Chess Piece Detection

by

Craig Belshe

Senior Project

ELECTRICAL ENGINEERING DEPARTMENT

California Polytechnic State University

San Luis Obispo

2021

TABLE OF CONTENTS

LIST OF TABLES, FIGURES, AND EQUATIONS

Tables *Page*

Figures

Equations

Abstract

This project determines the positions of each piece on a physical chessboard, so that a computer can record a game of chess by noting down the piece positions at the end of each turn. It determines each move so that the game can be replayed later without watching actual footage. The project allows for easy viewing of past games. An image of the chessboard is analyzed to detect each square on the board, and each piece's location. This is then done for each turn, so that the system can keep track of an entire game.

I. INTRODUCTION

The goal of this project is to be capable of identifying chess pieces and their locations on a chessboard using just an image of the chess game. This needs to be accurate and perform quickly enough to observe and record every move in a chess game. For each turn, the project aims to be able to know the location of all pieces on the chess board "grid." This involves creating a program capable of finding both chessboards and chess pieces solely through analyzing an image, and from there determining all positions on the board.

II. BACKGROUND

This project requires at least basic background knowledge of chess. A chessboard is divided into 64 equal sized squares in an eight-by-eight grid. Each square alternates between a dark or light color from its four neighbors. Two players each have 16 pieces each set on a square on the board, and each player has six different types of pieces. To function, this project must be able to differentiate between each type of piece, and place it to a location on the board.

III. CUSTOMER NEEDS ASSESSMENT

Customers will use this device primarily to record chess games in a more efficient and easier to view manner than simple video. This will be useful for people who wish to save or share their chess games but prefer to play chess on a physical board. It will allow people to review their own games so they can learn from them.

IV. REQUIREMENTS AND SPECIFICATIONS

The requirements and specifications were determined by looking at how the project will be used. To be of use, it must record games accurately, or else it could not be trusted. It also must be capable of working in multiple environments without much effort by the user. For example, it must function in both very low lighting and very high lighting, with possible glare, and with the camera in possibly different locations. These requirements were used to fill out the table below.

TABLE I

CHESS PIECE DETECTION REQUIREMENTS AND SPECIFICATIONS

Marketing Requirements	Engineering Specifications	Justification
1, 3, 4	Detect pieces from different camera angles, but works best when centered, on the side of the board (between the players), and "looking" at the center of the board. Camera angle can vary by at least 20 degrees so long as board is fully in view without affecting performance.	The device cannot work if any piece is out of frame, but the device must be able to withstand lack of precision in setup to be more user friendly. Mounting the device directly over the board would likely not be user-friendly, so mounting it to the side is better.
1, 3, 4	Detect pieces under varying light conditions. Must at least be capable of detection between 50 and 100,000 lux.	The device cannot work pieces are not visible, but the device must be able to work in different rooms which may have different lighting to be easy to install.
2	Cost under $50 for device, including camera, mounting system, and wiring; computer excluded.	Device must be affordable enough to be worthwhile for a game of chess. Since it only requires a camera and mount, it should not be too expensive.
1, 3, 4	Detect pieces from different camera positions. Camera can be moved at least up to 6 inches from optimal position so long as board is fully in view.	The device cannot work if any piece is out of frame, but the device must be able to withstand lack of precision in setup to be more user friendly.
1, 3	Function on a standard chessboard at minimum. Square sizes between 2 and 2.5 inches on the side, King should be 3.4 – 4.5 inches tall. King's base should be between 75-80% of the square size, and all other pieces are smaller. There is no standard for color; for this project the pieces will appear black and white, and the board color does not matter.	Be able to function on a normal chessboard for ease of use. Some chessboards are too abnormal to be practical.
5, 4	Record each turn and be capable of showing each move for at least the last 250 moves. Will record moves as changes in piece location (for example, piece moved from location (5,5) to (6,5) with (0,0) as the close left corner from the white player's perspective). To play back moves, the program goes step by step from the starting position.	It may not be practical to record an infinitely long game, but it should be capable of remembering most games. Recording changes in position, rather than the position of every piece, saves space.

5	Record each move as part of the same game, for at least one game recorded.	To be useful, must remember one game, so user may view the game from start to end. Records game as a series of turns so the game can be replayed turn by turn, not as video.
5, 3	Software must allow past game(s) that were recorded to be accessed by the user. Should run on a standard pc running Windows 10.0.19042 (This is the latest release version of Windows).	User must have a way to easily view the game back after recording it.
Marketing Requirements 1. High accuracy 2. Low cost 3. Easy to install and use 4. Capture entire board 5. Record games and allow them to be visualized by user		

TABLE II

CHESS PIECE DETECTION DELIVERABLES

Delivery Date	Deliverable Description
May 2021	Design Review
July 2021	EE 461 demo
July 2021	Detection of any piece and its position under specific conditions
July 2021	EE 461 report
Dec. 2021	EE 462 demo
Dec. 2021	ABET Sr. Project Analysis
Dec. 2021	Sr. Project Expo Poster
Dec. 2021	EE 462 Report

V. DESIGN

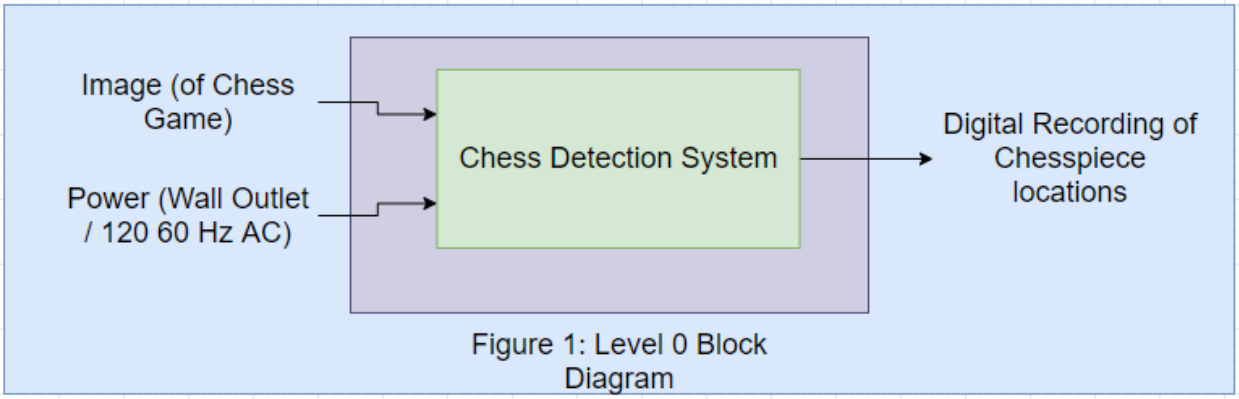


Figure 1: Level 0 Block Diagram

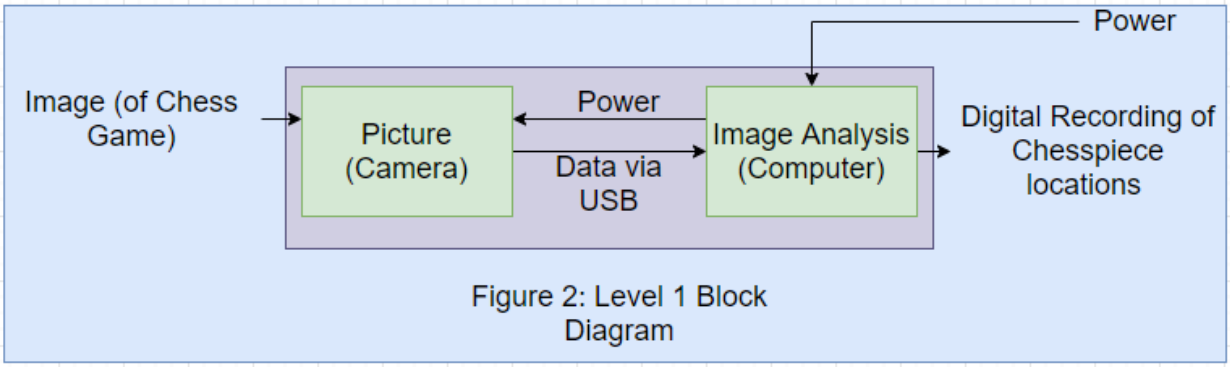


Figure 2: Level 1 Block Diagram

TABLE III

LEVEL 0 BLOCK DIAGRAM FROM FIGURE 1

Inputs	**Functionality**
Image	Light reflecting off the chess pieces is used as an input by the system to determine the piece locations.
Power	Electricity going to the computer, and the camera, is needed as an input for the system to run. Power enters system from US outlet, or 120 V RMS @ 60 Hz AC.
Outputs	
Digital Recording of Chess Piece Locations	Outputs a recording of the location of each piece at the end of each turn. For example, at the end of turn 1 the King might be at E1.
Overall Module	Returns the recording of chess locations for a game of chess.

TABLE IV

LEVEL 1 PICTURE BLOCK FROM FIGURE 2

Picture (Camera)		Takes pictures of the chessboard constantly to send to computer.
Image	Input	Camera takes Image of game as an input to take a picture.
Power	Input	Camera requires power to operate, received via the computer. Camera needs DC voltage.
Data	Output	Camera sends picture data to computer for analysis.

TABLE V

LEVEL 1 IMAGE ANALYSIS BLOCK FROM FIGURE 2

Image Analysis (Computer)		Performs computations on image to get final output recording.
Data	Input	Data containing picture from camera for analysis
Power	Input	Computer needs power to operate and to power camera.
Power	Output	Provides power to the camera while receiving data.
Digital Recording of Chess Piece Locations	Output	Returns the recording of chess locations for a game of chess.

TABLE VI

MATERIAL COST ESTIMATE

Parts:	
Optimistic Cost (a)	$40
Most Likely Estimate (b)	$60
Pessimistic Cost Estimate (c)	$90
Estimate Cost = (a+4b+c)/6	$62

The three anticipated components that will be needed are a camera, cable, and mounting device. At cheapest, these could come together in a $40 package. However, a specific camera and mounting device would be best suited, which would most likely cost slightly more to purchase separately. Higher quality components could increase the cost as much as $90. Using Equation 6 from Chapter 10 of *Design for Electrical and Computer Engineers* gives an overall estimate of $62 for parts.

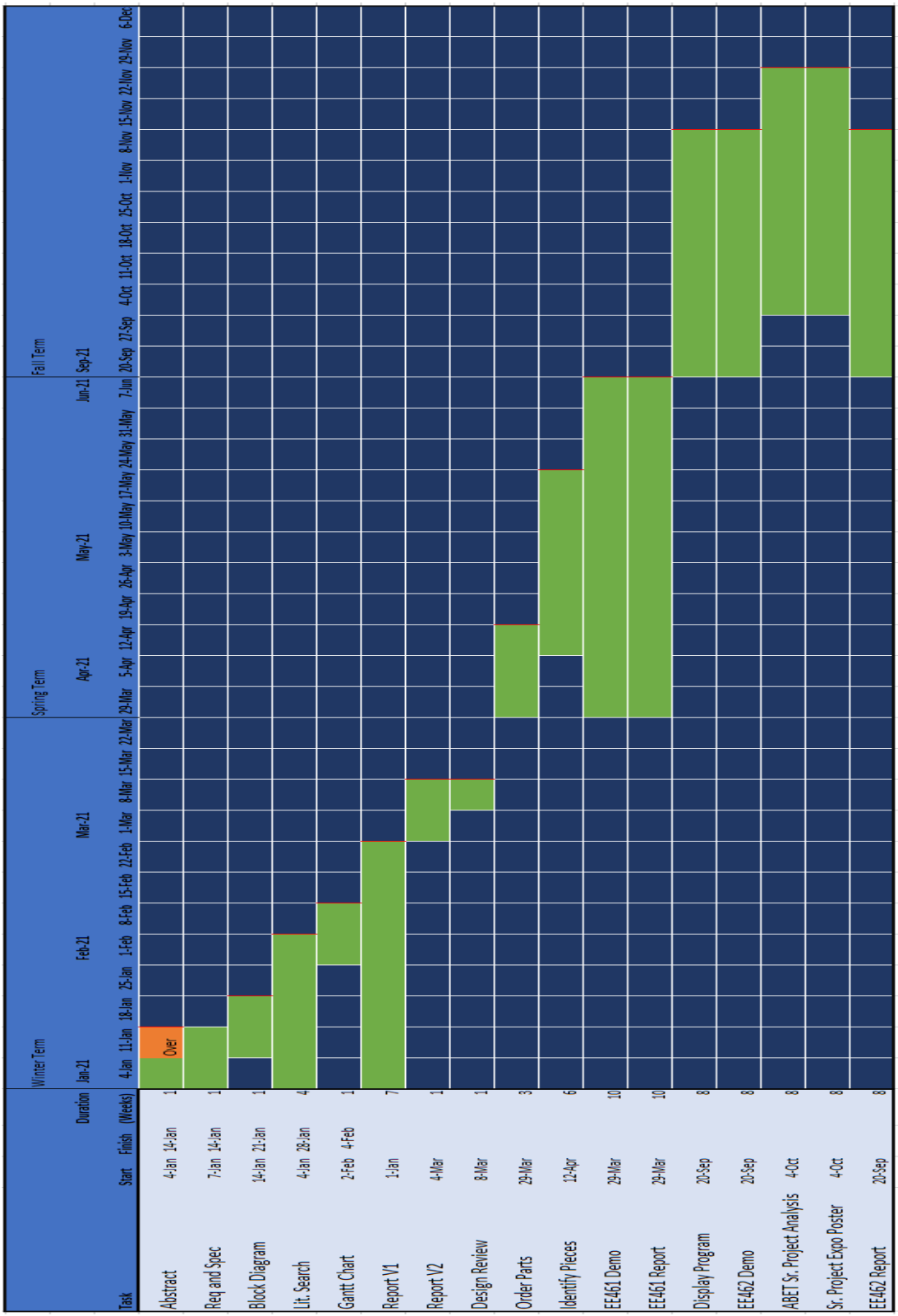
TABLE VII

LABOR COST ESTIMATE

Labor:	
Estimate Labor	180 hours
Estimate Labor Cost	$5400

Labor costs were calculated for one person, at $30 / hr. The number of hours was determined with 6 hours a week through 6 quarters at 10 weeks per quarter. Over the course of the project, this leads to 120 total hours of labor. At $30 / hr., this leads to $3600 in labor costs.

