INTRODUCTION TO COMPUTER VISION

CHESS ASSISTANT

December 10, 2017

Abhilesh Borode and Mason Wilie

Colorado School of Mines

INTRODUCTION

Motivation

Chess is a a game in which the objective is to capture the enemy player's king, while simultaneously protecting their own. Even being one of the most popular sports in the world with over 605 million players [1], it is a relatively difficult game to learn from the ground up. Each chess piece has different legal moves which they can take, moves which can be hard for a beginner to remember. With the help of computer vision, it is possible to make a "chess assistant" which allows the player to select the piece they would like to move and have the possible moves displayed on the board. This allows the player to, with no prior knowledge of chess, experiment with different pieces and over time memorize the moves that each piece can make.

Other augmented reality chess assistants exist on the market, though they are complicated and most require prior knowledge of the game or augmented reality to be able to play. One of the objectives of this project was to be able to create the assistant using things that can be cheap and easily obtained by individuals. This is why both a printed chess board and printed pieces have been chosen. This allows the player to use the chess assistant with nothing more than a computer that has a web-cam, and a printed out chess board and pieces, lowering the cost barrier for people to play and learn the game.

Assumptions

In order to accomplish this project, a few assumptions needed to be made. The first of which is relatively good and consistent lighting in the playing environment, with the camera pointing somewhat perpendicular to the playing board. This allows for the program to more easily detect the board, identify pieces, and identify the occupancy of squares. Another assumption made for this project is that the chess pieces are close to the center of the squares which they occupy at all times. This allows for easier matching for identification of the piece. It is also assumed that the pieces used by the player are the same as the ones that have been used in the creation of the program. Finally, piece selection a player must select a square that contains a chess piece. These assumptions are reasonable as all are realizable in a regular household environment.

Previous Works

Chess is an extremely popular game for the computer science community to analyze, simulate, and try to create algorithms which allow for optimal game play. Because of this, many different AR chess systems exist, most of which are complicated and focus more on actual game play than educating the player on how each piece can move. One of these projects, coming out of Baskent University in Turkey, is similar to the chess assistant in the sense that it focuses on the moves which a player has taken. However, this application focuses more on checking that the correct moves have been made rather than assisting the player in making the moves[3]. Most of the previous work mentioned above assumes people know how to play chess. Thus we are building on the preexisting models mentioned above in a way that our model becomes more of a computer vision chess assistant. It is a lot more user interactive which makes it better than any pre-existing chess game rule book or any video explanation. One can just clone our model by directly taking a print out of their own chessboard along with the chess pieces, allowing them to get started learning to play chess.

METHODS

Creating an Orthophoto

Firstly, we feed in a stream of frames of images of the chessboard and take a still image, shown in figure 1, which we need use to crate an orthophoto. This is an important because it simplifies our processing by segmenting the checkerboard image into identically sized squares. The user then selects the square with a chess piece in it, using the ginput function built into Matlab. The square is then manually cropped out, considering the dimensions of all the squares in a chess board are same this method turns out to be consistent. We use a modified version of the findcheckerboard function we created in class, which returns the corners of the chess board. We then make use of the fitgeotrans function, in which we pass in the dimensions of the predefined constant image points. This gives us a transformation matrix to an orthophoto. A reference frame is created of the same dimension using the imref2d function. Finally, using the imwarp function in Matlab we create the orthophoto, shown in figure 2, using the same transformation matrix from figeotrans. This conversion to an orthophoto is very significant as now we don't have to bother about dealing with problems in different frames such as glares, shadows, partial piece occlusion, of different pieces obstructing the squares.

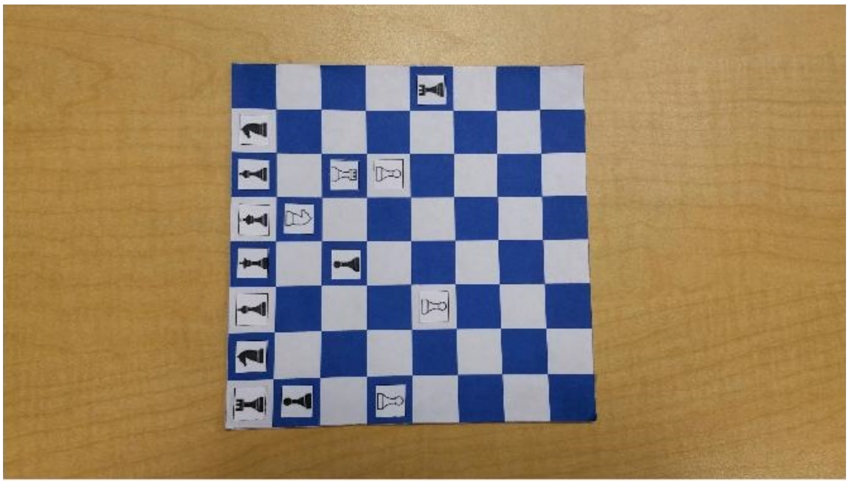

Figure 1: Original photo taken from the video of the chess board

Piece Identification

The most important part of the Chess Assistant program is the piece identification. If the piece is not identified correctly, the right moves will not be displayed and this might confuse

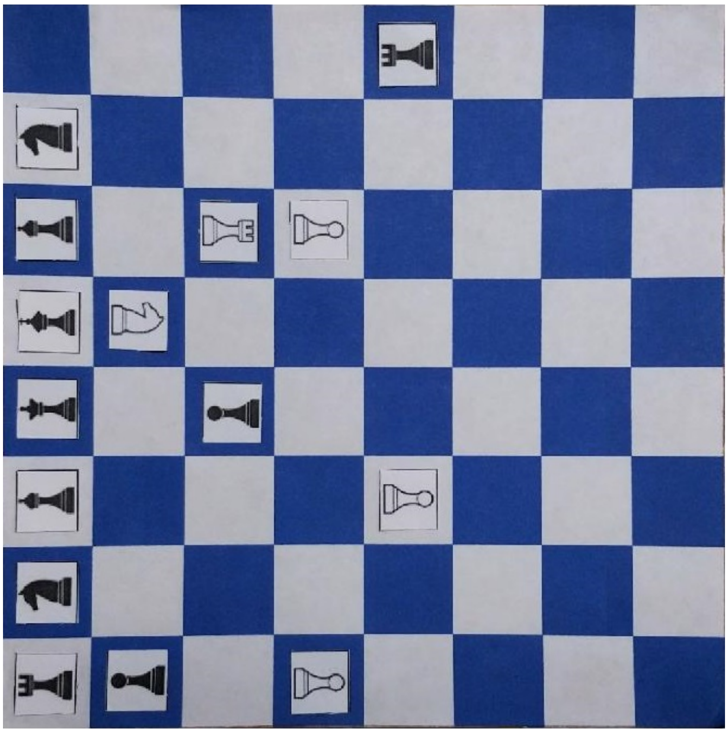

Figure 2: Orthophoto created from the input photo

the player. If the objective of the Chess Assistant is to teach a player chess, then feeding false information is detrimental to its cause. To ensure that this process is completed correctly, the program uses template matching to identify the piece.

A predefined set of 12 templates, one for each piece using both the white and blue backgrounds, is used to identify the pieces. The templates as well as the piece's square, which is selected by the user through a mouse click, are converted to grey-scale in order to be able to us the MatLab function "normxcorr2" to compute the normalized cross-correlation coefficients between each template and the image of the square which contains the piece. An example of a selected square can be seen in figure 3 with a template that will match to it shown in figure 4. The maximum cross-correlation between the template from both the piece with a white background and a blue background is then stored in a vector, the index of which represents the piece that the square is believed to contain. By finding the maximum value of that vector of cross-correlation coefficients and taking the index which corresponds to that value, the piece identification is obtained.

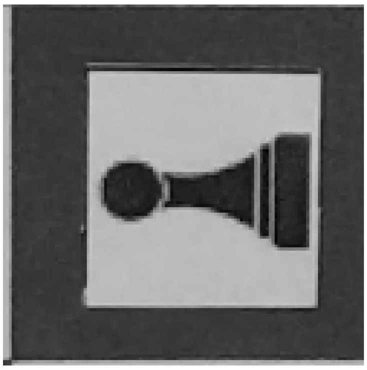

Figure 3: Chess piece selected by the user input

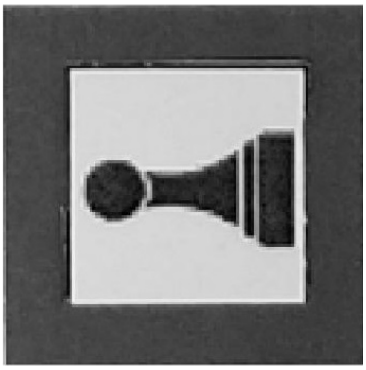

Figure 4: Image used to represent the pawn with a white background for template matching

This method proved to be extremely effective, identifying the piece correctly through all trials. This method can also be adapted to create the templates at the beginning of the game by asking the player to, in the first orthophoto, select each piece. By cropping out the square that holds the piece, a template can easily be made for use during the game. This allows for the dismissal of the assumption that the pieces in the game must be the same that were used to create the program.

