A CASE STUDY ON LOGGING VISUAL ACTIVITIES: CHESS GAME

A Thesis Submitted to
the Graduate School of Engineering and Sciences of
İzmir Institute of Technology
in Partial Fulfillment of the Requirements for the Degree of

MASTER OF SCIENCE

in Electrical and Electronics Engineering

by
Şükrü OZAN

November 2005
İZMİR

We approve the thesis of **Şükrü OZAN**

Date of Signature

. . . . . **1 November 2005**
Assist. Prof. Dr. Şevket GÜMÜŞTEKİN
Supervisor
Department of Electrical and Electronics Engineering
İzmir Institute of Technology

. . . . . **1 November 2005**
Prof. Dr. F.Acar SAVACI
Department of Electrical and Electronics Engineering
İzmir Institute of Technology

. . . . . **1 November 2005**
Assist. Prof. Dr. Salih DİNLEYİCİ
Department of Electrical and Electronics Engineering
İzmir Institute of Technology

. . . . . **1 November 2005**
Assist. Prof. Dr. Mustafa Aziz ALTINKAYA
Department of Electrical and Electronics Engineering
İzmir Institute of Technology

. . . . . **1 November 2005**
Assist. Prof. Dr. Serhan ÖZDEMİR
Department of Mechanical Engineering
İzmir Institute of Technology

. . . . . **1 November 2005**

Prof. Dr. F.Acar SAVACI
Head of Department
Department of Electrical and Electronics Engineering
İzmir Institute of Technology

. . . . . .
Assoc. Prof. Semahat ÖZDEMİR
Head of the Graduate School

ACKNOWLEDGEMENTS

First of all, I would like to thank my supervisor Assist.Prof.Dr. Şevket Gümüştekin for the encouragement, inspiration and the great guidance he gave me throughout my master study. I would also like to thank Prof.Dr.Ferit Acar Savacı, Assist.Prof.Dr. Salih Dinleyici, Assist.Prof.Dr. Mustafa Aziz Altınkaya and Assist.Prof.Dr. Serhan Özdemir for giving me technical and editorial guidance, corrections and suggestions which helped me a lot in revising the thesis. A special thank goes to Assist.Prof.Dr. Thomas F. Bechteler for encouraging and helping me to write this thesis in LaTeX format.

ABSTRACT

Automatically recognizing and analyzing visual activities in complex environments is a challenging and open-ended problem. In this thesis this problem domain is visited in a chess game scenario where the rules, actions and the environment are well defined. The purpose here is to detect and observe a FIDE (Fédération International des Échecs) compatible chess board and to generate a log file of the moves made by human players. A series of basic image processing operations have been applied to perform the desired task. The first step of automatically detecting a chess board is followed by locating the positions of the pieces. After the initial setup is established every move made by a player is automatically detected and verified. A PC-CAM connected to a PC is used to acquire images and implement the corresponding software. For convenience, *"Intel® Open Source Computer Vision Library (OpenCV)"* is used in the current software implementation.

ÖZET

Karmaşık sahnelerdeki görsel hareketliliklerin tanınması ve analiz edilmesi zor ve önü açık bir problemdir. Bu tez çalımasında konuyla ilgili olarak, kuralları ve hamleleri önceden tanımlanmış popüler bir oyun olan satranç üzerinde çalışılmıştır. Temel amaç, karmaşık bir sahne içerisinde tamamı görüntülenebilen FIDE standartlarına uygun bir satranç tahtasını tanımak ve ardından bu tahta üzerinde oyuncular tarafından oynanan hamleleri belirlemektir. Bunun için bir dizi temel görüntü işleme algoritması uygulanmıştır. Sistem satranç tahtasını belirledikten sonra taşların yerlerinin belirlenmesi işlemine geçer. Oyun başladıktan sonra oyuncuların yaptığı hamleler sistem tarafından otomatik olarak belirlenir ve doğrulanır. Bir PC'ye bağlı PC-CAM sistemi sayesinde görüntü alımı ve ilgili yazılım algoritmasının uygulanması sağlanmıştır. Bunun yanında uygulama kolaylığı açısından, açık kaynak kodlu *"Intel® Open Source Computer Vision Library (OpenCV)"* isimli kütüphane kullanılmıştır.

TABLE OF CONTENTS

LIST OF FIGURES

CHAPTER 1

INTRODUCTION

Interpreting visual activities is an open-ended research problem in Computer Vision. The recent availability of necessary computational power to process vast amount of video data motivated research on several aspects of video processing. The temporal component of video introduces an additional dimension compared to static images enabling us extraction of meaningful events. Temporal segmentation techniques are reviewed in a general terms in (Koprinska and Carrato 2001). Several interrelated applications such as content based image retrieval (Smoulders et al. 2000), video annotation (Dorado et al. 2004), and video indexing (Brunelli et al. 1999) have been a major focus of research. Most of these applications are based on low level processing of pixel data. Since there are an unlimited number of visual events that need to be recognized it is impossible to expect a general purpose algorithm to extract the meaning of all the actions in a video stream. The high level information in video data can be better interpreted if the application domain is limited. Several successful studies have been carried out in sports video applications (e.g. (Assfalg et al. 2003)(Ekin et al. 2003)). In this thesis, popular board game chess has been considered as the domain of study. Compared to the other board games chess is more interesting as a vision problem because of its relatively complicated rules and different types of pieces.

Chess is a game, played by two players. One player plays with the white pieces, and the other player plays with the black pieces. Each player has sixteen pieces in the beginning of the game: one king, one queen, two rooks, two bishops, two knights, and eight pawns. The game is played on a chessboard, consisting of 64 squares: eight rows and eight columns. The squares are alternately light (white) and dark colored. The board must be laid down such that there is a black square in the lower-left corner. To facilitate notation of moves, all squares are given a name. From the view of the white player, the rows are numbered as $1, 2, 3, 4, 5, 6, 7, 8$ from bottom to the top. The columns are named as a, b, c, d, e, f, g, h from left to right. A square is labeled by combining its column-letter and row-number, e.g. the square in the lower left corner (for white) is a1 (WEB_4 2005).

Most of the work on chess has been in the area of Artificial Intelligence aiming

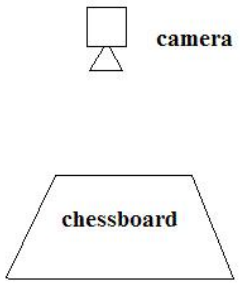


Figure 1.1: Best Case

to create computer chess players. The progress of these studies ended up as a computer defeating a human world champion (Campbell et al. 2002). Other related work involves reading printed chess moves (Baird and Thompson 1990). The studies similar to this thesis like; (Groen et al. 1992)(Uyar et al. 2004), also attack the problem of interpreting a chess game in action but they are focused on designing and implementing robots that can move pieces on the chess board.

Here, our main concern is "understanding the actions" on the chess board. An earlier version reported in the computer vision part of (Uyar et al. 2004) is reconsidered and improved. The system first detects and identifies an empty chess board in the field of view. This detection and identification process is possible only if the whole chessboard is visible. An algorithm in the implemented software, which will be explained in detail, in Chapter 2, finds the total of 49 inner chessboard corners. These corners identify the chessboard and they are used for the rectification of the view.

Figure1.1 illustrates the best case for camera vs. chessboard placement. In this case, camera is directly above the chessboard which makes the whole chessboard clearly visible, simplifying the detection and identification of moves. In this study the placement of the camera is not restricted. An arbitrary camera location can be seen in Figure1.2. The only restriction is that the whole chess board should be visible. In order to simplify the problem, a view projection is considered. The projection transforms the camera view to its ideal position illustrated in Figure1.1. This makes detection and identification of the moves possible. The details of the projection will be explained in Chapter 3.

After detection of the chessboard and projection of the view, the system becomes ready for logging process. The players place the chess pieces in their places then start

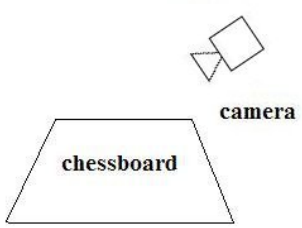


Figure 1.2: An Arbitrary Camera Placement

playing.

The system takes reference images at specific time instants (e.g. at the beginning of the game; after a move is detected). At any time, system finds the difference image between the reference image and current frame. As described in Chapter 4, by counting the amount of difference, the system can detect moves.

When a possible move is detected the scene is analyzed to identify the actions. this is done by using a combination of basic image processing techniques and a knowledge based approach. Difference images between consecutive reference frames give enough information about a move which is possibly performed. The knowledge based approach ensures that the decision is correct. Otherwise the move is considered as “not valid”. The analysis process is described in Chapter 5. Chapter 6 includes discussions with concluding remarks.

CHAPTER 2

CHESS BOARD DETECTION

In order to start solving the problem of identifying visual actions on the chessboard, firstly chess board should be detected. This chapter describes an efficient and robust method for chess board detection from real time video data. In this study, the image frames are gathered from a low cost camera with 320x240 resolution and 24-bit color depth in RGB format. Figure 2.1 represents an example chess board placement scenario. The images captured by the camera have been converted to 8 bit gray-scale images, since color information is not considered necessary for this application.

In order to detect a chessboard, 49 inner corners are located using a builtin function (*cvFindChessBoardCornerGuesses()*) of Intel® Open Source Computer Vision Library. This process can be briefly explained as follows: The function accepts 8-bit single channel gray scale images like the one shown in Figure 2.1. First, a gray scale dilation operation is applied to the image. This operation is explained in detail in Section 2.1. By this operation touching corners of the squares are separated from each other. After the separation, thresholding is applied to the figure in order to convert the gray scale image to binary image format. Generated binary image is used for contour tracing routine. While getting all the contour information, algorithm rejects contours with too small perimeter which are regarded as noise in the image. Contour analysis continues with searching for quadrangles. All the contours are checked and non-quadrangle contours are rejected. After these steps all the quadrangles and their corresponding contours in the source image are detected. This is done by finding the dominant corners and using this information.

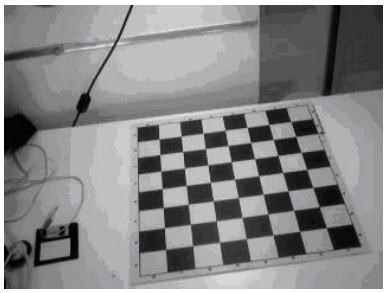


Figure 2.1: Chessboard View Example

A corner detection algorithm is applied to the contour image. After applying algorithm more precise information about the corners of the quadrangles are extracted. If the corner information is analyzed, it can be seen that some corners are very close to each other. Intuitively, one can easily deduce that these close corners are actually separated corners of the squares in the source image at the very first step of the algorithm. So the information obtained from these close corners yields to the information of the inner corners in the chessboard.

Lastly, 49 inner corners are reordered in the right order, hence it can be said that, after these steps, chessboard and its inner corners are detected properly. This procedure is described in more detail in the following sections.