TABLE VIII
GANTT CHART

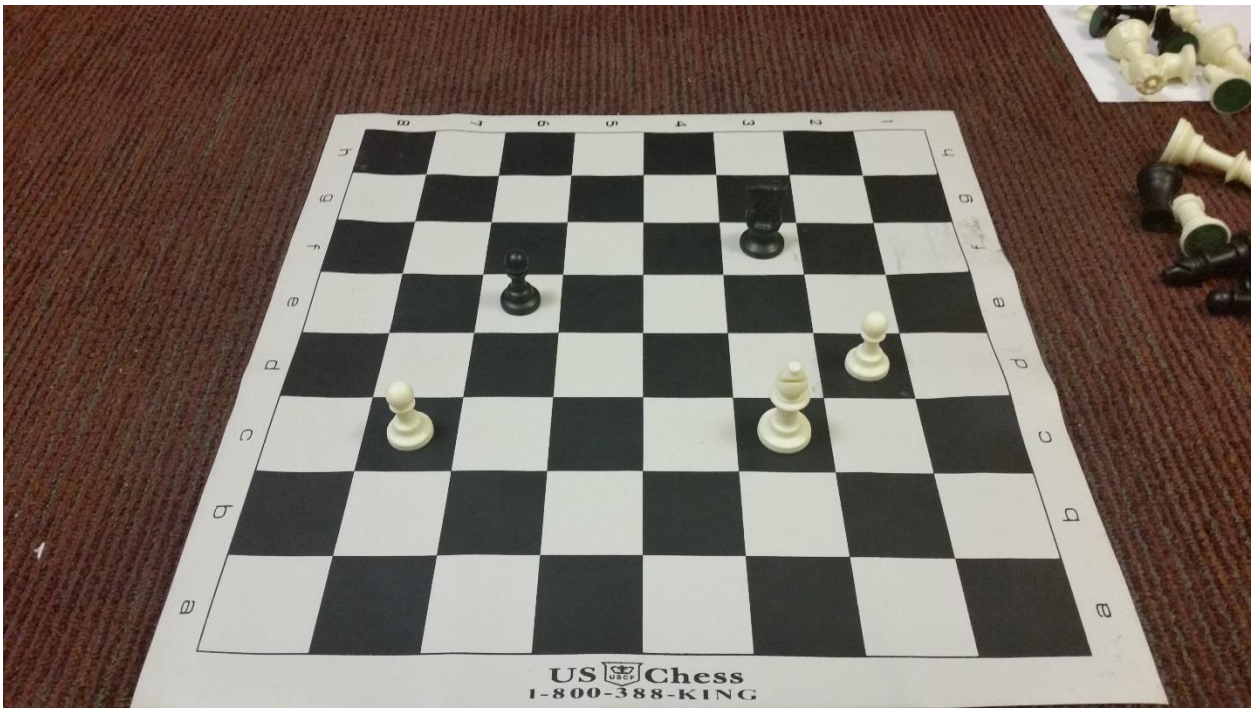


VI. DEVELOPMENT

Most of the development of this project was pure software – the majority of the difficulty in finding chess pieces on a chessboard lies in finding a way to accurately find both the board and the pieces in an image. This project was done entirely using Python, with its Open-Source Computer Vision (OpenCV) and PyTorch libraries. Most of the development was done on an Ubuntu 20.04.

The code was designed to work best with images in which the board was the largest object, was lying on a darker surface, and in which the camera caught the entire board from a slight angle. An example image is shown in Figure III below. This image is also used throughout this section to demonstrate how images are manipulated to get the chess piece locations.

FIGURE III
EXAMPLE IMAGE



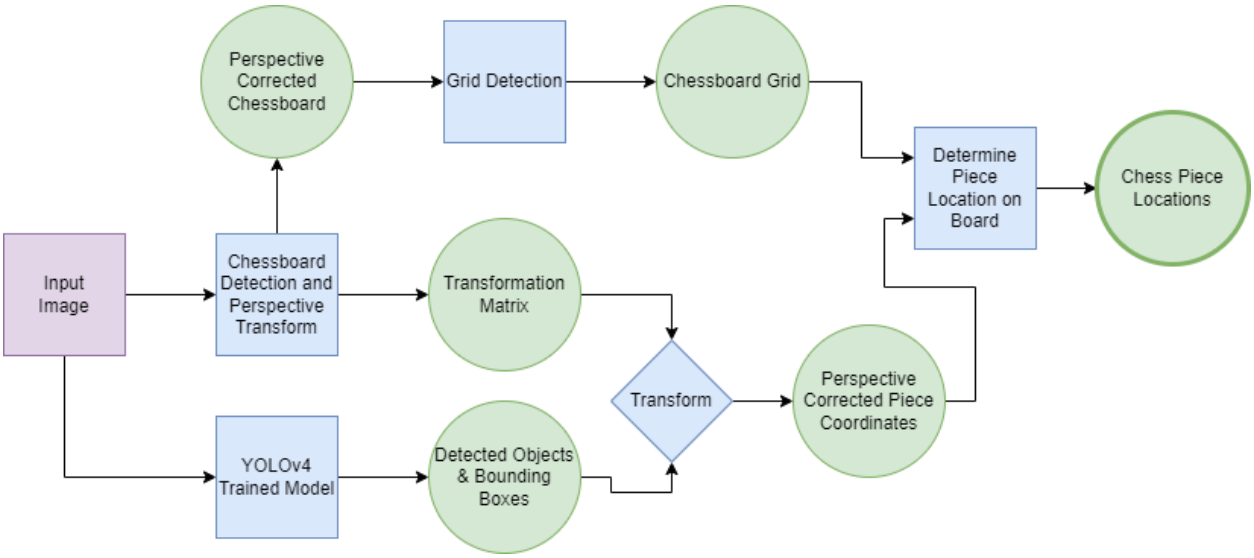
For simplicity, the project was split into 3 initial parts:

1. Board Detection
2. Grid Detection
3. Chess Piece Detection

In the first part, the goal was to separate the chessboard from the surrounding parts of the image, and then adjust the image so the chessboard was perfectly square. The second part involves finding the "grid" created by the 64 squares on the board. The final part searches the image and finds the location and type of any chess pieces present on the board. Once all three parts are complete, the location of the chess pieces found in the third part can be run through the same transformation as the board in part one to find their location on the grid found in part two. This results in the location of each detected chess piece on the grid, which is the goal of the project.

A flow diagram of the structure for this project is shown below in Figure IV. The blue squares stand for the methods, while the green circles are the outputs from each method. The final output of the project is the chess piece locations on the chessboard – for example, a list of pieces with their location on the board coordinate space, where the further left corner square is (0, 0) and the closest right corner square is (7, 7).

FIGURE IV
DESIGN FLOW CHART



A. BOARD DETECTION

For this part, the goal was to cut out the chessboard from the surrounding image and shift the perspective of the camera, so it appears completely square (as if the camera was directly above the board). This was necessary for a few reasons. The images of the chess game that the computer would be working with would include whatever else happened to be in frame. Objects unrelated to the game could "distract" from the actual chess pieces during later, more sensitive object detection. Further, to analyze a chess game in real time, the computer needs to process as few pixels as possible – so cutting out unneeded parts of the image would save computation time. Correcting the image was needed since when looked at from an angle the chessboard would appear more trapezoidal than square. Correcting the image to be square would allow the "grid" of squares in the board to be found later, since every square in the board would be perfectly square and they would all be the same size.

To start, it was necessary to find the four corners of the board. These would allow the board to be cut from the surrounding image and could be used to shift the board. To do this, the image of the board was first scaled down to a smaller resolution since the full-sized image would take far too long, and the added detail was unnecessary for finding a large object like the board. This was simply done by scaling down the image to have 1/5 the height and width. This preserved the original scale of the image. Test images varied slightly in size, since many were taken from the internet, but they all were at least 3,000 pixels in both dimensions. This resized image was only used for board and grid detection. An example image of a resized board is shown in Figure V below.

FIGURE V
RESIZED IMAGE

After resizing, the board could be converted from a color, RGB image to a simple grayscale one – this is because the actual colors of the board will have no effect on results. Chess usually utilizes two colors, a light one and a dark one (generally black and white), and the difference between the two will be preserved in a grayscale image. This also saves time, as the grayscale image only has one magnitude for the shade of gray (0-255) for each pixel, as opposed to the three needed for an RGB color image. As a result, less information needs to be worked with going forward. To convert to grayscale, the OpenCV function cv2.`cvtColor(image, cv2.COLOR_RGB2GRAY)` was used. This converts the red, green, and blue values to a single gray value using Equation I [14]. Y is the magnitude of the single gray scalar, and R, G, and B correspond to the color intensities.

EQUATION I

RGB TO GRAYSCALE

$$Y = 0.299 \cdot R + 0.587 \cdot G + 0.144 \cdot B$$

After resizing the board, the next step was to try and find the contour (the border) of the chessboard itself. However, there was a fair bit of difficulty with detecting the squares inside the board as part of the background. Therefore, the image was first changed so the insides of the board were irrelevant. This was done by first thresholding the grayscale image, to convert it to a binary image (where each pixel is either black or white). A threshold value of 140 was used to make sure that most board would be separated from a darker background. This meant that pixels with a magnitude higher than 140 would become pure black, while all values would become white. Next, the OpenCV function `cv2.findContours(image, cv2.RETR_LIST, cv2.CHAIN_APPROX_SIMPLE)` was used to find contours in the image. This would return a list of shapes found in the image. The contours returned are always the outlines of a closed shape. The CHAIN_APPROX_SIMPLE and RETR LIST are both constants which define how the contours are returned and how they are represented, but do not have any effect on the contours themselves. To find the chessboard, it was assumed that the board would be the largest object in the image – and would have the largest area in the list of contours. This would eliminate the smaller contours found of the pieces, squares, and other objects that may be in the image (such as a person's hand). However, this method fails if there is a larger object in the image. An example of the contour of a chessboard that was found this way is shown in Figure VI below. The found contour is plotted in green on top of the original image from Figure V.

FIGURE VI

CHESSBOARD CONTOUR

Once the contour following the border of the chessboard has been found, the chessboard corners can easily be found by assuming the upper-leftmost part of the contour is a corner, the upper-rightmost part is another corner, and so on. To do this, the formulas shown in Equation II below were applied to every point on the contour. These equations generally returned the corners of the contour. For example, the point with the largest sum of its x and y coordinates would be the bottom right point.

EQUATION II

CORNERS OF A SQUARE

$$Bottom\ Right = \max(x + y)$$
$$Bottom\ Left = \max(x - y)$$
$$Top\ Right = \min(x - y)$$
$$Top\ Left = \min(x + y)$$

Figure VII shows the corners found from the contour shown in Figure VI, using this method. Each of the four corners are drawn on the original image as red dots.

