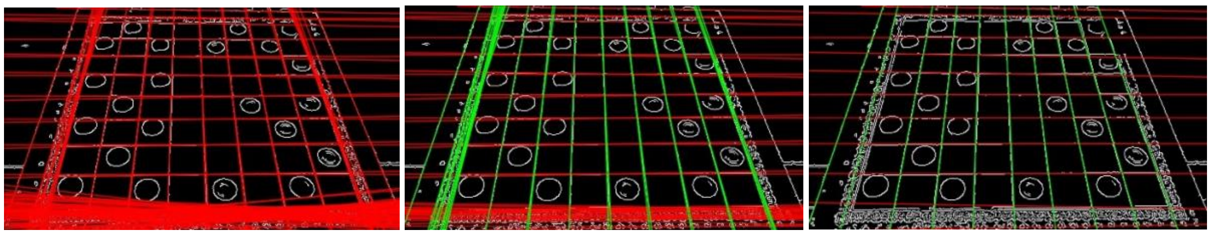


Figure 12 - Hough transform result (middle) | Vertical and Horizontal lines separation (left) | Close lines elimination (right)

At this point, only one step separates us from the final grid that represents the draughts board. Here, I just had to find 9 evenly spaced lines horizontally and vertically. Now, we just have to pick the first and last horizontal lines, and the first and last vertical lines. The intersection between those lines are the corners of the chessboard

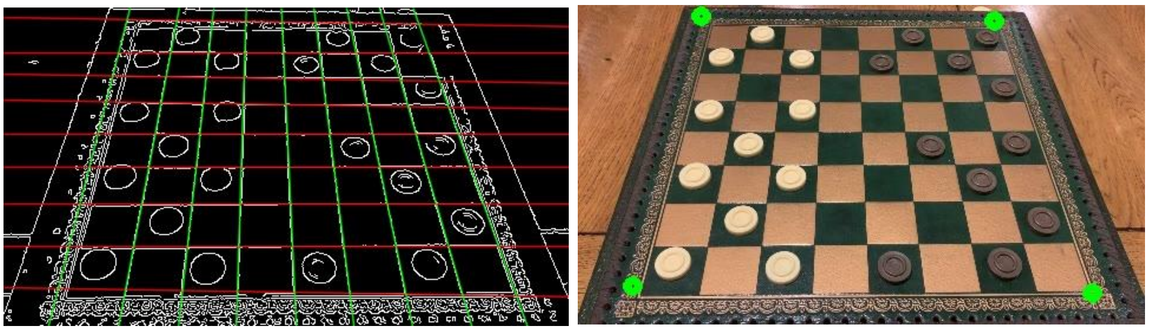


Figure 13 - Grid lines of the draughts board (Left) | Intersection between the 4 border Hough lines (Right)

Contour following and straight-line segment extraction

From a Canny-produced binary edge image, we can apply contour following to approximate edges with segments. This allows us to compute the angle between every two consecutive segments and select the intersections of segments that form approximately 90°. Setting a range of the length of segments is important though, to eliminate the segments that are too long (the edge of the draughts board's frame for instance) and segments that are too short. After getting this limited set of corners, we can now select the top-right most, the top-left most, the bottom-right most and the bottom-left most. Those are our estimated corners of the draughts board:

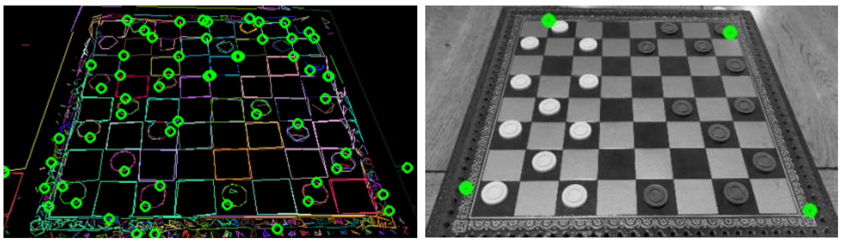


Figure 14 - Corner detection based on angles between line segments (left) | The estimated four corners (right)

We notice that this method is not accurate at all. Not only we miss a significant number of corners because of the perspective, but we also get corners in some pieces as they are approximated with segments.