2.1. Applying Dilation Operation to Source Image

Dilation is one of the basic morphological operations widely used in image processing (Gonzales and Woods 1993). In the following discussion a review of common operations in mathematical morphology is given.

2.1.1. Mathematical Morphology

The word morphology commonly denotes a branch of biology that deals with the form and structure of animals and plants. The same word is used in the context of mathematical morphology as a tool for extracting image components, such as boundaries, skeletons, and the convex hull. Mathematical Morphology offers a unified and powerful approach to many problems of image processing.

Sets in mathematical morphology represent the shapes of objects in an image. For example, the set of all black pixels in a binary image is a complete description of the image. In binary images, the sets in question are members in the 2D integer space Z^2, where each element of a set is a 2D vector whose coordinates are the (x, y) coordinates of a black pixel in the image. Gray scale digital images can be represented as sets whose components are in Z^3. In this case, two components of each element of the set refer to the coordinates of a pixel, and the third corresponds to its discrete intensity value. Sets in higher dimensional spaces can contain other image attributes, such as color and time varying components (Gonzales and Woods 1993).

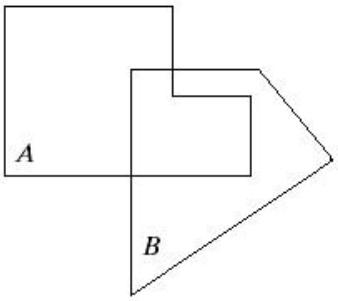


Figure 2.2: Two Sets A and B

(source:(Gonzales and Woods 1993))

2.1.2. Morphological Operations

Let A and B be sets in Z^2 (Figure 2.2) , with components $a = (a_1, a_2)$ and $b = (b_1, b_2)$ respectively. The translation of A by $z = (z_1, z_2)$, denoted $(A)_z$ (Figure 2.3), is defined as

$$(A)_z := \{c | c = a + z, \textit{for } a \in A\} \tag{2.1}$$

The reflection of B, denoted $\hat{B}$, is defined as (Figure 2.4)

$$\hat{B} := \{x | x = -b, \textit{for } b \in B\} \tag{2.2}$$

The complement of set A is (Figure 2.5)

$$(A)^c := \{x | x \ni A\} \tag{2.3}$$

Finally, difference of two sets A and B, denoted $A - B$, is defined as

$$A - B := \{x | x \in A, x \ni B\} = A \cap B^c \tag{2.4}$$

For sets A and B in Z^2, the erosion of A by B, denoted $A \ominus B$, is defined as

$$A \ominus B := \{x | (B)_x \subseteq A\} \tag{2.5}$$

With A and B as sets in Z^2 and denoting the empty set, the dilation of A by B, denoted $A \oplus B$, is defined as

$$A \oplus B := \{x | (B)_x \cap A \neq \emptyset\} \tag{2.6}$$

The dilation of A by B then is the set of all x displacements such that B and A overlap by at least one nonzero element. Set B is commonly referred as structuring element in

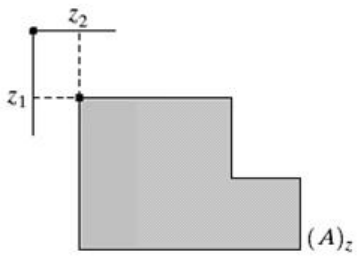


Figure 2.3: Translation of A by $z = (z_1, z_2)$

(source:(Gonzales and Woods 1993))

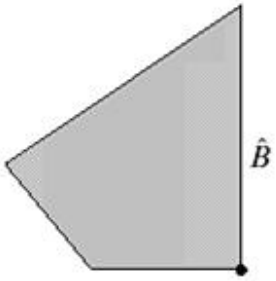


Figure 2.4: Reflection of B,$\hat{B}$

(source:(Gonzales and Woods 1993))

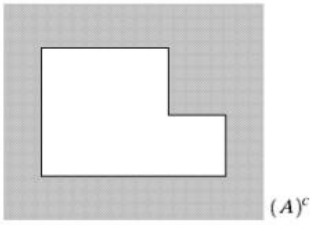


Figure 2.5: The complement of A, $(A)^c$

(source:(Gonzales and Woods 1993))

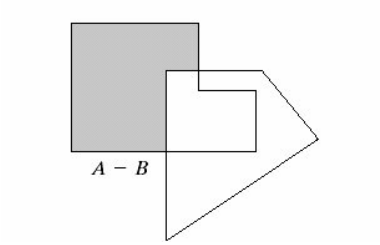


Figure 2.6: The difference between A and B
(source:(Gonzales and Woods 1993))

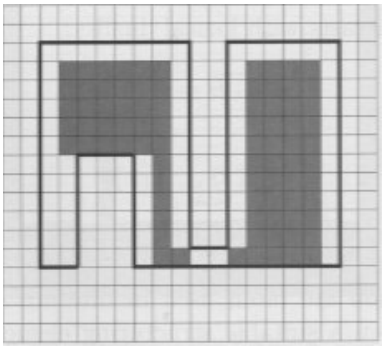


Figure 2.7: Erosion of A by B
(source:(Gonzales and Woods 1993))

morphological operations. In image processing combinations of dilation and erosion can be used. Two well known combinations are called opening and closing. The opening of set A by structuring element B, denoted $A \circ B$, is defined as

$$A \circ B := (A \ominus B) \oplus B \tag{2.7}$$

Which is simply erosion of A by B, followed by a dilation by B. The closing of set A by structuring element B, denoted $A \bullet B$, is defined as

$$A \bullet B := (A \oplus B) \ominus B \tag{2.8}$$

Figures 2.7, 2.8, 2.9 and 2.10 illustrates erosion, dilation, opening and closing respectively.

The binary morphological operations discussed in this section can be extended to gray scale images with minor modifications as discussed in the next section.

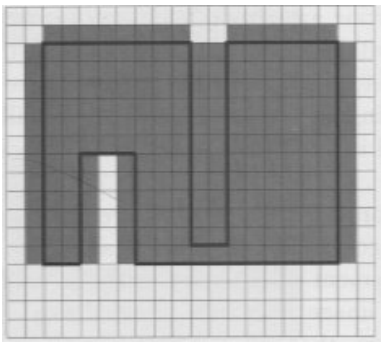


Figure 2.8: Dilation of A by B

(source:(Gonzales and Woods 1993))

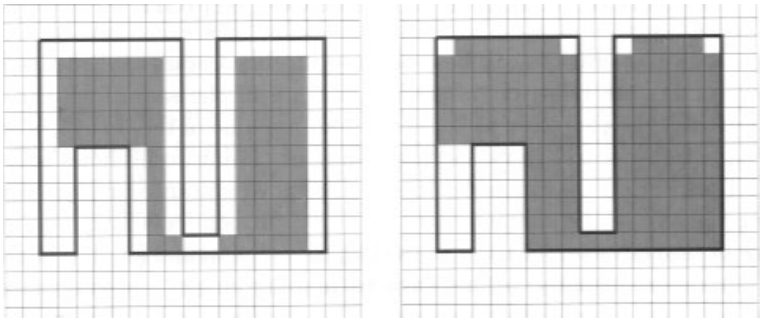


Figure 2.9: Opening

(source:(Gonzales and Woods 1993))

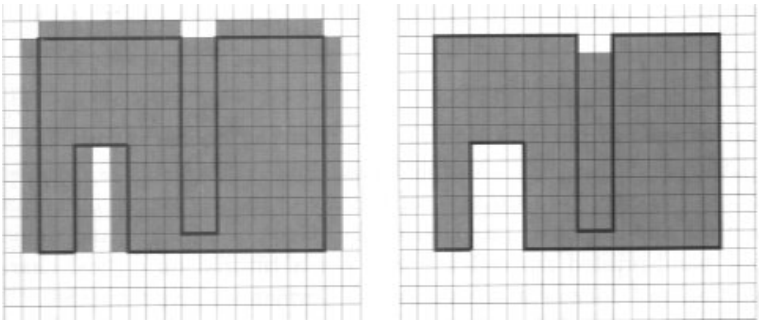
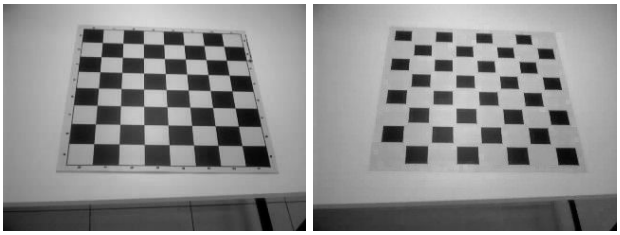


Figure 2.10: Closing

(source:(Gonzales and Woods 1993))



(a) **(b)**

Figure 2.11: (a) Image Gathered from Camera (b) Image After Dilation Operation

2.1.2.1. Extensions to Gray-Scale Images

If the source image is $f(x, y)$ and morphological structuring element is $b(x, y)$, gray scale erosion of f by b, denoted $f \ominus b$, is defined as

$$(f \ominus b)(s,t) := \min\{f(s+x, t+y) - b(x,y) | (s+x), (t+y) \in D_f; (x,y) \in D_b\} \quad (2.9)$$

Where D_f and D_b are the domains of f and b respectively. Gray scale dilation of f by b, denoted $f \oplus b$, is defined as

$$(f \oplus b)(s,t) := \max\{f(s-x, t-y) + b(x,y) | (s-x), (t-y) \in D_f; (x,y) \in D_b\} \quad (2.10)$$

Where D_f and D_b are the domains of f and b respectively. The expressions for opening and closing of gray-scale images have the same form as their binary counterparts. The opening of image f by structuring element b, denoted $f \circ b$, is defined as

$$f \circ b := (f \ominus b) \oplus b \quad (2.11)$$

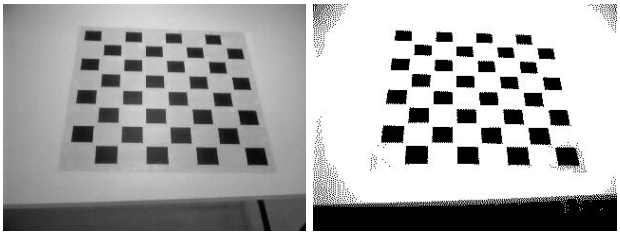
The closing of image f by structuring element b, denoted $f \bullet b$, is defined as

$$f \bullet b := (f \oplus b) \ominus b \quad (2.12)$$

A gray-scale dilation is performed using a 3x3 structuring element within *cvFindChessBoardCornerGuesses()* function. This dilation operation splits the touching squares (Figure 2.11).