FIGURE VII

CHESSBOARD CORNERS

After finding the four corners of the chessboard, the next step is to cut out the rest of the image and to "correct" the image so the chessboard is completely square – in the past images, the chessboard has been noticeably trapezoidal due to the angle the image was taken from. While that angle is necessary in order to determine the type of chess piece later (the pieces all look the same from above), the computer does not have the same ability that humans due to naturally understand that the board is actually square from this angle. Since we know the board is square, we can shift our image of the board to match.

Shifting the image is done through a perspective transformation, or homography. Since we know where the current corners of the chessboard are, and we know where we want them to go (into a square format), so we can find a matrix to transform the current image into the corrected one. This can be calculated as shown in Equation III below, where (u, v) are the new, shifted coordinates and (x, y) are the original coordinates of the corners.

EQUATION III

HOMOGRAPHY MATRIX

$$\begin{bmatrix} u \\ v \\ 1 \end{bmatrix} = \begin{bmatrix} h_{11} & h_{12} & h_{13} \\ h_{21} & h_{22} & h_{23} \\ h_{31} & h_{32} & h_{33} \end{bmatrix} \cdot \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$$

A code snippet of how this was done is shown in Table IX on the next page. An array containing the four corner points is created using NumPy, though the points are all shifted inwards to avoid including the border of the chessboard in the end result. The output chessboard needs to be a square, and it only needs to be large enough to recognize the location of each chess square. For this reason, the output was set to be 200x200 and an array with corners corresponding to the output square was created (with the upper left corner being (0, 0) and the upper right square being (199, 0)). These two sets of coordinates were then entered into the OpenCV function `cv2.getPerspectiveTransform(original_coord, new_coord)`. This found the homography matrix needed to shift the perspective of the image so that the chessboard would be square, using a method derived from Equation III.

TABLE IX

PERSPECTIVE TRANSFORM CODE

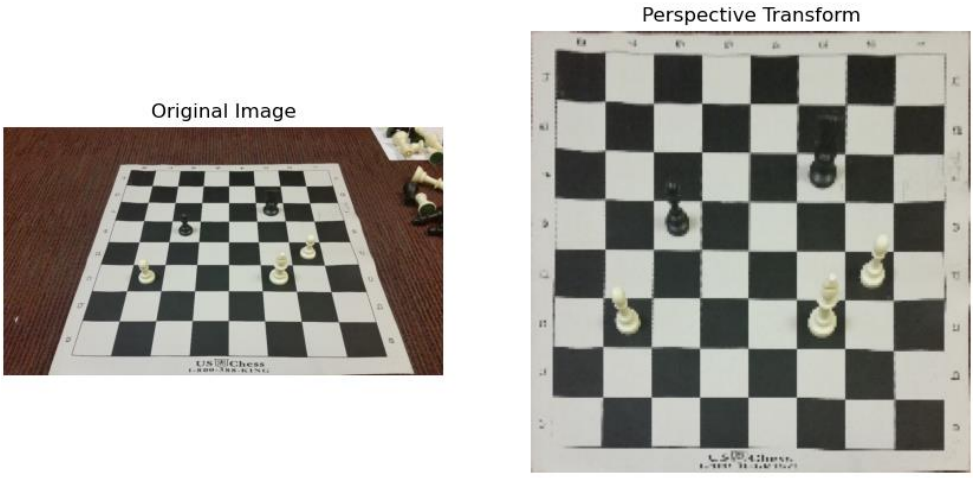
```
# tlp -> top left corner; trp -> top right corner
# blp -> bottom left corner; brp -> bottom right corner
# np -> numpy library; cv -> OpenCV library

# The plus/minus 5 are to exclude the direct borders
rect = np.array(((tlp[0]+5, tlp[1]+5), (trp[0]-5, trp[1]+5), (brp[0]-5, brp[1]-5),
(blp[0]+5, blp[1]-5)), dtype="float32")

width = 200
height = 200
dst = np.array([[0,0], [width-1,0], [width-1,height-1], [0,height-
1]],dtype="float32")
M = cv.getPerspectiveTransform(rect,dst)
warped_img = cv.warpPerspective(image, M, (width, height))
```

Once the perspective transformation matrix has been found, it can be used to “warp” or transform the original image. While this could be applied to the entire image, there is no need to do so as only the chessboard is needed. Using the OpenCV function `cv2.warpPerspective(image, matrix, (width, height))`, a square output image containing only the chessboard can be gained. An example of this output image is shown next to the original image in Figure VIII. It may be worth noting the images are scaled to fit the page (the original image is technically larger).

FIGURE VIII
WARPED PERSPECTIVE OF CHESSBOARD



The transformed chessboard is now square, and all the square spaced in the board are equal size. This will make it possible to find the grid defining each position on the board. However, the pieces on the board have also been stretched and distorted – this could make it difficult to recognize them through the transformation. To get around this, the pieces will be found in the original image, and then their coordinates can be shifted into the coordinate grid of the warped image through the OpenCV function `cv2.perspectiveTransform(src, matrix)`. The shifted piece coordinates can then be used to place them inside the grid.

B. GRID DETECTION

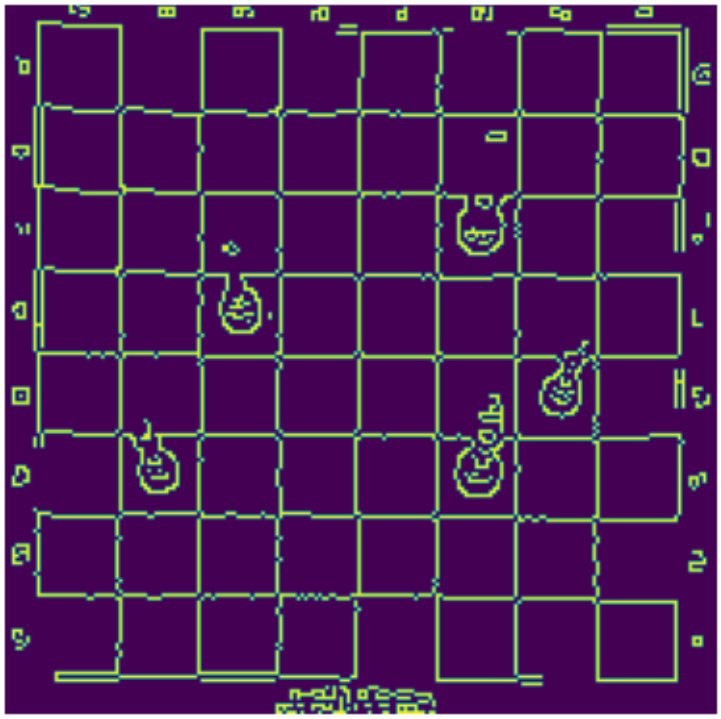
The goal of grid detection was to end up with a set of vertices for each of the 64 squares in the chessboard. This would allow the image coordinates of each piece to be mapped to the correct square, so that a list of each piece's location could be created. This part started with the transformed chessboard, such as the one in Figure VIII.

There are a few different ways to go about this. One of the initial attempts involved using a method called Harris Corner detection to directly find the corners of each square. This method did not seem to perform well, likely due to the proximity of other squares, so it was decided to use the lines formed by the squares to find the vertices.

First, Canny Edge detection was used to find a map of all the edges present in the image. This method works by finding the gradient, or change in pixel magnitude, in both the x and y directions for each pixel in the image. If the gradient is very high, then it must be located on an edge since there is a sharp change in intensity. Two hysteresis thresholds are applied as well, to suppress non-maxima edges (pixels whose gradients are high, but not higher than their neighbors). OpenCV implements Canny Edge detection, which can be used with `cv2.Canny(warped_image, 100, 150)`, where 100 and 150 are threshold values that were found worked well over multiple images. The output edges from applying the Canny edge detection to the same image as used in chessboard detection are shown below in Figure IX. The light-colored lines are the parts marked as edges.

FIGURE IX

CANNY EDGE DETECTION



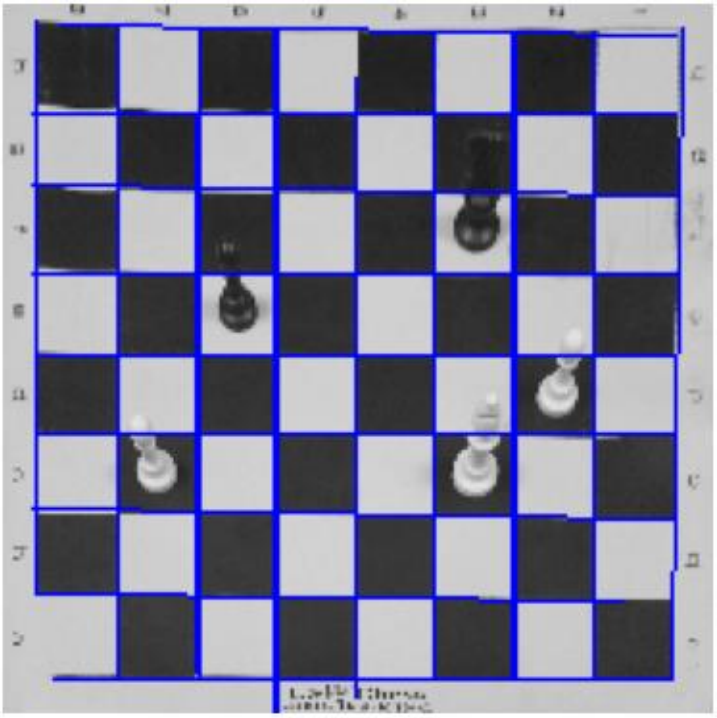
The Canny edge detection on its own has a few problems. It detects all locations where pixel magnitudes change greatly, so picks up numbers and words on the side of the board, as well as the chess pieces. These are not needed to find the underlying grid, so need to be removed. Using a stronger blurring function before applying the edge detection does help eliminate these, but not well enough to find the grid. Another problem is that many of the lines in the chess board have gaps – these are just caused by noise and other problems with the original image.

To get from this collection of edges to an actual grid, a method called the Hough line transform can be used to find the lines along each edge. The Hough transform involves graphing the image in such a way that any given line (such as $y = ax + b$, although lines are usually represented using θ and ρ) is given a point in the Hough space graph. Single points from the original image become sine waves in the Hough graph. This makes it so that the point where many lines intersect in the new graph represents a line in the original image.

To implement the Hough Transform using OpenCV, the function `cv2.HoughLinesP(canny_image, 1, np.pi/180, 60, minLineLength=40, maxLineGap=70)` was used. The P at the end of the function name calls a version of the function that returns lines using an easier to understand point based format – it returns the two endpoints for each line. The function takes the image generated using Canny edge detection, as well as a few threshold values, and returns a list of lines. The `minLineLength` parameter is used to define how long a line must be for the function to return it. This value was set relatively high at 40 since lines should be travelling from one side of the chessboard to the other. However, the `maxLineGap` was also set quite high, at 70, so that chess pieces that interrupt the flow of the line would not stop the function from returning the correct line (making it more tolerant to objects between the camera and the board). An example of the lines found using this function is shown in Figure X below. The lines are plotted in blue on top of the perspective transformed image.

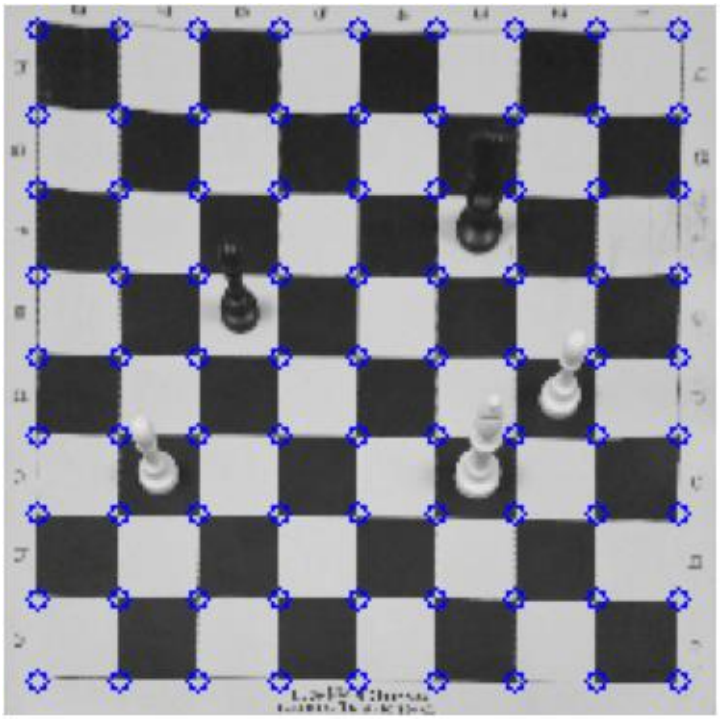
FIGURE X

HOUGH LINES



The Hough Lines function does a fairly good job finding all the lines. Some of the lines are not as long as they should be, or are drawn at slight angles, but it consistently finds every line making up the chessboard grid. In many cases, multiple Hough lines were drawn next to each other along the same edge. This was easy to eliminate by discarding lines withing a certain distance from each other. Any lines that were found but were not close to being either vertical or horizontal were also discarded, as they could not be part of the board. The remaining lines were shifted slightly to either be perfectly vertical or horizontal, as needed, and were assumed to extend along the entire image. From here, the intersection points of all the lines can be found to get the vertices of all the squares in the image. The vertices found using this method are shown in Figure XI below.