Determining Possible Moves

Once the piece in the square which the user selected has been identified, it is possible to determine the moves that the piece can take. This is accomplished by stepping through the adjacent squares in the direction of legal moves, identifying what the square contains, and branching out accordingly. For pieces that are allowed to move in multiple directions, this process first begins with picking a direction to start, testing the possible moves in that direction until no legal moves are left, then moving on to the next direction in a counter clockwise manner. If the square contains nothing, a green square is drawn in it to let the player know this is a legal move, if it contains an enemy a red square is drawn, and if the square contains a friendly piece no square is drawn. In order for this method to work, an accurate way of identifying the contents of a square is needed.

This problem can be broken up into two parts, first identifying if the square is empty and if it is not, determining if the piece inside is friendly or an enemy. To determine if the square is empty of not, the standard deviation of the red, green, and blue values was analyzed. Standard deviation is a measure of the expected value of how far data deviates from the mean of the data, shown in equation 1. In the case of an RGB image, standard deviation can be used to determine how consistent the color is over the entire image. A high standard deviation means that the square is not a constant block of color, and therefore is not empty. For example, figure 5 has an average standard deviation of RGB values of 44.86, while the empty square shown in figure 6 has an much lower average standard deviation of RGB values of 20.12. If the square is determined to be occupied, a test must then be performed to figure out whether the occupation is by a friendly or enemy piece.

$$sd = \sigma = \sqrt{\frac{\sum_{i=1}^{N}(x_i - \bar{x})^2}{N}} \tag{1}$$

To determine the team of the occupying piece, the assumption that a piece is either black or white as well as the fact that the piece is close to the center of the image is taken advantage of. Because of these assumptions, it was believed that the center point of the image can be used to determine team. To do this, the image would first be converted to a binary image using the MatLab function "im2bw", and then the center point of the image

Figure 5: Square with a high standard deviation showing that it has a chess piece occupying it

Figure 6: Square with a low standard deviation showing that it is empty

would be examined. If the center point has a value of 0, then the piece is black and therefore a friend. If the center point has a value of 1, then the piece is white and is an enemy. This method was found to be inconsistent due to variations in the placement and detection of the checker piece.

A method that was found to be more reliable was to take advantage of the bwlabel and regionprops functions. These allowed for better identification of the important regions in the square. By looking at the areas provided by regionprops, it was possible to locate the largest region in the square. This region will either be the white box that encompass the piece, or the chess board square. By locating these, the scope is narrowed and the centroid of these blobs is more closely related to the center of the actual chess piece. Now by checking if the value at the centroid is either 0 or 1, you can more accurately identify the team which the chess piece belongs to. The results of the first and second method can be shown in figure 7 and figure 8 respectively. It can be seen that the second method is the only one of the two that produces the correct result with this image.

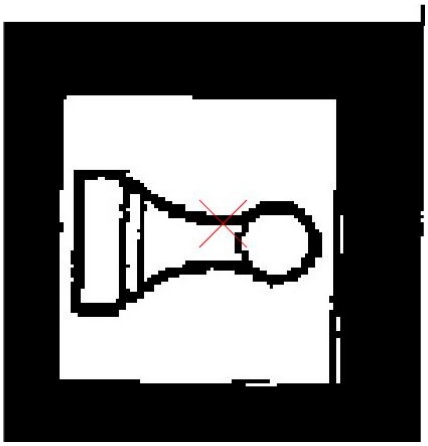

Figure 7: Method where the center point of the image is sampled to determine which team the piece belongs to

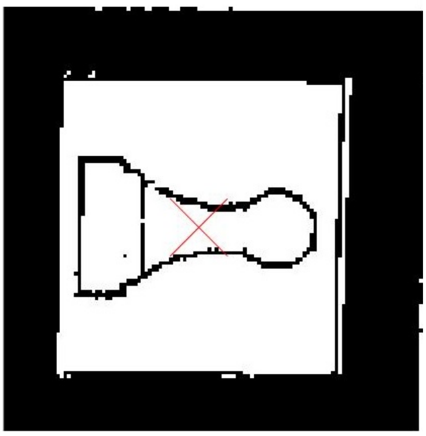

Figure 8: Method where the centroid of the largest binary figure is sampled to determine which team the piece belongs to

This method of identifying the occupation of the square has been turned into a function as it is needed to be called when both determining the moves that the piece can make, and keeping track of the score of the game. Score is kept by parsing through each square, while keeping a tally of white and black pieces to to show the player how many of each team's pieces are left on the board.

Experiments

Piece Identification

As previously stated, the most fundamental part of the Chess Assistant program is accurate identification of the pieces. Because of this, it is important to test the reliability of the recognition. To do this, we first created a new program which uses the same matching function as the Chess Assistant program, but prints out the found identification next to the piece for quick verification. This allows us to count the number of successful piece identifications and create a bar graph displaying the probability of each piece being identified correctly, shown in figure 9. This graph, which represents the results from 12 different arrangements of the pieces, shows that the probability of each piece being identified correctly is 1 for all but the pawn and bishop. These pieces frequently get identified as each other, as well as queens. The reason for this may be how non-distinct the features on these pieces are, as they take on the basic shape of most pieces.

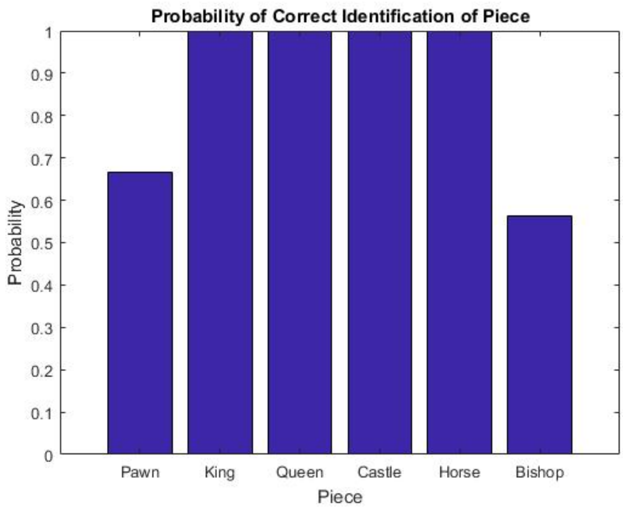

Figure 9: Bar graph representing the probability of each piece being successfully identified using our current method

Square Occupation Testing

Another important aspect of the Chess Assistant that is worth verifying works is the method that is used to find the occupation of the squares that the pieces can move into. To test this, another experimental function is made which parses through the entire board, checking what is in the squares and highlighting them accordingly. By visually inspecting each square, it is possible to verify that the method is working correctly. One cycle of this process is shown in figure 10. The highlighted red represents a detection as an enemy piece, green as an empty square, and no highlight as a friendly piece. This iteration resulted in 3 errors, with a total of 7 errors throughout the entire 3 test images which we used. Using this information, we can calculate the probability that the program will produce a false reading of the occupation of the square using equation 2.

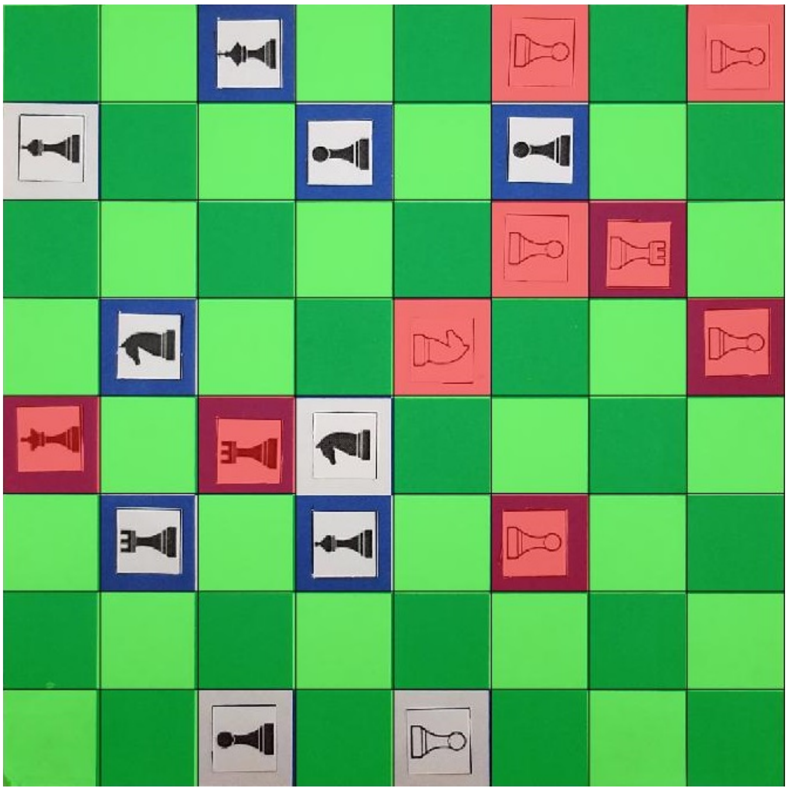

Figure 10: Image which shows the output of our experimental program

$$P(incorrect) = \frac{False}{Tested} = \frac{7}{192} = 0.036 \tag{2}$$

Testing With a Different Board Color

To test for robustness in board variations, it was necessary to experiment with a different board color. A black board was chosen because black and white is a very common chess board color. It is also beneficial because many people do not have access to color printing, and therefore might not be able to print a blue board, helping the goal of DIY (Do it yourself). In this experiment, everything but the threshold values for both the image conversions from RGB to black and white and the mean standard deviation for identification of empty squares has been kept constant.

