Chess Vision – Final Report

12/5/2016

Group Members

Amanpreet Walia - aspw02

Xue Ying Shi - cse03020

Youn Sun (Joy) Choi - cse13111

Table of Contents

Overview

The Chess Vision project is system that applies computer vision techniques and algorithms to capture a game of chess played in the real world and accurately analyze what moves were made. Through a camera positioned above a stationary chess board, the system aims to output a virtual depiction of a game of Chess played in real-time in the physical world by two players.

This report details the following major components of the Chess Vision project:

- Board Projection
- Board Detection
- Hand Detection
- Piece Detection
- Move Determination
- Chess Board Mapping

Board Projection

The primary task of the system is to isolate the chess board from any given frame of the video feed. This task is accomplished by an initial setup phase for the user. When the program is started, the system will present a static image taken by the camera. The user will then be expected to click on the four corners of the chess board in the image to proceed. (See Appendix A.1 & A.2).

Figure 1 - Example static image with added arrows to show where the user should click

Once the four corners are specified, the system first orders the points (see Appendix A.3), and then proceeds to calculate an orthogonal rectangle to which perspective transformation will be applied. The method four_point_transform (see Appendix A.4) achieves this. Figure 2 additionally shows how the box is derived.

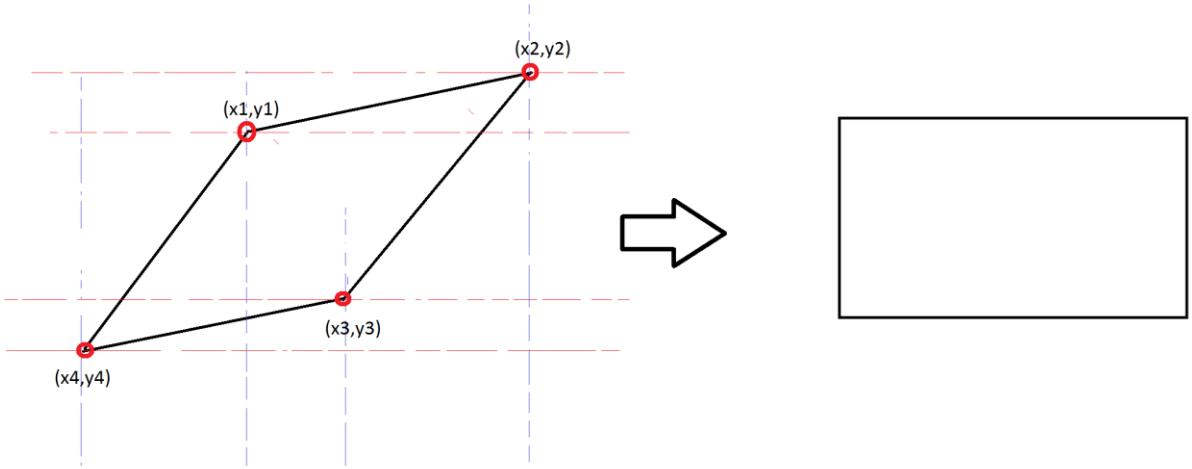


Figure 2 - Left shows a possible polygon formed from the 4 corners specified. Right shows the calculated rectangle.

$$\text{width} = \max(\sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2}, \sqrt{(x_3 - x_4)^2 + (y_3 - y_4)^2}$$

$$\text{height} = \max(\sqrt{(x_1 - x_3)^2 + (y_1 - y_3)^2}, \sqrt{(x_2 - x_4)^2 + (y_2 - y_4)^2}$$

After this point, the OpenCV getPerspectiveTransform function is used in the main program (see Appendix B.1) to get the perspective transformation matrix. The transformation can be generalized as the matrix equation:

p' = R p + T or p = Rt (p' – T)

The transformation matrix is applied to relocate points in the original image to match the shape of the derived rectangle.

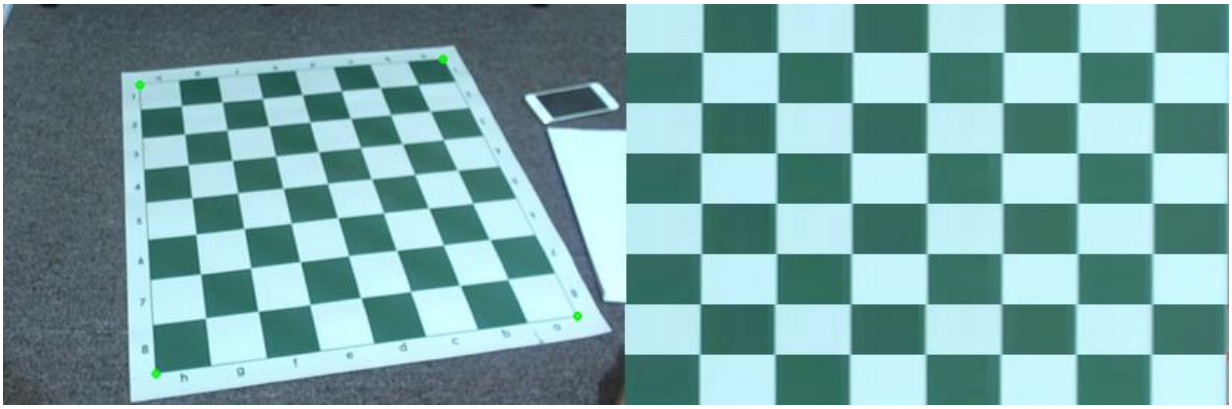


Figure 3 - Left shows a possible initial setup static image with the four corners clicked. Right shows the result of the board projection step, including perspective transform.

Board Detection

The next major step for the Chess Vision system is to detect the main feature of the board that is the grid squares. This is important, since we need to localize each grid square to be able to detect and determine whether or not a piece occupies the square. For this step we use a simple chess board dimensional analysis to give a practically very good estimate of corners with high accuracy.