FIGURE XI
SQUARE VERTICES



In this example, all the points were found. In other tests, all or nearly all points were generally found, but there were frequently extra points from Hough lines that were not part of the chessboard grid. These points needed to be removed. To do so, the side of a chessboard square was calculated using the list of square vertices. The distance to the closest point was calculated for each point, and then the median of the list of shortest distances was found. Since the closest point for any point on the grid would be roughly one square side, the median would be the length of a square. The median was used instead of the average so that it would be more resistant to outliers – so long as most points are correct, the distance in the middle should be the correct distance. The average could be skewed in one direction by an extreme case. For the example shown in Figure XI above, the side of a square was calculated to be 22.0 pixels using this method. Manually measuring the distance confirms it is accurate.

Now that the length of a square side has been found, any points that do not fit the grid can be eliminated based off their distance to the next point.

C. PIECE DETECTION

In many ways the most important part, the goal of Piece Detection was to search an image for chess pieces and return which pieces were present in the image as well as the location of the pieces. A few different methods were considered for this step – template matching, where the pieces would be compared to a template image, was decided to be too incapable of adjusting to new angles. Another method, SIFT, was not used since it is proprietary software, although it likely would have performed decently.

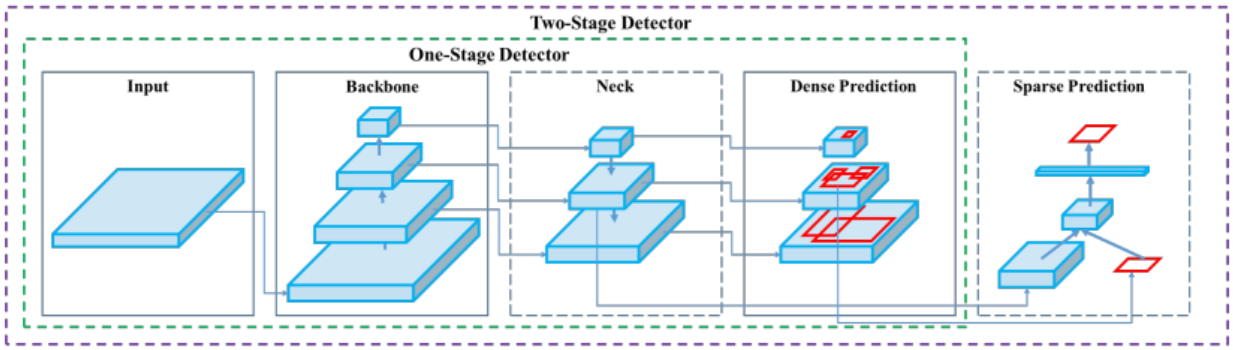
In the end, an Object Detection model known as YOLO v4 was used. The name stands for “You Only Look Once,” and after training the model, it can run fast enough to detect objects in real time, which is necessary in order for the chess piece detection to be relevant in an actual game of chess. It was chosen for being simultaneously fairly accurate but also very fast for a neural network-based object detection [15]. Once trained, it can detect objects in real time (even on a relatively old laptop, which was important for this project).

This method is faster than others for object detection in part because it predicts the location of the objects within the whole image, rather than applying object classification in various locations like other networks. YOLO splits the image into a grid of size S x S, and searches for objects within each grid cell. This makes it poor at finding close or overlapping objects, as only 2 objects can be within the same cell. However, since the bounding boxes are predicted, the model does a good job predicting objects in similar positions – similar to how chess pieces are always upright on a square in the chessboard. YOLO outputs both a bounding box and its probability of belonging to an object class.

YOLOv4 builds off the original YOLO model and the previous YOLOv3 detector to create a full single stage object detector. As an object detector, it consists mainly of two parts – a backbone, and a head. The backbone is pre-trained with images and then the head is used to predict the actual objects and their bounding boxes. Between the backbone and head, is often a layer used to collect the feature maps from previous stages. For the head, two-stage detectors such as the R-CNN models are common. As its name may suggest, YOLOv4 uses the single stage detector YOLO for its head. A diagram of the general structure of an object detector is shown in Figure XII [16].

FIGURE XII

OBJECT DETECTOR STRUCTURE [16]



YOLOv4 uses CSPDarknet53 for its backbone as a feature extractor. This is a convolutional neural network (CNN) that uses 53 convolutions written by the same guy that developed YOLO, and primarily used in YOLOv3 and YOLOv4. It is open source, and written in C. The creators claim it is significantly faster than other methods, such as ResNet-101 [17].

A convolutional neural network, such as is used in Darknet, is used to extract features from the image, such as might be used to identify chess pieces or other objects. It involves a convolution process in which filters are applied to an image many times to extract features. After some manipulation, these features are passed as arguments into a fully connected neural network which is trained to predict objects. The output can then be used to find the calculated probabilities of finding an object.

For the "neck" of the model, YOLOv4 uses two modules: Spatial Pyramid Pooling (SPP) and a modified Path Aggregation Network (PANet or PAN). SPP is used to increase the receptive field of the backbone by splitting the feature map into equal sized blocks, or "spatial pyramids," and then integrating the pyramids into a CNN and using max pooling to reduce the size (max pooling removes non-max features) [16]. PAN is a method for instance segmentation where the image is divided into a number of instances or layers. PAN uses a bottom-up path augmentation to shorten the information path, adaptive feature pooling to recover broken information between levels, and fully-connected fusion of layers [18].

YOLOv4 uses a variety of other modules, known as its "Bag of Freebies" (BoF). The name references that these methods almost manage to increase the performance of the model "for free" since the only computational cost is during the training phase – which is not super relevant to the model's performance – while allowing the cost of using the trained model to detect objects the same. During the backbone phase, YOLOv4 uses CutMix and Mosaic data augmentation to improve the variability of training images so the model is better able to handle differences in new images (such as different backgrounds), as well as DropBlock regularization and Class label smoothing. CutMix cuts out segments of two different images in the set and mixes them around. Mosaic works by combining segments of four different images. Class label smoothing helps to work around images that may be poorly labeled [16].

A variety of other methods are used as part of the BoF for the detector phase: YOLOv4 uses CIoU-loss, CmBN (based on CBN), Self-Adversarial Training, Eliminate grid sensitivity, and a few others [16].

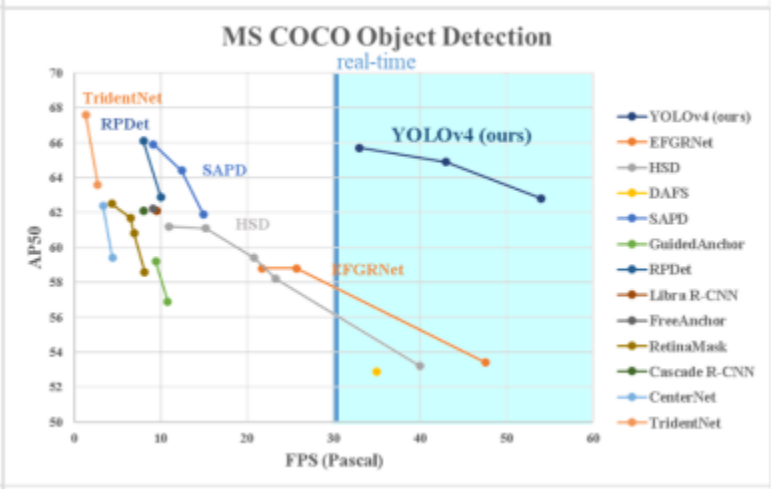
Another group of modules to increase performance, called the “Bag of Specials” (BoS) are utilized by YOLOv4. These are methods that may increase cost during the actual model application by a small amount, so are not “free,” but increase accuracy by enough to compensate. Some of the BoS methods in YOLOv4’s backbone phase include the Mish activation function, cross-stage partial connections (CSP), and multi-input weighted residual connections (MiWRC). The Mish Activation function is used on top of the CSPDarknet53 to more accurately train the neural network. CSP helps to save time by removing redundant gradient information between stages.

Some of the BoSs in the detector stage include the SPP and PAN blocks mentioned previously, as well as a modified Spatial Attention Module (SAM) block.

Overall, the YOLOv4 model, once trained is capable of object detection with good accuracy at real time speeds. A comparison of YOLOv4 to other object detectors is shown in Figure XIII. The comparison compares accuracy using the AP_{50} performance metric with the speed, in frames per second, on a specific set of hardware. All of the detectors were trained with a standard image dataset – MS COCO [16]. AP_{50} largely uses the precision (number of correct detections out of all detections) and recall (number of correct detections out of number of objects) to determine accuracy. The 50 in the name means that a detected object that overlaps the true object by at least 50% of their combined area.

FIGURE XIII

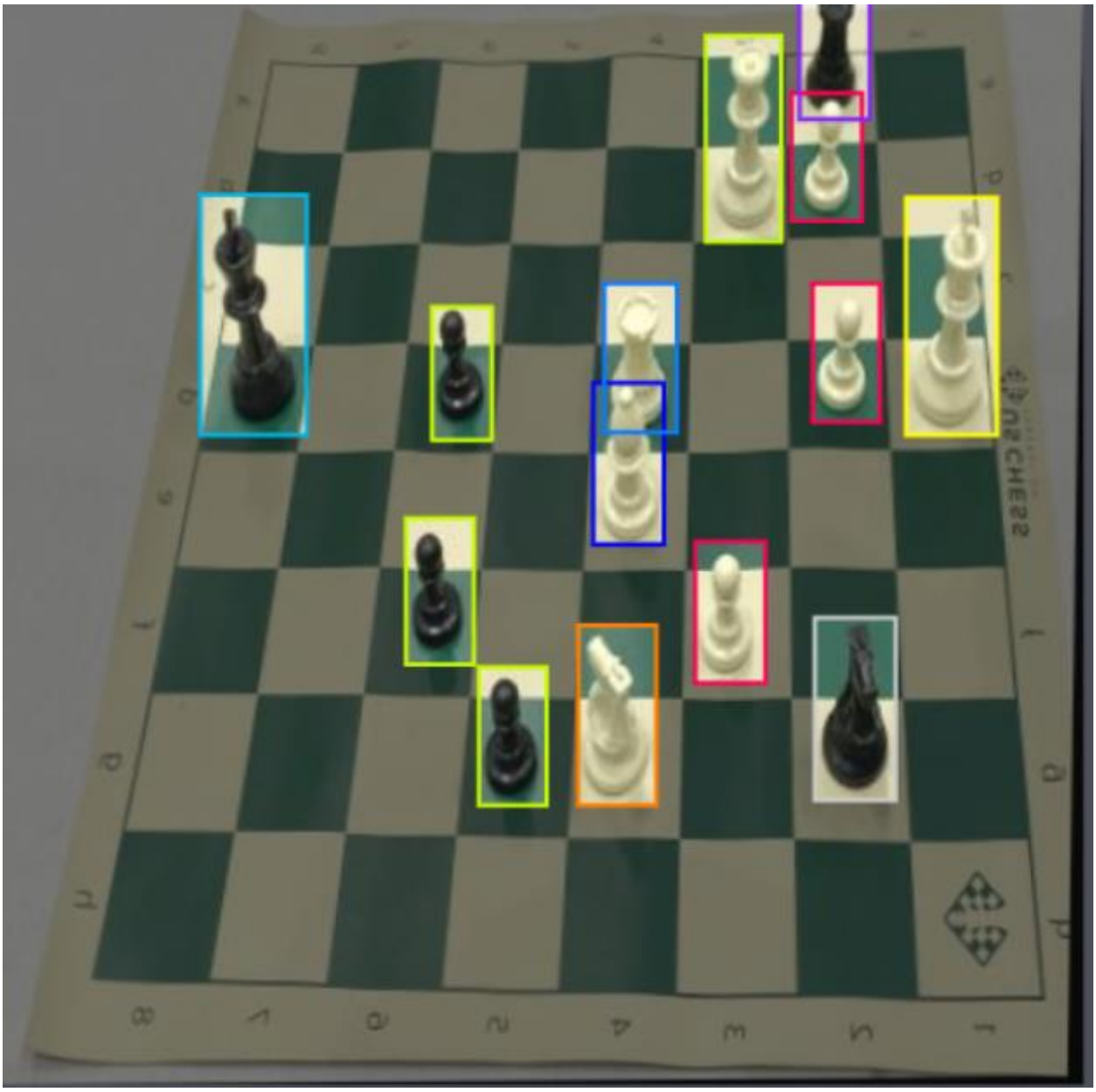
COMPARISON OF OBJECT DETECTORS [16]



To train the YOLOv4 model, a prebuilt dataset from Roboflow.com was used – a new image set was not created for this project. This included roughly 600 images of a chessboard with prelabeled chess pieces (each piece is labeled with the piece name and a box around it). Some adjustments were made to the original dataset – all the images were shifted to grayscale, and small rotations were applied to simulate possible changes in camera position.

