Abstract

This paper presents a project that implements real-time computer vision to allow OTB(Over the board) chess games to be played online. The complete solution is a compact system that operates on mobile devices, perceiving game state directly from the device camera and translate it to digital as well as playing moves against an online opponent by integrating with Chess site's APIs. An algorithm which detects chess moves was proposed. It compares two or more key frames that defines important states on the chess board, before, during and after making a move and detect pieces' motions based on the frame differences. Further image processing was implemented to reduce the effects of noise, image distortion and perspective differences with the aim of building a system which performs well under minimal constraints.

Below is a summary of my main contributions towards this project:

- I wrote a total of 3000 lines of source code, comprising a Java openCV real time motion detection model, a GUI (in React Native Javascript) and a back-end server in Python that integrates with Lichess APIs.
- I spent 100 hours researching and evaluating existing solutions to computer vision in Chess.
- I implemented a version of the Morphological based moving object detection with background subtraction method proposed in [22].
- I spent 100 hours collecting data, evaluating the effects of varying environments on the system performance.

Dedication and Acknowledgements

I would like to thank the following people for their help and support:

Ethics Statement

This project did not require ethical review, as determined by my supervisor, [Pui Anantrasirichai]

Supporting Technologies

This thesis has made use of the following third-party open-source software technologies:

- I used a parts of the OpenCV[18] computer vision library to capture images from a camera, and for various standard operations including threshold, edge detection, homography transformation, and feature detection.
- I used the open-source React Native [8] to build the mobile-interface component of my application.
- I used Python Websocket [7] to create a websocket server
- I used the official open-source Lichess API library Berserk[1] to connect with Lichess.
- I used Matplotlib[10] to create and visualise graphs and figures.

Notation and Acronyms

Throughout this thesis, I will use the following notations and acronyms:

OTB	:	Over the Board
DGT	:	Digital Game Technology
ROI	:	Region of Interest

Chapter 1

Introduction

The classic board game chess, which has been around for over 14 decades, has been seeing a rapid resurgence in popularity over the past couple of years due to the Covid-19 pandemic. The chess platform chess.com had nearly 30 millions members before the pandemic broke out in March, 2020. This number is currently over 85 millions as of April, 2022 with over 5 millions new members in the past 3 months [27]. While the game of chess has greatly benefited from rapid technological advancements, including new analytical chess engines, online playing environments and huge game databases, there have been relatively few advances in improving the experience of playing OTB chess. The full experience of playing on a physical chess board (e.g. tapping the clock, moving the pieces,..), which is highly respected among chess players, is often overlooked by developers.

I aim to improve this aspect of the game by creating a robust and compact system, which allow users to experience the fullness of playing on their favourite physical chess boards, while maintaining the convenience of online playing environments. In particular, by leveraging existing computer vision algorithms and open-source projects, I would like to build an end-to-end application that comprises of:

- A computer vision model that can detect chess moves on a physical board.
- A Text-to-speech subsystem that reads the state of the game including online opponent's moves and prompting users for illegal moves or false move detection.
- A subsystem that integrates with large online chess sites that provides necessary APIs to play a full game of chess.

Different strategies for the computer vision model at several stages will be discussed and investigated. The robustness of the system will be tested against varying environments including different light conditions and board angles.

1.0.1 Challenges

The typical impediments in digitalizing chess moves includes complex background, effects illumination and shadows, perspective distortion and occlusion. A subset of these will be discussed below.

illumination and shadow

In standard lighting conditions, pieces on the chess boards can cast shadow on the board surface. Algorithm that fails to account for this will produce false detection as shadows can often be misclassified as another piece occupying the squares. **??** shows the absolute difference between 2 key frames. one taken before the move and one taken after the move's been made. Because of

the long blurry tail produce by the shadows, determining the piece position on the chess board from this image is no trivial task.

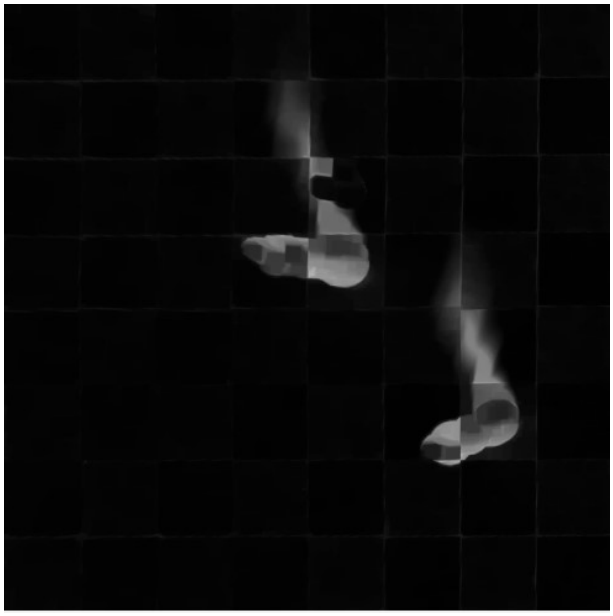
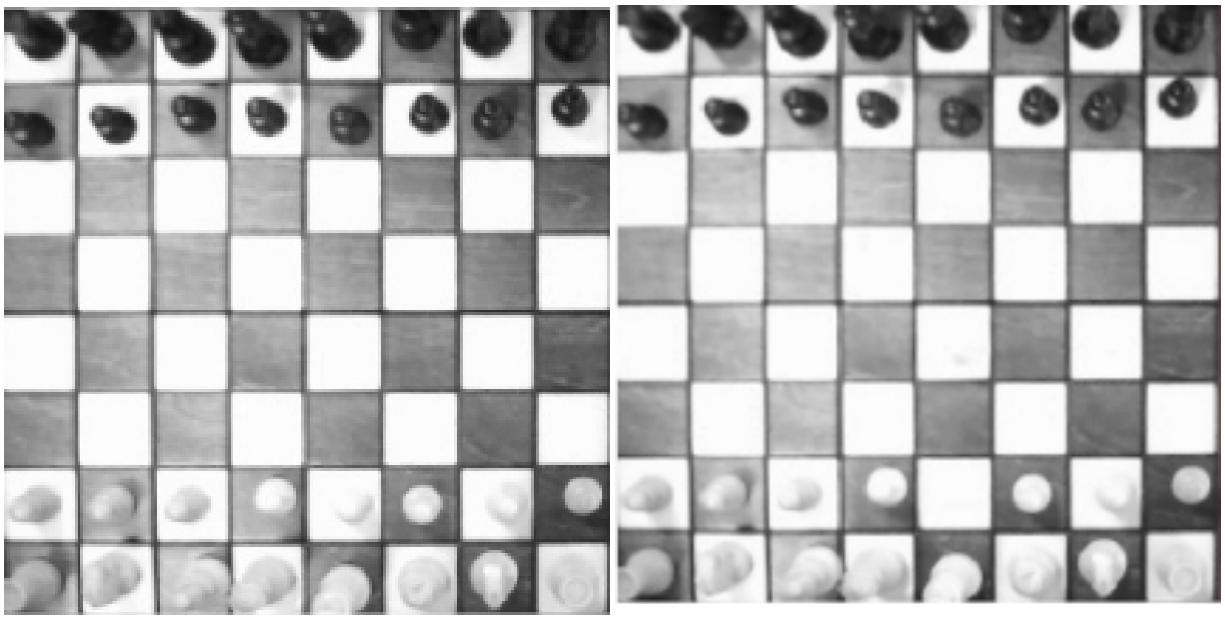


Figure 1.1: Absolute difference of 2 key frames

In another light condition where there is too much illumination, some systems. including human eyes, struggle to see a piece positioned at the illumination point. Figure 1.2 shows 2 frames captured before and after making the move e2e4 on a highly reflective board. Can you spot the pawn on e4?



