DRAUGHTS DETECTOR

INF573 - Image Analysis and Computer Vision: algorithms and applications

5 January 2020

Jean-Peïc CHOU: jean-peic.chou@polytechnique.edu
Pierre FERNANDEZ: pierre.fernandez@polytechnique.edu

Introduction & Motivation

Really often, as a chess player or a draughts game player, one can want to instantly turn a photo of the game into a digital display of it. It could have several applications. For instance, to save a "real life" game in order to continue it later online, or on a more advanced level, to take a picture of the game, so that a computer can tell you where to play for the next move, etc.

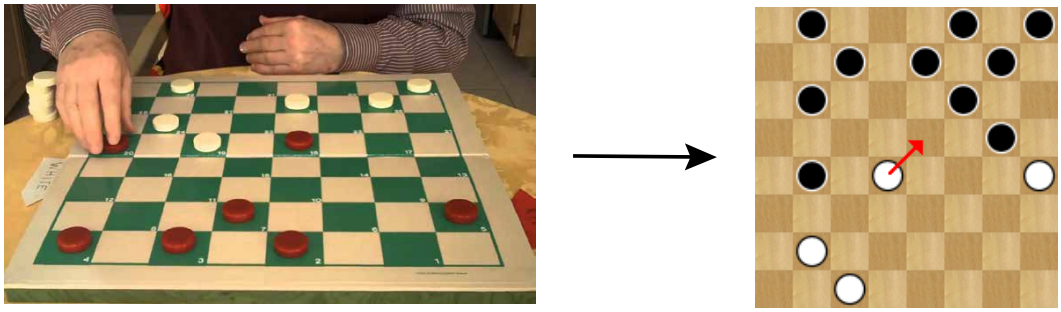

Figure 1: Goal of the program

To put in in a nutshell, the goal of the program is to process an image of a chessboard and to return the picture of the chessboard seen from above and a digital representation of the game (as a matrix of -1, 0 and 1).

Chessboard Detection

Detecting the lines

First things first, one must detect all the lines that could constitute the grid of the chessboard. To do so, we used the HoughLine feature of OpenCV which is used to detect straight lines. A line can be represented as (m, b), where m is the slope and b the coordinate at the origin, or in an other parametric form, as (ρ, θ) where ρ is the perpendicular distance from origin to the line, and θ is the angle formed by this perpendicular line and horizontal axis measured in counter-clockwise. The Hough transform uses this parametric form, which is useful to compute all the points belonging to the same line. The HoughLine feature of OpenCV precisely uses Hough transform. The results with a good choice of parameter and a simple preprocessing of the image are shown in Figure 2.

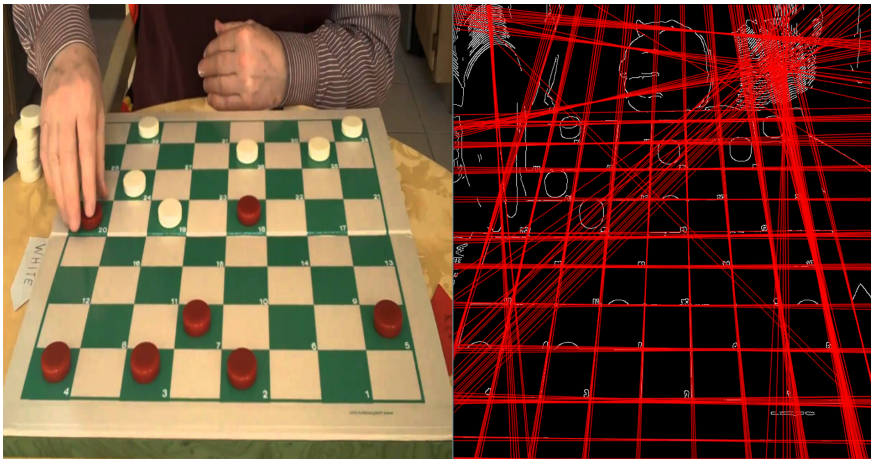

Figure 2: Lines detected using the HoughLines feature of OpenCV

To get to these first results, some preprocessing needed to be done. First, the image was blurred so that small stripes or small details that can damage the image's readability would not be taken into account in the next steps, but not too much so that lines would still be clearly distinguishable. Then, we used a Canny filter in order for the HoughLine feature to be more accurate.

After these steps, HoughLine detection can be used. However, one parameter is really important to take into account: the threshold, i.e. the minimum number of intersections to "detect" a line. Depending on this parameter, one picture can lead to many lines or none at all. In addition, the size of the image is really important : the larger it is, the more points there are, and the more the threshold need to be high. Therefore, in order to have a more stable solution, we used two methods: resize the image to a constant size (800px x 800px), use a moving threshold and choose one that gives enough lines.