There were a total of 606 training images, all 416x416 in size – while this size is not ideal, due to being a different size (and also square) than the images that were used to test and for the grid detection, it was not practical to try and create a new dataset for this project, due to the large number of images. All of the images appear to have been captured on a standard chessboard, although one that uses white and green squares. The background surrounding the chessboard is entirely white in all images. One image in the set contains no pieces. These images had a total of 7086 labelled pieces among them, for all of the 12 defined chess piece types. A labelled image is one where a person has gone through and defined bounding boxes for each piece. These labels can then be used by the training method to define what each piece is. An example of a pre-labelled image is shown in Figure XIV.

FIGURE XIV
EXAMPLE PRE-LABELED IMAGE



In Figure XIV, each piece has a box drawn about it representing its label – the boxes are different colors for different piece types. Since these images are all taken of situations that might occur in an actual game of chess, certain pieces are naturally more common. The number of labels throughout the entire online dataset for each chess piece type are included in Table X below.

TABLE X
DATASET LABEL NUMBERS

Piece	**Number of Labels**
Black Pawn	1,651
White Pawn	1,587
Black Rook	495
White Rook	460
Black Knight	478
White Knight	464
Black Bishop	337
White Bishop	422
Black Queen	211
White Queen	273
Black King	349
White King	359
Total	7,086

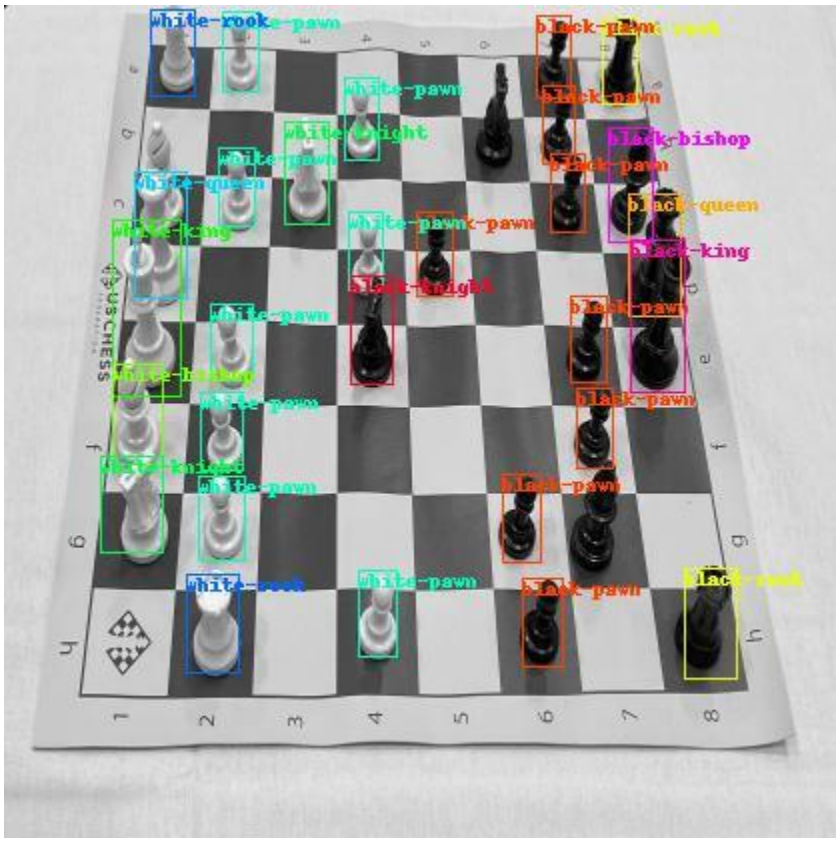
In the full dataset, pawns were easily the most prevalent. This over representation can negatively affect the training, as it can skew the results towards detecting pawns over other pieces. However, since pawns are also the most common in actual games of chess, it could make the detector more accurate with those pieces. The Queen pieces, on the other hand, were very under-represented in this dataset. If a new dataset were to be created for this project, it preferably would have another few hundred labelled Queens, Kings, and Bishops. Ideally, around a thousand of each type might be more effective, while avoiding over-training, but this would require many more images and better hardware to train the object detection model – so is outside the scope for this project.

Once the dataset was found, a publicly available Google CoLab provided by Roboflow, and PyTorch were used to run the training program – this took about 4 hours. Very little custom code was necessary here, as the YOLO model and programs to train it were already available. Once the model was finished, it was saved so that it can be used in the future without retraining. The copy of the CoLab code used can be found here.

Rather than rewrite the YOLOv4 model, a prebuilt python repository was used. This can be found at https://github.com/roboflow-ai/pytorch-YOLOv4. When the training module is run, the previous dataset can be fed in to train the object detection model. The standard values recommended by the module author were used – a batch size of 2, learning rate of 0.001, and only 50 epochs, or cycles of training (due to the fact that this was being run through Google CoLab, which would time out during longer computations).

When run on an image using the models module, the model returns a list of detected objects, including the four points defining the bounding box, and the chess piece type that has been detected. In order to run effectively, it needs to be run on a CUDA enabled device (CUDA is an API for GPU usage, which works fine with the Nvidia GTX 1050 on the device used to run the project). Images were not resized before feeding them into the model, though the code that was used to run the model would automatically resize them to 608x608. Figure XV shows an example output from the model.

FIGURE XV
YOLO MODEL RESULT



The model uses a black and white version of the original image. In this example image, only 3 pieces were missed – a white bishop, a black bishop, and a black knight. The other 29 pieces were all detected correctly. Since each object includes a bounded box, the location of the box can be used to determine the square the piece is on.

D. FULL CHESS DETECTION

With the first 3 parts working, they need to be combined in order to turn an image of a chessboard into a list of chess pieces and their coordinate locations. To do this, an image is first run through the chessboard detection and perspective transformation. The grid of chess squares is then found with the grid detection. Finally, the chess pieces themselves are found. To determine where each piece is on the grid, the bottom center point (the point centered between the bottom left and bottom right points) was used as the piece's location on the original image. This was because when looked at from an angle, the tops of the pieces often overlapped or fell into other squares on the grid. By choosing the bottom of the piece, the piece's location will always be directly on the square the piece is sitting on in reality.

However, the point selected for the chess piece will be located in the coordinate system of the original image – but the grid of squares is defined in the transformed image. To get around this, the coordinate of the piece can also be transformed using the same transformation matrix onto the transformed image. Since only a single point rather than an image is being transformed, the OpenCV method cv2.`transform([pts], matrix)` can be used to find the transformed point. From here it is relatively trivial to compare the piece's location to the grid to determine the location on the board. Repeating for each detected piece, and the system can return a list of pieces and which of the 64 squares they reside on.

VII. TESTING

To ensure that each stage was working correctly, it was decided to simply test it on testing images, which could then be examined by hand to check for accuracy. Roughly 20 images were used for this, although not all will be shown. The testing was done independently on two parts: the grid detection, and the chess piece detection.

A. BOARD AND GRID TESTING

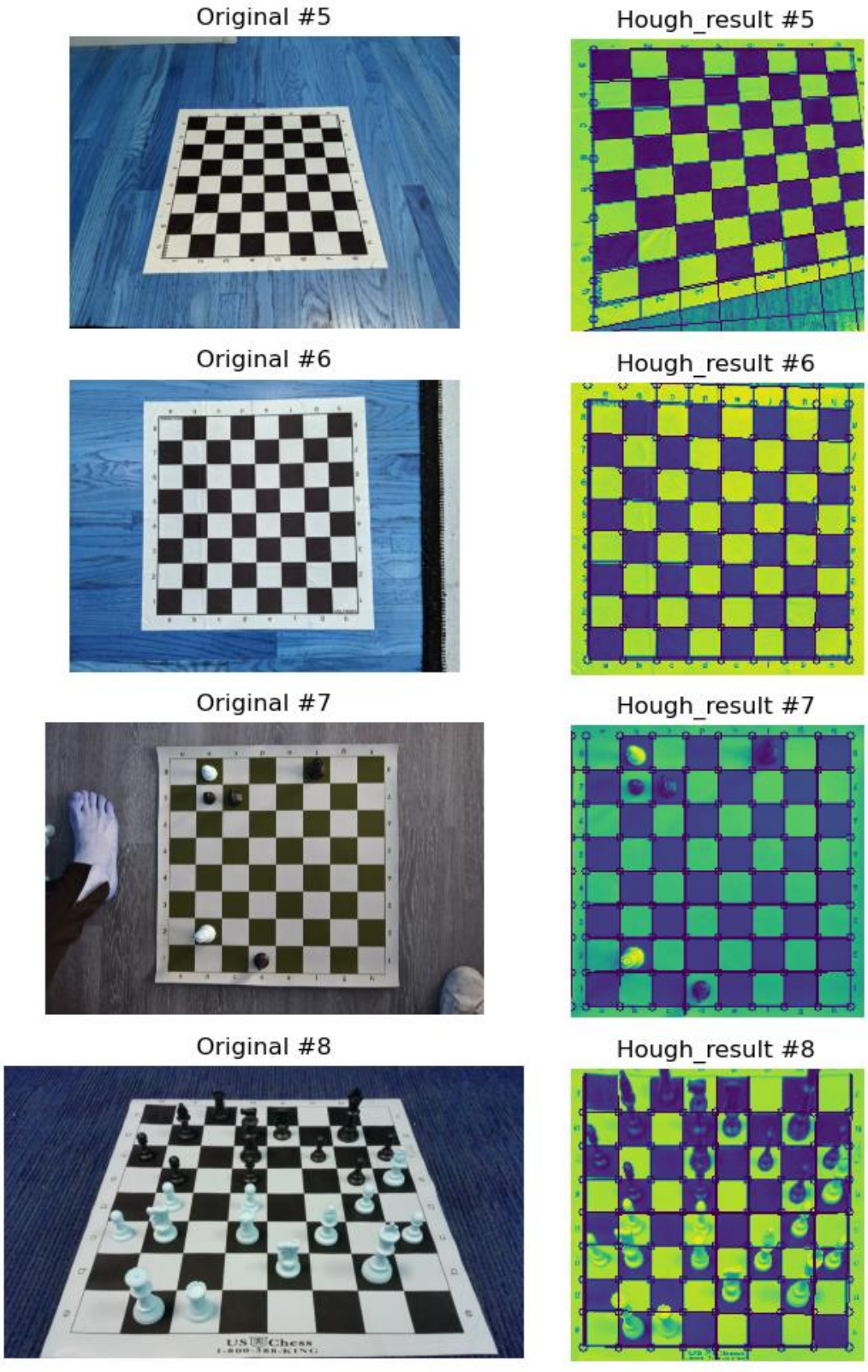
The goal of the 2 parts related to correcting the board and grid was to reliably determine the squares of the chessboard. Since this is being calculated using the vertices of each square, success was measured in the number of correct vertices in the output, as well as on the perspective transform correctly ensuring everything is square.

Figure XVI on the next page shows a sample of these test images. The coloring is a little off – this is just caused by the red and blue being switched during graphing (the image was graphed as BGR instead of RGB). Overall, the chessboard was detected in most images. Images where the chessboard was further away did not perform as well – in image number 5 the light part of the background is mistaken for part of the board, causing the perspective transform to be incorrect, which causes the grid to also be wrong. In most images such as images 6 and 7, extra points that were not part of the board were originally detected as part of the board, but those points can be filtered out as they are not equidistant from their neighbors. However, in some images such as image 8, points in the grid were not detected. Despite only missing a few points, this causes the entire grid to be off – so is just as wrong as image 5. Table XI shows the overall performance of the grid detection. All the test images are included in Appendix B.

TABLE XI
PERFORMANCE OF GRID DETECTION

Test Images	Labelled All Vertices	Incorrect Transform	Missing Vertices
19	13	6	6

FIGURE XVI
TESTING GRID DETECTION



B. CHESS PIECE TESTING

The goal of chess piece detection was to accurately find every piece in the image, as well as the correct location for the image. While testing this stage, success was measured based off how many pieces in the image were correctly labelled, compared to how many were present in the image. The same images used in testing the grid detection were used here for consistency, but vertical images were thrown out as the piece detection did not work at all from above – this was expected as there are not any defining features to chess pieces when looked at from the top.