2.2. Binary Image Processing

After splitting touching squares using gray scale erosion, thresholding operation is applied to convert gray level image to binary format. This conversion is needed due



(a) **(b)**

Figure 2.12: (a) Dilated Image (b) Dilated Image After Binary Thresholding

to the fact that the structural features of binary images are easier to be processed and analyzed compared to gray level images. Once an image is converted to binary format the following operations are mostly applied on binary images (Petrakis 2004):

- Noise suppression
- Run-length encoding
- Component labeling
- Contour extraction
- Medial axis computation
- Thinning
- Filtering (morphological)
- Feature extraction (size, orientation etc.)

Here, the thresholding is applied to the dilated image where the touching squares are separated. Mean intensity value of the image is used as threshold T . In an 8-bit single channel gray level image (I), intensity value is between 0 and 255. So the binary thresholding rule can be written as:

$$\textit{If } I[x,y] < T \textit{ then } B[x,y] = 0 \textit{ else } B[x,y] = 255 \tag{2.13}$$

Where B is the resulting binary image and [x,y] denotes an arbitrary pixel in either image. In Figure 2.12 dilated image before and after binary thresholding operation can be seen. Binary image obtained is to be used for contour extraction.

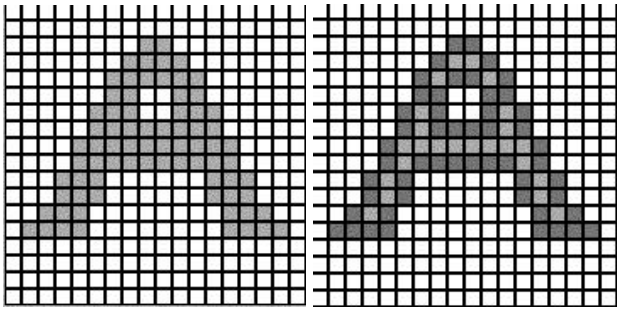
2.3. Contour Tracing

Until this step, a gray scale dilation operation and a thresholding operation has been performed respectively. In the image obtained (Figure 2.12.b), 32 separate black squares can be seen. The corners of these square pairs in close proximity enable us to identify the locations of inner corners. In order to do this, first the squares should be detected. Square detection is performed by simply approximating contours to nearest polygons. After rejecting negligibly small contours and non-quadrangles, corresponding squares are left.

Contour tracing, also known as boundary following or border following, is a technique that is applied to binary images in order to extract their boundary. The boundary of a given pattern is the set of border pixels. Since we are using a square tessellation, there are 2 kinds of boundary (or border) pixels: 4-border pixels and 8-border pixels. A black pixel is considered a 4-border pixel if it shares an edge with at least one white pixel. On the other hand, a black pixel is considered an 8-border pixel if it shares an edge or a vertex with at least one white pixel. A 4-border pixel is also an 8-border pixel. An 8-border pixel may or may not be a 4-border pixel. After identifying the boundary pixels of a pattern, an ordered sequence of the boundary pixels is needed to extract the general shape of the pattern.

Contour tracing is one of many preprocessing techniques performed on binary images in order to extract information about shapes of objects. Once the contour of a given pattern is extracted, its different characteristics can be examined and used as features for pattern classification. Therefore, correct extraction of the contour produces more accurate features increasing the chances of correctly classifying a given pattern. The contour pixels are generally a small subset of the total number of pixels representing a pattern. Therefore, the amount of computation is greatly reduced when feature extracting algorithms are implemented on the contour instead of on the whole pattern. Since the contour shares most of the descriptive features with the original pattern, despite a reduced data size, the feature extraction process becomes much more efficient when performed on the contour rather on the original pattern. In Figure 2.13.a, a gray “A” pattern can be seen on a white background. In Figure 2.13.b, the same pattern is shown after the contour tracing algorithm; pixels with dark gray color are boundary pixels.

Contour tracing is often considered as a major contributor to the efficiency of the



(a) **(b)**

Figure 2.13: (a) An “A” Pattern (b) Shows “A” Patterns Boundary Pixels (source:(WEB_1 2000))

feature extraction process which an essential process in the field of pattern recognition (Pavlidis 1982)(WEB_1 2000).

2.4. Searching for Quadrangles

Within the function *cvFindChessBoardCornerGuesses()*, a well known algorithm called “Douglas Peucker Algorithm”is used to approximate contours to nearest polygons.

2.4.1. Douglas-Peucker Algorithm

The Douglas Peucker line simplification algorithm has been widely used for data cleaning and simplification as a scale independent method for generalizing line features.

In their original paper (Douglas and Peucker 1973), the authors describe two methods for reducing the number of points required to represent a digitized line. The second method has been most widely implemented as adapted here. The first point on the line is defined as the anchor and the last point as a floater. These two points are connected by a straight line segment and perpendicular distances from this segment to all intervening points are calculated. If none of these perpendicular distances exceed a user specified tolerance (distance value), then the straight line segment is accepted to represent the whole line in simplified form. If this condition is not met, then the point with the greatest perpendicular offset from the straight line segment is selected as a new floating point.

The cycle is repeated, the new straight line segment being defined by the anchor

and the new floater. Offsets for intervening points are then recalculated perpendicular to this new segment. This process continues; the line being repeatedly subdivided with selected floating points being stored in a stack, until the tolerance criterion is met. Once the tolerance criterion is met, the anchor is moved to the most recently selected floater, and the new floating point is selected from the top of the stack of previously selected floaters. The selection process is repeated.

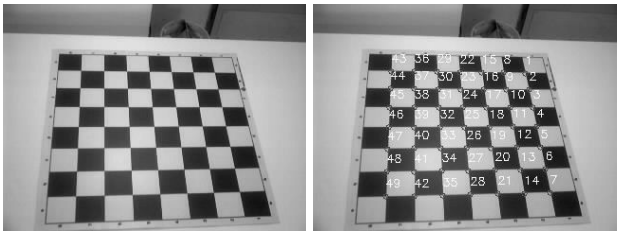
Eventually, the anchor point reaches the last point on the line, and the simplification process is complete. Points previously assigned as anchors are connected by straight line segments to form the simplified line. Note that specifying a low tolerance value results in little line detail being removed whereas specifying a high tolerance value results in all but the most general features of the line being removed (Visvalingam and Whyatt 1990). In the OpenCV implementation *cvFindChessBoardCornerGuesses()* function simply asks for the list of all the contours approximated as a polygon using Douglas Peucker algorithm, and rejects anything that does not have four vertices of projected squares. Components that satisfy the expectations on the structure and size are added to the list.

2.5. Corner Finding

The chessboard detection loops through the detected squares and looks for adjacent corners. The midpoint between the adjacent corners is accepted as a corner intersection. In OpenCV, *cvFindChessBoardCornerGuesses()* function is able to find the corner intersection points to an accuracy of half a pixel. If the function returns a positive value, further refinement of the corner intersection locations is done within the image using *cvFindCornerSubPix()* call. This function is able to find the location of the intersections to an accuracy of 0.1 pixel.

The *cvFindCornerSubPix()* function works on the original, gray-scale image, and the initial guesses of the intersection pointers. For each point, the function iterates until a corner measure at a subpixel intersection location is above a threshold. Within OpenCV the following method is used to find corners (WEB_2 2005):

Method: Assuming D_x as the first derivative in the direction of x just as D_y in the direction of y, D_{xx} is the second derivative in the direction of x just as D_{yy} in the direction of y, D_{xy} as the partial derivative with respect to x and y, sobel gradient operators are used to take the derivatives in x and y directions, after which a small region



(a) **(b)**

Figure 2.14: (a) A Chessboard View (b) Detected and Labeled Corners After Chessboard Detection Algorithm

of interest is defined to detect corners in. A 2x2 matrix of the sums of the derivatives x and y is subsequently created for the region of interest as follows:

$$C = \begin{bmatrix} \Sigma D_x^2 & \Sigma D_x D_y \\ \Sigma D_x D_y & \Sigma D_y^2 \end{bmatrix} \tag{2.14}$$

The eigen values are found by solving $det(C - \lambda I) = 0$, where λ is a colomn vector of the eigenvalues and I is the identity matrix. For the 2x2 matrix of the equation above, the solutions may be written in a closed form:

$$\lambda_{1,2} = \frac{\Sigma D_x^2 + \Sigma D_y^2 \pm \sqrt{(\Sigma D_x^2 + \Sigma D_y^2)^2 - 4(\Sigma D_x^2 \Sigma D_y^2 - (\Sigma D_x \Sigma D_y))^2)}}{2} \tag{2.15}$$

If $\lambda_1, \lambda_2 > t$, where t is some threshold, then a corner is found within the region of interest. This technique for corner detection has been widely used for object and shape recognition.

After all the inner corners of the chessboard is found and well defined, then chess board is considered to be found. The chess board is assumed to be empty while being detected. In Figure 2.14.a, an example view and in 2.14.b labeled corners after chess board detection can be seen.

CHAPTER 3

PROJECTION OF THE VIEW

The view of the board shown in Figure 3.1 is the case where the cameras view plane is parallel to the chessboard. In order to obtain a general solution for various perspectives, a perspective projection transformation is needed to be constructed that maps an arbitrary view to the desired view. By using the inner corner information obtained in Chapter 2, and the view constraints on the desired view, corresponding transformation can be found. The constructed transformation is then performed by mapping every point in the source view to the desired view, Figure 3.1.

3.1. Geometric Transformations

A geometric transformation is a mapping that relocates image points. Transformations can be global or local in nature. Global transformations are usually defined by a single equation which is applied to the whole image. Local transformations are applied to a part of image and they are harder to express concisely. Some of the most common global transformations are affine, perspective and polynomial transformations. The first three cases of the Figure 3.2 are typical examples for the affine transformations. The remaining two are the common cases where perspective and polynomial transformations are used, respectively (Gümüştekin 1999).