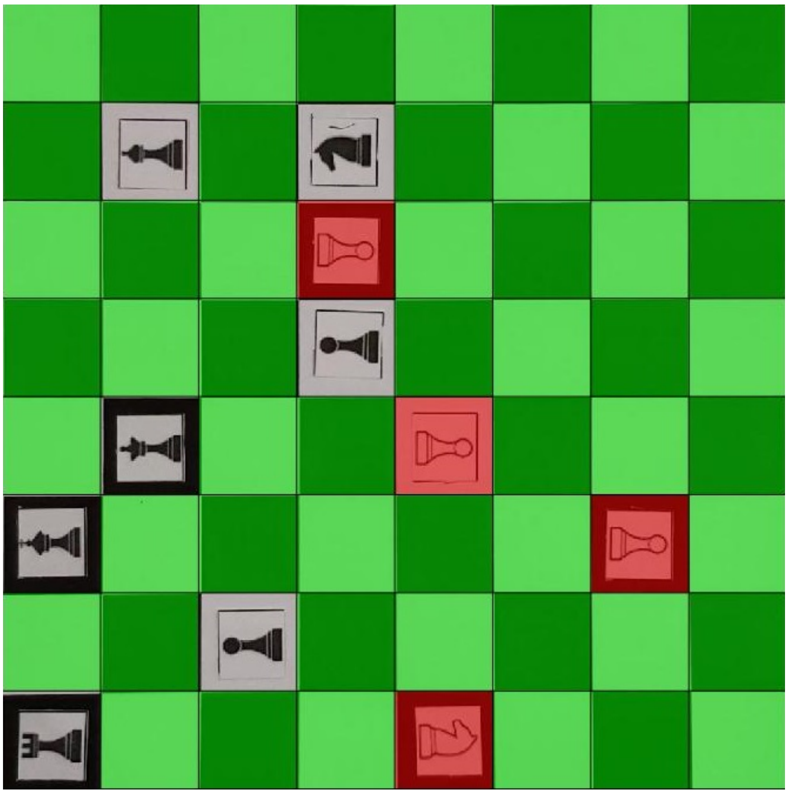

Figure 11: Testing the accuracy of the occupation testing of the black and white board

Figure 11 shows the results of running the black and white board through the experiment which tested what occupied the squares. Using the black and white board, every square was identified to contain the correct thing. This result is not surprising, as black and white are easier to differentiate than blue and white. A more surprising result is that of the identification of the selected pieces. It was found that, even though the background color for each template is different than the background color of the board, all but one piece was able to be identified correctly and display the legal moves as show for the queen in figure 12.

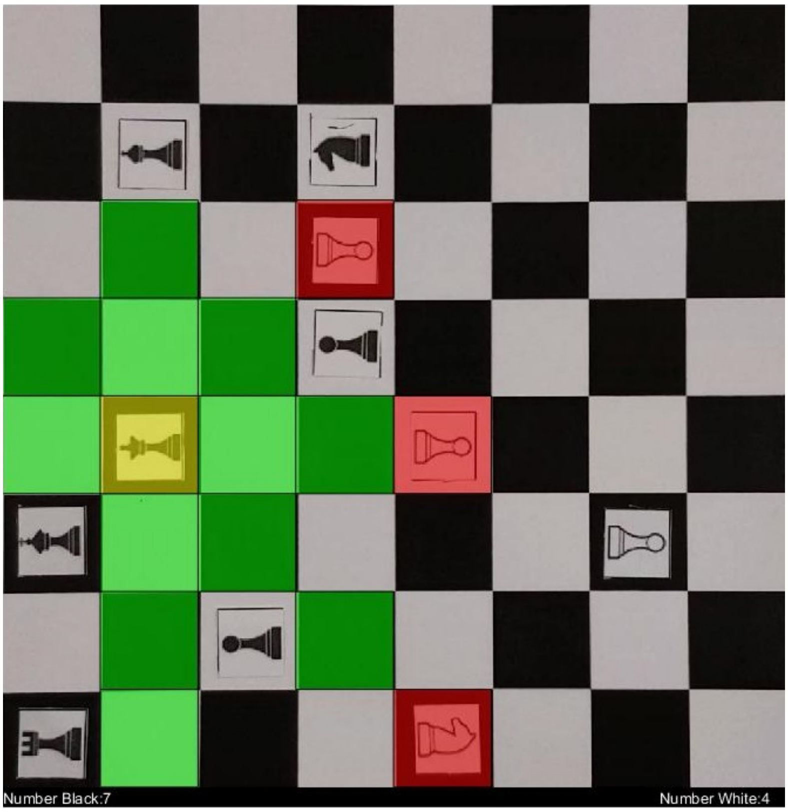

Figure 12: Legal moves for the queen piece as tested on the black board

DISCUSSION

Achievements

The Chess Assistant is successfully able to accurately assist the user in learning the basics of chess. It successfully maps the possible moves which each piece can take, explicitly showing kill move along with the regular moves which do not result in a piece captured, this can be seen in both figure 13, which shows the moves of the queen, and figure 14, which shows the moves of the knight. The Chess Assistant displays the selected piece in yellow, along with the possible moves in green and red, green represents regular moves while red represents kill moves that the piece can take.

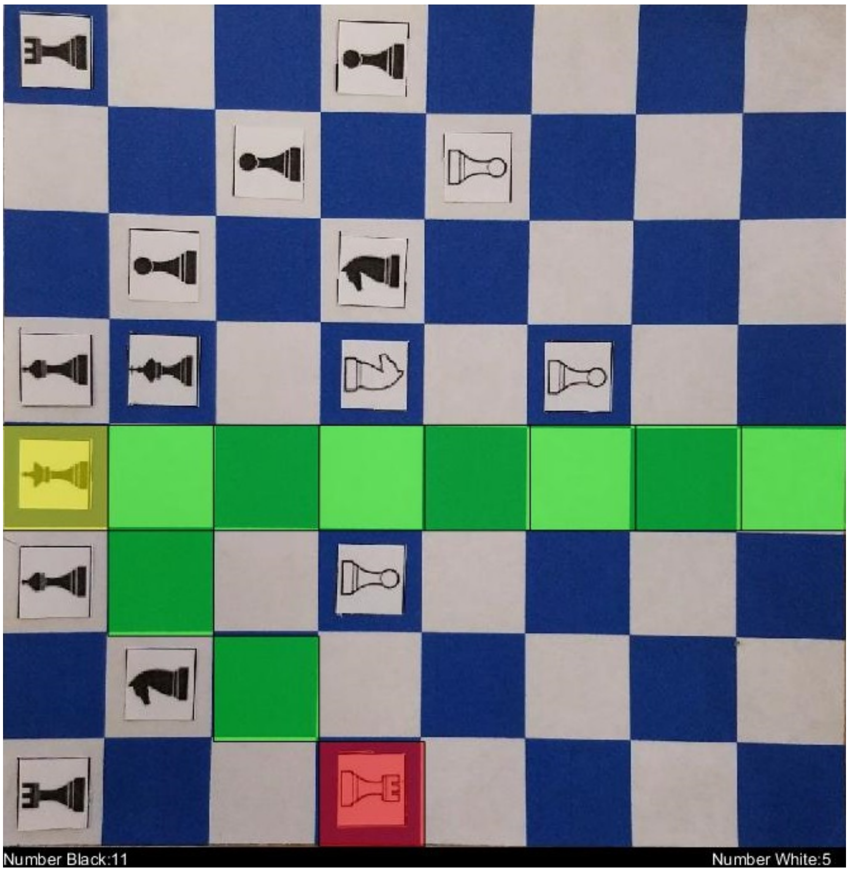

Figure 13: Moves that the queen piece can make as produced by the Chess Assistant

Limitations

One of the main limitations of our algorithms is that the pieces have to maintain the same predefined orientation and location for simplicity of template matching. This could be a problem because it is sometimes hard to orient the pieces exactly right, and small bumps of the board or any other movement of the pieces while trying to identify them could lead