The dataset originally used to train the chess piece detection program also included a training set. However, the images in the image set were very consistent, and performance was very high as a result (well above 98%). For this reason, the chess piece detection was tested with the more varied set from before.

Images with many chess pieces performed significantly worse than ones with fewer pieces. The Queen and black Knight were the worst performing pieces. Table XII summarizes the performance of the chess piece detection as a whole, as well as for each piece type. For each piece, the number of times it was correctly detected and labelled in the correct location is counted through all test images, as well as the number of times it is detected when it is not present (the number of false positives). The number of times each piece occurs throughout all images is included for reference – the pawns are overrepresented due to their prevalence in chess (there are 8 pawns on side, out of 16 pieces each at the start of a game). The performance over the test set is relatively poor, however for specific images at the right angle the results are higher. This means this detection form is not as consistent as the grid detection. The actual test images are shown in Appendix C, but a single test image is also shown in Figure XVII on the next page.

TABLE XII

PERFORMANCE OF PIECE DETECTION

Piece Type	Number in Test Images	Number Correctly Detected in Test Images	Percent Correct	Number of False Positives
All Pieces	104	84	81	11
All Black Pieces	55	38	69	9
All White Pieces	49	46	94	2
Black Pawn	28	25	89	1
White Pawn	23	23	100	1
Black Rook	7	5	71	1
White Rook	6	5	83	0
Black Knight	8	3	38	3
White Knight	6	6	100	1
Black Bishop	5	3	60	1
White Bishop	7	6	86	0
Black King	5	2	40	1
White King	4	4	100	0
Black Queen	2	0	0	2
White Queen	3	2	67	0

It is also very clear from the testing that the black chess pieces were not detected nearly as well as the white ones – while well over 90% of white pieces were correctly detected, only 69% of black pieces were found. This is likely caused by the black pieces being harder to see and blurring into the background squares due to their low intensity. The difference between the two is most profound for the more complicated pieces, such as the Knight and King – showing that the distinguishing features are being lost for black pieces. The black pawns still performed nearly as well as the white pawns, at 89%, likely due to a combination of their simple shape and prevalence in the training image set. Pieces that showed up less frequently in the training image set, due to being less frequent in chess games in general, performed worse than more common pawns. This suggests a model trained on more images could be considerably more accurate. More complicated pieces such as the Knight and Queen also performed worse, which makes sense.

FIGURE XVII
TESTING PIECE DETECTION

In Figure XVI, the angle is close to ideal – almost all the pieces are detected, even the knights. However, one of the black knights is still not detected and there are none of the low accuracy black kings or queens, so this image does not properly represent the failings of the chess piece detection model.

VIII. CONCLUSION & RESULTS

While this project is not perfectly accurate, it displays the potential to be adjusted to consistently determine the locations of chess pieces on a chessboard. The methods to do so run quickly enough to be used to record a chess game in real time. Further, both the chessboard detection and the chess piece detection work decently enough using images of different boards, with different backgrounds and camera angles. However, the camera does need to be aimed at the board from within a few inches away, and must look at the board at a shallow angle from the side.

There are a few ways to improve this project. The first would be to implement the GUI and save state functionality needed for the chess piece and chessboard detection to be used to record an actual game of chess with ease – currently this project is only working with single images, though it runs fast enough that images could be sampled from a video feed to implement for the full game. If the program was keeping track of the state of a chess game over multiple moves, it could be made to be much more accurate by assuming all moves are legal (within the bounds of the game). Since any detected chess piece not fitting the game would be known to be false, the next highest probability prediction could be used instead. This would likely increase the accuracy to near 100%. Alternatively, the program could be used to tell the player if they made an illegal move – but this would require much more accuracy.

Another big method would be to improve detection accuracy for chess piece detection. It is likely that a different object detection model than the YOLO model that was implemented would be able to do so with more accuracy – although there might be a trade-off with speed. Feature-based detection methods, especially, might be effective due to the noticeable features in each piece. For example, SIFT would likely perform quite well (there is actually a GitHub repo that does this with chess pieces, which works fairly accurately). Sticking with the YOLO model, a larger and more diverse set of images would certainly improve performance, possibly by quite a bit.

Finally, the grid or chessboard detection could be improved. The current method of finding the lines of the chessboard through Hough lines and then deriving the vertices of the squares has many steps – and as a result, many points where failure could occur. There is definitely a less convoluted way to reach the same conclusion – likely by either detecting the squares directly, using their solid colors, or by trying to detect the corners of the squares directly. Even more simply, the corners of the squares could be bypassed if the inner corners of the chessboard were to be found, assuming the perspective transform works properly. Since the squares are all the same size, if the inner corners are found, the chessboard size could just be divided into 64 equal segments to get each of the squares. This was not done for two reasons. The first is that this would make the grid detection fail completely if any error occurs in the perspective transformation. The second is that only the outer corners of the chessboard were found, and due to the surrounding whitespace on the board, it was difficult to reliably find the corners of the chessboard.

This project allows chess pieces to be found on a chessboard with decent accuracy when the camera is set up correctly. One of the things that was learned from this project is that the accuracy of most computer vision algorithms decreases very drastically with changes to the angle of the camera, even with perspective transformation. There was also a lot of difficulty with using different chessboards. Had a single chessboard and exact camera angle been chosen at the beginning of this project, the program would probably function better. The original idea for the range of angles and boards that would work was far too ambitious.

Overall, this project fulfills the goal of detecting chess pieces on a chessboard. While it does not do so with the predicted accuracy and is not as resistant to changes in image layout as was hoped, it does work well enough when images are taken properly. It does not ever approach 100% accuracy, but there is plenty of room for improvement – this project shows a possible way to determine the locations of pieces on a chessboard, and this method can be improved to be accurate enough to be useful.

IX. BIBLIOGRAPHY

[1] IEEE Board of Directors, "IEEE Code of Ethics - IEEE." IEEE. Jun. 2021. https://www.ieee.org/about/corporate/governance/p7-8.html [Accessed Mar. 15, 2021].

[2] "NSPE Code of Ethics." NSPE. https://www.nspe.org/resources/ethics/code-ethics [Accessed Mar. 15, 2021].

[3] D. Barik and M. Mondal, "Object identification for computer vision using image segmentation," *2010 2nd International Conference on Education Technology and Computer*, Shanghai, 2010, pp. V2-170-V2-172, doi: 10.1109/ICETC.2010.5529412. Available: https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=5529412&isnumber=5529317 [Accessed 1/29/2021].

[4] P. F. Felzenszwalb and R. Zabih, "Dynamic Programming and Graph Algorithms in Computer Vision," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 33, no. 4, pp. 721-740, April 2011, doi: 10.1109/TPAMI.2010.135. Available: https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=5518769&isnumber=5715914 [Accessed 1/29/2021].

[5] Szeliski, Richard. *Computer Vision: Algorithms and Applications*, 1st Ed. 2011. ed. London: Springer, 2011.

[6] Ana Tomé, Martijn Kuipers, Tiago Pinheiro, Mario Nunes, Teresa Heitor, "Space–use Analysis through Computer Vision." *Automation in Construction*, Vol. 57 pp 80-97, 2020. Available: https://cpslo-primo.hosted.exlibrisgroup.com/permalink/f/prbv08/TN_cdi_gale_infotracacademiconefile_A520883827 [Accessed 1/29/2021].

[7] Ashish Kumar Gupta, Ayan Seal, Mukesh Prasad, Pritee Khanna, "Salient Object Detection Techniques in Computer Vision—A Survey," *Entropy*, Vol. 22, No. 10, 2020. Available: https://cpslo-primo.hosted.exlibrisgroup.com/permalink/f/prbv08/TN_cdi_doaj_primary_oai_doaj_org_article_fcfd3872624d4f509b8767126972e977 [Accessed 1/29/2021].

[8] E. Dandil and K. K. Çevik, "Computer Vision Based Distance Measurement System Using Stereo Camera View," *2019 3rd International Symposium on Multidisciplinary Studies and Innovative Technologies* (ISMSIT), Ankara, Turkey, 2019, pp. 1-4, doi: 10.1109/ISMSIT.2019.8932817. Available: https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=8932817&isnumber=8932719 [Accessed 1/29/2021].

[9] F. Elian, F. I. Hariadi, and M. I. Arsyad, "Implementation of computer vision algorithms for position correction of chip-mounter machine," *2017 International Symposium on Electronics and Smart Devices* (ISESD), Yogyakarta, 2017, pp. 90-94, doi: 10.1109/ISESD.2017.8253311. Available: https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=8253311&isnumber=8253290 [Accessed 1/29/2021].

[10] T. Romih, M. Malajner, D. Gleich and P. Planinsic, "Educational computer vision system for object detection and tracking," 2008 50th International Symposium ELMAR, Zadar, 2008, pp. 369-372. Available: https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=4747520&isnumber=4747505 [Accessed 1/29/2021].

[11] W. Gui and T. Jun, "Chinese chess recognition algorithm based on computer vision," *The 26th Chinese Control and Decision Conference* (2014 CCDC), Changsha, 2014, pp. 3375-3379, doi: 10.1109/CCDC.2014.6852759. Available: https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=6852759&isnumber=6852105 [Accessed 1/29/2021].

[12] E. Nishani and B. Çiço, "Computer vision approaches based on deep learning and neural networks: Deep neural networks for video analysis of human pose estimation," *2017 6th Mediterranean Conference on Embedded Computing* (MECO), Bar, 2017, pp. 1-4. Available: https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=7977207&isnumber=7977112 [Accessed 1/29/2021].

[13] Seeed Technology Co., "OV2649 Camera," OV2649 Datasheet, Date not available.

[14] OpenCV, "OpenCV Documentation," https://docs.opencv.org/4.5.4/

[15] J. Redmon, S. Divvala, R. Girshick and A. Farhadi, "You Only Look Once: Unified, Real-Time Object Detection," *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, Las Vegas, NV, USA, 2016 pp. 779-788. https://doi.ieeecomputersociety.org/10.1109/CVPR.2016.91

[16] Bochkovskiy, Alexey, et al. "YOLOv4: Optimal Speed and Accuracy of Object Detection". 2020. https://csu-calpoly.primo.exlibrisgroup.com/permalink/01CALS_PSU/hls1s0/cdi_arxiv_primary_2004_10934

[17] J. Redmond, A. Farhaldi, "YOLOv3: An Incremental Improvement," YOLOv3.pdf (pjreddie.com)

[18] S. Liu, L. Qi, H. Qin, J. Shi and J. Jia, "Path Aggregation Network for Instance Segmentation," 2018 IEEE/CVF Conference on Computer Vision and Pattern Recognition, 2018, pp. 8759-8768, doi: 10.1109/CVPR.2018.00913.

Appendix A: Senior Project Analysis

1. Summary of Functional Requirements

The project will be capable of using a camera to record a chess game and determine the locations of each piece on the board at the end of each turn. It will use this to record a digital version of a game of chess. This means that a single camera mounted facing a chess board will be able to determine which position each piece is at. One potential application is to easily make a recording of a game where each turn is clearly marked digitally.

2. Primary Constraints

Without a way to move the camera (since such a system would be overcomplicated), the whole board can only be observed with one angle, and parts of pieces and the board are obstructed. The most significant challenge will be to determine which piece is which from that data. Other limiting factors include the impracticality of only working with the camera in one exact position or only capable of recording for one specific chess set. This poses a challenge in that the project must be able to adjust to minor changes. Different chess sets use different appearances for pieces and different environments could change lighting, making detection further difficult.

3. Economic

This project could save human time, and work, by converting a game to a format ready for a computer without human intervention. It would have relatively little effect on natural or manufactured capital beyond any increase in camera usage since the largest component needed is a camera, which are already produced. The primary cost throughout the project's lifecycle is the cost of the camera used, as well as the time needed to code the project. The initial estimate for the cost of all parts is $60. This project likely will not earn much beyond that from anyone interested in recording a chess game but uninterested in playing a virtual game. The product should exist for some time, until wear on the camera causes it to stop working which would take years of use. The end product requires little to no maintenance. The primary operation cost is the cost of electricity. The estimated time requirement for this project is about 30 weeks, or 3 quarters, to complete. Afterwards, the software for it can be released for usage.

4. If Manufactured on a Commercial Basis:

If this device is manufactured, I predict sales around 3,000 units per year. I think the cost of manufacturing can brought down to about $35 with the cheapest to produce camera with the required quality. If the markup on the purchase price is 15% then the total cost will be around 40 dollars. With $5 of profit per device sold, I would estimate around $15,000 in profit per year. The cost for the user to operate the device is mostly reliant on the power usage of the user's computer. With the average 200 W computer with 13.3 cents / kWh (the average in the United States), the device would likely cost around 2.6 cents per hour in electricity, a negligible amount to the average user.