Processing the lines

Once we got the lines, we decided to group and order them according to their directions and distance to the origin. That will be useful later in detecting the board's frame (cf. subsection "Detecting the board's frame").

To do so we used k-means clustering with $k = 2$ and a small number of iterations. But, given the parametric representation of a line (ρ, θ) we had to group together θ and $\pi - \theta$. Moreover, by creating a histogram and computing the size of each bar of the histogram, we have been able to remove lines that were "alone" in their directions. It gave the following result (Fig. 3 &

Fig. 4) for the same example as given above :

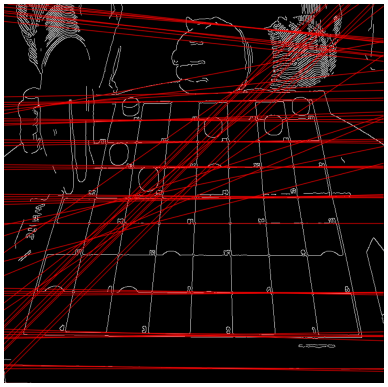

Figure 3: Batch corresponding to horizontal lines

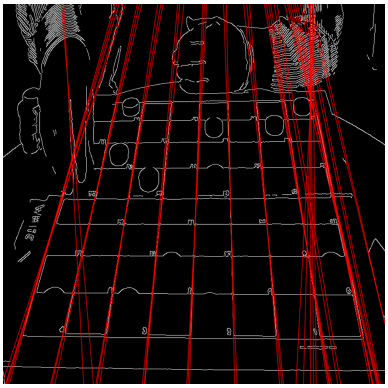

Figure 4: Batch corresponding to vertical lines

Detecting the board's frame

We now find the lines which constitute the board's borders by using a RANSAC algorithm that chooses 2 random lines of each batch and score them. To do so, the part of the image framed by the 4 lines is warped to get a black and white square of a fixed number of pixels (128 in our implementation) by finding the intersections and sorting them. This square is scored according to the following method. This output is divided into 64 squares, 8 by 8. We attribute a mean value of the pixels for each of these squares, from 0 (black) to 255 (white). We compute the score by summing the positive or negative differences of every juxtaposed values so that the result of these differences is always of the same sign if it's a board. This way, we are also making sure to take into consideration the picture's variations of light. If the 4 lines correspond to the right frame, the score should be higher than $\Delta \times (7*8*2) = 8960$, for $\Delta = 80$, Δ being the difference

between two contrasted squares. However, if we suppose that there are 12 white pawns lying on 12 black squares. The maximal score should be higher than $\Delta \times (7*5+4*8) = 5360$, for $\Delta = 80$.

Some examples are given in Figure 5 to better understand how a possible board is scored.