Dimensional analysis simply involves dividing the length and the width of the board into 8 equal segments to get the subsequent length and width of each grid square. The code implementing the dimensional analysis and ChessSquare object creation can be seen from Appendix B.2. Appendix C shows the definition of the ChessSquare class, and it can be highlighted that each ChessSquare holds the following key properties:

- The centroid coordinate
- The four corner coordinates
- Its position in the grid in Cartesian coordinates.

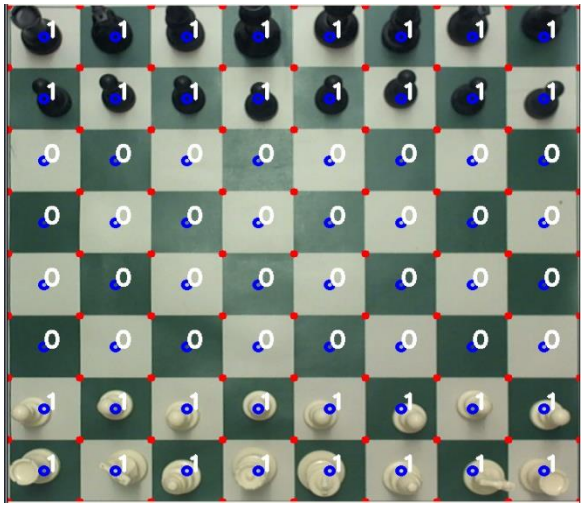


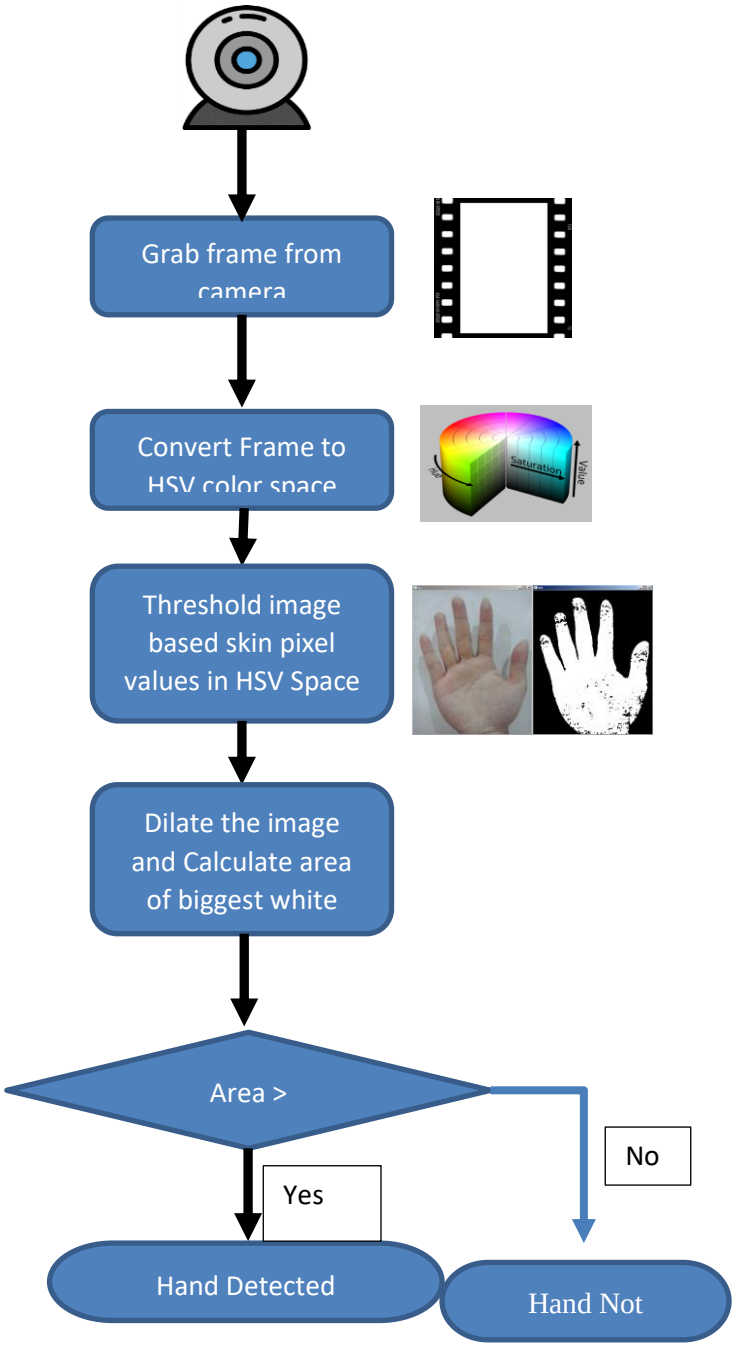
Figure 4 - After completion of board detection, we can see the result as blue circles indicating the centroid and red dots indicating the corners of each ChessSquare

Hand Detection

Next, before getting to the most significant task of the system, which is to detect chess pieces and their movements on the board, it is necessary to determine whether a hand is obstructing the camera view. In the physical world, players move pieces with their hands, and no processing of movement should be done while the player is in progress of making a move. **The Challenge is to** determine if a hand is in the image – do not try to process chess piece movement if a player is in the process of moving a piece.Define a range of skin tones ((H,S,V)=>[0, 48, 80]to[20, 255, 255]) to be detected on an HSV scale.Convert the relevant image points to HSV values and determine if within range. Create a binary image based on which points are inside the range and which points are not.