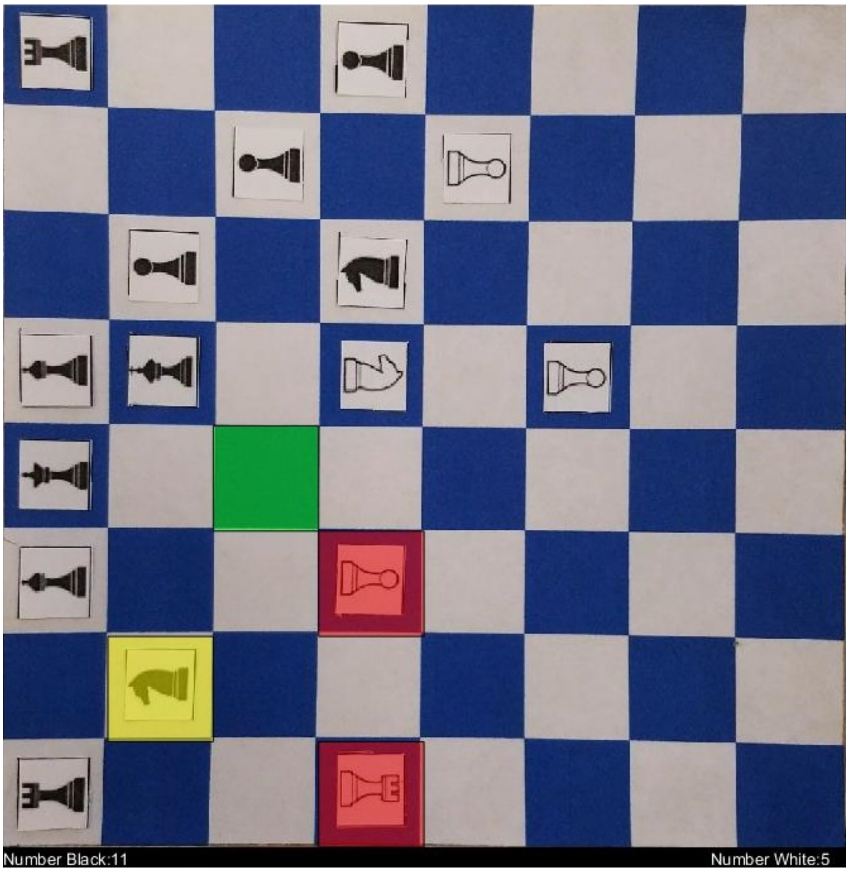

Figure 14: Moves that the knight piece can make as produced by the Chess Assistant

to false identification. Another limitation is the need for consistent lighting. If the lighting changes, the predefined thresholds for the im2bw function as well as the threshold for the mean of the standard deviation for identifying empty squares needs to be changed. Finally, our program is not yet exactly a real time model. The user has to rerun the entire program to play for each different move.

The problem of needing to place the pieces in the exact location could possibly be solved with more complicated computer vision matching techniques such as SIFT. This would allow for the pieces to be both oriented differently, as well as located in different parts of the square as compared to the template. The solution to the lighting problem would be to use a consistent lighting source, such as a desk lamp or any other consistent light source. Lastly by using OpenCV instead of MatLab, it might be possible to run the Chess Assistant in real time rather than having to step through blocks of frames as it currently does.

Possible Future Work

Given enough time, this project could be expanded upon to create a more effective learning environment for the player. One thing that could be added is the ability for 2 players to receive information on possible moves. This could be achieved by incorporating turns into the program, as well as another set of templates for matching the pieces of player two. Another possible addition to the program might be implementing a priority move option, allowing the user to make smarter and more strategic moves which will benefit the player later in the game. This can be done through weighting the pieces in relation to their importance in the game, as well as implementing different popular game play strategies, such as Castling the King[4]. More complicated rules could also be included, such as being forced to block kill moves on the king if at all possible[2]. It is also possible to help the player select which piece to resurrect if their pawn reaches the opponents side. This could be accomplished through tracking which pieces have been killed, and using a similar weighting system as that which selects priority kill moves to decide which piece is more important to return to the board.

The Chess Assistant could also be used for the base of an augmented reality chess game, which combines the extravagance of augmented reality with its current teaching ability. This could be accomplished by replacing the current chess pieces with ArUco Markers, allowing for easy computation of things such as pose. This would make it relatively simple to overlay three-dimensional graphics on the board by using popular graphics software which can be combined with OpenCV, such as OpenGL. This would allow for the user to feel more as though they are actually playing chess.

Bibliography

[1] Harriet Dennys. Agon releases new chess player statistics from yougov, 2012.

[2] The United States Chess Federation. Learn to play chess, 2007.

[3] Can Koray and Emre Sumer. A computer visino system for chess game tracking, 2016.

[4] A.R. Rostami. Advanced chess game and method therefor, December 2 1997. US Patent 5,692,754.

Appendices

Appendix A

Video

Link to demonstration video: `https://www.youtube.com/watch?v=fUx4NC4V_gU`

Appendix B

MatLab Code

```
        %% Chess Assistant
%  By Abhilesh Borode and Mason Wilie

clear all;
close all;

%% Reading In Templates for Matching
global TbishopW
global TcastleW
global ThorseW
global TkingW
global TpawnW
global TqueenW

global TbishopB
global TcastleB
global ThorseB
```

```
global TkingB
global TpawnB
global TqueenB

TbishopW = imread('BishopW.jpg');
TcastleW = imread('CastleW.jpg');
ThorseW = imread('HorseW.jpg');
TkingW = imread('KingW.jpg');
TpawnW = imread('PawnW.jpg');
TqueenW = imread('QueenW.jpg');

TbishopB = imread('BishopB.jpg');
TcastleB = imread('CastleB.jpg');
ThorseB = imread('HorseB.jpg');
TkingB = imread('KingB.jpg');
TpawnB = imread('PawnB.jpg');
TqueenB = imread('QueenB.jpg');

%% Constants
global SQUARE_LEN
M = 20;
RES = 1000;
SQUARE_LEN = round(RES / 8);

% Resizing Templates to Fit Square --------------------
TbishopW = imresize(TbishopW, [(RES / 8), (RES / 8)]);
```

```
TcastleW = imresize(TcastleW, [(RES / 8), (RES / 8)]);
ThorseW = imresize(ThorseW, [(RES / 8), (RES / 8)]);
TkingW = imresize(TkingW, [(RES / 8), (RES / 8)]);
TpawnW = imresize(TpawnW, [(RES / 8), (RES / 8)]);
TqueenW = imresize(TqueenW, [(RES / 8), (RES / 8)]);

TbishopB = imresize(TbishopB, [(RES / 8), (RES / 8)]);
TcastleB = imresize(TcastleB, [(RES / 8), (RES / 8)]);
ThorseB = imresize(ThorseB, [(RES / 8), (RES / 8)]);
TkingB = imresize(TkingB, [(RES / 8), (RES / 8)]);
TpawnB = imresize(TpawnB, [(RES / 8), (RES / 8)]);
TqueenB = imresize(TqueenB, [(RES / 8), (RES / 8)]);

% Converting Templats to Grayscale for normxcorr2
TbishopW = rgb2gray(TbishopW);
TcastleW = rgb2gray(TcastleW);
TkingW = rgb2gray(TkingW);
ThorseW = rgb2gray(ThorseW);
TpawnW = rgb2gray(TpawnW);
TqueenW = rgb2gray(TqueenW);

TbishopB = rgb2gray(TbishopB);
TcastleB = rgb2gray(TcastleB);
TkingB = rgb2gray(TkingB);
ThorseB = rgb2gray(ThorseB);
TpawnB = rgb2gray(TpawnB);
```

```
TqueenB = rgb2gray(TqueenB);

video = VideoReader('board_moved.mp4'); % Reading in video
nFrames = video.NumberOfFrames; % Getting number of frames in video

playIndex = 2;

for i =60:10:nFrames

    I = read(video, i); % Read current frame
    iFrame = I;
    [xREZ, yREZ, z] = size(I);

    [corners, nMatches, avgErr] = findCheckerBoard(I); % Gets the
        corners of the checkerboard

    transform = fitgeotrans(corners, [0, 0; RES, 0; RES, RES; 0, RES],
        'projective'); % Creates a transformation from camera to
        orthonormal view


    ref = imref2d([RES, RES],... % Creates reference frame for
        orthonormal view
            [0 RES],...
            [0 RES]);