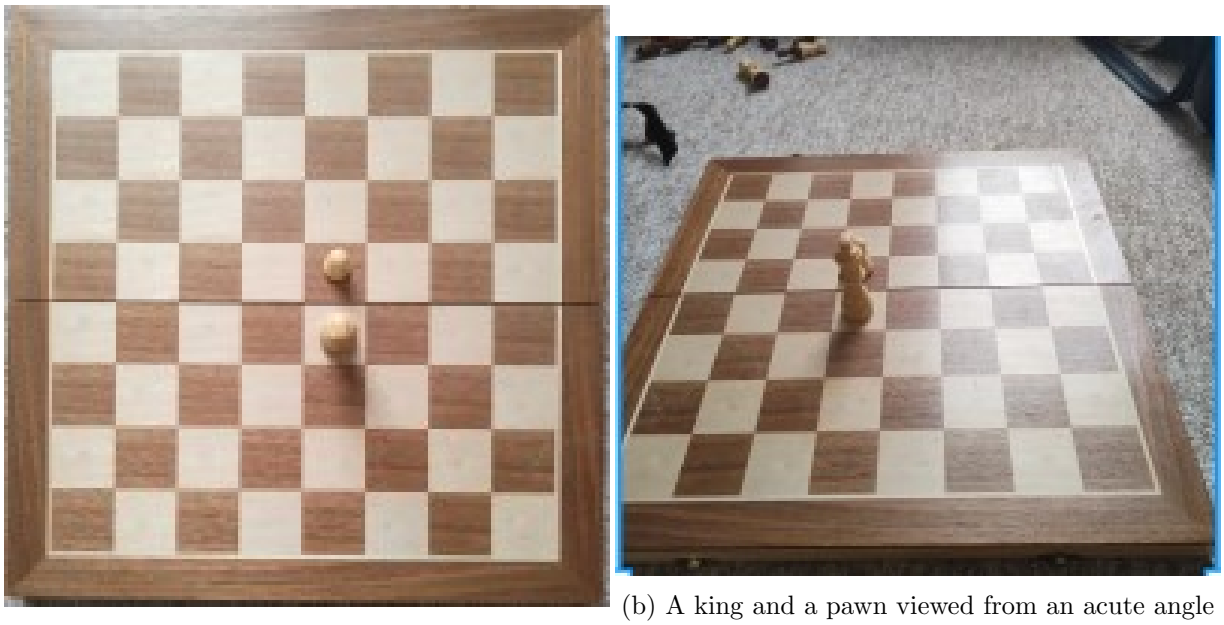
(a) Frame captured before making the move (b) Frame captured after making the move

Figure 1.2: Illumination problem, chess move e2e4, images from another study[12]

Occlusion

From certain angle, some pieces might be occluded by other pieces and appear hidden to the camera lenses. Without any additional information, the motion of those pieces will be completely

silent, and many algorithms will fail.



(a) A king and a pawn viewed from top down angle

(b) A king and a pawn viewed from an acute angle

Figure 1.3: Illustration of the occlusion problem

1.0.2 Related work

There are many published systems related to autonomous chess playing. However, the majority of these were developed as subsystems of an expensive solution such as Raspberry Turk[24] or DGT eboards. However these solutions are not easily accessible to the chess community and often requires a lot of set up and configurations that are not available in standard environments. An average DGT eboards also requires the users to pinch down the piece in order to register a move, taking away the naturalness of piece movement in OTB chess. Other solutions, carried out by independent hobbyists, independent of established research institutions, rely on heavily on simplifications of the challenges such as:

- The board is perceived at bird-eye view, leading to minimal perspective distortion.
- Dual or multiple cameras, leading to negligible occlusion.
- Controlled lighting conditions, with minimal shadow and illumination problems.
- Static detection: The user has to manually notify the system when a move has been made.

There are a number of chess move detection algorithms, which assumes 1 or more of the above simplifications, already exist in literature. In a study conducted by Ismail M. Khater and Ahmed S. Ghorab, a chessboard recognition system, which identify the name, location and the color of the pieces, was proposed [23]. The system extracts relevant features that define different square types (empty, or occupied by a piece) on a 2D chess board and train a classifier that matches these features with their respective square type. M. Piškorec presented a computer vision system for the chess game reconstruction [25]. The system solve the occlusion problem by using 2 cameras, one mounted at the top of the chess board and one on the side of the chess board. Maciej A. Czyzewski proposed a novel algorithm for digitizing chess board confiurations with the support of neural networks [20]. In another study by Sokic and Ahic-Djokic[26], a more simple, low-cost computer vision system for chess playing robot was discussed. The algorithm compares and analyses key frames in order to define a played chess move. The concentration of

this project will therefore, not to study new chess computer vision algorithms, but to implement an extended version of the above proposed algorithm and build an end-to-end system that is easily accessible to the community. I aim to solve only a subset of the problems discussed above, and make simplifications where only minimal extra setup overhead is required. In particular the final system is expected to be pragmatic and useful as a general-use program that maximizes robustness while requiring minimal setup overhead. The detail approach taken will be discussed in later sections of this report.

Chapter 2

Technical Background

2.0.1 Terminology

Board coordinates: a (2x1) vector that represents a square on a chessboard. e.g. (0,0) represents the bottom left corner a1 of the chess board and (8,8) represents the top right corner h8 of the chess board. More on standard algebraic notation in chess [15].

2.0.2 Proposed Approach

As discussed in the previous section, I will be implementing a version of the algorithm proposed by Sokic and Ahic-Djokic. This approach works based on the Morphological based moving object detection with background subtraction method proposed in [22]. The general idea of the approach is that several frames that defines states of a chess move are analysed and compared with each other, through which information about piece's motion can be extracted.

2.0.3 Devices setup

The finished system is a compact application that runs on a mobile device and requires minimal initial setup overhead. The system will assume that the phone camera will be capturing frames from the side of the board and from the camera perspective, white is always on the right side. This is an important simplification that will help us solve many problems that will be discussed later in this report. The chess board and pieces are also expected to have standard color and size. In the setup I used to test the system, the phone is positioned at the side of the chess board by a phone holder looking at the chess board at a 60degree angle. This is not a requirement, however the system shows better detection accuracy on higher camera angle due to minimal occlusions.

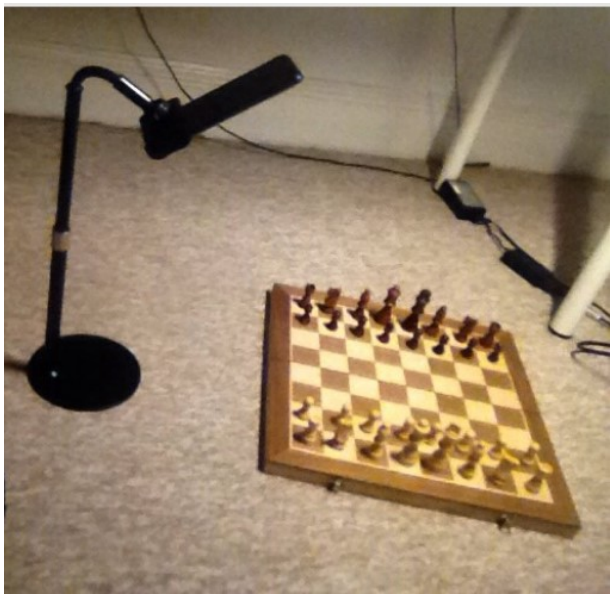


Figure 2.1: Example image of the setup used to test the system

2.0.4 A high level overview of the framework

Figure 2.2 describes the general framework that I will be following in this project to build the computer vision model for the system. The framework starts at the initialization phase where the first frame, which will be used as a reference point to detect motion is saved. In the subsequent frames, we try to locate the 4 outermost corners of the chess board. The location of the found corners are then used to find the Homography transformation matrix, with which we can perform image rectification that isolates the region that contains the chess board from the rest of the frame. The system waits until a hand is detected inside this region, during which a move is potentially being made. When it is finally safe to analyse the key frame (hand has left screen), the system performs motion detection by first computing the absolute difference between the current frame and the reference frame. By applying further image processing techniques, we can easily obtain the coordinates of the move, represented by the non-black pixels on the resulting image. The final steps involve matching the resulting difference image with the reference frame and the current frame to obtain the exact location of the piece and where the piece has moved to. Generally, the region where the difference image matches with the reference image is where the piece was originally; the region where it matches with the current frame is where the piece has moved to.