1. Dilate the image.
2. Find the largest contour and area of 1s.
3. If the area exceeds a set threshold, a hand is detected to be in the image

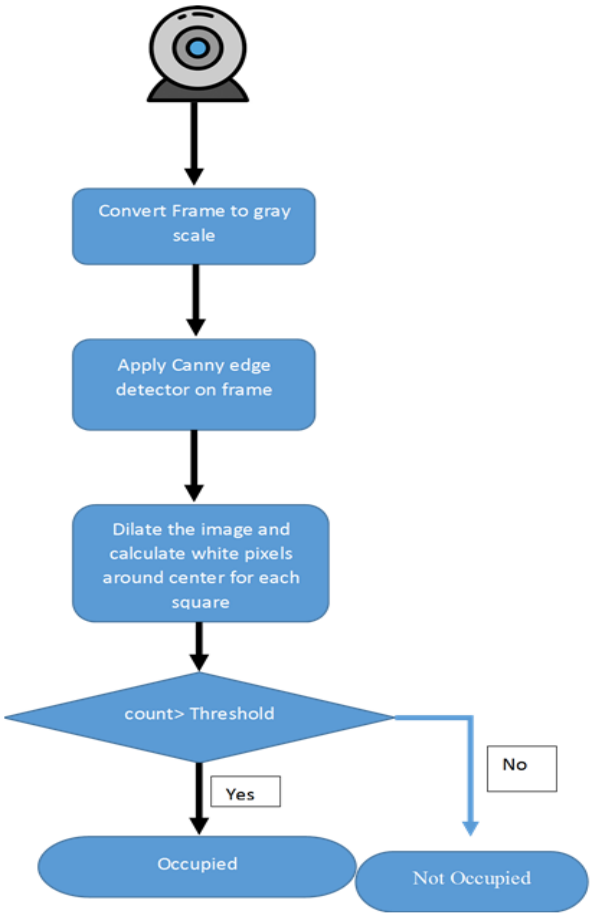
Figure 5 - Left shows a hand in the frame: successfully halts piece detection. Right shows a grayscale image after a skin mask is applied, and the largest contour and area of skin is detected.


Grab frame from camera
Convert Frame to HSV color space
Saturation
Value
Threshold image based skin pixel values in HSV Space
Dilate the image and Calculate area of biggest white
Area >
No
Yes
Hand Detected
Hand Not

Piece Detection

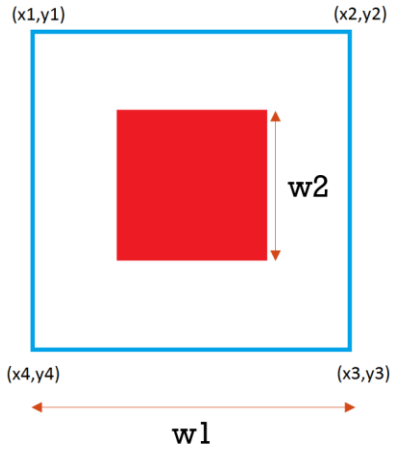
The next step, being the most important, focuses on the content of each localized grid square to determine whether it contains a piece, and additionally, whether the piece is white or black.

The first algorithm that is applied to a frame determined to be free of hand movement is the Canny edge detection algorithm (see Appendix C.2). It is applied to a greyscale image of the current frame to get the edges of each piece.



For determining the central points, a certain percentage of length of chess square is chosen & all the pixels inside this square are checked for how many white pixels are

there inside this square area. This value is then compared against threshold value to determine whether a chess square should be considered as occupied or not. This threshold value depends on various external factors like lightening conditions, shadows etc., however we found for this project that in most settings any non-zero value works fairly well in differentiating occupied squares from non-occupied ones.



Individual Chess square

$w2$ *is calculated by using* % *threshold of* $w1$

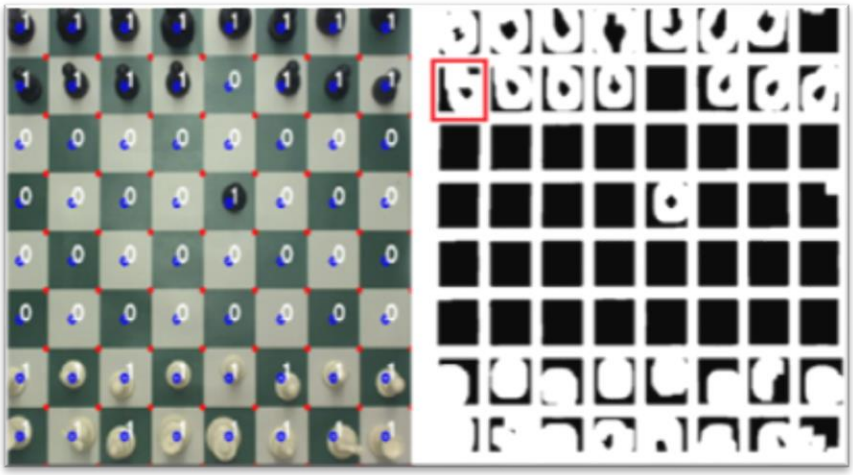


Figure 6- Left shows the chess board frame. Right shows the result of Canny edge detection.

Next, to determine whether the occupied grid square contains a black piece or a white piece, a black color mask is applied to the frame, and the number of black pixels within each ChessSquare is counted. If the count passes a threshold set in modules.py (see Appendix F - min_black_pixels), the piece is determined to be black. It is otherwise determined to be white (given that it also passes the white piece threshold).