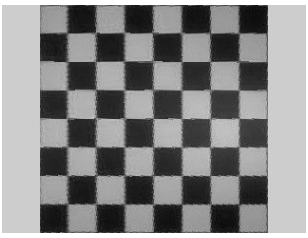


Figure 3.1: Desired View (Projected View)

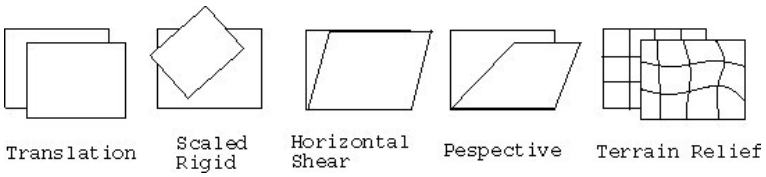


Figure 3.2: Common Geometric Transformations

(source:(Gümüştekin 1999))

3.1.1. Projective Models for Camera Motion

Having $p = (x, y)^T$ as old and $p' = (x', y')^T$ as the new coordinates of a pixel, a 2D affine transformation can be written as:

$$\begin{aligned} p' &= A * p + t \\ \begin{bmatrix} x' \\ y' \end{bmatrix} &= \begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} + \begin{bmatrix} t_x \\ t_y \end{bmatrix} \end{aligned} \tag{3.1}$$

The vector t is the translation component of the above equation. The matrix A controls scaling, rotation and shear effects:

$$A = s \begin{pmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{pmatrix} \quad : \ scale\ and\ rotation \tag{3.2}$$

$$A = \begin{pmatrix} 1 & 0 \\ a & 1 \end{pmatrix} \quad : \ vertical\ shear \tag{3.3}$$

$$A = \begin{pmatrix} 1 & a \\ 0 & 1 \end{pmatrix} \quad : \ horizontal\ shear \tag{3.4}$$

$$A = \begin{pmatrix} s_x & 0 \\ 0 & s_y \end{pmatrix} \quad : \ altered\ aspect\ ratio \tag{3.5}$$

The affine transformation can be represented by a single matrix multiplication in homogeneous coordinates:

$$\begin{bmatrix} X' \\ Y' \\ 1 \end{bmatrix} = \begin{bmatrix} a & b & c \\ d & e & f \\ 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} \tag{3.6}$$

Where equation 3.1 and 3.6 are related by ($a_{11} = a,\ a_{12} = b,\ a_{21} = d,\ a_{22} = e,\ t_x = c,\ t_y = f$). Introducing a scaling parameter W, the transformation matrix A can be modified to handle perspective corrections:

$$\begin{bmatrix} X' \\ Y' \\ W \end{bmatrix} = \begin{bmatrix} a & b & c \\ d & e & f \\ g & h & 1 \end{bmatrix} \cdot \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} \tag{3.7}$$

The image coordinates can be obtained by

$$x' = \frac{X'}{W} \tag{3.8}$$

$$y' = \frac{Y'}{W} \tag{3.9}$$

3.2. Constructing Projection Matrix

The perspective projection matrix described in 3.7 relates old (x, y) and new (x', y') coordinates by a matrix multiplication. The 3x3 matrix A has 8 unknown parameters.

If the expressions for X', Y' and W are written explicitly from equation 3.7 and substituted in equations 3.8 and 3.9 respectively, following equations are obtained for x':

$$x' = \frac{a * x + b * y + c}{g * x + h * y + 1} \tag{3.10}$$

and y' :

$$y' = \frac{d * x + e * y + f}{g * x + h * y + 1} \tag{3.11}$$

If the equations 3.10 and 3.11 are rearranged, equations 3.12 and 3.13 are obtained.

$$x * a + y * b + 1 * c + 0 * d + 0 * e + 0 * f - x * x' * g - y * x' * h = x' \tag{3.12}$$

$$0 * a + 0 * b + 0 * c + x * d + y * e + 1 * f - x * y' * g - y * y' * h = y' \tag{3.13}$$

For convenience, both equations are written in the same format with respect to 8 variables namely a, b, c, d, e, f, g and h. To find these variables, 8 equations are needed. Mapping of a pre-known point to another pre-known point gives two linearly independent equations according to equations 3.12 and 3.13.

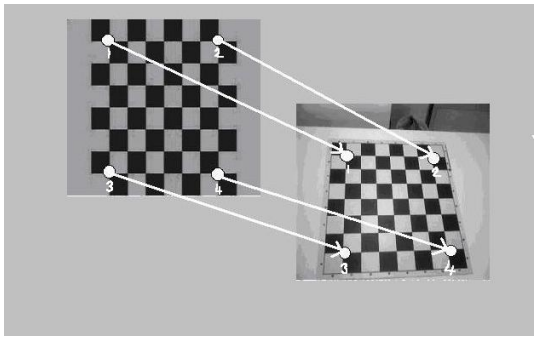


Figure 3.3: Points for Mapping

Hence, mapping of 4 points (Figure 3.3) provides enough information to find the unknown parameters of the projection matrix. 8 linearly independent equations can be written in matrix representation as;

$$\begin{bmatrix} x_1 & y_1 & 1 & 0 & 0 & 0 & (-x_1 * x_1') & (-y_1 * x_1') \\ x_2 & y_2 & 1 & 0 & 0 & 0 & (-x_2 * x_2') & (-y_2 * x_2') \\ x_3 & y_3 & 1 & 0 & 0 & 0 & (-x_3 * x_3') & (-y_3 * x_3') \\ x_4 & y_4 & 1 & 0 & 0 & 0 & (-x_4 * x_4') & (-y_4 * x_4') \\ 0 & 0 & 0 & x_1 & y_1 & 1 & (-x_1 * y_1') & (-y_1 * y_1') \\ 0 & 0 & 0 & x_2 & y_2 & 1 & (-x_2 * y_2') & (-y_2 * y_2') \\ 0 & 0 & 0 & x_3 & y_3 & 1 & (-x_3 * y_3') & (-y_3 * y_3') \\ 0 & 0 & 0 & x_4 & y_4 & 1 & (-x_4 * y_4') & (-y_4 * y_4') \end{bmatrix} . \begin{bmatrix} a \\ b \\ c \\ d \\ e \\ f \\ g \\ h \end{bmatrix} = \begin{bmatrix} x_1' \\ x_2' \\ x_3' \\ x_4' \\ y_1' \\ y_2' \\ y_3' \\ y_4' \end{bmatrix} \tag{3.14}$$

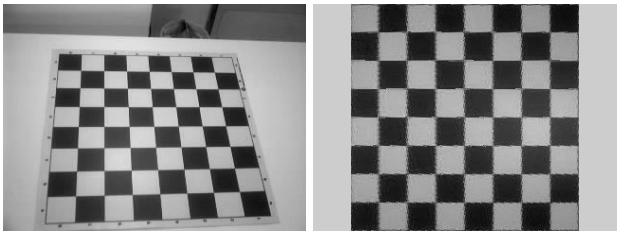
Equation 3.14 can be described by

$$\mathbf{P}.\mathbf{x} = \mathbf{b} \tag{3.15}$$

Selecting 4 control points that are distinct from each other as illustrated in Figure 3.3 helps avoiding numerical problems associated with a bad conditioned matrix. After constructing the matrix **P**, by using 4 inner corners that are distant from each other the unknown parameters can easily be calculated by

$$\mathbf{x} = \mathbf{P}^{-1}.\mathbf{b} \tag{3.16}$$

The equation 3.16 gives us a, b, c, d, e, f, g, h which are the elements of the projection matrix defined in equation 3.7. After the projection matrix calculation the scene can be projected. In Figure 3.4.b the projected view of the scene in Figure 3.4.a can be seen.



(a) **(b)**

Figure 3.4: (a) A Proper Scene (b) Perspective Projection of The Scene

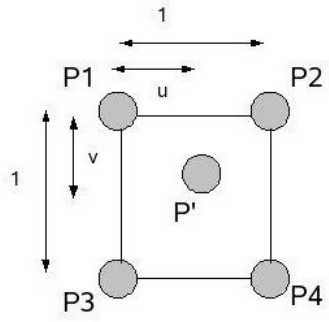


Figure 3.5: Interpolation Exemplar

3.3. Bilinear Interpolation

There are two types of mapping that can be used for geometric transformations: forward mapping and backward mapping. Forward mapping is done by scanning input image, finding new coordinates and assigning the pixel value to it. Backward mapping is done by scanning output image, finding old coordinates and assigning the pixel value from it.

Neither backward nor forward mapping guarantees a one to one mapping between the samples of input and output coordinates. Forward mapping produces some holes in the target image since some of the output coordinates are never visited. Backward mapping is preferred since it guarantees that the target image is completely filled. The problem that we need to face in backward mapping is to find the value of P' from the input image coordinates, Figure 3.5. The most commonly used interpolation techniques are nearest neighbor interpolation and bilinear interpolation

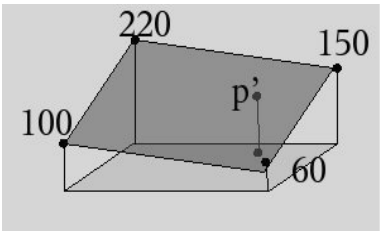


Figure 3.6: Bilinear Interpolation
(source:(Gümüştekin 2004))

- **Nearest Neighbor interpolation :** P' gets its value from the nearest pixel. In the example it is P_2.
- **Bilinear interpolation :** P' gets its value from the plane on the top of 4 neighboring pixels. This can be illustrated by Figure 3.6, where the intensity values of the neighbor pixels are given as; $P_1 = 220$, $P_2 = 150$, $P_3 = 100$ and $P_4 = 60$. Intensity value of P' can be found with the bilinear equation 3.17.

$$P' = A_1 * (1 - v) + A_2 * v \tag{3.17}$$

where

$$\begin{aligned} A_1 &= 220 * (1 - u) + 150 * (u) \\ A_2 &= 110 * (1 - u) + 60 * (u) \end{aligned} \tag{3.18}$$

In Figure 3.7.a and 3.7.b, close up view to 4 squares can be seen. The result in 3.7.a is obtained by nearest neighbor interpolation where in 3.7.b, bilinear interpolation has been applied. It can be seen that the additional computational complexity introduced by bilinear interpolation provides a higher quality image.

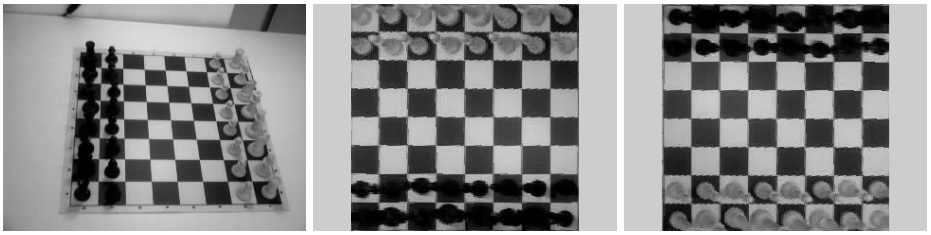
3.4. Preparation of the Scene

The inner corner detection algorithm detects the corners and labels them in an order. After labeling, corners with label $1, 7, 43$ and 49 are used for projection. For further analysis on deciding the moves, a predefined chessboard standard (shown in Figure 3.8) is to be considered. After players fill the board with their pieces, the projected view of the scene may not satisfy the assumed standard. In this case, before starting the game the projected view should be adjusted to make it compatible with the standard.

(a) **(b)**

Figure 3.7: (a) Distorted Image (b) Enhanced Image After Applying Gray-level Interpolation

Figure 3.8: Assumed Chessboard Standard



(a) **(b)** **(c)**

Figure 3.9: (a) Source (b) View of the Black Player (c) View of the White Player

In Figure 3.9, the projected view of a chessboard before and after projection adjustment can be seen. The adjustment is done by rotating the projected view by 90 degrees clockwise or counterclockwise direction until the correct projection is achieved. After the projected view is adjusted according to the standard, the activities in the chess game is observed within the boundaries of the chessboard.

CHAPTER 4

MOVE DETECTION

In previous chapters, detection and identification of the chessboard and projection and preparation of the scene were explained. After these steps, the system should continuously observe the scene and detect the actions that are typical to a chess game. In this chapter our approach to move detection will be explained. Difference images, that are due to players hand presence in the scene, are used to detect the instances where possible moves are made.