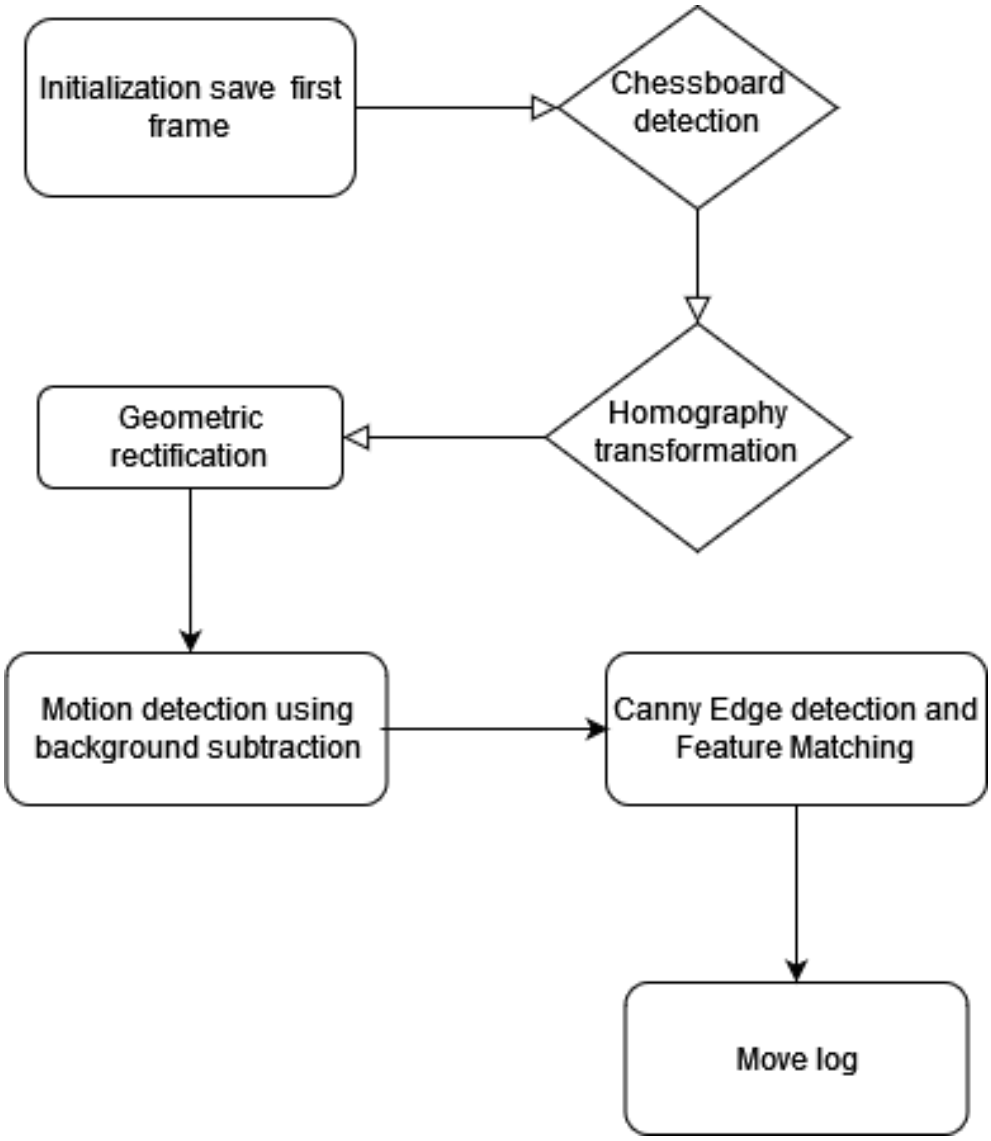


Figure 2.2: High level diagram of the computer vision subsystem

2.0.5 Background subtraction

Background subtraction is a widely used technique which allows the image's foreground to be extracted from a static background for further image processing. . The rationale in the approach is that of detecting the moving objects from the difference between the current frame and a reference frame, often called "background image", or "background model" [17]. The convention approach in implementing this is to take an image as background and take the frames obtained at time t, denoted I(t) to compare with the background image denoted by B. We can segment out the objects by simply taking the difference between the pixel values of I(t) denoted P(I(t)) and the pixel values of B denoted P(B). In mathematical equation, this can be written as:

$$P[F(t)] = P[I(t)] - P[B]$$

F(t) will only show pixel intensity for pixel values at the locations, which contains the foreground object we want to segment. This is then compared with the difference at time t' F(t') by again taking the absolute difference between the 2 frames:

$$P[D(t)] = |P[F(t)] - P[F(t')]|$$

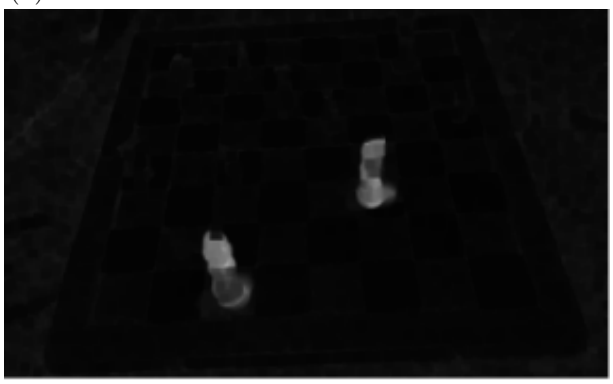
which can also be written as:

$$P[D(t)] = (P[I(t)] - P[B]) - (P[I(t')] - P[B]) = P[I(t)] - P[I(t')]$$

The difference image D(t) would only show pixel intensity for pixel locations which have changed in the two frames. Further image processing techniques such as median filter or image thresh holding can be used to reduce noise and better segment the regions that show difference. The difference image D(t) is concluded to contain motion if it's pixel intensities are higher than some threshold value, which can be set statically or computed dynamically.

(a) Frame at time t. (Reference frame)

(b) Frame at time t' after the move has been made



(c) Computed absolute difference D(t)

Figure 2.3: Frame differencing in background subtraction

2.0.6 Homography transformation

In the field of computer vision, any two images of the same planar surface in space are related by a homography (assuming a pinhole camera model).[13, 14]. In other words, a homography is a mapping between two planar projections of the same image. This mapping is represented mathematically as a 3x3 transformation matrix in a homogeneous coordinates space as:

$$\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = H \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} = \begin{bmatrix} h_{11} & h_{12} & h_{13} \\ h_{21} & h_{22} & h_{23} \\ h_{31} & h_{32} & h_{33} \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} \quad [6]$$

In a coordinate plane, it looks like the following:

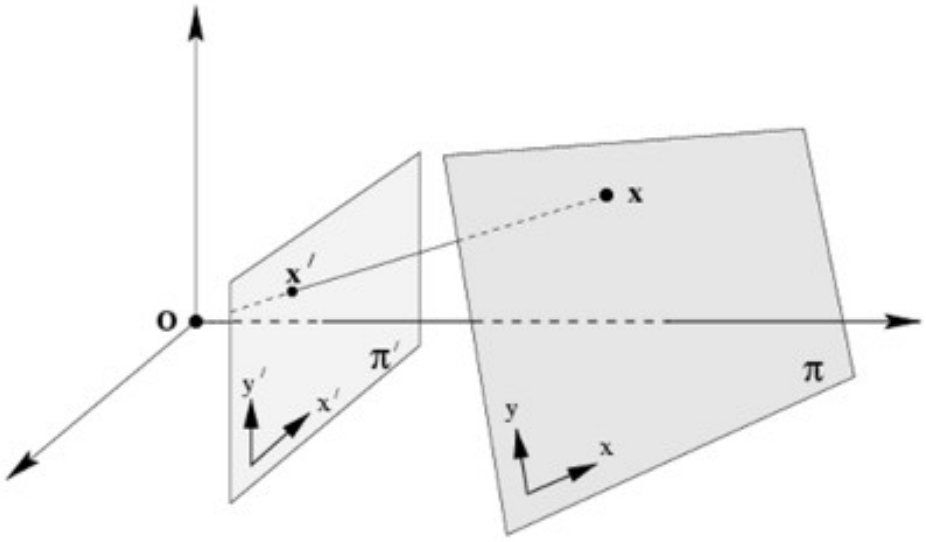


Figure 2.4: transformation between 2 planes
[21]

As observed in the illustration image, every pixel on a plane has a corresponding projection on another plane in a homogeneous coordinate space. The resulting projection of the original plane contains the same information, but in a transformed perspective.

2.0.7 Morphological operations

Morphology is a broad set of image processing operations that process images based on shapes. Morphological operations apply a structuring element to an input image, creating an output image of the same size. In a morphological operation, the value of each pixel in the output image is based on a comparison of the corresponding pixel in the input image with its neighbors. [**?**]. The most basic morphological operations are Erosion and Dilation. They have a wide array of uses, a subset of which I am going to be using in my implementaion including noise removal and isolation of individual elements. The operation of dilation and erosion consists of scanning the kernel B over an image A. The value of the of the pixel value on A, overlapped by B, is replaced with the new computed value. In dilation, we compute the maximal pixel value on A overlapped by B, whereas in erosion we calculate the minimal pixel value on A overlapped by B. Intuitively, dilation causes bright region in an image to "grow" as nearby pixels are filled with the local maximal pixel intensity. In erosion, we can observe an opposite effect.