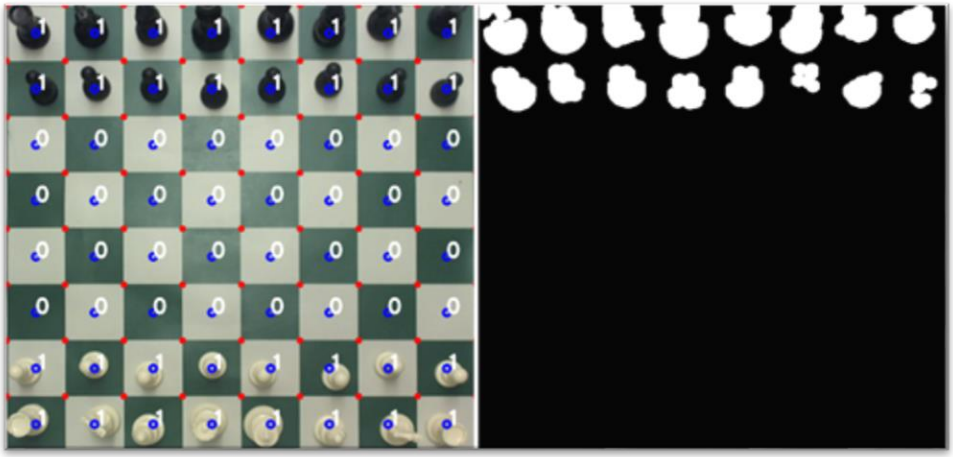
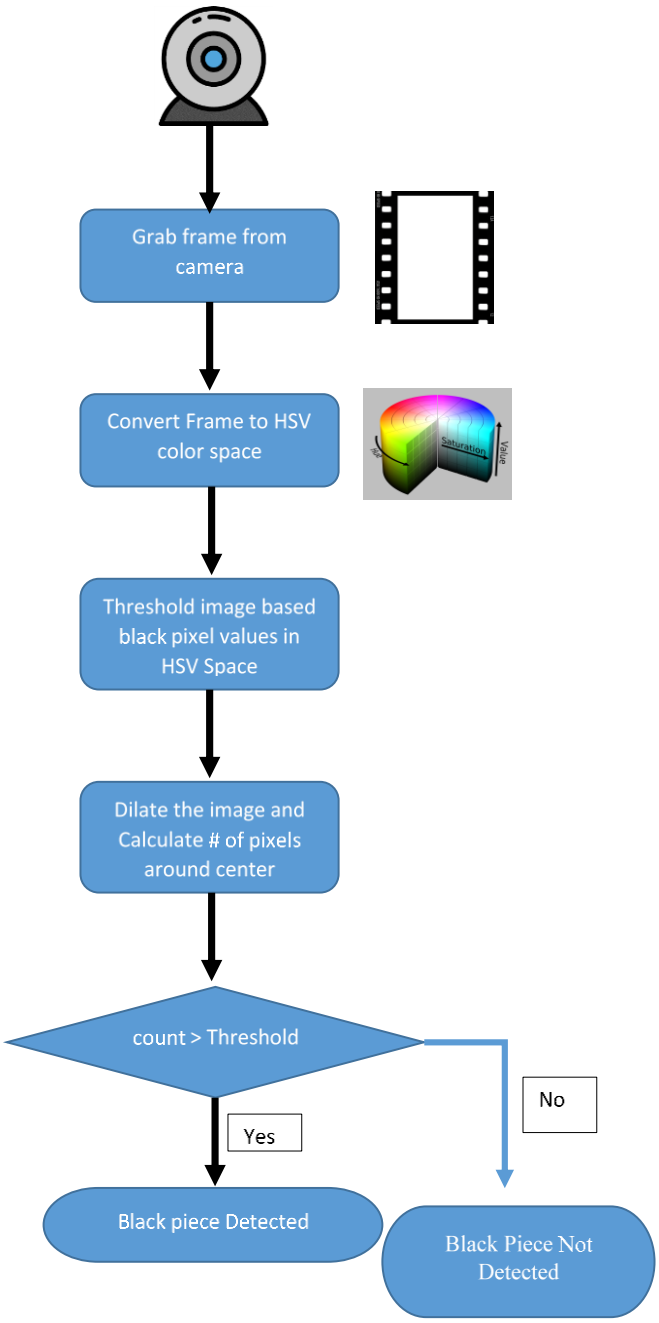


Figure 7 - Left shows the chess board frame. Right shows the concentration of detected black pixels in the image.

Algorithm for black piece detection

- Define a range of black color{(H,S,V)=[0, 0, 0]to [180, 255, 30]} to be detected on an HSV scale.
- Convert the relevant image points to HSV values and determine if within range. Create a binary image based on which points are inside the range and which points are not.
- Dilate the image.
- Calculate the number of pixels around center using same technique as before.
- If the count exceeds a set threshold, a black is detected to be in the image.


Grab frame from camera
Convert Frame to HSV color space
Threshold image based black pixel values in HSV Space
Dilate the image and Calculate # of pixels around center
count > Threshold
No
Yes
Black piece Detected
Black Piece Not Detected

Move Determination

Once it is determined which squares contain black pieces, white pieces, or no pieces, the information for each ChessSquare is compiled and sent off to the ChessBoard module (see Appendix E) for the move determination step. The format by which the information is compiled is as follows: a 2D 8x8 array is populated based on the corresponding ChessSquare. A 'b' is set in the array if the corresponding grid position is determined to contain a black piece; a 'w' is set for white pieces; and '0' is set if it is determined that there are no pieces in the ChessSquare. The compiled array is then shipped to the ChessBoard module's update method (see Appendix E.2), where the array is compared to the previous frame's array.

If the system finds a difference between the arrays, the following logic is applied to determine what move was made:

- If a square in the previous state contained a piece, but current state does not: the piece in the square from the previous state was moved -> the Cartesian coordinate of the square is the movement starting position.
- The destination is either: a square in the current state that contains a piece, but did not contain a piece in the previous state; or a square that contains a piece of a different color in the current state than the previous state.

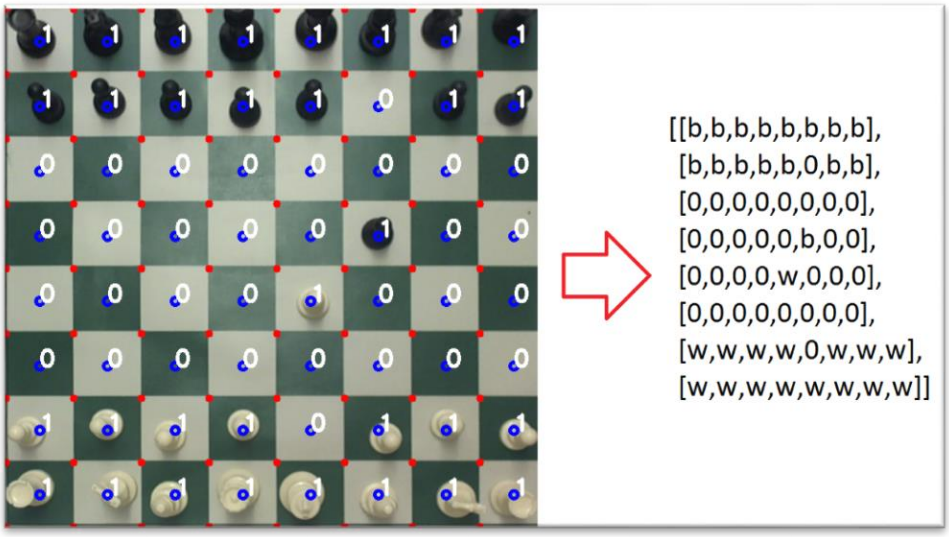


Figure 8 - Left shows the chess board frame. Right shows the corresponding 2D 8x8 array representation that is sent to ChessSquare.update