4.1. Difference Image

The difference between two images f(x,y) and h(x,y), expressed as

$$g(x, y) = |f(x, y) - h(x, y)| \tag{4.1}$$

is obtained by computing the difference between all pairs of corresponding pixels from f and h. Image subtraction has numerous important applications in segmentation and enhancement. A classic application of equation 4.1 for enhancement is an area of medical imaging called mask mode radiography. In this case $h(x, y)$, the mask, is an x-ray image of a region of patients body capture by an intensifier and TV camera (instead of traditional x-ray film) located opposite an x-ray source. The image $f(x, y)$ is one sample of series of similar TV images of the same anatomical region but acquired after injection of a dye into the bloodstream. The net effect of subtracting the mask from each sample in the incoming stream of TV images is that only the areas that are different between $f(x, y)$ and $h(x, y)$ appear in the output image as enhanced detail. Because images can be captured in TV rates, this procedure gives a movie showing how the dye propagates through the various arteries (Gonzales and Woods 1993).

Besides biomedical applications, difference images are used in a variety of applications such as video surveillance, fault detection, remote sensing etc. Similarly, in this study difference images are used to detect changes in the scene. Any change in the scene stimulates the system, so that the recent activities can be analyzed.

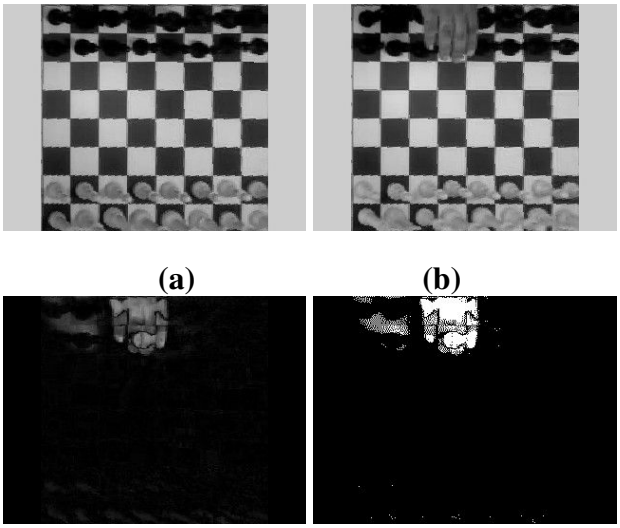


Figure 4.1: (a) Reference Frame Before the Move (b) The Hand is in the Scene (c) Difference Image (d) Binary Thresholded Version of (c)

4.2. Procedure for Move Detection

Within the context of detecting chess moves, $f(x, y)$, one of the images used in equation 4.1, has been considered as the frame before a move, which is regarded as reference image. After a move is detected the reference image is to be updated. All the upcoming frames after reference frame has been treated as $h(x, y)$. Figure 4.1.a is the scene at the beginning of the game. This frame is taken as reference image. In Figure 4.1.b, the players hand is in the view. When the difference image between (a) and (b) is calculated (c) is obtained. In this difference image the hand of human player extracted from the rest of the scene can be observed.

For convenience, the difference image in Figure 4.1.c is converted to binary format (Figure 4.1.d). By using binary information, difference at any time can be represented as the number of white pixels. The plot in Figure 4.2 has been generated by using these count values. Horizontal axis is the frame number, vertical axis is the difference between reference frame and current frame. It is due to the opening move of the pawn in *f2* (Figure 4.3.a and 4.3.b). This plot can be examined as follows;

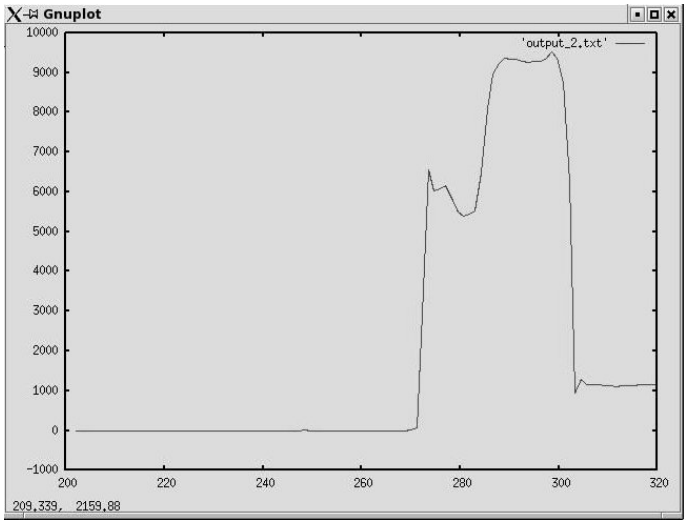
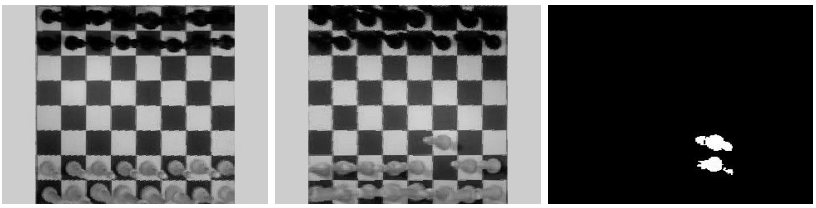


Figure 4.2: Difference Plot



(a) **(b)** **(c)**

Figure 4.3: (a) Reference Frame (b) Scene After The Pawn is moved from f2 to f3 (c) Difference Between (a) and (b)

- Frame #0 is taken a as reference. The difference is approximately 0 until frame #272

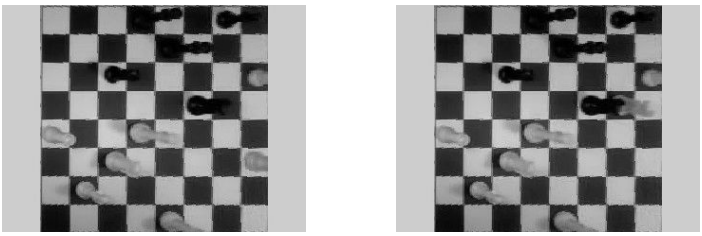
- Starting from frame #272, the players hand enters to the view. Between frame #274 and frame #285 the difference does not change so much. Because within that period the player grabs the piece and hand position does not change.

- The difference increases till frame #287, between frame #287 and frame #301, the difference is almost constant. Because within this period the player puts the piece in its new place and hand stays there for a while. Starting from frame #301, difference decreases until frame #306.

- After frame #306, the difference remains nearly constant. It fluctuates around 1150. This flat region, after frame #306, indicates that the hand is off the screen. Fixed difference is due to the position change of the piece which has been manipulated (Figure 4.3.c).

In Figure 4.4 and 4.5, two more moves and their corresponding difference plots are given. In Figure 4.4, white knight at *h3* is moved to *g5*. The move is forward. So the difference when player takes the piece is lower than the difference when player puts the piece in its new place; whereas, in Figure 4.5, white queen at *d4* is moved to *f2* which is a backward move. In the corresponding difference plot, difference when player takes the piece is higher than the difference when player puts the piece in its new place. Either it is a forward move or backward move; the difference plot has a flat region after the move is completed. This flat region can be detected by observing the difference between consecutive frames within a predefined period. In this study, program considers a move to be complete if the differences between *25* consecutive frames are lower then a threshold. This threshold is considered as *2000* pixels and it has been found as a result of experiments performed.

The move detection approach described in this chapter identifies the time instances of the beginning and end of each move. The image frames at these critical instances are analyzed further as described in the next chapter to interpret the actions within the context of chess game.



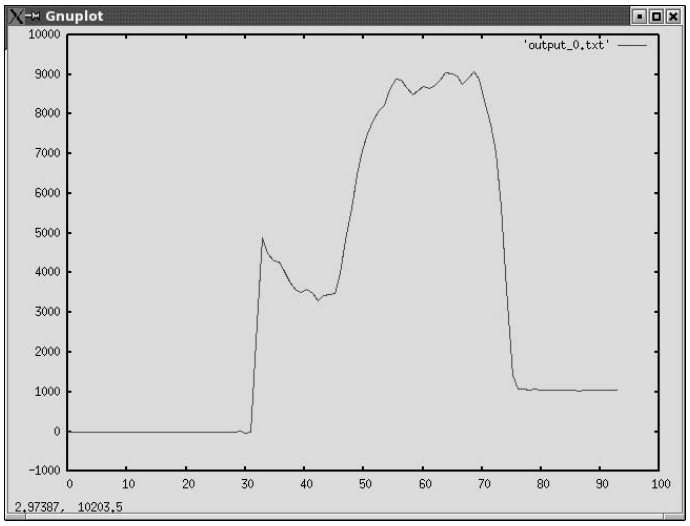


Figure 4.4: Move (h3g5) and its corresponding difference plot

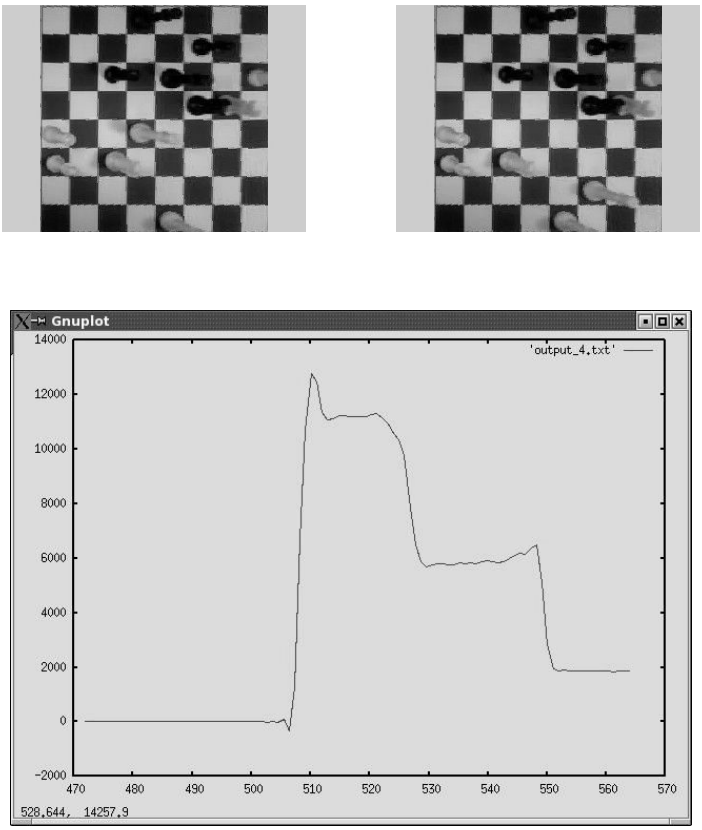


Figure 4.5: Move (d4f2) and its corresponding difference plot

CHAPTER 5

MOVE DECISION

When a possible chess move is detected the next step is to verify the move and determine the new positions of pieces. The decision is based on the information about the static and dynamic changes in the scene. The possible movements indicated by difference images are checked to see if they are valid by using the moves allowed by chess rules at the current state of the game. This is done by using a move table representation (e.g. Figure 5.1) for each piece which is updated after each move.