```

```
    I = imwarp(I, transform, 'OutputView', ref); % Creates orthonormal
        view
    corners = imwarp(corners, transform); % Transforms the corner
        points of the checkerboard

    [imagePoints, boardSize] = detectCheckerboardPoints(I);

    newFrameOut = getframe;

    if (boardSize(1) ~= 8 || boardSize(2) ~= 8) continue; end

    imshow(I);
    displayI = zeros(RES + 25, RES, 3);
    displayI(1:RES, 1:RES, 1) = I(:,:,1);
    displayI(1:RES, 1:RES, 2) = I(:,:,2);
    displayI(1:RES, 1:RES, 3) = I(:,:,3);

    displayI = uint8(displayI);
     if (playIndex < 1)
         imshow(displayI); % Displays the orthonormal view of the
             checkerboard
     end

     imshow(displayI);

     [x,y] = ginput(1); % Gets user input of selected square
```

```
    square = [floor(x / (RES / 8)) + 1, floor(y / (RES / 8)) + 1]; % Translates the image points of the selected square to square points on an 8x8 grid
    rectangle('Position', [SQUARE_LEN * (square(1) - 1), (SQUARE_LEN * (square(2) - 1)), SQUARE_LEN, SQUARE_LEN], 'FaceColor', [1, 1, 0, 0.5]);


    squareIm = I(((square(2) - 1 )* SQUARE_LEN + 1):((square(2) - 1 )* SQUARE_LEN + SQUARE_LEN),((square(1) - 1 )* SQUARE_LEN + 1):((square(1) - 1 )* SQUARE_LEN + SQUARE_LEN), :); % Gets the image of the square that we want to check what the piece is
    squareIm = imresize(squareIm, [RES / 8, RES / 8]); % Resizes the square to be the same size as the templates (Already should be, but just in case)
    piece = identifyPiece(squareIm); % Identifies what friendly piece occupies the square which the user selected
    [numBlack, numWhite] = findScore(I);
    string = strcat('Number Black: ' ,num2str(numBlack));
   % if (numBlack == 0 || numWhite == 0) break; end % ends the game if one of the colors is completely gone, optional


    text(0, RES + 12, string,'Color', 'white');
    string = strcat('Number White: ' , num2str(numWhite));
    text(840, RES + 12, string, 'Color', 'white');
```

```
%% Moves
    figure(1),hold on;

% Moves of the Pawn
  ------------------------------------------------------------------------------------------------

    piece = identifyPiece(squareIm);
    piece
if (strcmp(piece, 'pawn')) %% Pawn Moves
        rectY = ((square(2) - 1 )* SQUARE_LEN + 1);
        rectX = ((square(1) - 1 )* SQUARE_LEN + 1);

        tempImage = I(rectY:(rectY + SQUARE_LEN) - 3, (rectX +
            SQUARE_LEN):((rectX + SQUARE_LEN) + SQUARE_LEN - 3),:);
        if (strcmp(findRelation(tempImage), 'Empty')); % Front square,
            only draws green square if empty
            rectangle('Position', [rectX + SQUARE_LEN, rectY, SQUARE_LEN
                , SQUARE_LEN], 'FaceColor', [0, 1, 0, 0.5]);
        end
        if (square(2) > 1) % Top diagonal, only draws red square if has
             enemy
             tempImage = I((rectY - SQUARE_LEN):(rectY - 3), (rectX +
                 SQUARE_LEN):((rectX + SQUARE_LEN) + SQUARE_LEN - 3),:);
             if (strcmp(findRelation(tempImage), 'Enemy'))
```

```
                rectangle('Position', [rectX + SQUARE_LEN, rectY -
                    SQUARE_LEN, SQUARE_LEN, SQUARE_LEN], 'FaceColor',
                    [1, 0, 0, 0.5]);
            end
        end
        if (square(2) < 8) % Bottom Diagonal, only draws red square if
            has enemy
            tempImage = I((rectY + SQUARE_LEN):(rectY + 2 * SQUARE_LEN
                - 3), (rectX + SQUARE_LEN):((rectX + SQUARE_LEN) +
                SQUARE_LEN - 3),:);
            if (strcmp(findRelation(tempImage), 'Enemy'))
                rectangle('Position', [rectX + SQUARE_LEN, rectY +
                    SQUARE_LEN, SQUARE_LEN, SQUARE_LEN], 'FaceColor',
                    [1, 0, 0, 0.5]);
            end
        end
    end


% Moves of the Bishop
    ---------------------------------------------------------------------------

    if (strcmp(piece, 'bishop'))
        for i = 1:4 % Iterate through each diagonal section
            currentSquare = square; % Sets the current square to the
                one that the user selected
```

```
        while (currentSquare(1) <= 8 && currentSquare(1) >= 1 && currentSquare(2) <= 8 && currentSquare(2) >= 1) % Loop, breaks when current square is off board
            if (i == 1) currentSquare = [currentSquare(1) + 1, currentSquare(2) - 1]; end % Up and Right Diagonal
            if (i == 2) currentSquare = [currentSquare(1) - 1, currentSquare(2) - 1]; end % Up and Left Diagonal
            if (i == 3) currentSquare = [currentSquare(1) - 1, currentSquare(2) + 1]; end % Down and Left Diagonal
            if (i == 4) currentSquare = [currentSquare(1) + 1, currentSquare(2) + 1]; end % Down and Right Diagonal

            if (currentSquare(1) == 9 || currentSquare(1) == 0 || currentSquare(2) == 9 || currentSquare(2) == 0) break; end % Exits loop when out of bounds

            tempImage = I(((currentSquare(2) - 1) * SQUARE_LEN + 1):(SQUARE_LEN * currentSquare(2)),((currentSquare(1) - 1) * SQUARE_LEN + 1):(SQUARE_LEN * currentSquare(1)), :); % Gets the image of the square traveling into
            relation = findRelation(tempImage); % Finds what is in the square
            if (strcmp(relation, 'Empty')) % Draws a green rectangle if the square is empty and continues through the loop
```

```
                    rectangle('Position', [SQUARE_LEN * (currentSquare(1) - 1), (SQUARE_LEN * (currentSquare(2) - 1)), SQUARE_LEN, SQUARE_LEN], 'FaceColor', [0, 1, 0, 0.5]);
                    continue;
                elseif (strcmp(relation, 'Enemy')) % Draws a red square if the box has an enemy in it, stops checking the next squares (can't travel past)
                    rectangle('Position', [SQUARE_LEN * (currentSquare(1) - 1), (SQUARE_LEN * (currentSquare(2) - 1)), SQUARE_LEN, SQUARE_LEN], 'FaceColor', [1, 0, 0, 0.5]);
                    break;
                elseif (strcmp(relation, 'Friend')) % Does not draw anything if the box has a friend in it, stops checking the next squares (can't travel past)
                    break;
                end
            end
        end
    end

% Moves of the Castle ---------------------------------------------------------------------------------------
    if (strcmp(piece, 'castle'))
```

```
    for i = 1:4 % Iterate through each diagonal section
        currentSquare = square; % Resets the current square to user
            selected square

        while (currentSquare(1) <= 8 && currentSquare(1) >= 1 &&
            currentSquare(2) <= 8 && currentSquare(2) >= 1) % Loops
            while the square is in bounds
            if (i == 1) currentSquare = [currentSquare(1) + 1,
                currentSquare(2)]; % Moving right
            elseif(i == 2) currentSquare = [currentSquare(1),
                currentSquare(2) - 1]; % Moving up
            elseif (i == 3) currentSquare = [currentSquare(1) - 1,
                currentSquare(2)]; % Moving left
            else currentSquare = [currentSquare(1), currentSquare(2)
                 + 1]; % Moving down
            end

            if (currentSquare(1) == 9 || currentSquare(1) == 0 ||
                currentSquare(2) == 9  || currentSquare(2) == 0)
                break; end % Checks to make sure that the square is
                in bounds


            tempImage = I(((currentSquare(2) - 1) * SQUARE_LEN + 1):
                (SQUARE_LEN * currentSquare(2)),((currentSquare(1) -
                1) * SQUARE_LEN + 1):(SQUARE_LEN * currentSquare(1))
```

```
                , :); % Gets the image of the square that we are observing
            relation = findRelation(tempImage); % Finds out what is in that square

            if (strcmp(relation,'Empty')) % Draws a green square in the space if there is nothing in it
                rectangle('Position', [SQUARE_LEN * (currentSquare(1) - 1), (SQUARE_LEN * (currentSquare(2) - 1)), SQUARE_LEN, SQUARE_LEN], 'FaceColor', [0, 1, 0, 0.5]);
                continue;
            elseif (strcmp(relation,'Enemy')) % Draws a red square in the space if there is an enemy, and does not check the squares after it (can't move past)
                rectangle('Position', [SQUARE_LEN * (currentSquare(1) - 1), (SQUARE_LEN * (currentSquare(2) - 1)), SQUARE_LEN, SQUARE_LEN], 'FaceColor', [1, 0, 0, 0.5]);
                break;
            elseif (strcmp(relation,'Friend')) % Does not draw anything if the square contains a friendly piece, does not continue checking the pieces after (can't move past)
                break;
            end
```

```

            end

        end
    end




% Moves of the Queen --------------------------------------------------------------------------------

        if (strcmp(piece, 'queen'))
        for i = 1:8 % Iterate through each diagonal section
            currentSquare = square; % Resets the current square to the user selected square

            while (currentSquare(1) <= 8 && currentSquare(1) >= 1 && currentSquare(2) <= 8 && currentSquare(2) >= 1) % Loops until the current square is out of bounds
                if (i == 1) currentSquare = [currentSquare(1) + 1, currentSquare(2)]; end % Right
                if (i == 2) currentSquare = [currentSquare(1), currentSquare(2) - 1]; end % Up
                if (i == 3) currentSquare = [currentSquare(1) - 1, currentSquare(2)]; end % Left
```

```
        if (i == 4) currentSquare = [currentSquare(1), currentSquare(2) + 1]; end % Down
        if (i == 5) currentSquare = [currentSquare(1) + 1, currentSquare(2) - 1]; end % Up and Right Diagonal
        if (i == 6) currentSquare = [currentSquare(1) - 1, currentSquare(2) - 1]; end % Up and Left Diagonal
        if (i == 7) currentSquare = [currentSquare(1) - 1, currentSquare(2) + 1]; end % Down and Left Diagonal
        if (i == 8) currentSquare = [currentSquare(1) + 1, currentSquare(2) + 1]; end % Down and Right Diagonal

        if (currentSquare(1) == 9 || currentSquare(1) == 0 || currentSquare(2) == 9  || currentSquare(2) == 0) break; end % Breaks out of the loop if out of bounds

        tempImage = I(((currentSquare(2) - 1) * SQUARE_LEN + 1):(SQUARE_LEN * currentSquare(2)),((currentSquare(1) - 1) * SQUARE_LEN + 1):(SQUARE_LEN * currentSquare(1)), :); % Gets the image of the square that we are checking
        relation = findRelation(tempImage); % Finds out what is in that square
        if (strcmp(relation, 'Empty')) % Draws a green square there if there is nothing in it, continues to check the squares following it
```

```
                rectangle('Position', [SQUARE_LEN * (currentSquare(1) - 1), (SQUARE_LEN * (currentSquare(2) - 1)), SQUARE_LEN, SQUARE_LEN], 'FaceColor', [0, 1, 0, 0.5]);
                continue;
            elseif (strcmp(relation, 'Enemy')) % Draws a red square if there is an enemy, does not check the squares following it (can't move past)
                rectangle('Position', [SQUARE_LEN * (currentSquare(1) - 1), (SQUARE_LEN * (currentSquare(2) - 1)), SQUARE_LEN, SQUARE_LEN], 'FaceColor', [1, 0, 0, 0.5]);
                break;
            elseif (strcmp(relation, 'Friend')) % Does not draw anythin if there is a friendly piece in that square, does not check the squares following it (can't move past)
                break;
            end
        end
    end
end


% Moves of the Horse
```

```
    if (strcmp(piece, 'horse'))
        for i = 1:4 % Iterate through each diagonal section

            % Checks the 4 Ls
            currentSquare = square; % Resets the square to the user selected square
            if (i == 1) currentSquare = [currentSquare(1) + 2, currentSquare(2) - 1]; end % L long side going right, short side going up
            if (i == 2) currentSquare = [currentSquare(1) - 1, currentSquare(2) - 2]; end % L long side going up, short side goine left
            if (i == 3) currentSquare = [currentSquare(1) - 2, currentSquare(2) + 1]; end % L long side going left, short side going down
            if (i == 4) currentSquare = [currentSquare(1) + 1, currentSquare(2) + 2]; end % L long side going down, short side going right
            if (currentSquare(1) < 9 && currentSquare(1) > 0 && currentSquare(2) < 9  && currentSquare(2) > 0) % Makes sure that the square is in bounds
                tempImage = I(((currentSquare(2) - 1) * SQUARE_LEN + 1):(SQUARE_LEN * currentSquare(2)),((currentSquare(1) - 1) * SQUARE_LEN + 1):(SQUARE_LEN * currentSquare(1)), :); % Gets the image of the square which we want to
```

```
check
                relation = findRelation(tempImage); % Finds out what is in that square
                if (strcmp(relation, 'Empty')) % If the square is empty, draw a green square over it
                    rectangle('Position', [SQUARE_LEN * (currentSquare(1) - 1), (SQUARE_LEN * (currentSquare(2) - 1)), SQUARE_LEN, SQUARE_LEN], 'FaceColor', [0, 1, 0, 0.5]);
                elseif (strcmp(relation, 'Enemy')) % If the square is an enemy, draw a red square on it
                    rectangle('Position', [SQUARE_LEN * (currentSquare(1) - 1), (SQUARE_LEN * (currentSquare(2) - 1)), SQUARE_LEN, SQUARE_LEN], 'FaceColor', [1, 0, 0, 0.5]);
                end % Does not do anything if the square contains a friendly piece
            end
            currentSquare = square; % Resets the square to the user selected square
            if (i == 1) currentSquare = [currentSquare(1) + 2, currentSquare(2) + 1]; end % L long side going right, short side going down
            if (i == 2) currentSquare = [currentSquare(1) + 1, currentSquare(2) - 2]; end % L long side going up, short side goine right
```

```
        if (i == 3) currentSquare = [currentSquare(1) - 2, currentSquare(2) - 1]; end % L long side going left, short side going up
        if (i == 4) currentSquare = [currentSquare(1) - 1, currentSquare(2) + 2]; end % L long side going down, short side going left
        if (currentSquare(1) < 9 && currentSquare(1) > 0 && currentSquare(2) < 9  && currentSquare(2) > 0) % Makes sure that the square we are checking is still in bounds
            tempImage = I(((currentSquare(2) - 1) * SQUARE_LEN + 1):(SQUARE_LEN * currentSquare(2)),((currentSquare(1) - 1) * SQUARE_LEN + 1):(SQUARE_LEN * currentSquare(1)), :); % Gets the image of the square which we are concerned with
            relation = findRelation(tempImage); % Finds out what is in that square
            if (strcmp(relation, 'Empty')) % If the square is empty, draw a green square over it
                rectangle('Position', [SQUARE_LEN * (currentSquare(1) - 1), (SQUARE_LEN * (currentSquare(2) - 1)), SQUARE_LEN, SQUARE_LEN], 'FaceColor', [0, 1, 0, 0.5]);
            elseif (strcmp(relation, 'Enemy')) % If the square is an enemy, draw a red square on it
                rectangle('Position', [SQUARE_LEN * (currentSquare(1) - 1), (SQUARE_LEN * (currentSquare(2) - 1)),
```

```
                    SQUARE_LEN, SQUARE_LEN], 'FaceColor', [1, 0, 0,
                    0.5]);
                end % Does not do anything if the square contains a
                    friendly piece
            end
        end
    end



% Moves of the King
    --------------------------------------------------------------------------------

    if (strcmp(piece, 'king'))
        for i = 1:8 % Iterate through each diagonal section
            currentSquare = square; % Resets the current square to the
                user selected square


            if (i == 1) currentSquare = [currentSquare(1) + 1,
                currentSquare(2)]; end % Right
            if (i == 2) currentSquare = [currentSquare(1), currentSquare
                (2) - 1]; end % Up
            if (i == 3) currentSquare = [currentSquare(1) - 1,
                currentSquare(2)]; end % Left
```

```
        if (i == 4) currentSquare = [currentSquare(1), currentSquare(2) + 1]; end % Down
        if (i == 5) currentSquare = [currentSquare(1) + 1, currentSquare(2) - 1]; end % Up and Right Diagonal
        if (i == 6) currentSquare = [currentSquare(1) - 1, currentSquare(2) - 1]; end % Up and Left Diagonal
        if (i == 7) currentSquare = [currentSquare(1) - 1, currentSquare(2) + 1]; end % Down and Left Diagonal
        if (i == 8) currentSquare = [currentSquare(1) + 1, currentSquare(2) + 1]; end % Down and Right Diagonal

        if (currentSquare(1) == 9 || currentSquare(1) == 0 || currentSquare(2) == 9  || currentSquare(2) == 0) continue; end % Skips iteration of loop if out of bounds

        tempImage = I(((currentSquare(2) - 1) * SQUARE_LEN + 1):(SQUARE_LEN * currentSquare(2)),((currentSquare(1) - 1) * SQUARE_LEN + 1):(SQUARE_LEN * currentSquare(1)), :); % Gets the image of the square that we are checking
        relation = findRelation(tempImage); % Finds out what is in that square
        if (strcmp(relation, 'Empty')) % Draws a green square there if there is nothing in it, continues to check the squares following it
            rectangle('Position', [SQUARE_LEN * (currentSquare(1) - 1), (SQUARE_LEN * (currentSquare(2) - 1)), SQUARE_LEN
```

```
                , SQUARE_LEN], 'FaceColor', [0, 1, 0, 0.5]);
            elseif (strcmp(relation, 'Enemy')) % Draws a red square if
                there is an enemy, does not check the squares following
                it (can't move past)
                rectangle('Position', [SQUARE_LEN * (currentSquare(1) -
                    1), (SQUARE_LEN * (currentSquare(2) - 1)), SQUARE_LEN
                    , SQUARE_LEN], 'FaceColor', [1, 0, 0, 0.5]);
            elseif (strcmp(relation, 'Friend')) % Does not draw anythin
                if there is a friendly piece in that square, does not
                check the squares following it (can't move past)
            end
        end

    end
    drawnow;
    hold off;
    return; %Take out if you want to run more than one frame

end


%% Find Relation Function
function relation = findRelation(RGB)
% findRelation - Finds out what is in the square related to the piece
%   Function which finds the relationship between what fills the
    selected
```

```
% square and the black piece, either "Empty", "Friend", or "Enemy". It
% determines if the square is empty by determining if the standard
% deviation of the RGB values is low. If it is low, that means the color is
% relatively consistant across the square and therefore it is empty. We
% determine if the piece is friendly or an enemy by seeing if the center of
% the black and white image is black or white. Black represents a friendly
% piece and white represents an enemy piece



    imageBW = im2bw(RGB); % Creates a black and white image of the square that we want to find what is in to black and white
    RGB = double(RGB); % Converts the RGB image from default uint8 to double

    [M, N, x] = size(imageBW); % Gets the size of the image

    stdR = std(RGB(:,:,1)); % Gets the standard deviation of the red value of the RGB
    stdG = std(RGB(:,:,2)); % Gets the standard deviation of the blue value of the RGB
```

```
    stdB = std(RGB(:,:,3)); % Gets the standard deviation of the green value of the RGB

    meanSTD = mean([stdR, stdG, stdB]); % Finds the average standard deviation


    if (meanSTD <  25) % If the mean SD is lower than 25, the square is most likely empty
        relation = 'Empty';
        return;
    end


    [L, num] = bwlabel(imageBW);

    blobs = regionprops(L, 'Area', 'Centroid');

    maxIndex = 1;
    maxVal = -9999999;

    for i = 1:num
        if (blobs(i).Area > maxVal)
            maxVal = blobs(i).Area;
            maxIndex = i;
        end
```

```
    end

    if (imageBW(round(blobs(maxIndex).Centroid(1)),round(blobs(maxIndex
        ).Centroid(2))) == 1) % If the center of the square is white, it
         is an enemy
        relation = 'Enemy';
        return;
    end
    if (imageBW(round(blobs(maxIndex).Centroid(1)),round(blobs(maxIndex
        ).Centroid(2))) == 0)% If the center of the square is black, it
        is a friend
        relation = 'Friend';
        return;
    end

    error('Error in findRelation function, could not determine what was
         in the square');
    return;

end
%% identifyPiece Function
% Idendifies the piece in the selected square and returns the decision
    in a
% string

function identification = identifyPiece(Ipiece)
```

```
% Gets the template images
global TbishopW
global TcastleW
global ThorseW
global TkingW
global TpawnW
global TqueenW

global TbishopB
global TcastleB
global ThorseB
global TkingB
global TpawnB
global TqueenB

corScores = zeros(6, 1); % Creates a matrix to store the
    correlation scores
Ipiece = rgb2gray(Ipiece);

corScores(1) = max([max(max(normxcorr2(TbishopW, Ipiece))), max(max
    (normxcorr2(TbishopB, Ipiece)))]); % Correlation scores for
    comparing to the bishop templates
corScores(2) = max([max(max(normxcorr2(TcastleW, Ipiece))),max(max(
    normxcorr2(TcastleB, Ipiece)))]); % Correlation scores for
    comparing to the castle templates
```

```
    corScores(3) = max([max(max(normxcorr2(ThorseW, Ipiece))),max(max(
        normxcorr2(ThorseB, Ipiece)))]); % Correlation scores for
        comparing to the horse templates
    corScores(4) = max([max(max(normxcorr2(TkingW, Ipiece))), max(max(
        normxcorr2(TkingB, Ipiece)))]); % Correlation scores for
        comparing to the king templates
    corScores(5) = max([max(max(normxcorr2(TpawnW, Ipiece))),max(max(
        normxcorr2(TpawnB, Ipiece)))]); % Correlation scores for
        comparing to the pawn templates
    corScores(6) = max([max(max(normxcorr2(TqueenW, Ipiece))), max(max(
        normxcorr2(TqueenB, Ipiece)))]); % Correlation scores for
        comparing to the queen templates

    [maxVal, index] = max(corScores); % Finds which template matched
        the best

    switch index % Identifies which piece the index corrisponds to
        case 1
            identification = 'bishop';
            return;
        case 2
            identification = 'castle';
            return;
        case 3
            identification = 'horse';
            return;
```

```
        case 4
            identification = 'king';
            return;
        case 5
            identification = 'pawn';
            return;
        case 6
            identification = 'queen';
            return;
        otherwise
            identification = 'error';
            return;
    end
end
%% Find Score Function
function [numBlack, numWhite] = findScore(I)
    global SQUARE_LEN; % Gets global variable

    numBlack = 0; % initializes the variables
    numWhite = 0;

    for i = 1:8 % loops through all the squares on the board
        for j = 1:8
            currentSquare = [i,j]; % sets the current square
            squareImage = I(((currentSquare(2) - 1) * SQUARE_LEN + 1): (
                SQUARE_LEN * currentSquare(2)),((currentSquare(1) - 1) *
```

```
            SQUARE_LEN + 1):(SQUARE_LEN * currentSquare(1)), :); %
            Gets the image of the square
        fill = findRelation(squareImage); % Checks what is in the
            square
        switch(fill) % Determines what to add to based on the fill
            case 'Friend'
                numBlack = numBlack + 1;
                continue;
            case 'Enemy'
                numWhite = numWhite + 1;
                continue;
            case 'Empty'
                continue;
        end
    end

  end
end

%% findCheckerBoard Function

function [corners, nMatches, avgErr] = findCheckerBoard(I)
    % Find a 8x8 checkerboard in the image I.
    % Returns:
    % corners: the locations of the four outer corners as a 4x2 array,
        in
```

```
% the form [ [x1,y1]; [x2,y2]; ... ].
% nMatches: number of matching points found (ideally is 81)
% avgErr: the average reprojection error of the matching points
% Return empty if not found.
corners = [];
nMatches = [];
avgErr = [];
if size(I,3)>1
    I = rgb2gray(I);
end
% Do edge detection.
[~,thresh] = edge(I, 'canny'); % First get the automatic
E = edge(I, 'canny', 5*thresh); % Raise the threshold

% Do Hough transform to find lines.
[H,thetaValues,rhoValues] = hough(E); % Extract peaks from the
    Hough array H.


myThresh = 0.1;
NHoodSize = ceil([size(H,1)/50, size(H,2)/50]);
% Force odd size
if mod(NHoodSize(1),2)==0 NHoodSize(1) = NHoodSize(1)+1; end
if mod(NHoodSize(2),2)==0 NHoodSize(2) = NHoodSize(2)+1; end
peaks = houghpeaks(H, ...
 30, ... % Maximum number of peaks to find
```

```
    'Threshold', myThresh, ... % Threshold for peaks
    'NHoodSize', NHoodSize); % Default = floor(size(H)/50);


     % Display Hough array and draw peaks on Hough array.


% Find two sets of orthogonal lines.
[lines1, lines2] = findOrthogonalLines( ...
 rhoValues(peaks(:,1)), ... % rhos for the lines
 thetaValues(peaks(:,2))); % thetas for the lines

% Sort the lines, from top to bottom (for horizontal lines) and
    left to
% right (for vertical lines).
lines1 = sortLines(lines1, size(E));
lines2 = sortLines(lines2, size(E));


[xIntersections, yIntersections] = findIntersections(lines1, lines2
    );

% Define a "reference" image.
IMG_SIZE_REF = 100; % Reference image is IMG_SIZE_REF x
    IMG_SIZE_REF
% Get predicted intersections of lines in the reference image.
```

```
    [xIntersectionsRef, yIntersectionsRef] = createReference(
        IMG_SIZE_REF);


    % Find the best correspondence between the points in the input
        image and
    % the points in the reference image. If found, the output is the
        four
    % outer corner points from the image, represented as a a 4x2 array,
         in the
    % form [ [x1,y1]; [x2,y2]; ... ].
    [corners, nMatches, avgErr] = findCorrespondence( ...
     xIntersections, yIntersections, ... % Input image points
     xIntersectionsRef, yIntersectionsRef, ... % Reference image points
     I);

end


%% findOrthogonalLines Function - Following Code From EENG437/507 Class
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Find two sets of orthogonal lines.
% Inputs:
% rhoValues: rho values for the lines
% thetaValues: theta values (should be from -90..+89 degrees)
% Outputs:
```

```
% lines1, lines2: the two sets of lines, each stored as a 2xN array,
% where each column is [theta;rho]
function [lines1, lines2] = findOrthogonalLines( ...
        rhoValues, ... % rhos for the lines
    thetaValues) % thetas for the lines
    % Find the largest two modes in the distribution of angles.
    bins = -90:10:90; % Use bins with widths of 10 degrees
    [counts, bins] = histcounts(thetaValues, bins); % Get histogram
    [~,indices] = sort(counts, 'descend');
    % The first angle corresponds to the largest histogram count.
    a1 = (bins(indices(1)) + bins(indices(1)+1))/2; % Get first angle
    % The 2nd angle corresponds to the next largest count. However, don
        't
    % find a bin that is too close to the first bin.
for i=2:length(indices)
    if (abs(indices(1)-indices(i)) <= 2) || ...
        (abs(indices(1)-indices(i)+length(indices)) <= 2) || ...
        (abs(indices(1)-indices(i)-length(indices)) <= 2)
        continue;
    else
        a2 = (bins(indices(i)) + bins(indices(i)+1))/2;
        break;
    end
end