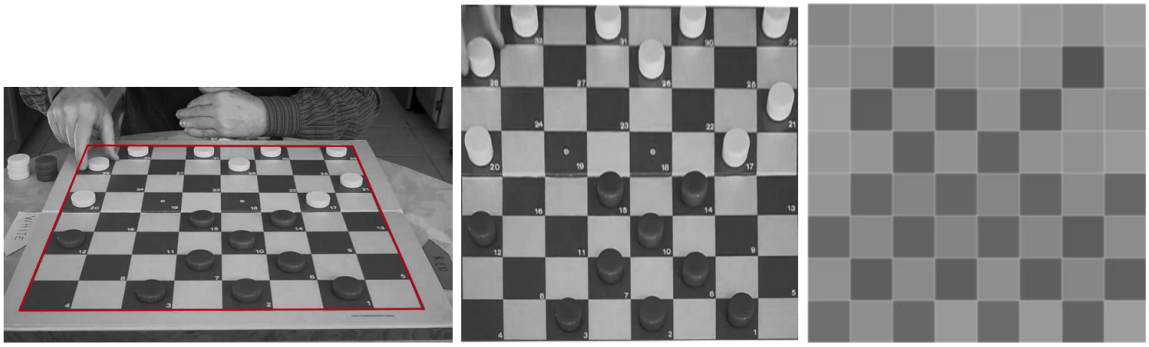

Figure 5: The chosen lines match the board's frame - score of 8986

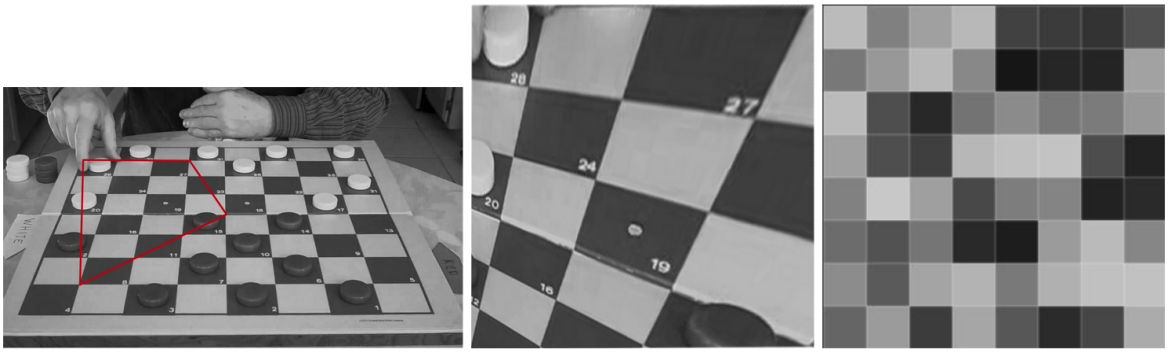

Figure 6: The chosen lines do not match the board's frame - score of 1262

The main problem with this method is that we need to define Δ, the values' difference between light and dark squares, which depends a lot on the board and its colours (green and white, light and dark brown...), but also on the light conditions (in the dark or overexposed with reflects...). That leads to 2 major issues:

- by keeping the maximum score, we sometimes end up with a 6 by 8 board because of the white pawns on black squares, as illustrated in Figure 7.
- by keeping the maximum score, we sometimes end up with a line outside the board that takes the place of a real line, as illustrated in Figure 8

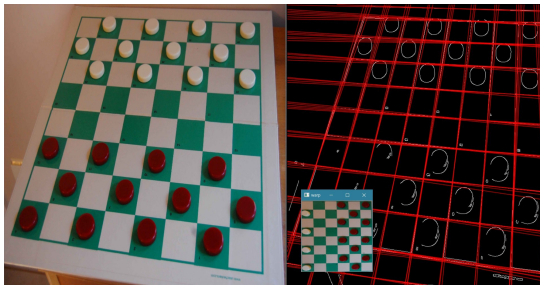

Figure 7

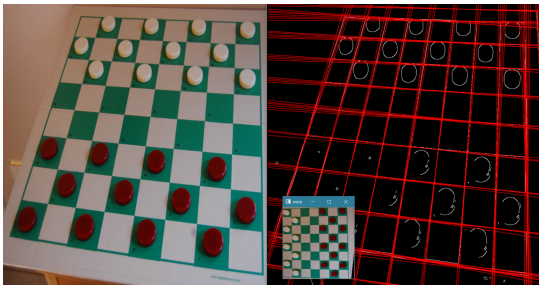

Figure 8

To avoid those issues, we could multiply the score by the size of the board but this is neither a good solution because one would really often get the outside borders of the board, which is not suitable for the pawn detection to come. We eventually considered the borders with the size of the board, saying that the contribution of the borders to the output should be responsible for at least 20% of the score, as a necessary condition. To take into account the size of the board, we kept the boards that presented the best scores and amongst them, we choose the final one to be the biggest one.

At last, using the findHomography function of OpenCV with, as inputs : the corners of the input board and the created corners for the new image, we can isolate the chessboard. Some results are given in Figure 9

Figure 9: Some results obtained with our program

Pawns Detection

Pawns' presence

Now that we have the grid of pixels picture divided into 64 squares, we can detect the pawns. The first idea is from using the output of Canny edge detector. In each square, we count the number of white pixels, excluding the ones too close to the borders, which correspond to the lines of the grid, or the middle, which can be due to flaws or small details like dust on the grid.

We also split each square in 4 parts. In at least 3 of them, we need to find white pixels to assert the presence of a pawn. However, this method is limited by the precision of the Canny edge detector. Besides, we know that pawns can only be found on white or black squares but not both. So we only keep the squares of the most "used" colours. This way, if we detect the presence of something unusual like a finger, we might not take it into account.

The second idea uses again the Hough transform, but this time with circles. As the HoughLine feature, there is an HoughCircle that detects circles in the image. We used almost the same preprocesing as before, that is to say : blurred the image, used a canny filter, and used a moving threshold to not detect too many circles. Once the circles are detected, one can check that they are mainly inside a square by checking that the points obtained by center $\pm$ radius /3 (arbitrary) are still in the square. This method has proven to be more accurate in many cases than the first one, that is why we used it in the final form of the program.