Chess Board Mapping

Once it is determined that there was chess piece movement from a given start grid position to target grid position, the final step remaining for the system is to graphically show which piece was moved. The Game class (see Appendix E.3) keeps track of the state of the board and which pieces are at which position at any given time. Calling the domove method (see Appendix E.4), feeding in the paramaters of the start position and the target position, will simply update the board to move the pieces accordingly. With each domove call, the system prints out the board (see Appendix E.5), showing which pieces are currently at which position.

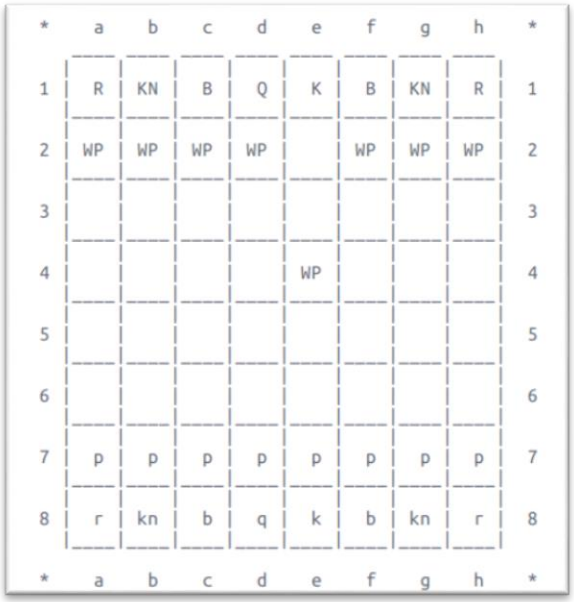


Figure 9 - Example graphical representation of the chess board state as console output.

Appendix A – PerspectiveTransform.py

```
from modules import *

class PerspectiveTransform(object):
```

A.1 - Constructor

```
    def __init__(self, image):
        self.coordinates = []
        self.img = image
        clone = self.img.copy()
        cv2.namedWindow("Chessboard")
        cv2.setMouseCallback("Chessboard", self.draw_circle)

        while True:
            cv2.imshow("Chessboard", self.img)
            key = cv2.waitKey(1) & 0xFF

            if key == ord('r'):
                PerspectiveTransform.img = clone.copy()
            if key == ord('c'):
                break
        cv2.destroyAllWindows()
```

```
        if len(self.coordinates) == 4:
            self.four_point_transform(self.img, np.asarray(self.coordinates))
```

A.2 - draw_circle

```
    # static class data members
    def draw_circle(self, event, x, y, flags, param):
        if event == cv2.EVENT_LBUTTONDBLCLK:
            cv2.circle(self.img, (x, y), 4, (0, 255, 0), 3)
            self.coordinates.append((x, y))
```

A.3 - order_points

```
    def order_points(self, pts):
        # initialzie a list of coordinates that will be ordered
        # such that the first entry in the list is the top-left,
        # the second entry is the top-right, the third is the
        # bottom-right, and the fourth is the bottom-left
        rect = np.zeros((4, 2), dtype="float32")

        # the top-left point will have the smallest sum, whereas
        # the bottom-right point will have the largest sum
        s = pts.sum(axis=1)
        rect[0] = pts[np.argmin(s)]
        rect[2] = pts[np.argmax(s)]

        # now, compute the difference between the points, the
        # top-right point will have the smallest difference,
        # whereas the bottom-left will have the largest difference
        diff = np.diff(pts, axis=1)
        rect[1] = pts[np.argmin(diff)]
        rect[3] = pts[np.argmax(diff)]

        # return the ordered coordinates
        return rect
```

A.4 - four_point_transform

```
    def four_point_transform(self,image, pts):
        # obtain a consistent order of the points and unpack them
        # individually
        rect = self.order_points(pts)
        (tl, tr, br, bl) = rect

        # compute the width of the new image, which will be the
        # maximum distance between bottom-right and bottom-left
        # x-coordiates or the top-right and top-left x-coordinates
        widthA = np.sqrt(((br[0] - bl[0]) ** 2) + ((br[1] - bl[1]) ** 2))
        widthB = np.sqrt(((tr[0] - tl[0]) ** 2) + ((tr[1] - tl[1]) ** 2))
        maxWidth = max(int(widthA), int(widthB))
```

```
        # compute the height of the new image, which will be the
        # maximum distance between the top-right and bottom-right
        # y-coordinates or the top-left and bottom-left y-coordinates
        heightA = np.sqrt(((tr[0] - br[0]) ** 2) + ((tr[1] - br[1]) ** 2))
        heightB = np.sqrt(((tl[0] - bl[0]) ** 2) + ((tl[1] - bl[1]) ** 2))
        maxHeight = max(int(heightA), int(heightB))

        # now that we have the dimensions of the new image, construct
        # the set of destination points to obtain a "birds eye view",
        # (i.e. top-down view) of the image, again specifying points
        # in the top-left, top-right, bottom-right, and bottom-left
        # order
        dst = np.array([
            [0, 0],
            [maxWidth - 1, 0],
            [maxWidth - 1, maxHeight - 1],
            [0, maxHeight - 1]], dtype="float32")

        # compute the perspective transform matrix and then apply it
        self.transfmatrix = cv2.getPerspectiveTransform(rect, dst)
        self.dim = (maxWidth, maxHeight)
```