```

```
    % Get the two sets of lines corresponding to the two angles. Lines
        will
    % be a 2xN array, where
    % lines1[1,i] = theta_i
    % lines1[2,i] = rho_i
    lines1 = [];
    lines2 = [];
    for i=1:length(rhoValues)
        % Extract rho, theta for this line
        r = rhoValues(i);
        t = thetaValues(i);

        % Check if the line is close to one of the two angles.
        D = 25; % threshold difference in angle
        if abs(t-a1) < D || abs(t-180-a1) < D || abs(t+180-a1) < D
            lines1 = [lines1 [t;r]];
            elseif abs(t-a2) < D || abs(t-180-a2) < D || abs(t+180-a2)
                < D
            lines2 = [lines2 [t;r]];
        end
    end
end

%% sortLines Function
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Sort the lines.
```

```
% If the lines are mostly horizontal, sort on vertical distance from yc
    .
% If the lines are mostly vertical, sort on horizontal distance from xc
    .
function lines = sortLines(lines, sizeImg)
    xc = sizeImg(2)/2; % Center of image
    yc = sizeImg(1)/2;
    t = lines(1,:); % Get all thetas
    r = lines(2,:); % Get all rhos
    % If most angles are between -45 .. +45 degrees, lines are mostly
    % vertical.
    nLines = size(lines,2);
    nVertical = sum(abs(t)<45);
    if nVertical/nLines > 0.5
        % Mostly vertical lines.
        dist = (-sind(t)*yc + r)./cosd(t) - xc; % horizontal distance
            from center
    else
        % Mostly horizontal lines.
        dist = (-cosd(t)*xc + r)./sind(t) - yc; % vertical distance
            from center
    end
    [~,indices] = sort(dist, 'ascend');
    lines = lines(:,indices);
end