(a) Original image (b) Dilated image (c) Eroded image

Figure 2.5: Illustraion of dilation and erosion, taken from [3]

2.0.8 Image Thresholding

The most basic form of image thresholding involves comparing each pixel (i.j) 's intensity value $I_{i,j}$ with a fixed value called the threshold T. The pixel (i,j) is replaced with a white pixel if $I_{i,j}$ is greater than T or with a black pixel if the pixel intensity is smaller than T. The results in a binary image (containing only black and white pixels) that can be used as inputs in many image processing frameworks, including motion detection. The threshold T can be computed and set manually by the user, or automatically by an algorithm. The most famous and widely-used automatic thresholding algorithm is Otsu's Binarization. Consider a bimodal image with 2 peak a,b (i.e: the majority of pixels has intensity values either close to a or close to b). A good threshold would be in the middle of a and b. Otsu's algorithm finds a value of T which lies in between two peaks such that variances to both classes, separated by T are minimal. We will not dive deep into the mathematical details of algorithm, but intuitively, Otsu's Binarization will try to find the threshold value T so that the variance on both size of T is minimized.

2.0.9 Median Blur

Image blurring(image smoothing) is a process of removing high frequency content from an image by convolving the image with a low-pass filter kernel. (i.e. keep low frequency). Blurring is an important preprocessing step before performing thresholding and edge detection algorithms. The idea is that, when we blur an image, we smoothes out rapid changes in pixel intensity from one side of an edge to another, reducing the number of low-information edges produced by noise. In the context of thresholding, blurring will removes the "salt and pepper" noise that we don't want to include in our resulting image. Median blur is a type of blurring operation where the central element is replaced by the median of all the pixels in the kernel area. Gaussian blur determines the value of the central element by computing the 2D Gaussian function of the pixel values overlapped by the kernel.

2.0.10 Canny Edge detection

Canny Edge Detection is a popular edge detection operator that uses a multi-stage algorithm, developed by John F.Canny in 1986. [16]. An evaluation of the edge detection algorithms shows that canny edge detection algorithm is the most popular edge detector because of its simplicity and accuracy. [19].The algorithm has the following stages:

1. Noise reduction

   The first step is to remove the low-information edges produce by noise in the image by convolving the image with a Gaussian filter.

2. Finding Intensity Gradient of the image

   In this stage, a 3x3 kernel called Sobel is convolved with the original image to calculate approximations of the derivatives in the horizontal direction, denoted G_x and in the vertical direction, denoted G_y. If we define A as the region on the source image where the kernel overlaps. Approximations of horizontal and vertical derivatives at the central element can be computed as follows: [4]

$$G_x = \begin{bmatrix} +1 & 0 & -1 \\ +2 & 0 & -2 \\ +1 & 0 & -1 \end{bmatrix} * A \; and \; G_y = \begin{bmatrix} +1 & +2 & +1 \\ 0 & 0 & 0 \\ -1 & -2 & -1 \end{bmatrix} * A$$

   From this 2 values, we can find edge gradient and direction of each pixel as follows:

$$Edge\_Gradient(G) = \sqrt{G_x^2 + G_y^2}$$
$$Angle(\theta) = tan^{-}1(G_y/G_x)$$

3. Non-maximum Suppression In this stage, every pixel is checked against neighbors along its' gradient directions. Only the local maximums is retained while others are suppressed(put to zero).

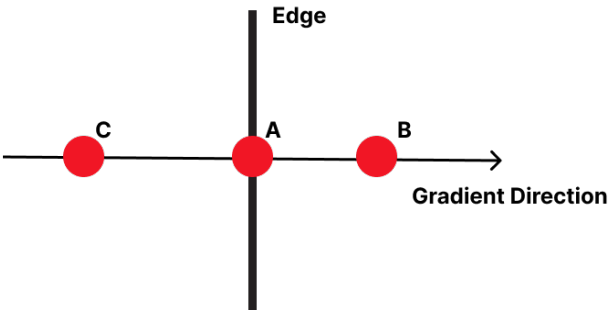


Figure 2.6: Non-maximum suppression

   Point B and C lies in the gradient direction of point A. So point A is checked against B and C to see if it forms a local maximum. If it does, point A must lie on an edge perpendicular to the gradient direction.

4. Hysteresis Thresholding [2]

   In this stage, we decide which edges to keep, and which edges to be discarded by introducing 2 threshold values, minVal and maxVal. Any edges with intensity gradient greater than maxVal are sure to be edges, and those whose intensity gradient smaller than minVal are sure to be non-edges. Those with intensity between maxVal and minVal are considered valid edges if they are connected with a sure-edge.

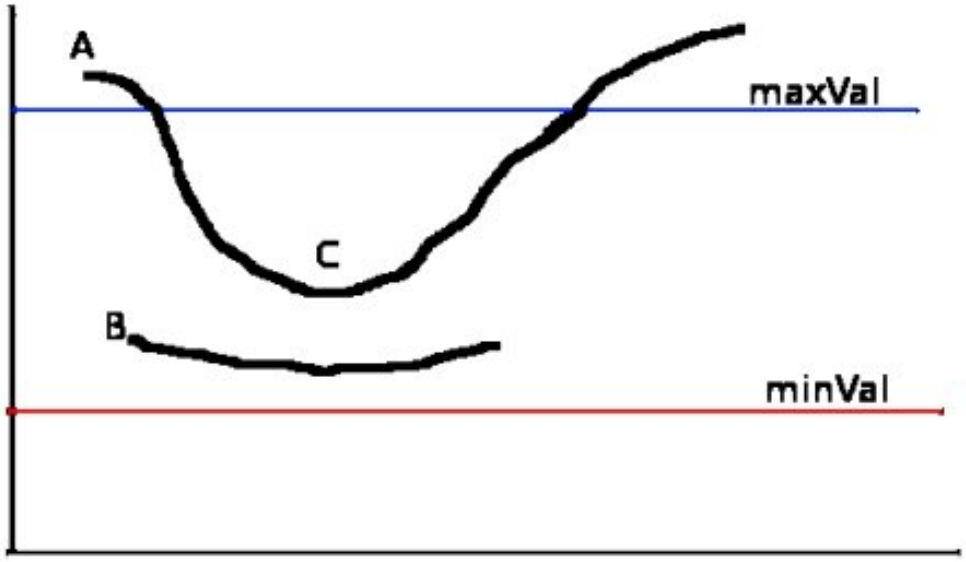


Figure 2.7: Hysteresis Thresholding
[2]

A is above maxVal, so it is considered a sure-edge. C lies between maxVal and minVal, but its connected to A, so it will also be considered a valid edge. B lies between maxVal and minVal, however, does not connect to any sure-edge, so it is discarded. [2]. In general, when implementing canny edge detection in openCV, we only have to select the suitable value for minVal, maxVal and the size of the Gaussian filter kernel based on the problem at hand, the remainder of the algorithm is carried out automatically by openCV.

2.0.11 Template matching

. Template Matching is a method for searching and finding the location of a template image in a larger image []. In openCV implementation, the template image is simple slided over the input image (as in 2D convolution) and compared with the patch of the input image underneath it. The return values contain information about the region on the input image that best matches with the template image.

2.0.12 Feature extraction and feature matching

A feature is a piece of information containing specific structures in the image such as points, edges, or objects. In computer vision, feature detection and matching are the building blocks of many e.g.object detection, structure-from-motion frameworks. There are many well-established system for finding features in an image such as SIFT, SURF, FAST, or ORB and algorithms for matching features such as Brute-Force Matcher of FLANN. In this project I will be using ORB descriptors and Brute-Force Matcher. Intuitively, ORB descriptor encode interesting information (features) into a matrix that acts as a numerical footprint that uniquely defines that feature. Brute-Force matcher takes the descriptor of one feature in first set and match it with all other features in second set using some distance calculation. And the closest one is returned. [5].

Chapter 3

Project Execution

3.0.1 Design

This chapter covers the design of the system to match the aims and objectives laid out in previous chapters.