Though it is not perfect due to sometimes unclear distinctions between pawn and chessboard, it rarely fails in more than 2 pawns of the chessboard (cf. Figure 10).

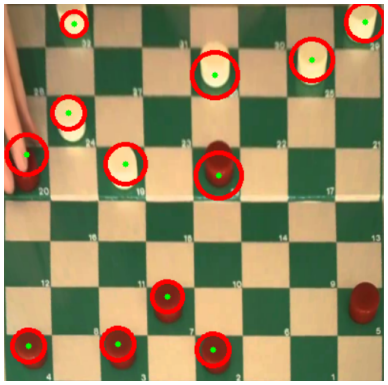

Figure 10: Circles detected using the HougCircle feature of OpenCV

Pawns' colour

Once we have detected every pawn on the board, we can simply determine the colour of each pawn by taking the mean colour of each square containing a pawn. White ones correspond to colour values superior to a certain value (123 = 255/2 for instance) and black ones to values inferior to the same value. However, this does not work for pawns that are red or green (as seen in the examples above). Instead of 123 as the threshold, we can separate the squares of the 8 by 8 output in which we detected a pawn into two batches of colours thanks to a k-means

algorithm (with k = 2), and using RGB colours instead of just a 0-255 value. The lighter one corresponds to the squares with white pawns and the darker one, with the black pawns.

Some results are shown in Figure 11,12 & 13. In the matrix, -1 corresponds to white pawn, 1 to black ones, and 0 to no pawn

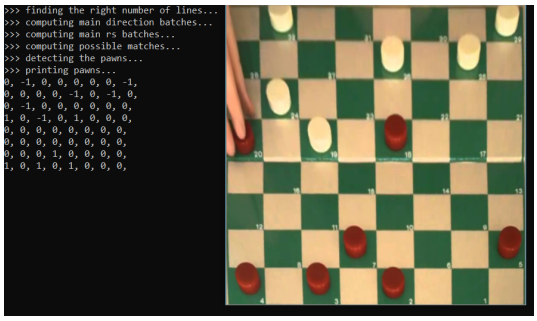

Figure 11: One pawn is not detected on the bottom right corner

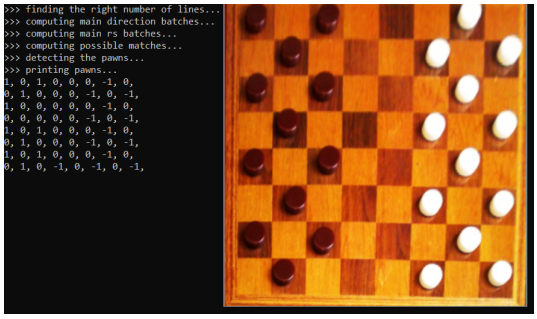

Figure 12: One pawn is detected while it should not

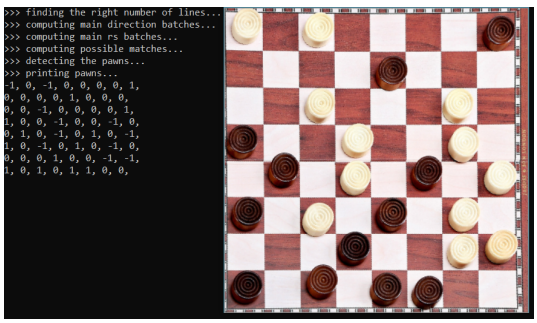

Figure 13: Same as in Figure 12

Conclusion and further work

To conclude with, using OpenCV tools and implementations of Canny edge detection, Hough line detection, and Hough circle detection, we were able to obtained really good results. Over the 7 examples we tested our program, in only one case is the program unable to achieve a good perspective. Really often, there is at more one pawn missed or added and the colour of the pawn is quite accurate (never fails in the 7 examples). One case that remains a clear issue is the fact that the program tends to create a wrong board with a false line as shown in the Figure 9. One way to get rid of this issue would be to improve the scoring system and to add a score based on RGB colours for instance.

Moreover, although it is not directly in link with Computer Vision algorithms, the program would be really quick to adapt into a smartphone application, given that React Native framework allows to include C++ code into an app. By improving the algorithms a little, it is thinkable to make it work for 10x10 chessboards and by recognising different chess pawns (e.g. king, queen, knight, etc. though it is trickier), it could be implemented into a chess app, allowing players to take a picture of their "real-life" game and save their plays. To go further, it could even allows to recreate entire chess games from a video. In short, there are many possibilities and improvements to such a program, and we are sure they will quickly appear in our phones and computers.