```

```
%% findIntersections Function
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Intersect every pair of lines, one from set 1 and one from set 2.
% Output arrays contain the x,y coordinates of the intersections of lines.
% xIntersections(i1,i2): x coord of intersection of i1 and i2
% yIntersections(i1,i2): y coord of intersection of i1 and i2
function [xIntersections, yIntersections] = findIntersections(lines1, lines2)
 N1 = size(lines1,2);
 N2 = size(lines2,2);
 xIntersections = zeros(N1,N2);
 yIntersections = zeros(N1,N2);
 for i1=1:N1
     % Extract rho, theta for this line
     r1 = lines1(2,i1);
     t1 = lines1(1,i1);

     % A line is represented by (a,b,c), where ax+by+c=0.
     % We have r = x cos(t) + y sin(t), or x cos(t) + y sin(t) - r = 0.
     l1 = [cosd(t1); sind(t1); -r1];

     for i2=1:N2
         % Extract rho, theta for this line
         r2 = lines2(2,i2);
```

```
                t2 = lines2(1,i2);

                l2 = [cosd(t2); sind(t2); -r2];

               % Two lines l1 and l2 intersect at a point p where p = l1
                   cross l2
               p = cross(l1,l2);
               p = p/p(3);

                xIntersections(i1,i2) = p(1);
                yIntersections(i1,i2) = p(2);
        end
       end

end


%% createReference Function
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Get predicted intersections of lines in the reference image.
function [xIntersectionsRef, yIntersectionsRef] = createReference(
    sizeRef)
    sizeSquare = sizeRef/8; % size of one square
    % Predict all line intersections.
    [xIntersectionsRef, yIntersectionsRef] = meshgrid(1:9, 1:9);
    xIntersectionsRef = (xIntersectionsRef-1)*sizeSquare + 1;
```

```
    yIntersectionsRef = (yIntersectionsRef−1)∗sizeSquare + 1;
    % Draw reference image.
    Iref = zeros(sizeRef+1, sizeRef+1);
    %figure(13), imshow(Iref), title('Reference image');
    % Show all reference image intersections.
    %hold on
    %plot(xIntersectionsRef, yIntersectionsRef, 'y+');
    %hold off
end


%% findCorrespondence Function
% Find the best correspondence between the points in the input image
    and
% the points in the reference image. If found, the output is the four
% outer corner points from the image, represented as a a 4x2 array, in
    the
% form [ [x1,y1]; [x2,y2], ... ].
function [corners, nMatchesBest, avgErrBest] = findCorrespondence( ...
 xIntersections, yIntersections, ... % Input image points
 xIntersectionsRef, yIntersectionsRef, ... % Reference image points
 I)
% Get the coordinates of the four outer corners of the reference image,
% in clockwise order starting from the top left.
pCornersRef = [ ...
     xIntersectionsRef(1,1), yIntersectionsRef(1,1);
```

```
    xIntersectionsRef(1,end), yIntersectionsRef(1,end);
    xIntersectionsRef(end,end), yIntersectionsRef(end,end);
    xIntersectionsRef(end,1), yIntersectionsRef(end,1) ];
M = 4; % Number of lines to search in each direction
DMIN = 4; % To match, a predicted point must be within this distance
nMatchesBest = 0; % Number of matches of best candidate found so far
avgErrBest = 1e9; % The average error of the best candidate
N1 = size(xIntersections,1);
N2 = size(xIntersections,2);
for i1a=1:min(M,N1)
    for i1b=N1:-1:max(N1-M, i1a+1)
        for i2a=1:min(M,N2)
            for i2b=N2:-1:max(N2-M, i2a+1)

                % Get the four corners corresponding to the intersections
                % of lines (1a,2a), (1a,2b), (1b,2b, and (1b,2a).
                pCornersImg = zeros(4,2);
                pCornersImg(1,:) = [xIntersections(i1a,i2a) yIntersections
                    (i1a,i2a)];
                pCornersImg(2,:) = [xIntersections(i1a,i2b) yIntersections
                    (i1a,i2b)];
                pCornersImg(3,:) = [xIntersections(i1b,i2b) yIntersections
                    (i1b,i2b)];
                pCornersImg(4,:) = [xIntersections(i1b,i2a) yIntersections
                    (i1b,i2a)];