Software architecture

As discussed in previous chapters, I aim to build a system that is as compact and requires the least amount of setup possible while maintaining a reasonable degree of accuracy and robustness. This project will be implemented as a mobile application, following the client-server model. []. The high level overview of the application architecture is illustrated in the diagram below.

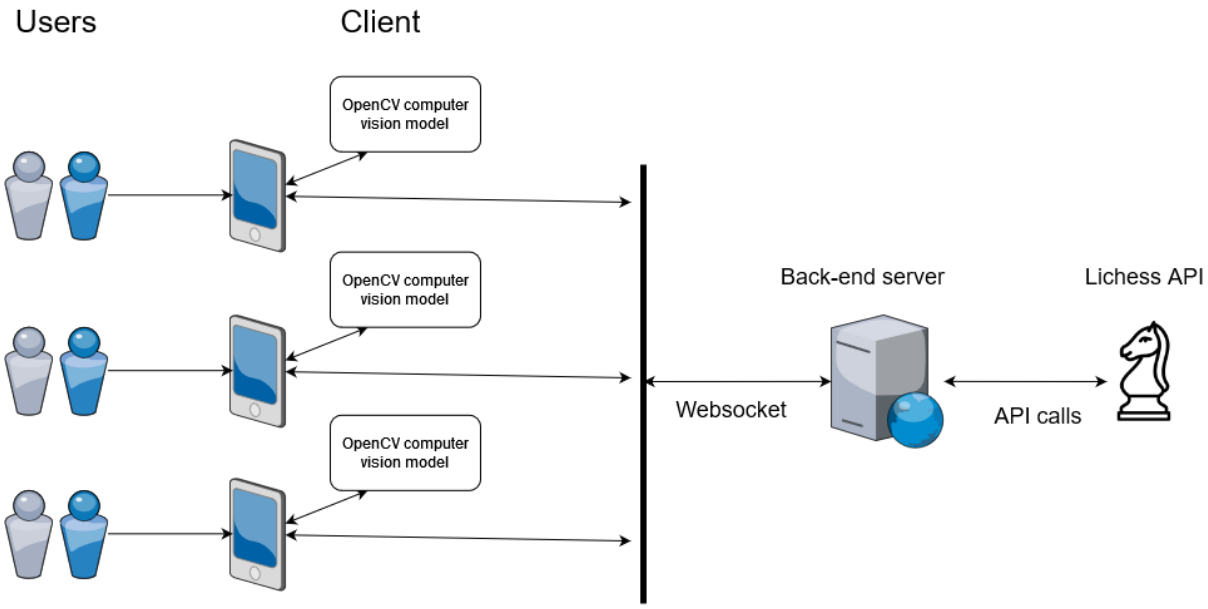


Figure 3.1: System architecture

User flow

The user flow should be simple and intuitive. Flexibility(users can take different paths to the same feature) is also a requirement in many applications' UX design. However, due to time limitations, I only built a 1 direction user flow path illustrated in the diagram below.

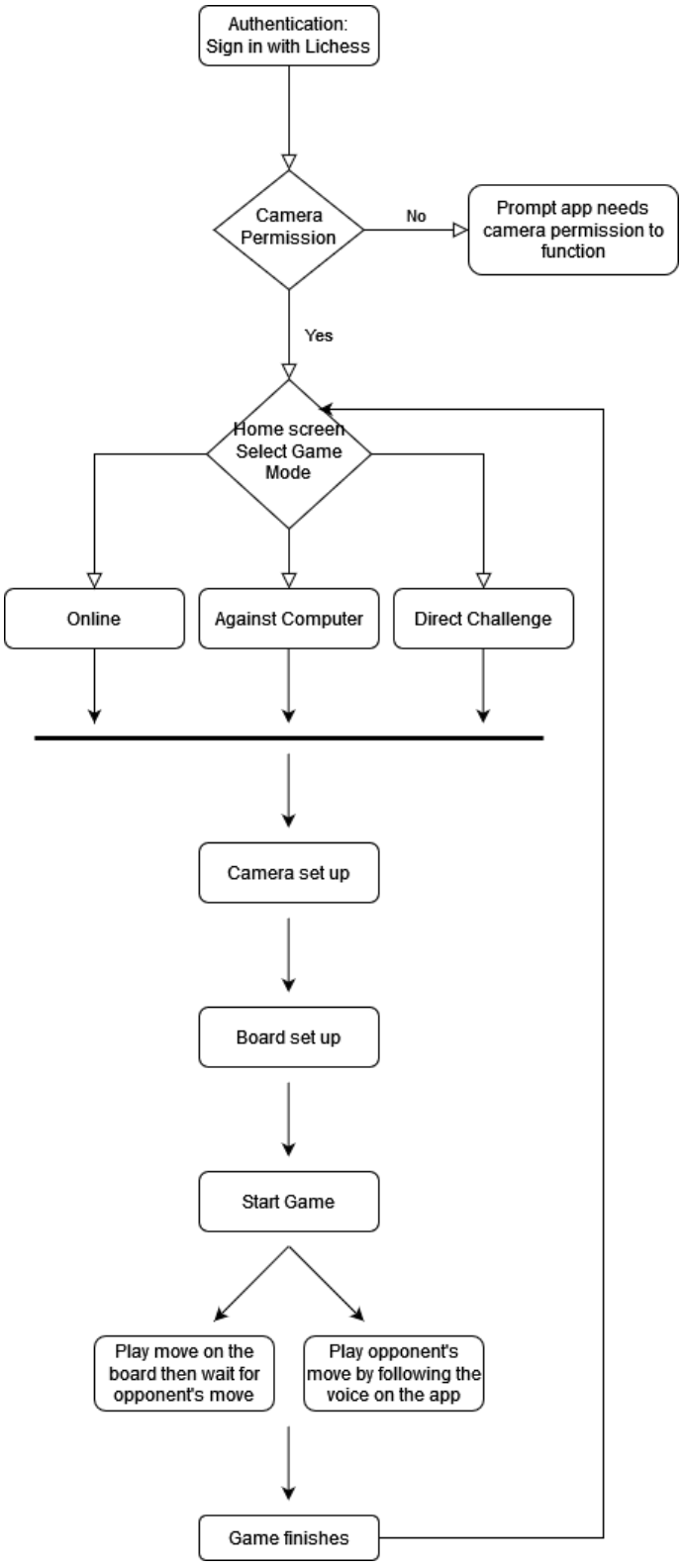
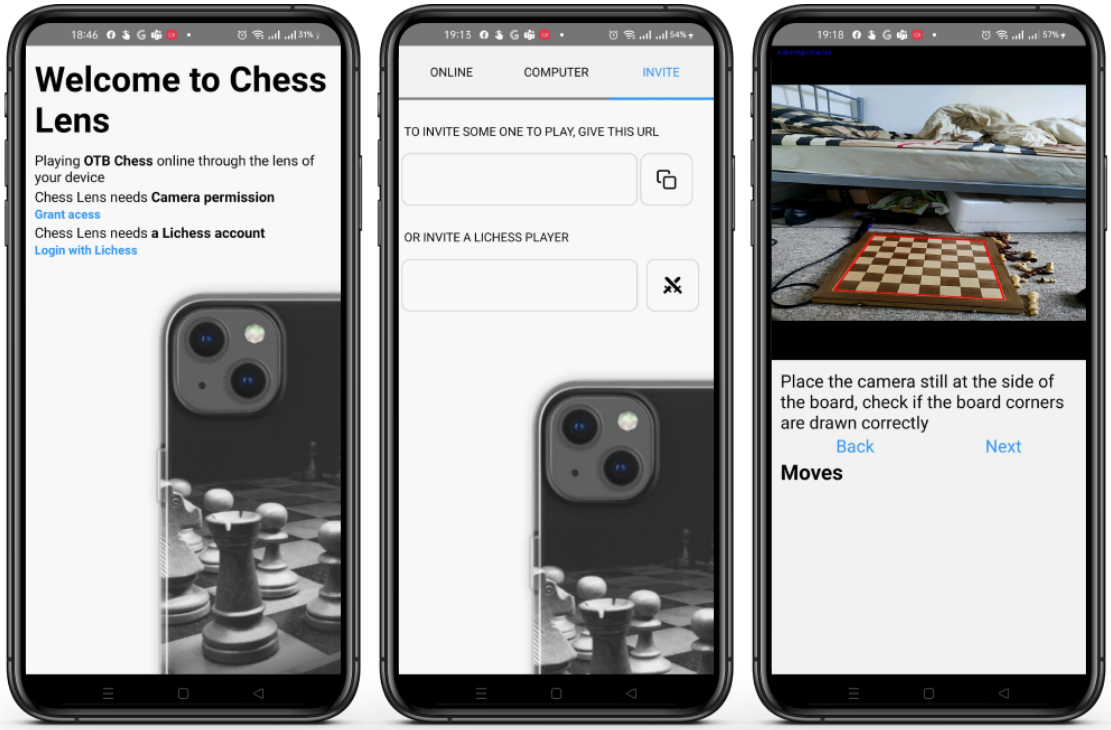