Using findChessboardCorners() routine from OpenCV

This function tried to find inner corners of the chessboard and displays them in order (from left to right, from top to bottom). This function works perfectly on the empty draughts board image, but fails completely on frame where there are pieces on the draughts board

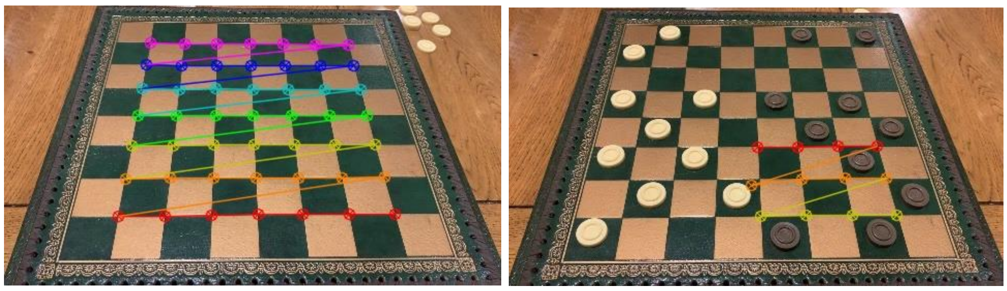


Figure 15 - findChessboardCorners() function with (7x7) filter (left) | findChessboardCorners() method with 3x4 filter(right)

In the picture in the right, I had to loop through different sizes to find one that fits.

Some other images work with different filter sizes. My guess of this is that when there are pieces on the draughts board, some of them overlap on the edges of the squares. That pushes the function to think that the corners that are next to the concerned edges are not really corners.

4.3. Robustness

The only really efficient method to find the corners of the draughts board is the Hough lines method. Although efficient in our examples, where the draughts board's horizontal lines are parallel with the x axis of the image, this method will definitely fail if the draughts board is slightly rotated. Indeed, the vertical and horizontal lines separation will no longer work because I used predefined θ ranges. To overcome this, K-means with $k = 2$ would be a good separation algorithm.

In contrast, contour following method gives really bad results for our examples and returns random results on different images. Moreover, it is highly sensitive to the presence of pieces that overlap with the edges of squares. This would not be the turn-to method in a real-world implementation.

5. Distinguishing between normal pieces and kings

5.1. Concepts

The goal of this part is to distinguish between normal pieces and kings. The main method that I'll use in this part is statistical pattern recognition. I'll illustrate the process in this part

5.2. Application and results

So, the general Idea I had for this part is statistical pattern recognition. Kings are two pieces one on top of the other. When we apply a perspective transformation to the draughts board with the given corners, we notice that those kings have more elongatedness than the normal pieces.

Though, the difficulty now is how to compute this elongatedness. The idea I had was subtracting the empty draughts board image from the frame we are processing, to get only the pieces. Even though grayscale subtraction doesn't seem to return good results when the images involve some shadow, subtraction of the hue channel does a bit better. This is an example of a thresholded difference image of a hue channel of a frame, and the hue channel of the empty draughts board image (The thresholded image has been closed then opened with a 3×3 mask to fill holes and remove noise)

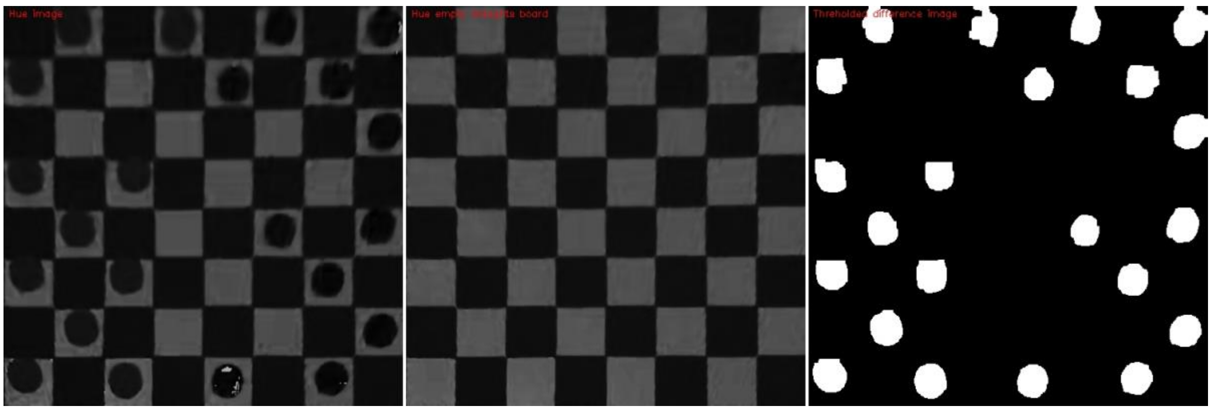


Figure 16 - Hue channel of current frame (left) | Hue channel of empty draughts board (middle) | difference image

We can now divide this image into the 32 black squares that are likely to contain pieces. In each of those, we can do connected component analysis to detect regions.

There is a problem though: In some squares we might get some noise despite the opening we've done, resulting on two regions from our connected component analysis. To overcome this, we just have to keep the largest region in each square.