```

```
                % Make sure that points are in clockwise order.
                % If not, exchange points 2 and 4.

                v12 = pCornersImg(2,:) - pCornersImg(1,:);
                v13 = pCornersImg(3,:) - pCornersImg(1,:);
                if v12(1)*v13(2) - v12(2)*v13(1) < 0
                     temp = pCornersImg(2,:);
                    pCornersImg(2,:) = pCornersImg(4,:);
                    pCornersImg(4,:) = temp;
                end


                % Fit a homography using those four points.
                T = fitgeotrans(pCornersRef, pCornersImg, 'projective');

                % Transform all reference points to the image.
                pIntersectionsRefWarp = transformPointsForward(T, ...
                [xIntersectionsRef(:) yIntersectionsRef(:)]);


                % For each predicted reference point, find the closest
                % detected image point.
                dPts = 1e6 * ones(size(pIntersectionsRefWarp,1),1);
                for i=1:size(pIntersectionsRefWarp,1)
                x = pIntersectionsRefWarp(i,1);
               y = pIntersectionsRefWarp(i,2);
```

```
            d = ((x-xIntersections(:)).^2 + (y-yIntersections(:)).^2)
                .^0.5;
            dmin = min(d);
            dPts(i) = dmin;
        end

        % If the distance is less than DMIN, count it as a match.
        nMatches = sum(dPts < DMIN);

        % Calculate the avg error of the matched points.
        avgErr = mean(dPts(dPts < DMIN));

        % Keep the best combination found so far, in terms of
        % the number of matches and the minimum error.
        if nMatches < nMatchesBest
        continue;
        end
        if (nMatches == nMatchesBest) && (avgErr > avgErrBest)
        continue;
        end

        % Got a better combination; save it.
        avgErrBest = avgErr;
        nMatchesBest = nMatches;
        corners = pCornersImg;

```

```
                        end
                    end
                end
end
end
```