Figure 3.2: User flow diagram

Front-end

The front-end of the application is built with React Native framework. Notice that the OpenCV computer vision model is drawn as a separate component in the system architecture. This is because the UI logic and the communication with the back-end server is written in Javascript,

while the computer vision model is written separately as a native module in Java. They communicate with each other through a channel known as the "bridge", and on a deployed system, they are compiled into a single AAB [], which can be installed and run on a single client side device. When user opens the application, they are greeted with a login screen where the app requests camera permission and a Lichess account 3.3a. Sucessfull logins will be redirected to a game-like home page where the user can select between different modes including playing against a computer, playing against a random opponent or directly challenging another player, all of which are supported by Lichess API. At the time of writting this report, the application supports playing through direct invitation only 3.3b. Before starting a game, the user will be guided through a camera setup stage 3.3 where each step is detailed. Detected moves and game events will be generated under **Moves**.



(a) Login screen (b) Challenge screen (c) Game screen

Figure 3.3: Example screenshots of the application

Back-end server

The back-end server is written in Python. The back-end should supports bidirectional communication because game events such as opponents moves picked up on the client side. This can often be achieved with SSE (Server Sent Event), HTTP streaming, HTTP polling or Websockets. In this project I will be using extensively the websocket protocol.

Unlike most systems, which use a full fledged REST framework like Django, I used a lightweight library websocket to set up the back-end server and manages connections purely through the websocket protocol. A system built this way is sub-optimal since websocket puts a lot of demand on the server and limits its scalability (This will be discussed further in later chapters). I have selected this approach primarily because of the limited development time I had.

I will follow the consumer & producer design pattern for processing messages in a Websocket server illustrated in the official documentaion. [11].

Computer vision model

As described in the previous chapters, the computer vision model will be developed separately as a native module. In java Android (implementation on swift will be similar), this is a class that is responsible for managing the camera view rendered in 3.3. This class implements an interface from the openCV library and is responsible for the following task in the framework:

1. Load OpenCV, initiates the camera view.
2. Continuously read frames by overriding onCameraFrame method.
3. Before the start of a game, detect and save homography matrix and board corners' locations.
4. At the start of a game, save the perspective corrected initial frame.
5. Every subsequent frames will be first perspectie corrected using the saved homorgraphy
6. Detect if there is a hand above the chess board on the frame by using background subtraction.
7. Detect if the hand has left the frame by using background subtraction. At this point, a move is assumed to have been made.
8. Comparing the current frame with the initial frame to find the move made.
9. Return the move made, push it to the chess state and play it on Lichess.

3.0.2 Implementation

Finding the chess board and the board-image Homography

The function Calib3d.findChessBoardCorners attempts to determine whether a chessboard pattern is in the input image and locate the all the internal chessboard corners. Running findChessBoardCorners on a frame containing an empty chessboard will return the following

Figure 3.4: drawn by OpenCV draw chessboard corner

We are only interested in the location of the outermost corners returned, marked red. Our goal is to find the outer corner of the chess board. We can do this by first perform a homography

transformation, mapping the top left corner with (100, 100), top right corner with (700,100), bottom right corner with (700,700) and bottom left corner with (100,700) from the initial plane to a resized (800x800) plane. (see illustration).

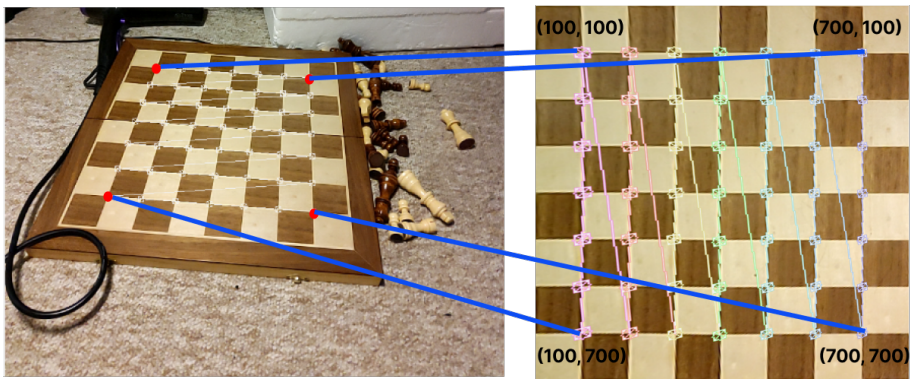


Figure 3.5: illustration of perspective correction

The four corners of the resized plane will match the four outer corners of the chess board. We can than perform a inverse perspective transformation to figure out the location of these corners on the original frame. We then draw lines connecting the found corners and show them on the UI.

This is the first step in the setup process, after this step, the camera view is assumed to be fixed and the homography and corners are saved. If the camera view changed, the user must go through this step again.

Hand enter/left screen detection

On every camera frame F, we will first perform perspective correction and geometric rectification using the saved Homography matrix to segment out only the interesting region of the frame. i.e. The region containing the chess board. We will implement background subtraction method to segment out foreground object on the current frame. 3.6a shows the result of the operation on a frame containing a hand. We can than apply a median blur before thresholding the difference image to better segment out the foreground object and get the image R as seen in 3.6b. Following this procedure, on every frame we can compare the average pixel intensity of R, denoted P(R) with a threshold value T, and assume that a hand is in F if $P(R) > T$.

The procedure for detecting when the hand has left the screen is similar. The system assume that the hand has left the screen if the difference between the frames is smaller than some value T.

(a) Absolute difference between the current frame and the initial frame

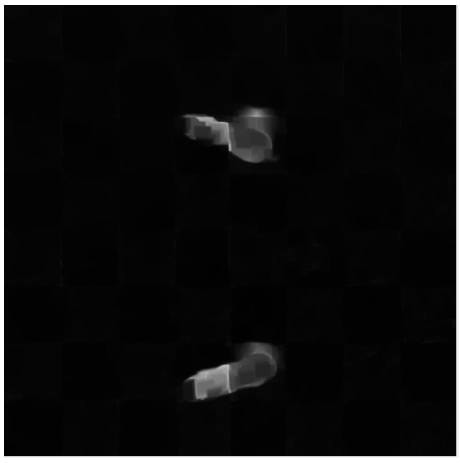
(b) After further processing operation

Chess move detection

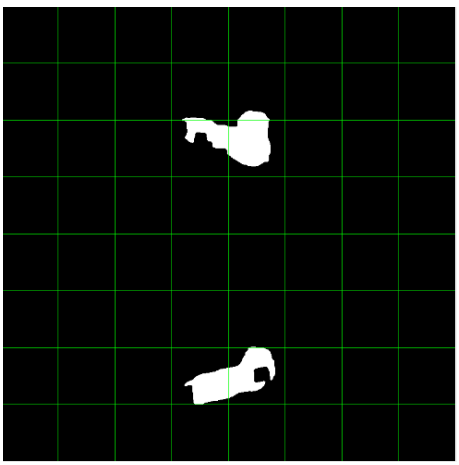
The implementation of the move detection is based on frame differencing. This move detection function only runs after a hand that was detected inside the screen have left screen. This is to make sure that the algorithm only runs on stable key frames which contains the move information. We start by computing the absolute difference D(t) of the current frame and the saved reference frame. The difference image after performing perspective correction is shown in **??**. Next, we will do some post processing to segment out the pixel intensities of the move and those produced by noise. We will first apply a morphological operation by first dilate and then erode the difference image D(t). This is called a "closed" operation. We then remove the small isolated dots in the image (a.k.a 'salt and pepper' noise) by applying a median filter. Lastly, we will threshold the image to make the object more prominent.

```
diff = Core.absdiff(mGray, InitialFrame);
corrected_diff = warpPerspective(difff, perspective
, new Size(800,800));
kernel = Mat.ones(10,10,CvType.CV_32F);
dst = morphologyEx(result, Imgproc.MORPH_CLOSE, kernel);

blur = medianBlur(threshHolded, 5);
```