Appendix B – main.py

```
from PerspectiveTransform import PerspectiveTransform
from framegrabber import *
from modules import *
from ChessSquare import *
from SkinDetector import *
from chessBoard import *


def sharpen_image(frame):
    # Create the identity filter, but with the 1 shifted to the right!
    kernel = np.zeros((9, 9), np.float32)
    kernel[4, 4] = 2.0  # Identity, times two!
    # Create a box filter:
    boxFilter = np.ones((9, 9), np.float32) / 81.0
    # Subtract the two:
    kernel = kernel - boxFilter
    # Note that we are subject to overflow and underflow here...but I believe
that
    # filter2D clips top and bottom ranges on the output, plus you'd need a
    # very bright or very dark pixel surrounded by the opposite type.
    return cv2.filter2D(frame, -1, kernel)


#frame grabber object to grab frames
frames = frame_grabber()

# Create a new instance of the board interface for a new game
```

```
boardInterface = BoardInterface()

#object of class skin Detector
skinDetector = SkinDetector(lower,upper,skinThresh)
```

B.1 – Apply Perspective Transform

```
#Determining homorgraphy matrix for chess board
init_frame = frames.get_frame()
perspectransform = PerspectiveTransform(init_frame)
init_per_frame = cv2.warpPerspective(init_frame, perspectransform.transfmatrix,
perspectransform.dim)
```

B.2 – Get ChessSquare Objects

```
#Chess square corners
(width,height) = perspectransform.dim
x = np.linspace(0,width,9,dtype=np.int)
y = np.linspace(0,height,9,dtype=np.int)

#predicted corners of chess board based on dimension calculation
pred_corners = np.transpose([np.tile(x,len(y)),np.repeat(y,len(x))])
pred_corners_list = pred_corners.tolist()

#reshaping the corners to make it suitable for conevrsion into chess square
format
pred_corners = pred_corners.reshape(9, 9, 2)
pred_corners_list1 = pred_corners.tolist()

#using these corners to initialise multidimensional array of chess squares class
corners = np.zeros((8,8,4,2),dtype=np.int)
for i in range(0,8):
    for j in range(0,8):
        corners[i,j,0] = pred_corners[i,j]
        corners[i,j,1] = pred_corners[i+1,j]
        corners[i,j,2] = pred_corners[i+1,j+1]
        corners[i,j,3] = pred_corners[i,j+1]

#list of chess square objects
squares = []

# Declaring chess square object relating to its (x,y) & associating corners to it
and
# initial cropped image of that square
for x in range(0,8):
    for y in range(0,8):
        s =
ChessSquare((x,y),corners[x,y],init_per_frame,min_white_count,min_black_pixels)
        squares.append(s)

# setting font for text
font = cv2.FONT_HERSHEY_SIMPLEX
```

```
(w, h) = perspectransform.dim


'''
This is main loop where all the processing happens, a frame is grabbed from
frame-grabber class and perspective projection is applied to it to extract the
ROI i.e. chessboard. After that
'''
while True:
    # frame is garbbed from frame grabber class
    frame = frames.get_frame()
    # perspective projection is applied on the grabbed frame to re orient the
image and extarct the ROI from image
    show_frame = cv2.warpPerspective(frame, perspectransform.transfmatrix,
perspectransform.dim)
```

B.3 – Hand Detection

```
   # Converting to HSV frame
    hsv = cv2.cvtColor(show_frame,cv2.COLOR_BGR2HSV)
    black_mask = cv2.inRange(hsv, black_lower, black_upper)
    kernel = cv2.getStructuringElement(cv2.MORPH_ELLIPSE, (11, 11))
    black_mask = cv2.dilate(black_mask, kernel, iterations=2)

    # Checking if significant portion of skin is found in frame
    # converting the image into grayscale
    gray_frame = cv2.cvtColor(show_frame, cv2.COLOR_BGR2GRAY)
    # blurring the image with gaussian filter to remove noise
    gray_frame = cv2.GaussianBlur(gray_frame,G_kernel_size, 0)
    #sharpening the image for better edge detection
    gray_frame = sharpen_image(gray_frame)
    # Applying Canny Edge detection to detect the edges
    detected_edges = cv2.Canny(gray_frame, canny_l_threshold, canny_u_threshold)
    # dilating the image to make binary image more suitable for piece detection
    dilated = cv2.dilate(detected_edges, np.ones((19, 19)))

    if not skinDetector.check_skin(show_frame):
        board = [[0 for x in range(w)] for y in range(h)]

        for x in range(0, 8):
            for y in range(0, 8):
                index = 8*x + y

                text,isBlack =
squares[index].check_square(show_frame,dilated,black_mask)

                if text == 0:
                    board[x][y] = text
                elif isBlack:
                    board[x][y] = "b"
                else:
```

```
                    board[x][y] = "w"

                cx, cy = squares[index].centroid
                cv2.circle(show_frame, (cx, cy), 5, (255, 0, 0), 3)

                #print(text)
                cv2.putText(show_frame, str(text), (cx, cy), font, 1, (255, 255,
255), 3)
                # Marking corners on the frame shown for each corner calculated
with blue colored circles

        boardInterface.update(board)
    for j in pred_corners_list:
        x, y = j
        cv2.circle(show_frame, (x, y), 5, (0, 0, 255), -1)


    cv2.imshow('Chess Board', show_frame)
    cv2.imshow('Piece detection',dilated)
    cv2.imshow('Black Pieces',black_mask)
    if cv2.waitKey(1) & 0xFF == ord('q'):
        break
```

Appendix C – ChessSquare.py

```
from modules import *

class ChessSquare(object):
```