5.1. Procedure for Move Decision

Move tables are constructed according to the chess rules (e.g. pawns only move forward; the pieces can not jump with the exception of knights, etc.). After a move table is constructed it is stored in an array of 8 unsigned characters. This is made by representing each line in the move table as a string of 8 bits and converting it to a decimal number. Making this conversion reduces the memory usage and simplifies computation. For example the table in Figure 5.1 can be stored as:

$$[34\ 36\ 168\ 112\ 255\ 112\ 168\ 36\ 34]$$

To generate a move table at any time during the game, the overall piece locations should be known. To be able to code this information, an 8x8 matrix has been used. The pieces

8	0	0	1	0	0	0	1	0
7	0	0	1	0	0	1	0	0
6	1	0	1	0	1	0	0	0
5	0	1	1	1	0	0	0	0
4	1	1	1	1	1	1	1	1
3	0	1	1	1	0	0	0	0
2	1	0	1	0	1	0	0	0
1	0	0	1	0	0	1	0	0
	a	**b**	**c**	**d**	**e**	**f**	**g**	**h**

Figure 5.1: Move Table of a Queen Located at c4 With no Other Piece Blocking

	a	b	c	d	e	f	g	h
8	r1	n1	b1	q	k	b2	n2	r2
7	p1	p2	p3	p4	p5	p6	p7	p8
6								
5								
4								
3								
2	P1	P2	P3	P4	P5	P6	P7	P8
1	R1	N1	B1	Q	K	B2	N2	R2

(a)

	a	b	c	d	e	f	g	h
8	-8	-7	-6	-2	-1	-3	-4	-5
7	-16	-15	-14	-13	-12	-11	-10	-9
6	0	0	0	0	0	0	0	0
5	0	0	0	0	0	0	0	0
4	0	0	0	0	0	0	0	0
3	0	0	0	0	0	0	0	0
2	16	15	14	13	12	11	10	9
1	8	7	6	2	1	3	4	5

(b)

Figure 5.2: (a) Representation of Figure 3.8 (b) Coded Version of The Chessboard at the Beginning of the Game

on the chessboard are represented by different labels. These labels have positive values for white pieces and negative values for black pieces. Considering Figure 3.8 , the board can be represented as Figure 5.2.a and the pieces can be labeled as Figure 5.2.b. For empty squares on the board, zero label is used. The other labels can be listed as:

Black rook(Left) at a8 = r1 = -8	Black rook(Right) at h8 = r2 = -5
Black knight(Left) at b8 = n1 = -7	Black knight(Right) at g8 = n2 = -4
Black bishop(Left) at c8 = b1 = -6	Black bishop(Right) at f8 = b2 = -3
Black queen at d8 = q = -2	Black king at e8 = k = -1
Black pawn [1. . . 8] at [a7. . . h7] =	[p1. . . p8] = [-16. . . -9]
White rook(Left) at a1 = R1 = 8	White rook(Right) at h1 = R2 = 5
White knight(Left) at b1 = N1 = 7	White knight(Right) at g1 = N2 = 4
White bishop(Left) at c1 = B1 = 6	White bishop(Right) at f1 = B2 = 3
White queen at d1 = Q = 2	White king at e1 = K = 1
White pawn [1. . . 8] at [a2. . . h2] =	[P1. . . P8] = [16. . . 9]

White pieces are shown with capital letters and labeled with positive integers where black pieces are shown with small letters and labeled with negative integers. The table in Figure 5.2.b is to be updated after every move. For example, considering an opening move for *P6* in *f2* to *f4*; the updated position matrix can be represented as Figure 5.3.b.

	a	b	c	d	e	f	g	h
8	-8	-7	-6	-2	-1	-3	-4	-5
7	-16	-15	-14	-13	-12	-11	-10	-9
6	0	0	0	0	0	0	0	0
5	0	0	0	0	0	0	0	0
4	0	0	0	0	0	0	0	0
3	0	0	0	0	0	0	0	0
2	16	15	14	13	12	11	10	9
1	8	7	6	2	1	3	4	5

(a)

	a	b	c	d	e	f	g	h
8	-8	-7	-6	-2	-1	-3	-4	-5
7	-16	-15	-14	-13	-12	-11	-10	-9
6	0	0	0	0	0	0	0	0
5	0	0	0	0	0	0	0	0
4	0	0	0	0	0	11	0	0
3	0	0	0	0	0	0	0	0
2	16	15	14	13	12	0	10	9
1	8	7	6	2	1	3	4	5

(b)

Figure 5.3: (a) Position Matrix Before the Move (b) Position Matrix After the Move

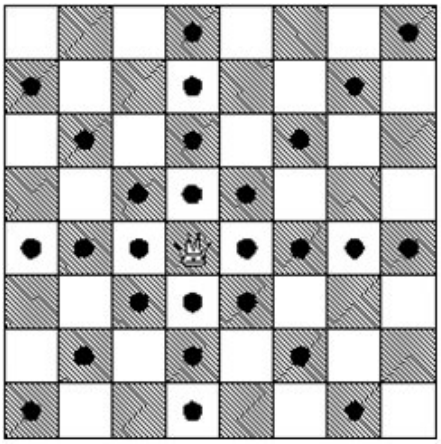
Hence, after every move, the updated position matrix gives sufficient position information for every single piece. This information is implicitly used for move table generation for the pieces.

5.1.1. Move Table Generation

Before every move, move tables for every piece should be updated. Position matrices are used to determine move tables. Procedure, while updating move tables, can be explained by using “white queen” as an example.

The queen has the combined moves of the rook and the bishop, i.e., it can move in any straight line, horizontal, vertical, or diagonal (Figure 5.4.a). In Figure 5.4.b, move table representation of the queen at d4 in 5.4.a can be seen.

The move tables are generated according to movement rules of chess using the position matrix which is updated after every turn. For the case of white queen (at *d4*) in Figure 5.5.a, first the move table given in Figure 5.4.b is constructed discarding other pieces on the board. Then this move table is updated so that 1’s in the table are deleted (i.e changed to 0’s), if there exists a chess piece between the original position (d4) and the current position. The resulting move table is given in Figure 5.5.b. For all chess pieces except for the knight, the move table is updated by checking whether or not the possible position are occupied by other pieces.



(a)

	a	b	c	d	e	f	g	h
8	0	0	0	**1**	0	0	0	**1**
7	**1**	0	0	**1**	0	0	**1**	0
6	0	**1**	0	**1**	0	**1**	0	0
5	0	0	**1**	**1**	**1**	0	0	0
4	**1**	**1**	**1**	**1**	**1**	**1**	**1**	**1**
3	0	0	**1**	**1**	**1**	0	0	0
2	0	**1**	0	**1**	0	**1**	0	0
1	**1**	0	0	**1**	0	0	**1**	0

(b)

Figure 5.4: (a) Illustrated Movement of Queen (b) Move Table Representation

	a	b	c	d	e	f	g	h
8	0	0	0	-2	-1	-3	0	0
7	-8	0	0	-9	0	0	0	0
6	0	0	0	0	0	-4	0	-5
5	0	0	0	0	0	0	0	0
4	0	0	0	2	0	11	0	0
3	0	0	0	0	0	0	0	0
2	0	0	0	0	12	0	10	9
1	8	7	6	0	1	3	4	5

(a)

	a	b	c	d	e	f	g	h
8	0	0	0	**0**	0	0	0	0
7	**1**	0	0	**1**	0	0	0	0
6	0	**1**	0	**1**	0	**1**	0	0
5	0	0	**1**	**1**	**1**	0	0	0
4	**1**	**1**	**1**	**1**	**1**	0	0	0
3	0	0	**1**	**1**	**1**	0	0	0
2	0	**1**	0	**1**	0	**1**	0	0
1	0	0	0	**1**	0	0	0	0

(b)

Figure 5.5: (a) Scenario in Movement Matrix Representation (b) Move Table of the White Queen with Label “2” in (a)

5.1.2. Estimating the Move from Difference Images

In the previous chapter our approach to move detection in a chess game scenario was explained. If the Figure 4.3 is reconsidered; 4.3.c shows the difference image between 4.3.a which is the reference image before the move and 4.3.b which is the image taken soon after the player hand is off the camera view. The difference image shown in 4.1.c displays some ambiguity caused by the perspective view. However it still provides the necessary information to extract the semantic context of the move. Due to perspective effects of non-ideal viewing positions, the problem of correctly detecting the position of chess pieces can be a challenging task. The problems arise from occlusions and coverage of several squares by each piece. The problem is attacked by initially estimating the coverage area at each position for an arbitrary perspective view.

5.1.2.1. Estimating Coverage Area of Chess Pieces

In order to estimate the coverage areas of pieces at each board position, an approximation is used instead of fully calibrating the camera and implementing a 3D mapping (Trucco and Verri 1998). The method is based on repositioning the board at a higher level, capturing the board image, and applying the perspective transformation derived from the original image as described in Chapter 3. Two possible approaches for repositioning the board are illustrated in Figure 5.6. The board is moved up in the Z axis at a distance comparable to the height of a typical chess piece, or the camera is moved down at the same distance. The more practical second approach is adopted by moving the camera. The new board plane imaged here is used to represent the imaginary plane placed at the top of chess pieces (Ozan and Gümüştekin 2005).

The original board image and the one captured by the method described above are mapped by the same perspective transformation. For either case, pixel coordinates of inner corners are used. These inner corners can be related to 49 squares' lower right corners. But on a chessboard there are 64 squares. The remaining 15 corner information can be extrapolated by using the known 49 inner corners. When the coordinates corresponding to each square is calculated, the deviation in the pixel coordinates of each square can be represented numerically. This deviation is due to the position change in z axis which can actually represent the estimated coverage areas of the chess pieces on the board.