(a) Absolute difference between the current frame and the initial frame



(b) After post processing operation

Next, we need to identify the location in board coordinates of the piece that has moved. The easiest implementation is a for loop going through each square on the board, a square is assumed to contain the piece if it has more than threshold T of white pixels. Due to perspective, a tall piece may span a lot of squares, so our goal here is group all squares containing the piece into 1 of 2 groups, one corresponds to the location of the piece before the move, one corresponds to the location of the piece after the move. i.e 2 white pixels islands on the image. This is a classic and well studied finding disjoint sets problem in data structure and algorithms and can be solved with the following union find approach implementation:

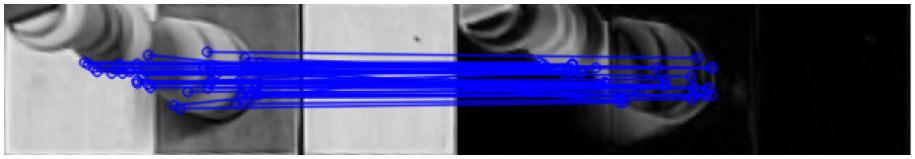
```
UnionFind uf = new UnionFind(image);
for (int r = 0; r < 8; ++r) {
  for (int c = 0; c < 8; ++c) {
    if (image[r][c] contains piece) {
      if (r - 1 >= 0 && grid[r-1][c] contains piece) {
        uf.union({r, c}, {r-1, c});
      }
      if (r + 1 < nr && grid[r+1][c] contains piece) {
        uf.union({r, c}, {(r+1), c});
      }
      if (c - 1 >= 0 && grid[r][c-1] contains piece) {
        uf.union({r, c}, {r, c - 1});
      }
      if (c + 1 < nc && grid[r][c+1] contains piece) {
        uf.union({r, c}, {r, c + 1});
      }
    }
  }
}
return uf
```

An example output for Figure 3.7b is Set1 = [(3,2), (4,2)] and Set2 = [(3,6), (4,6)]. Now we need to know the exact position of the piece in board coordinates. Because the camera is assumed to be capturing frames from one side of the chess board, if a tall piece spans multiple squares, its base must be at the square closest to the camera. In Figure 3.7b the camera is looking at the chess board from the right. Therefore, the base of the piece must be (3,2) and (3,6). We however, do not know which square match with which frame. i.e Did the piece in Figure 3.7b move from (3,2) to (3,6) or the other way around.

We will compare the ROIs (i.e square at board coordinate (3,2)) of the difference frame and the reference frame. If they match, the piece is assumed to have moved from (3,2) to (3,6). To achieve this, we will find edges in the ROI of both frames by applying Canny edge detection algorithm then apply feature matching on the Canny edged frames. This approach works because the edge information of the moved piece is preserved in the difference frame. Figure 3.8

(a) Canny edge of the Difference Frame's ROI

(b) Canny edge of the Reference Frame's ROI



(c) Matches detected

Figure 3.8: Feature Matching

Keeping the internal state of the chess game

The system can only detects motion i.e: The board coordinates of the moved piece, However it has no information regarding which piece (e.g King or Queen) has been moved. To overcome this, the system assumes that every chess game starts from a standard position [], and keep an internal state of the game. This internal state is updated whenever a moved is made. I used an open-source library chess.JS in order to achieve this.

Authentication

To connect to Lichess and start playing games, users will have to login to their Lichess account through Oauth. Lichess supports PKCE (Proof Key for Code Exchange) flow, which is a modern flow of Oauth2, which introduces a secret created by the calling application that can be verified by the authorization server. My implementation follows directly from the official documentation. [9]

Listening to Lichess Events

Lichess API only supports listening to live events such as opponent moves or game starts/finishes through HTTP streaming, which is blocking, therefore must be implemented as a threading operation on our back-end as it also needs to be communicating with the front-end in the mean time. Events listened on the front-end will be translated to speech by using React-native-tts library. With this set up, users can play a full game of chess without looking or interacting physically with the screen.

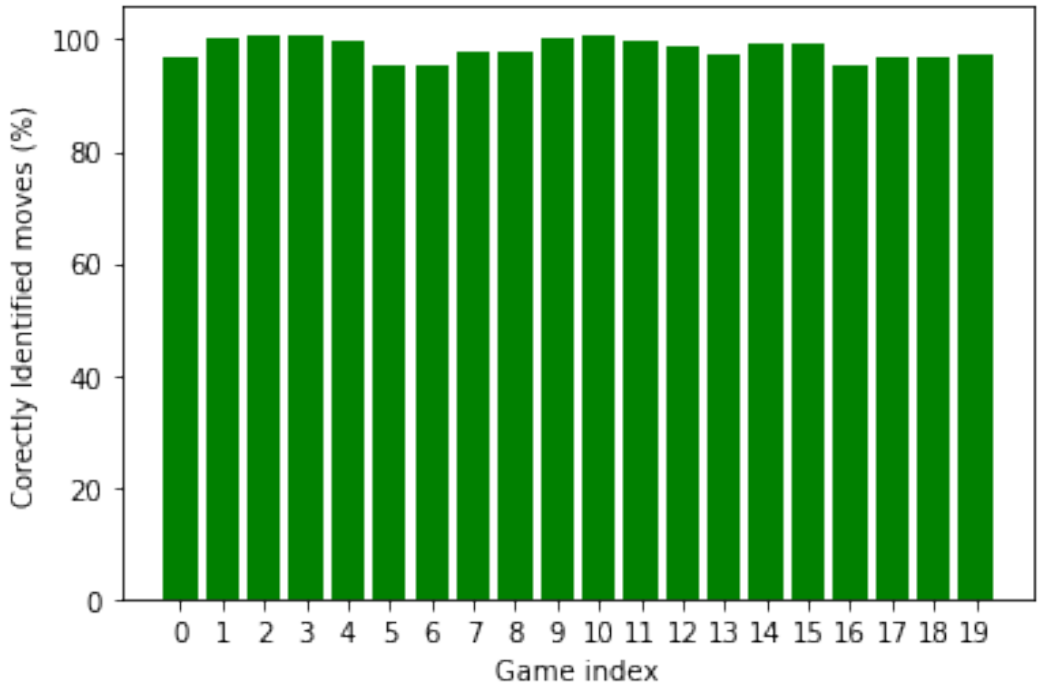
Chapter 4

Critical Evaluation

This chapter analyses the results of testing the application against varying environments, with the objective of assessing its performance and robustness.

4.1 Live game testing

I used the program to play a total of 20 complete games (approximately 1000 single player moves in total) under controlled environments and determine calculate the percentage of moves that the program was able to detect correctly. All 20 games were played in a dark room with 1 single light source right above the chess board. The light source was positioned far enough from the board so that chess pieces cast minimal shadows. The board was chosen to be non-reflective and the camera was positioned at 70deg angle to the center of the chess board.



The result was promising: The system was able to sucessfully detect at least 95% of moves in all games. In this regard, the system was found to be satisfactory to detect moves in real-time.

4.2 Effects of varying environments

4.2.1 Camera angle

In general, we expect the system's performance to be invariant under horizontal camera angle changes (Moving camera side to side). However, lower vertical camera angle (Camera view closer to chess board surface) should negatively affect the performance of the system.