C.1 - Constructor

```
    def __init__(self,coord,pts,init_frame,white_count_thresh,black_piece_thresh):
        #chess sqaure cartesian coordinate
        self.coordinate = coord

        #Four points associated with this square
        self.points = pts

        #centroid of this square
        self.centroid = np.asarray((sum(np.asarray(pts))/4),dtype=np.int)



        #Store initial square coordinates
        self.init_square = self.crop_square(init_frame,self.points)
        self.init_square_gray = cv2.cvtColor(self.init_square,cv2.COLOR_BGR2GRAY)

        #whether a chess square is filled or not
        self.filled = False
```

```
        #thresholding parameters
        self.white_count_thresh = white_count_thresh
        self.black_piece_thresh = black_piece_thresh

        #name = "Coordinate #"+str(self.coordinate)
        #cv2.imshow(name,self.init_sqr)

    def name(self):
        return str(self.coordinate)

    def show_square(self,image_frame):
        return self.crop_square(image_frame,self.points)

    def crop_square(self,frame,coordinates):
        # image of initial square of the chess board
        # top leftcorner
        (x1, y1) = coordinates[0]
        # bottom right corner
        (x3, y3) = coordinates[2]
        return frame[y1:y3, x1:x3]
```

C.2 - check_square

```
def check_square(self,image_frame,canny_image,black_mask):

        # getting width of the square
        sq_width = abs((self.points[3,:]-self.points[0,:])[0])
        (cx,cy) = self.centroid - self.points[0,:]
        perc_area =  0.50
        thres_square_dim = int(sq_width*0.4)
        #cropping the square for
        curr_square = self.crop_square(canny_image, self.points)
        black_piece = self.crop_square(black_mask,self.points)
        count_canny = 0
        count_black_piece = 0
        for i in range(cx-thres_square_dim/2,cx+thres_square_dim/2):
            for j in range(cy-thres_square_dim/2,cy+thres_square_dim/2):
                if curr_square[i,j] == 255:
                    count_canny += 1
                if black_piece[i,j] == 255:
                    count_black_piece += 1

        if count_canny > self.white_count_thresh:
            canny_piece_detected = True
        else:
            canny_piece_detected = False

        if count_black_piece > self.black_piece_thresh:
            black_piece_detected = True
        else:
```

```
            black_piece_detected = False

        if canny_piece_detected and not black_piece_detected:
            return (1, False)
        elif black_piece_detected:
            return (1, True)
        else:
            return (0, False)

    def background_sub(self,image_frame):
        curr_square = self.crop_square(image_frame, self.points)
        curr_square_gray = cv2.cvtColor(curr_square, cv2.COLOR_BGR2GRAY)
        sub = cv2.subtract(self.init_square_gray,curr_square_gray)
        return sub
```

Appendix D – SkinDetector.py

```
from modules import *

class SkinDetector(object):
```

D.1 - Constructor

```
    def __init__(self,lower,upper,skinThresh):

        # HSV Model for skin color using upper and lower threshold values
        self.lskinRange = lower
        self.uskinRange = upper

        # parameter to decide how much skin area is sufficient to reject the
frame
        self.skinAreaThresh = skinThresh

        # five states of skin detector which are initially set to true to
initially
        # detect that skin is there, this is just transition step and hence will
not
 # create problem
        self.skin_state1 = True
        self.skin_state2 = True
        self.skin_state3 = True
        self.skin_state4 = True
        self.skin_state5 = True
```

D.2 – Find the largest contour and area of skin

```
    def get_largest_contour_area(self,contours):
        index = 0
        temp = 0
        count = 0
```

```
        for c in contours:
            area = cv2.contourArea(c)
            if temp < area:
                temp = area
                index = count
            count += 1
        return index, temp

    def get_frame_area(self,frame):
        (width,height,dim) = frame.shape
        return width*height

    def check_skin(self,frame):

        frame_area = self.get_frame_area(frame)
        skin_area = self.get_skin_area(frame)

        rel_skin_area = skin_area/frame_area
        if rel_skin_area < self.skinAreaThresh:
            return self.skin_state_stabiliser(False)
        else:
            return self.skin_state_stabiliser(True)

    def get_skin_area(self,frame):
        # converting the frame to HSV color space to apply skin detection model
        hsv_frame = cv2.cvtColor(frame, cv2.COLOR_BGR2HSV)
        #grayscale frame
        gray_frame = cv2.cvtColor(frame,cv2.COLOR_BGR2GRAY)
        # detecting all the pixels which are lying in range of skin color model
        skin_mask = cv2.inRange(hsv_frame, self.lskinRange, self.uskinRange)

        # apply a series of erosions and dilations to the mask
        # using an elliptical kernel
        kernel = cv2.getStructuringElement(cv2.MORPH_ELLIPSE, (11, 11))
        skin_mask = cv2.erode(skin_mask, kernel, iterations=2)
        skin_mask = cv2.dilate(skin_mask, kernel, iterations=2)

        # detecting the contours on skin blobs
        contours, hierarchy = cv2.findContours(skin_mask, cv2.RETR_TREE,
cv2.CHAIN_APPROX_SIMPLE)

        #getting index of the largest contour and its area
        index,skin_area = self.get_largest_contour_area(contours)

        zeros = np.zeros(gray_frame.shape, dtype=np.uint8)
        cv2.drawContours(zeros, contours, index, 255, -1)
        ROI = cv2.bitwise_and(gray_frame, gray_frame, mask=zeros)
        cv2.imshow('Detected Skin',ROI)

        return skin_area
```

```
    # This function is necessary to remove bumpy reading from skin detector as
during
    # transition phase detector reading oscillate between False and True, Hence
for
    # stabilising this five consecutive false are required to make detector vote
for
    # no skin detected, hence transition phase is tackled.
    def skin_state_stabiliser(self,skin_curr_state):
        res = self.skin_state1 or self.skin_state2 or self.skin_state3 or
self.skin_state4 or self.skin_state5
        self.skin_state5 = self.skin_state4
        self.skin_state4 = self.skin_state3
        self.skin_state3 = self.skin_state2
        self.skin_state2 = self.skin_state1
        self.skin_state1 = skin_curr_state
        return res
```