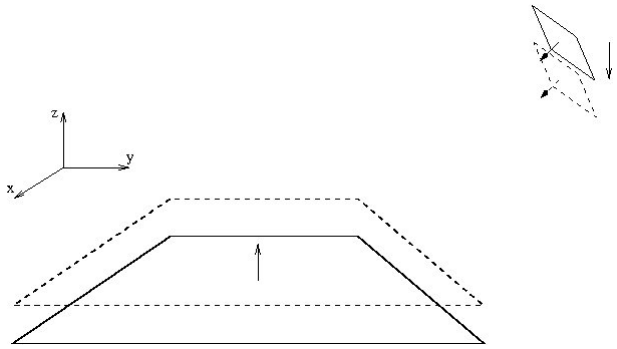


Figure 5.6: Two Approaches for Repositioning the Board for Approximating Perspective View of Pieces

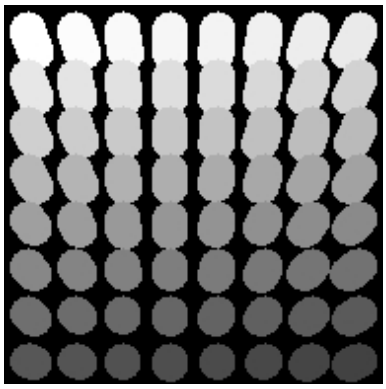


Figure 5.7: A Generated Coverage Area Example

From the deviation information of the squares, a deviation vector can be assigned to each square. By using a circular structuring element like 5.8.b and dilating the image of the deviation vectors with this structuring element, estimated coverage areas can be found. For convenience, different gray levels are assigned to each area in Figure 5.7. This approach simplifies the problem of finding correlation for each square individually. The procedure described in this section is based on physically changing the perspective view to estimate the coverage areas. An alternative method described in the next section may be applied when the view change illustrated in Figure 5.6 is not possible.

5.1.2.2. Alternative Method to 5.1.2.1

During the experiments it has been observed that in difference images, manipulation of a piece generates two circular shapes (one of them is due to the position before the move and the other is due to the position after the move). The circular shape occurs

Figure 5.8: (a) A Difference Image (b) Circular Structuring Element

because bottom of a piece has a circular shape. Hence finding a correlation between a circular structuring element and the difference image can be used as an alternative method (Figure 5.8).

A typical structuring element that can be used is shown in Figure 5.8.b. The size of the structuring element should be proportional to the size of a square.

5.1.2.3. Finding Correlation

Methods described in 5.1.2.1 and 5.1.2.2 are used for estimating the coverage areas of chess pieces for each square on the chess board. Assuming that a move has been detected and a difference image is obtained, estimated coverage area information is applied to the difference image such that a correlation value for every chessboard square can be obtained.

The correlation is simply calculated by counting the common foreground pixels. For each ellipse in binary estimated coverage area image, corresponding difference pixels are counted and they are considered as correlation values. All the values are normalized with respect to the highest value and considering the highest correlation value as "1".

For the second approach described in 5.1.2.2, the correlation can be calculated by simply counting common white pixels in the structuring shape and the chessboard squares. But this method sometimes gives incorrect results due to occlusion and non-ideal perspectives. In order alleviate this problem, an additional criterion can be added to the correlation algorithm, besides increasing the corresponding correlation by 1 for common white pixels, a penalty term is introduced which decrements the correlation value for white pixels in difference image which is correlated with a black pixel in the structuring

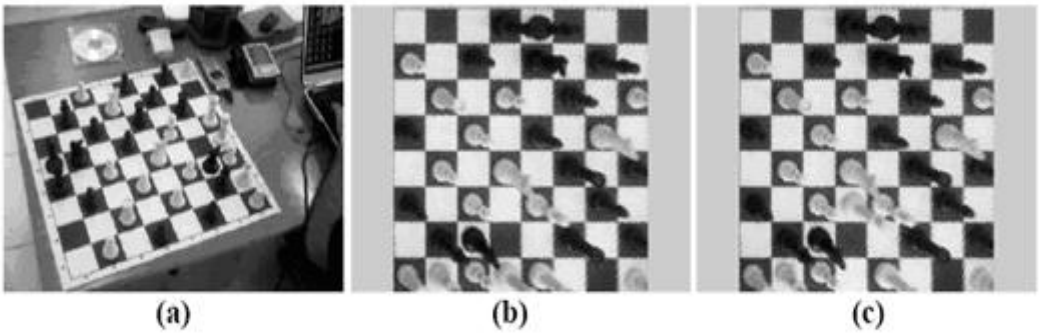


Figure 5.9: (a) Original View of a Sample Game (b) Reference Image Taken Before the Move (c) Reference Image Taken After the Move

element.

A typical example of a chess move is shown in Figure 5.9. The move is made by moving white queen in d1 to d3. Figure 5.9.a shows the original view of the scene, whereas, Figure 5.9.b is the projected view before the move and Figure 5.9.c is the projected view after the move. The table in Figure 5.10 is the correlation table which is extracted from static difference image between 5.9.b and 5.9.c. Second correlation method is used for the extraction. Highlighted squares indicates the "1" s in the move table which has been discussed in Section 5.1.1. These squares are the squares which the corresponding chess piece can move. Here the moving piece is white queen in d1. If the highlighted squares are examined, two biggest correlation numbers can be seen as 1 in *d1* and 0.721 in *d3* (d1 has the highest correlation and all the correlation values are normalized by the highest value). Hence the system reports a move information between squares *d1* and *d3*. The direction of the movement is determined by looking at the position matrix of the game which has been defined in Section 5.1. The program analyzes the position matrix and according to the movement type, knowing that it is white player's turn, it evaluates the following options:

• "White queen in d1 to d3" if d3 is empty in position matrix

• "White queen in d3 to d1" if d1 is empty in position matrix

• "White queen in d1 takes black pawn in d3" if the black pawn is in d3

In this example d3 is empty so the movement is found out to be: "White queen in d1 to d3". When a move is detected and reported, the system writes the result to a log file.

8	0	0	0	0	0.0021	0	0	0
7	0.0231	0	0	0	0	0	0	0
6	0	0.0042	0	0	0	0	0	0
5	0.0378	0.0021	0.0084	0	0	0	0	0
4	0	0	0	0	0	0	0	0
3	0.0462	0	0.0252	0.721	0.275	0	0	0
2	0.0021	0.0042	0.042	0.0084	0.174	0.0063	0	0
1	0	0.00196	0.00313	1	0.197	0	0	0
	a	**b**	**c**	**d**	**e**	**f**	**g**	**h**

Figure 5.10: Weights Extracted from Static Difference Image

CHAPTER 6

CONCLUSION

In this thesis a case study on logging visual activities has been considered. A well-known board game, chess was chosen as domain of study. Although the chess game is confined to well defined boundaries, it introduces some challenge as a vision problem with its three dimensional pieces and relatively complex rules.

A computer vision system which identifies a chessboard, and interprets the actions on the board is described. The domain knowledge is represented and updated at each state of the game. The visual data is combined with the encoded domain knowledge in order to recognize actions.

System starts with detecting and identifying the empty chessboard. These two tasks are performed by a built-in function of OpenCV, called *cvFindChessBoard-CornerGuesses()*. The identification of chessboard gives us precise pixel coordinates of 49 inner corners. Furthermore all the inner corners are numbered in a consecutive order from 1 to 49 (refer to Figure 2.14). If all the inner corners are found and labeled the system is considered to be ready for further processes. If all the inner corners are not found then system returns with an error and waits for the necessary correction. The reason for failure may be an object on the chessboard, lack of illumination or inappropriate viewing angle.

After finding the inner corners system calculates the parameters of a projection matrix to view the scene in optimal condition as illustrated in Figure 1.1. This over the top view simplifies the analysis of visual activities. In Figure 3.4 a game scene and its projected view is shown. The projection is performed by using pixel coordinates of inner corners labeled as 1, 7, 43 and 49.

Once the projection is performed, players put the pieces in their places and start to play the game. During the game reference images are taken. First reference image is taken at the very beginning of the game. For each frame, the difference between reference image and current frame is calculated. The difference image is converted to binary format and the total number of white pixels is taken as difference amount. By observing the difference, changes in the scene can be detected. When a player hand enters the scene,

the difference amount increases. After the player makes a move and takes his hand off the screen, the difference amount remains nearly constant. When move is detected, the last difference image is considered as one of the sources for move analysis.

Second source for the move analysis is knowledge based data. At the beginning of the game the starting positions of chess pieces are well defined and during the game, at any time, the turn information (i.e. whether it is black players turn or white players turn), possible moves of every single piece are known. At each step, knowledge based data and the difference image information are combined and the move is interpreted.

Lastly, when a move is detected and a correct decision is made about it, the move is written to an output file, the reference image is renewed (i.e. a new reference image is taken) and knowledge database is updated according to the move. At the end, a log file of the game is generated. Current version of code implementation has the capability to detect invalid moves and produce a warning signal.

In this thesis, the well defined domain of the chess game makes it possible to produce a complete solution for the task of logging visual activities. Different problem domains that may be used to perform the same task can be compared as follows:

- Chess game analysis
- Sports video analysis
- Traffic scene analysis
- Security monitoring
- Context extraction from video

For chess game the event triggers and objects are very distinctive. Change detection is used for the analysis. Well defined rules of chess game are easily integrated in decision making. A well defined grammar can be used for coding actions (like c3c4, h3g5 or d4f2).

In sports video analysis, event triggers and objects are much more ambiguous compared to chess game analysis. Motion detection can be used for the analysis. Well defined rules of the football game can be integrated in decision making and in accordance with that, actions can be coded (e.g. A goal scored by team A, etc.)

In traffic scene analysis, just like sports video analysis, event triggers and objects are much more ambiguous compared to chess game analysis and motion tracking can be used for the analysis. Application specific rules can be defined and actions can be coded (e.g. Traffic flow rate is high, etc.)

In security monitoring, event triggers and objects are ambiguous. Motion tracking can be used for the analysis. Application specific rules can be defined and actions can be coded (e.g. Intruder alert!, etc.)

In context extraction from video, the case possible solutions are much more complicated than the specific examples described above. It requires advanced research on video analysis and artificial intelligence to process data and to select appropriate rules and grammars.

This study aims to be a first step into task oriented understanding of actions in video data. The well defined environment and the rules of the chess game provide us a good framework in order to use domain knowledge in a vision guided decision making process. The experience gained here can be extended to more complicated vision tasks involving more complicated environments. Besides attacking more complex problems, the future work involves making the algorithm more robust in terms of extracting reliable data and decision making.

APPENDIX A

CODE IMPLEMENTATION

This study requires a hard code implementation. OpenCV library and Microsoft Visual C++ is used for coding. The algorithm has been separated into two parts and those parts have been implemented individually. These parts are:

- Detecting chessboard
- Playing the game

A.1 Detecting Chessboard

Corresponding executable is *detect.exe*. Within the code, first a window is needed to observe the data gathered from the usb cam. The camera used in the experiments is a PC-CAM. The frames have the size of 320x240 in pixels. The images are defined as;
IplImage source=cvCreateImage(cvSize(320,240),8,3);*

Here $(320, 240)$ is the size of the image object, where 8 is the color depth and 3 is the number of channels (RGB format is used, hence number of channels is 3 here). To see the frames on screen a window is to be defined as;
cvNamedWindow("SOURCE",1);

Here "SOURCE" is the name and label of the window object. Parameter *1* is given as a default number. In order to gather frames from camera a *capture* object is to be defined.
*CvCapture * capture = NULL;*
and
capture =cvCaptureFromCAM(1);
is added to capture frames from the assumed camera. The object capture and source image are related with following routines:
source = GrabFrame(capture);
cvShowImage("SOURCE",source);

When the setting (i.e. installation of the board and camera) is ready the board is detected by using a function called
MyFindChessBoardCorners();

The floating point pixel coordinates of 4 inner corners are written in a text file called *world_coordinates.txt* for a later use. X and Y coordinates are written in separate lines for each corner. Hence the text file looks like;

$$\begin{bmatrix} 226.443649 & 134.403763 \\ 138.345444 & 40.220333 \\ 126.905121 & 189.404572 \\ 35.888817 & 115.054184 \end{bmatrix}$$

If the detection is successful, user can stop the program and pass to the next routine which is playing the game.