In this experiments, 100 moves were played in the set up described in the previous section. However with the camera mounted at varying angles both in the horizontal direction and in the vertical direction. The results of the experiment are shown in Figure 4.1

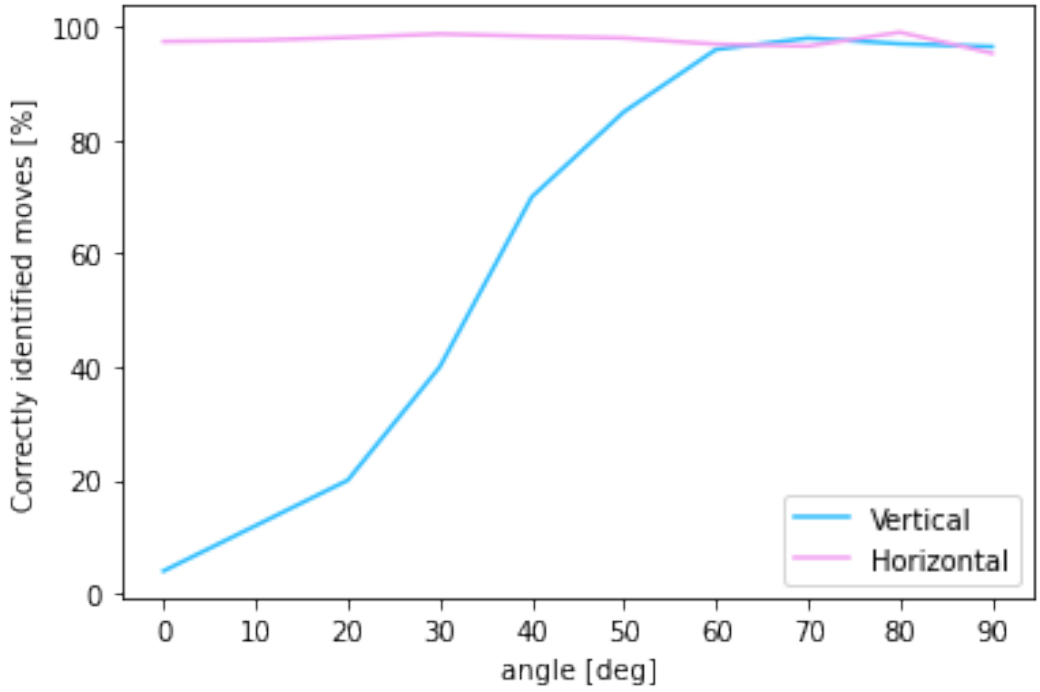


Figure 4.1

The graph shows that the horizontal angle changes have minimal impact on the performance of the system (assuming the vertical angle is unchanged at 70deg). The percentage of correctly detected moves remained above 95% at all angles. This is inline with the hypothesis proposed at the start of the section.

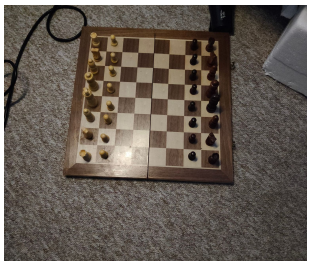
The graph shows a direct correlation between the performance of the system and the vertical angle of the camera. At lower angle (camera closer to the surface of the chess board), the system shows poor accuracy. This is primarily due to the occlusion problem, where the piece is almost or completely invisible from the camera perspective. At 50 deg angle or above, the system shows satisfactory accuracy. 50 deg is achievable in most standard setup (i.e: without a phone holder), suggesting the usability and accessability of the system to the community. Another interesting observation is that the fewer pieces there are on the board, the more accurate the system will be because there is less chance for occlusion to happen.

4.2.2 Lighting conditions

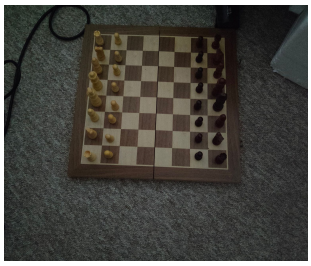
the first experiment I did involves checking the program's performance under "normal" lighting conditions. Lighting condition is considered "normal" when its likely to occur in real-world

scenario with standard setup. e.g: The program is used in a slightly darker or slightly brighter room. The program was tested at 3 different lighting conditions.

(a) lighter room



(b) darker room



(c) very dark room

Figure 4.2: Different lighting conditions

The performance of the program remained roughly the same in lighter and darker room (consistently at higher than 95%). However, in extremely dim conditions, some of the black pieces further away from the camera are misclassified as unoccupied dark squares. This decreases the performance of the system.

In the second experiment, I tried to create extreme lighting conditions in order to challenge the system. I used a highly reflective board and move the primary light source closer to the board surface so it introduces illumination. I also moved the light source to an acute angle to the board so the pieces would cast very long shadows. For this experiment, I also recorded the system ability to detect the board and correctly computed the homography.

Light conditions	Correctly detect board	Correctly detect moves
With illumination	90%	95%
With long shadows	100%	16%

Table 4.1: Caption

The first observation from table 4.1 is that the board detection maintains relatively high accuracy, despite extreme conditions. In lighting conditions that would create saturation, the system maintains high accuracy as only the moves made at the saturated region are affected. The system performs poorly when there are long shadows. This is an expected outcome and its one of the shortcomings of the system.

In a real-world scenario, it is unlikely that the lighting source would be so close to the chess board and the pieces' shadows are that prominent. The above experiment results suggest that the system's performance is satisfactory as a functional application.

Chapter 5

Conclusion

The aim of this project was to create an application that is meaningful and accessible to the general community by leveraging and improving on open-source libraries and existing works. Under standard environments, the application should have high accuracy with minimal setup overhead. This criteria has been successfully achieved as shown in the experiments.

In the system implemented the move detection robustness could potentially still be improved. The current implementation ignores the effect of shadows, therefore only works well in lighting conditions that produce small/no shadows and performs poorly in environments where pieces have long shadows. In less extreme lighting conditions, the experiments in the previous section shows that the system can deliver reliable accuracy in a range of lighting conditions. Some parameters passed into image processing operations were set manually, this is sufficient given the performance delivered, however, far from ideal. Potential future work includes implementing algorithms to find the optimal parameters values automatically.

Due to time limitations, some solvable problems were left untouched in this project. One of which is a sub problem of the occlusion problem: In a lower camera angle set up, if a tall piece moves laterally by 1 or 2 squares, its image in the reference frame might overlap its image in the comparing frame. The difference image might shows some unintuitive values for the overlapping area. 95%, which was the average accuracy of the system is actually still not acceptable to be used in real games as it is about equivalent to losing 1/5th of the games because of false detection. Future work will involve improving this number by tackling the unhandled problems in this implementation.

After a game is finished, users got redirected back to the home screen and have to go through the same setup phase again if they want to start a new game. Many test users found this feature very annoying and time consuming. I was aware of this and will put flexibility of the application as on the of the main priorities in future work.

In conclusion, the application developed was robust enough to play a full game of chess without a false detection in controlled light environments and can deliver reliable accuracy in standard environments. The program is still far from production ready, but a prototype version supporting limited game modes have been successfully implemented.

Bibliography

[1] Berserk: official lichess python api.

[2] Canny edge detection opencv illustration.

[3] Eroding and dilating in opencv.

[4] feature detectors - sobel edge detector.

[5] Feature matching in opencv.

[6] Homography transformation in opencv.

[7] Python websockets.

[8] React native · learn once, write anywhere.

[9] Rfc 7636 - proof key for code exchange by oauth public clients.

[10] Visualization with python.

[11] Web socket design patterns.

[12] Computer vision system for a chess-playing robot, May 2019.

[13] Homography (computer vision), Dec 2021.

[14] Pinhole camera model, Jun 2021.

[15] Algebraic notation (chess), Apr 2022.

[16] Canny edge detector, Jan 2022.

[17] Foreground detection, Feb 2022.

[18] Opencv, Apr 2022.

[19] Kevin Bowyer, Christine Kranenburg, and Sean Dougherty. Edge detector evaluation using empirical roc curves. *Computer Vision and Image Understanding*, 84(1):77–103, 2001.

[20] Maciej A. Czyzewski. An extremely efficient chess-board detection for non-trivial photos. *CoRR*, abs/1708.03898, 2017.

[21] Christiano Gava and Gabriele Bleser. 2d projective transformations (homographies). In *Technische Universität Kaiserslautern*.

[22] Rudrika Kalsotra and Sakshi Arora. Morphological based moving object detection with background subtraction method. *2017 4th International Conference on Signal Processing, Computing and Control (ISPCC)*, 2017.

[23] Ismail M. Khater, Ahmed S. Ghorab, and Inad A. Aljarrah. Chessboard recognition system using signature, principal component analysis and color information. *2012 Second International Conference on Digital Information Processing and Communications (ICDIPC)*, 2012.

[24] Joey Meyer. Raspberry turk.

[25] M. Piškorec, N. Antulov-Fantulin, J. Ćurić, O. Dragoljević, V. Ivanac, and L. Karlović. Computer vision system for the chess game reconstruction. In *2011 Proceedings of the 34th International Convention MIPRO*, pages 870–876, 2011.

[26] Emir Sokic and Melita Ahic-Djokic. Simple computer vision system for chess playing robot manipulator as a project-based learning example. *2008 IEEE International Symposium on Signal Processing and Information Technology*, pages 75–79, 2008.

[27] Chess.com Team. Chess.com update: Over 5 million new members, Feb 2022.