5. Environmental

The manufacture of this device would require the production of a digital camera, which is similar to a computer in terms of waste, creating certain chemicals, metals and semiconductors, and during production. The camera stand and cable for the camera would likely contain plastic, which has its own impact since it does not break down easily and has byproducts involved in its production. The product

also indirectly impacts the environment by requiring the production of a chess set, which is generally made of plastic, and a computer. However, the product is unlikely to directly impact other species beyond the general impact of the waste produced. The last environmental impact is the small electrical usage of the device, since most forms of power generation have some impact on the environment.

6. Manufacturability

This product should not have any major problems with manufacturing. The largest issue would be creating the camera in a way that the user can easily position it properly, but since cameras are already manufactured this would be a small issue.

7. Sustainability

The main challenge with maintaining the system is just that the casing on the camera, the stand, and the cables cannot be broken, snapped, or cut. Without taking damage, they should continue to work. The other challenge is any updates needed for the software to continue to work on new or updated computers. The project negatively impacts the sustainable usage of resources by encouraging the production of cameras and computers which generally become waste after their lifecycle has ended or the user finds a replacement. A method to improve the project design is to incorporate the computing power with the camera, rather than relying on a separate computer, since this would keep the project self-contained and reduce its usage of power. However, this would increase the cost of production, and would make running the software more difficult on limited hardware.

8. Ethical

All the materials used in the production of the project should be ethically sourced, and the end product should contain as few nonrecyclable products as possible. Unethical use of the system is unlikely, other than to spy on another person's chess game. One of IEEE's codes of ethics is "to avoid injuring others, their property, reputation, or employment by false or malicious actions, rumors or any other verbal or physical abuses" [1]. This product follows IEEE's Code of Ethics by not putting anyone in danger, being entirely within the law, and treating all users equally. Another framework for ethics that this follows is the National Society of Professional Engineers' requirement to "Hold paramount the safety, health, and welfare of the public." This framework is satisfied in the same way.

9. Health and Safety

It is unlikely that this project will hurt someone during use. So long as all electrical components are properly insulated to avoid shock and the design of the camera stand makes it ineffective as a weapon, this project should not hurt the user. Manufacturing of the project should occur under good conditions where any manufacturers do not put health to the side to increase production.

10. Social and Political

This project could change the social aspects of chess by making it easier to share chess games with other people in an easy to view manner. Politically, it has little importance other than the legality of recording another person's game with it. The project impacts those who play chess on a physical chessboard and those who view chess games, the direct stakeholders. The indirect stakeholders would include those who invested in the device as well as stores that would sell the device. This device would benefit anyone who wishes to be able to easily record their chess games. It would also help anyone trying to view other

people’s games. These stakeholder’s benefit the most, but also pay the most as they must purchase the product. The other stakeholders benefit from the project’s success. Some stakeholders may not have the money to purchase this device, or the computer to support it, in which case they would benefit little compared to other stakeholders. Those who already play chess on a computer or who use incompatible boards would also benefit little.

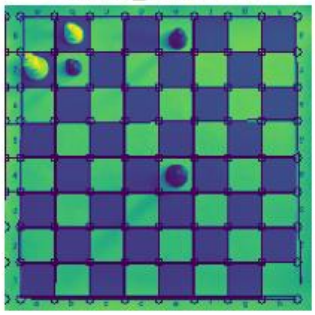
11. Development

During the planning for the project, I learned about various planning tools. This includes Gantt charts for keeping the project on track, cost estimates to keep it on budget, and block diagrams to get a good system overview of the project. I believe I will learn a lot about teaching computers to recognize objects through this project, as well as a thing or two about chess. I will also learn how to match recognized objects to recognized locations on the board. This will hopefully improve my understanding of computer vision based concepts.