So, for each largest region in each square, we can compute the elongatedness. A problem we have here is that depending on the colour of the piece, we get more or less noise (There is more noise for white pieces):

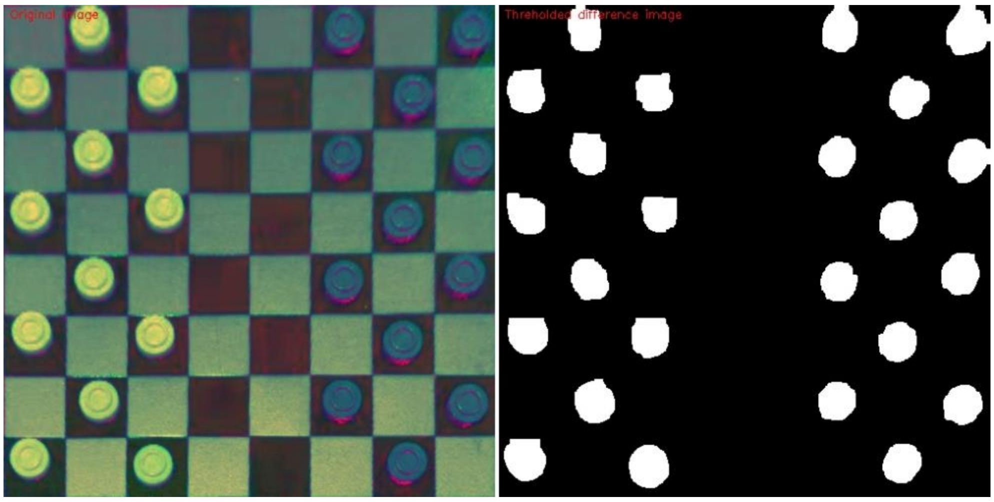
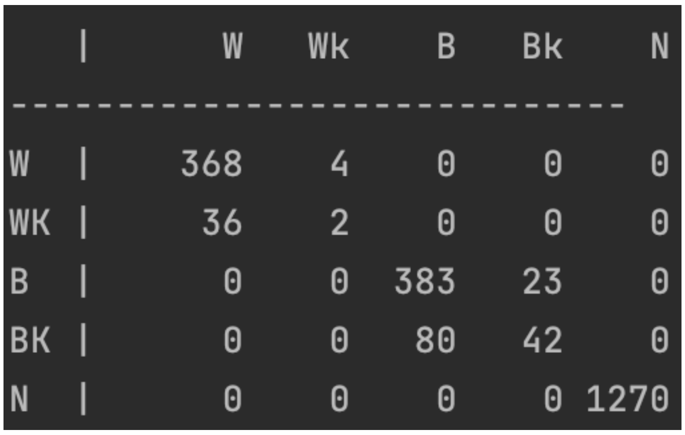


Figure 17 - HLS image and thresholded difference image (Different noise levels)

Therefore, the elongatedness threshold that we apply to white and black pieces is different. After testing different values, I've come with those thresholds that are not the worst:

Minimum elongatedness for white kings	**Minimum elongatedness for black kings**
2	1.7

With those values, I obtain this confusion matrix:



```
   |        W    Wk      B    Bk     N
--------------------------------------------
W  |      368     4      0     0     0
WK |       36     2      0     0     0
B  |        0     0    383    23     0
BK |        0     0     80    42     0
N  |        0     0      0     0  1270
```

Figure 18 - Extended confusion matrix that includes kings detection

5.3. Robustness

We notice that this method gives us a lot of false negatives, and most importantly, lots of false positives. This is mostly because of the accuracy of the hue channels' difference. Indeed, when the pieces are very far from the camera (the first row of the draughts board), they are significantly less detected than the closer ones. Moreover, the similarity in hue value between white squares and white pieces gives us truncated white pieces in some cases:

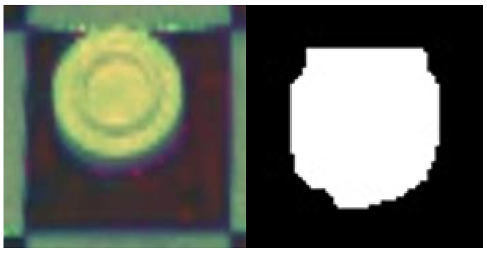


Figure 19 - truncated white piece because of it overlapping on the white square

This leads the system to sometimes give random results.

Canny edge detection might do better in this example it locates the edges more accurately. We can for instance imagine a system that takes a slightly larger squares to be sure to get all the piece. After this, we can use Canny edge detection to get the contours, and then fill the circular regions to get the piece. From the regions we get, we can imagine that elongatedness gives us better results.

Finally, we can overcome all those problems in vision by tracking the moves of the pieces using the video. The system would track every black piece, and as soon as it reaches the leftmost column of the draughts board, the system labels it as a king. Similarly, it would track every white piece, and as soon as it reaches the rightmost column of the draughts board, it labels it as a king.