Appendix E – ChessBoard.py

```
import os,sys

class BoardInterface(object):
```

E.1 - Constructor

```
   def __init__(self):
      self.binaryBoard = [[0]*8 for i in range(8)]
      self.initialized = False;
      self.game = Game()
      self.game.printboard()

E.2 - update
   def update(self, updatedBoard):
    startrow = 0
    startcol = 0
    endrow = 0
    endcol = 0
    changed = 0

    if self.initialized == False:
        self.binaryBoard = updatedBoard
        for row in range(0,8):
          for col in range(0,8):
              print(self.binaryBoard[row][col])
        self.initialized = True
    else:
        for row in range(0,8):
          for col in range(0,8):
            if self.binaryBoard[row][col] != updatedBoard[row][col]:
              if updatedBoard[row][col] == 0:
```

```
                changed = 1
                startrow = row
                startcol = col
              else:
                endrow = row
                endcol = col

    if changed == 1:
      temp = self.binaryBoard[startrow][startcol]
      self.binaryBoard[startrow][startcol] = 0
      self.binaryBoard[endrow][endcol] = temp

      for row in range(0, 8):
          for col in range(0, 8):
              print(self.binaryBoard[row][col])

    start = (startrow, startcol)
      end = (endrow, endcol)

      self.game.domove(start, end)

class Player(object):

    allsquares = [(x, y) for x in range(8) for y in range(8)]

    def __init__(self, colour):
        self.colour   = colour

    def __str__(self):
        return self.colourself.colour

    def getpieces(self, board):
        return [pos for pos in board if board[pos].colour is self.colour]

class Piece(object):

    def __init__(self, piecename, position, player):
        self.colour    = player.colour
        self.piecename = piecename
        self.position  = position
        self.nrofmoves = 0

    def __str__(self):
        if self.colour is 'white':
            if self.piecename is 'p':
                return 'WP'
            else:
                return self.piecename.upper()
        else:
            return self.piecename
```

E.3 - Game Class

```
class Game(object):

    def __init__(self):

        self.board = dict()
        playera = Player('white')
        playerb = Player('black')

        for player in [playera, playerb]:
            if player.colour is 'white':
                brow, frow = 0, 1
                player.enpassantrow = 4
            else:
                brow, frow = 7, 6
                player.enpassantrow = 3

            player.longrook  = (brow, 0)
            player.longrook_target = \
            (player.longrook[0], player.longrook[1]+3)

            player.shortrook = (brow, 7)
            player.shortrook_target = \
            (player.shortrook[0], player.shortrook[1]-2)


            [self.board.setdefault((frow,x), Piece('p', (frow,x), player)) \
            for x in range(8)]
            [self.board.setdefault((brow,x), Piece('r', (brow,x), player)) \
            for x in [0,7]]
            [self.board.setdefault((brow,x), Piece('kn',(brow,x), player)) \
            for x in [1,6]]
            [self.board.setdefault((brow,x), Piece('b', (brow,x), player)) \
            for x in [2,5]]
            self.board.setdefault((brow,3),  Piece('q', (brow,3), player))
            self.board.setdefault((brow,4),  Piece('k', (brow,4), player))
```

E.4 - domove

```
  def domove(self, start, target):

        self.savedtargetpiece = None
        if target in self.board:
            self.savedtargetpiece = self.board[target]

        if self.board[start]:
            self.board[target] = self.board[start]
            self.board[target].position = target
            del self.board[start]

        self.printboard()
```

```
    def getpiece(self, grid):

        startcol  = int(ord(grid[0].lower())-97)
        startrow  = int(grid[1])-1
        start     = (startrow, startcol)

        if start in self.board:
          return self.board[start].piecename
```

E.5 - printboard

```
    def printboard(self):

        topbottom=['*','a','b','c','d','e','f','g','h','*']
        sides=['1','2','3','4','5','6','7','8']
        tbspacer=' '*6
        rowspacer=' '*5
        cellspacer=' '*4
        empty=' '*3

        print
        for field in topbottom:
            print "%4s" % field,
        print

        print tbspacer+("_"*4+' ')*8

        for row in range(8):
            print(rowspacer+(('|'+cellspacer)*9))
            print "%4s" % sides[row],('|'),
            for col in range(8):
                if (row, col) not in self.board:
                    print empty+'|',
                else:
                    print "%2s" % self.board[(row, col)],('|'),
            print "%2s" % sides[row],
            print
            print rowspacer+'|'+(("_"*4+'|')*8)
        print

        for field in topbottom:
            print "%4s" % field,

        print "\n"
```

Appendix F – modules.py

```
import cv2
import numpy as np
import time
```

```
'''
Parameters Used inside Code
'''
#Gaussian kernel size used for blurring
G_kernel_size = (3,3)

#canny thresholding parameters
canny_u_threshold = 200
canny_l_threshold = 80

# define the upper and lower boundaries of the HSV pixel
# intensities to be considered 'skin'
lower = np.array([0, 48, 80], dtype = "uint8")
upper = np.array([20, 255, 255], dtype = "uint8")

black_lower = np.array([0, 0, 0], dtype = "uint8")
black_upper = np.array([180, 255, 30], dtype = "uint8")

#threshhold for % of skin area detected
skinThresh = 0.00025

#Minimum number of white pixels needed for square to be counted as occupied
min_white_count = 1
#minimum number of black detected pixels in square
min_black_pixels = 200
```

Appendix G – FrameGrabber.py

```
from modules import *

class frame_grabber(object):

    def __init__(self):
        self.video = cv2.VideoCapture(1)
        self.video.set(3, 1280)
        self.video.set(4, 1024)
        self.video.set(15, 0.8)
        self.reduceFactor = 2

    def __del__(self):
        self.video.release()

    def get_frame(self,redfactor=1):
        success, image = self.video.read()
        [height,width] = image.shape[0],image.shape[1]
        self.reduceFactor = redfactor
        image =
cv2.resize(image,(width/self.reduceFactor,height/self.reduceFactor),interpolation
=cv2.INTER_CUBIC)
        return image
```