A.2 Playing the Game

Corresponding executable is play.exe. Within the code, first program reads floating point pixel coordinates fom the text file world_coordinates.txt, which has already been generated in chessboard detection phase. By using these coordinates, projection matrix is calculated by the function;

MyProjectionMatrixCalculate()

For this part two helper windows are created. One for the source view and one for the projected view. User can watch the scene from those windows. After the program start, players put their pieces in their places. If the projected view is not consistent with the Figure 3.8, necessary adjustment can be done by pressing **"r"** for a couple of times. Each press causes a 90^0 degrees of rotation. When the projected view is consistent with the chessboard standard representation, the player can start playing the game.

The program assumes that players do know "how to play chess", which means, the program does not consider improper movements of the players. According to the chess rules the starting positions of the pieces are well defined (shown in Figure 3.8). Taking this as an initial condition, program starts to analyze the game.

A.2.1 Analysis

As it has been mentioned before, the move table knowledge and chessboard knowledge is updated before every move. Hence, at the very beginning of the game this should be done. The first move is white players move. So the corresponding move table updates are performed for white player first. The updates are performed according to the chess rules. The data structure used to hold move table information for each piece

is an 8x8 array.

For every piece a generic function is defined to generate corresponding move table. For example for queen a function is defined as;

void MoveTableQueen(int queen_move[8][8],int color)

Where “int queen_move[8][8]”is an 8x8 integer array which represents the move table, whereas *int color* is an integer which indicates the color of the piece. 1 stands for white and -1 stands for black. Code example of *void MoveTableQueen()* can be given to demonstrate the move table generation algorithm.

```
void MoveTableQueen(int queen_move[8][8],int color){ \\ function decleration
int i,j,k; \\ variable declerations
for (i=0;i<8;i++)
for (j=0;j>8;j++) queen_move[i][j]=0; \\ initialization of the corresponding move table
for (i=0;i<8;i++)
for (j=0;j<8;j++)
{
if (chess_board[i][j]==2*color) \\ current position of the queen is determined
{
queen_move[i][j]=1;
k=1;
while ((i-k)≥0 && (i-k)<8 && (j-k)≥0 && (j-k)<8 ) \\ lower-left diagonal is going to be checked
{
if (chess_board[i-k][j-k]==0) {queen_move[i-k][j-k]=1;k++;} \\ if the position is empty it is considered as 1 in the move table
and program keeps on checking in the corresponding direction
else if (chess_board[i-k][j-k]*color<0) {queen_move[i-k][j-k]=1;k=10;} \\ if the position is occupied by an opponent piece it is
also considered as 1 and checking in the corresponding direction is stopped
else k=10; \\ otherwise it is occupied with a relative piece, then the checking terminates and the position is considered as 0
}
k=1;
while ((i-k)≥0 && (i-k)<8 && (j+k)≥0 && (j+k)<8 ) \\ upper-left diagonal is going to be checked
{
if (chess_board[i-k][j+k]==0) {queen_move[i-k][j+k]=1;k++;}
else if (chess_board[i-k][j+k]*color<0) {queen_move[i-k][j+k]=1;k=10;}
else k=10;
} \\ the analysis is the same
k=1;
while ((i+k)≥0 && (i+k)<8 && (j-k)≥0 && (j-k)<8 ) \\ lower-right diagonal is going to be checked
{
if (chess_board[i+k][j-k]==0) {queen_move[i+k][j-k]=1;k++;}
else if (chess_board[i+k][j-k]*color<0) {queen_move[i+k][j-k]=1;k=10;}
else k=10;
} \\ the analysis is the same
k=1; while ((i+k)≥0 && (i+k)<8 && (j+k)≥0 && (j+k)<8 ) \\ upper-right diagonal is going to be checked
{
if (chess_board[i+k][j+k]==0) {queen_move[i+k][j+k]=1;k++;}
else if (chess_board[i+k][j+k]*color<0) {queen_move[i+k][j+k]=1;k=10;}
```

```
else k=10;
} \\ the analysis is the same
k=1;
while ((i-k)≥0 && (i-k)<8 ) \\ left direction is going to be checked
{
if (chess_board[i-k][j]==0) {queen_move[i-k][j]=1;k++;}
else if (chess_board[i-k][j]*color<0) {queen_move[i-k][j]=1;k=10;}
else k=10;
} \\ the analysis is the same
k=1;
while ((j-k)≥0 && (j-k)<8 ) \\ down direction is going to be checked
{
if (chess_board[i][j-k]==0) {queen_move[i][j-k]=1;k++;}
else if (chess_board[i][j-k]*color<0) {queen_move[i][j-k]=1;k=10;}
else k=10;
} \\ the analysis is the same
k=1;
while ((i+k)≥0 && (i+k)<8 ) \\ right direction is going to be checked
{
if (chess_board[i+k][j]==0) {queen_move[i+k][j]=1;k++;}
else if (chess_board[i+k][j]*color<0) {queen_move[i+k][j]=1;k=10;}
else k=10;
} \\ the analysis is the same
k=1;
while ((j+k)≥0 && (j+k)<8 ) \\ up direction is going to be checked
{
if (chess_board[i][j+k]==0) {queen_move[i][j+k]=1;k++;}
else if (chess_board[i][j+k]*color<0) {queen_move[i][j+k]=1;k=10;}
else k=10;
} \\ the analysis is the same
}
}
}
```

All the move tables are analyzed and updated likewise. After the move table updates, the difference which is described in 5.1.2 is considered to predict the move. Method in 5.1.2.1 or method in 5.1.2.2 can be used to find correlation between difference image and chessboard positions.

Gathered correlation information is reevaluated with respect to the move tables. If an exact correspondence is found between them, the move is considered as a valid move and it is recorded to the log file.

REFERENCES

Assfalg J., Bertini M., Colombo C., Bimbo A.D. and Nunziati W. 2003. "Semantic annotation of soccer videos: automatic highlights identification", Computer Vision and Image Understanding, Vol.92, Issues 2-3, November-December 2003, pp.285-305.

Baird H.S., Thompson K. 1990. "Reading chess", Pattern Anal. Machine Intell., Vol.12, Issue 6, June 1990,pp.552-559.

Brunelli R., Mich O. and Modena C.M. 1999. "A Survey on the Automatic Indexing of Video Data", Journal of Visual Communication and Image Representation, Vol.10, Issue 2, June 1999, pp.78-112.

Campbell M., Hoane A.J. and Hsu F.H. 2002. "Deep Blue", Artificial Intelligence, Vol.134, Issues 1-2, January 2002, pp.57-83.

Dorada A., Calic J., Izquierdo E. 2004. "A rule-based video annotation system", IEEE Transactions on Circuits and Systems for Video Technology, Vol.14, Issue 5, May 2004, pp.622633.

Douglas D.H. and Peucker T.K. 1973. "Algorithms for the reduction of the number of points required to represent a digitised line or its caricature", The Canadian Cartographer, Vol.10, issue 2, pp.112-122.

Ekin A., Tekalp A.M., Mehrotra R. 2003. "Automatic soccer video analysis and summarization"; IEEE Transactions on Image Processing, Volume 12, Issue 7, July 2003, pp.796-807.

Gonzales R.C., Woods R.E., 1993. "Digital Image Processing", edited by Addison Wesley, pp.296-302 and 518-548.

Groen F.C.A, Den Boer G.A., Van Inge A. and Stam R. 1992. "A chess-playing robot: lab course in robot sensor integration", IEEE Transactions on Instrumentation and Measurement, Vol.41, Issue 6, December 1992, pp.911-914.

Gümüştekin Ş. 2004. "EE542 Lecture Notes at IYTE", 2004.

Gümüştekin Ş. 1999. "New Methods for the Construction of Panaromic and Plenoptic Images with Computer Vision Applications", PhD Thesis, 1999, University of Pittsburgh

Koprinska I. and Carrato S. 2001. "Temporal video segmentation: A survey", Signal Processing: Image Communication, Vol.16, Issue 5, January 2001, pp.477-500.

Ozan Ş. and Gümüştekin Ş. 2005. "A Case Study on Logging Visual Activities: Chess Game", Proceedings of Turkish Symposium on Artificial Intelligence and Neural Networks, June 2005, pp.88-97

Pavlidis T., 1982. "Algorithms for Graphics and Image Processing", edited by Computer Science Press, Rockville, Maryland, 1982.

Petrakis E.G.M 2004. "Binary Image Processing Lecture Notes", 2004.

Smoulders A.W.M., Worring M., Santini S., Gupta A. and Jain R. 2000. “Content based image retrieval at the end of the early years”, IEEE Trans. Pattern Anal. Machine Intell., vol.22, December 2000, pp.13491380.

Trucco E. and Verri A., 1998. “Introductory Techniques for 3D Computer Vision”, edited by Prentice Hall, 1998.

Uyar E., Gümüştekin Ş., Ozan Ş. and L.Çetin 2004. “A Computer Controlled Vision Oriented Robot Manipulator for Chess Game”, Prococeedings of Workshop on Research and Education in Control and Signal Processing, REDISCOVER 2004, Cavtat, Croatia.

Visvalingam M.and Whyatt J.D. 1990. “The Douglas Peucker Algorithm for Line Simplification: re evaluation through visualization”, Computer Graphics Forum University of Hull, Vol.9, Issue 3, pp.213-228.

WEB_1, 2000. Abeer George Ghuneim “Contour Tracing” Lecture Notes, 05/02/2005. http://www.cs.mcgill.ca/ aghnei/

WEB_2, 2005. Intel Open Source Computer Vision Library, 01/02/2005. http://www.intel.com/research/mrl/research/opencv/

WEB_3, 2004. “Actual Handbook Fédération International des Échecs”, 21/12/2004. http://www.fide.com/handbook.asp

WEB_4, 2005. “The Chess Variant Pages”, 03/03/2006. http://www.chessvariants.com/chess.html

WEB_5, 2000. Chris Halsall’s “How the CamCal Program Works”, 12/03/2005. http://www.linuxdevcenter.com/pub/a/linux/2000/09/22/camcal.html

Zimmermann J.Z., 1993. “Fuzzy Set Theory and its Applications”, edited by Kluwer Academic Publishers, 1993