Appendix B: Grid Detection Test Images

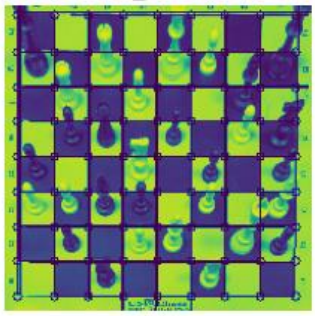
Original #1

Hough_result #1

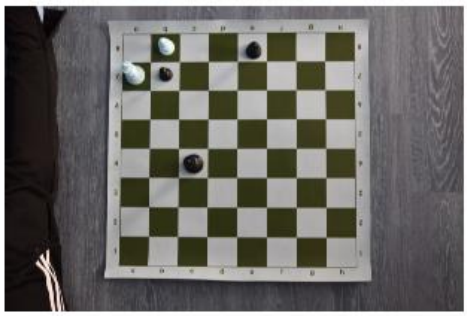


Original #2

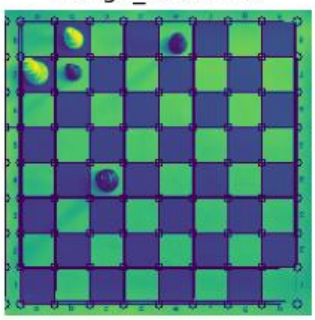
Hough_result #2



Original #3

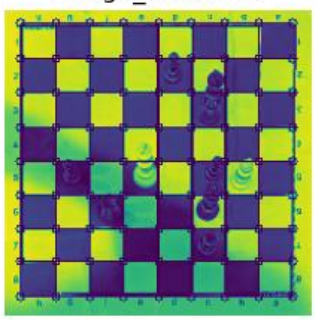


Hough_result #3



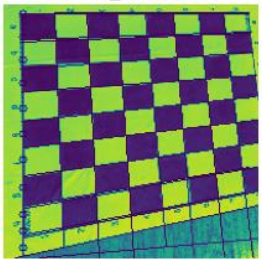
Original #4

Hough_result #4

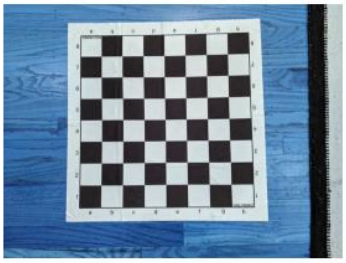


Original #5

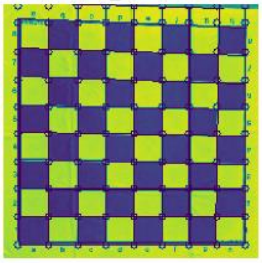
Hough_result #5



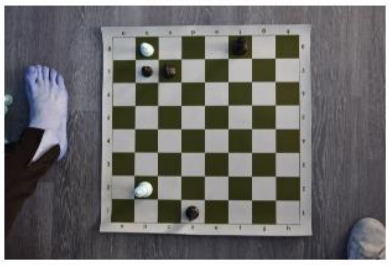
Original #6



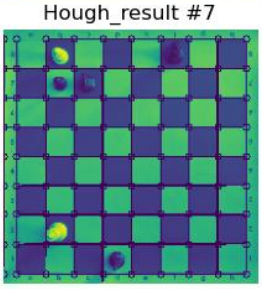
Hough_result #6



Original #7

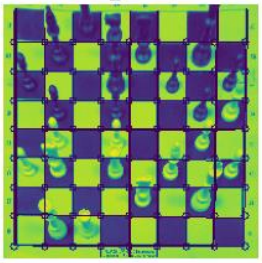


Hough_result #7



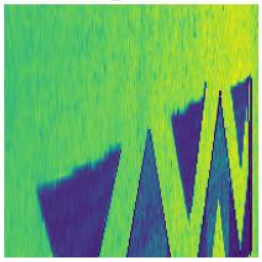
Original #8

Hough_result #8



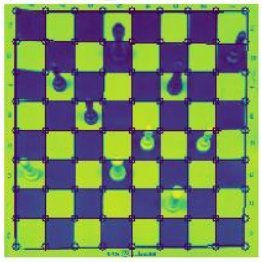
Original #9

Hough_result #9



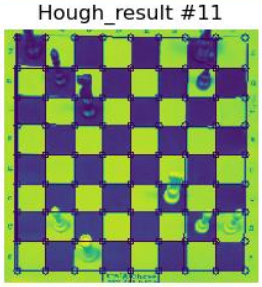
Original #10

Hough_result #10



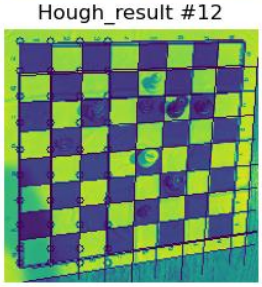
Original #11

Hough_result #11



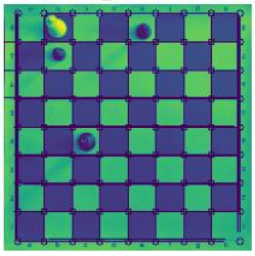
Original #12

Hough_result #12



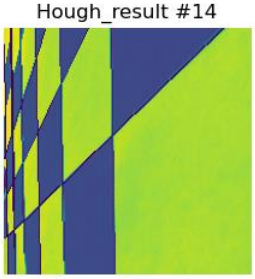
Original #13

Hough_result #13



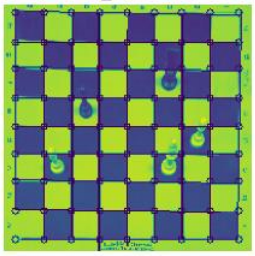
Original #14

Hough_result #14



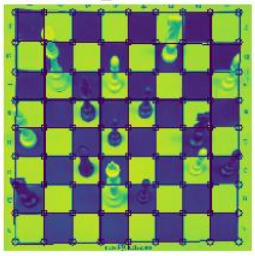
Original #15

Hough_result #15

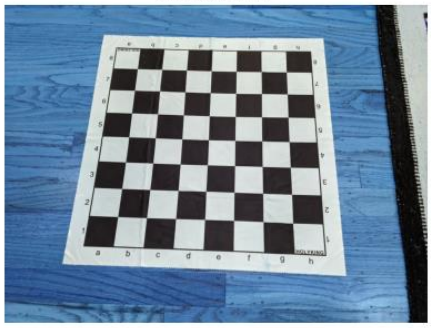


Original #16

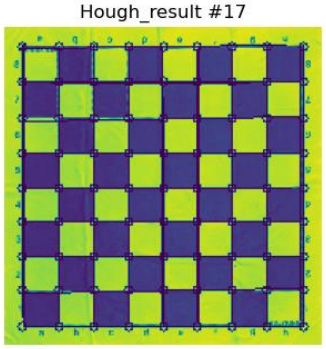
Hough_result #16



Original #17

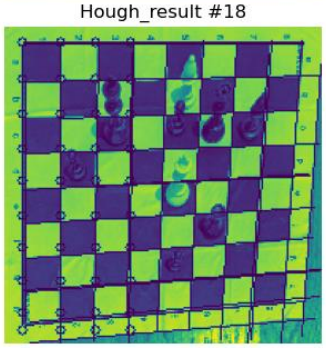


Hough_result #17

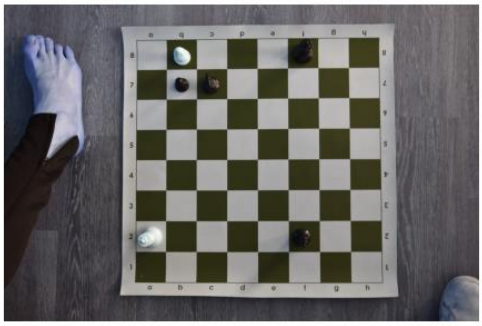


Original #18

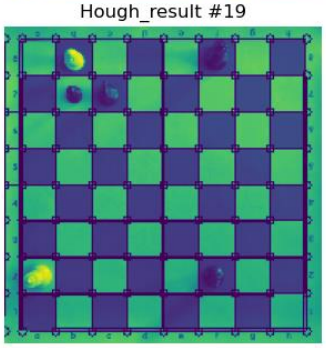
Hough_result #18



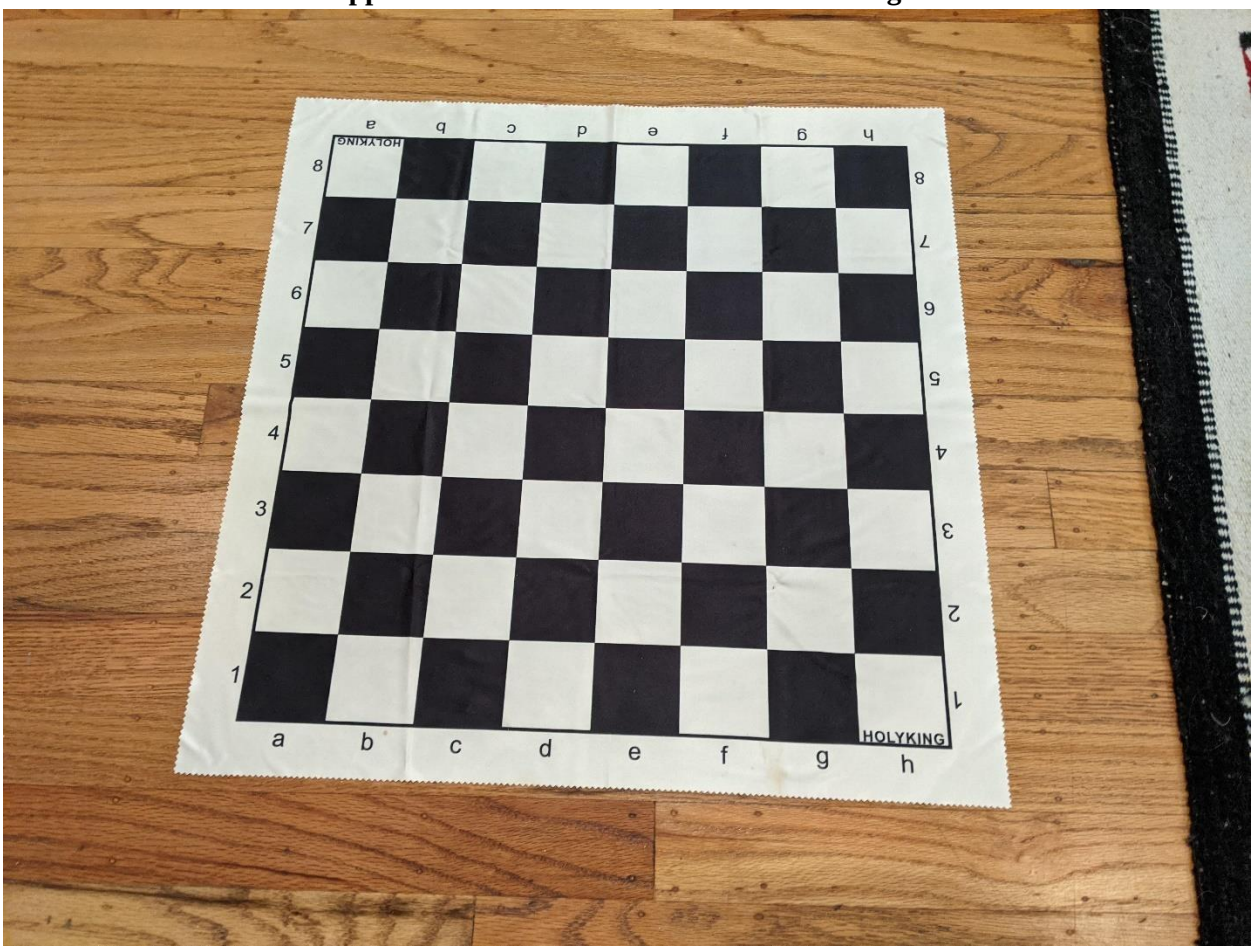
Original #19

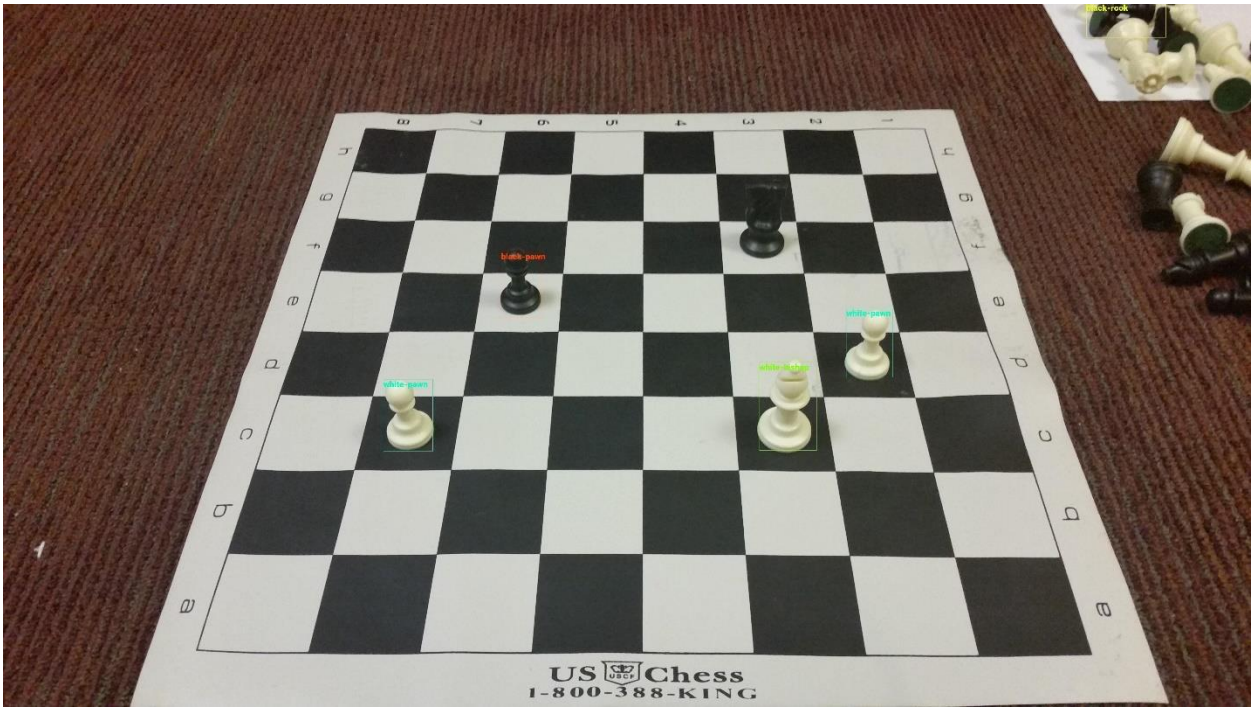


Hough_result #19



Appendix C: Chess Piece Detection Test Images



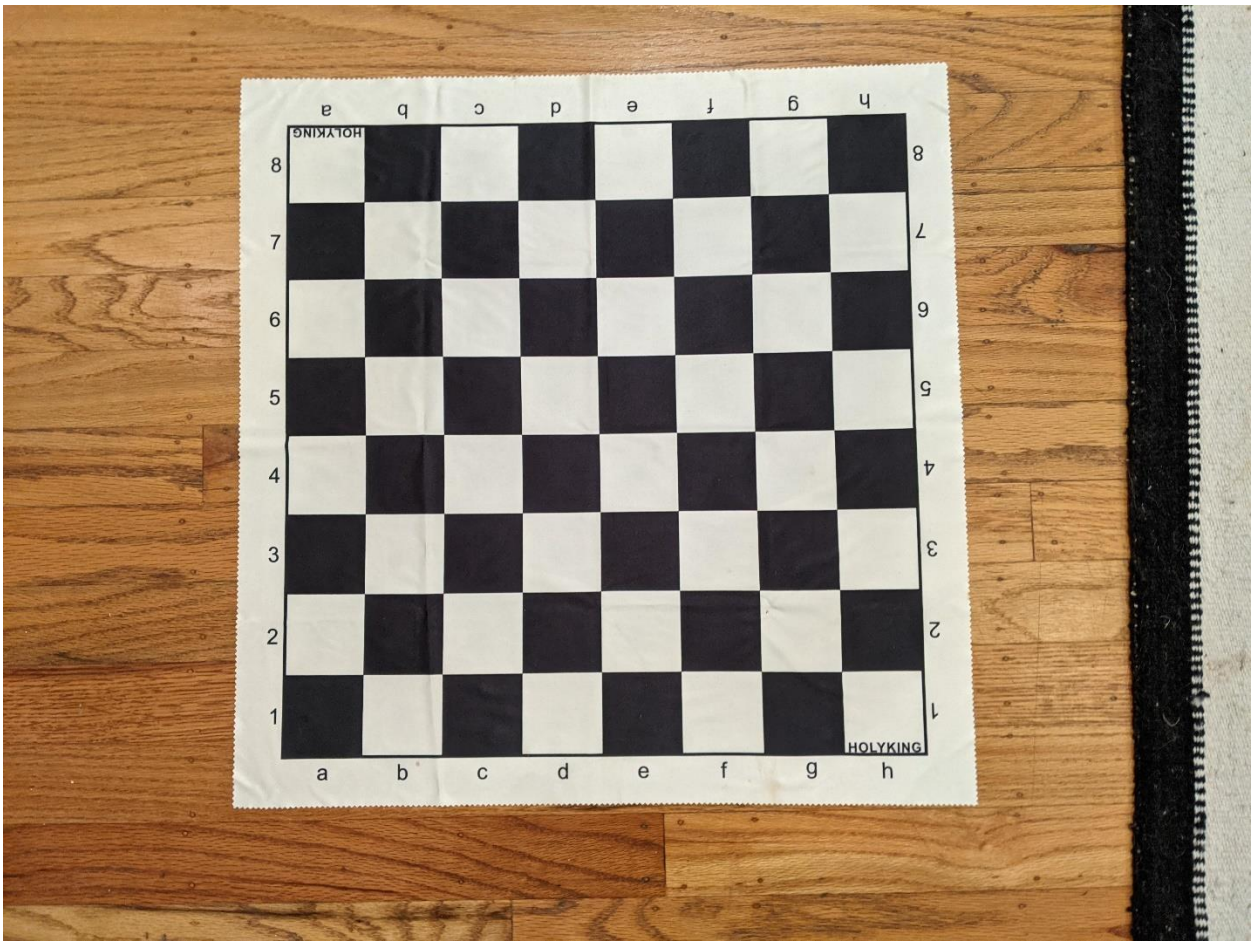

black-rook
black-pawn
white-pawn
white-knight
white-pawn
US Chess
1-800-388-KING


black-king
black-knight
black-pawn
black-pawn
white-knight
white-pawn
white-pawn
white-pawn
white-rook
US Chess
1-800-388-KING


black-pawn
black-rook
black-pawn
white-pawn
white-pawn
white-rook
white-pawn
US Chess


black-bishop
black-king
white-pawn
white-queen
black-pawn
black-bishop
white-king
black-pawn
black-rook
black-pawn
HOLYKING
HOLYKING


black-knight
black-queen
black-bishop
black-knight
black-knight
black-knight
black-pawn
black-pawn
black-pawn
black-pawn
black-pawn
white-bishop
white-pawn
white-pawn
white-pawn
white-pawn
white-knight
white-rook
white-bishop
white-pawn
white-king
white-knight
white-knight
white-rook
US Chess
1-800-388-KING


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


black-pawn
black-queen
white-queen
black-king
white-pawn
white-rook
white-pawn
black-pawn
black-rook
black-pawn
white-pawn
white-pawn
white-knight
black-pawn
black-bishop
white-pawn
white-pawn
white-king
white-bishop
black-pawn
white-pawn
black-pawn
white-knight
US Chess
1-800-388-KING

Appendix D: Related Code

There is a fair bit of code related to this project. Below is the code for running the perspective transform, Hough, and square vertices.

```
# from PIL.Image import new
import os
import cv2 as cv
import numpy as np
import matplotlib.pyplot as plt
import statistics
from tkinter import *
# from PIL import Image, ImageTk

# # img1 = cv.imread('board_images/5_1.jpg')
# img1 = cv.imread('board_images/1B2b3-Kp6-8-8-4k3-8-8-8.JPG')


def find_med_smallest_dist_between_pts(pts):

    all_smallest = []
    for pt in pts:
        smallest = 10000
        for pt2 in pts:
            if pt != pt2:
                d = dist(pt, pt2)
                if d < smallest:
                    smallest = d
        all_smallest.append(smallest)

    return statistics.median(all_smallest)

# Need to remove duplicate points
def dist(i,p): # finds distance between pts, kinda
    # return ((i[0] - p[0])**2 + (i[1] - p[1])**2)**0.5
    res = (abs(i[0]-p[0]) + abs(i[1]-p[1]))
    if (res == 0): return 100 # this is just to exclude duplicatepts
    return res

def remove_duplicates(list):
    # list = set(list)
    # return [p for p in list if all(dist(i,p) > 3 for i in list)]
    out = []
    l = len(list)
    for i in range(l):
        c = True
        if all(dist(list[i], p) > 6 for p in out):
            out.append(list[i])

    return out

def find_intersections(lines, warped_img):
```

```
    new_lines_v = []
    new_lines_h = []

    for line in lines:
        x1, y1, x2, y2 = line[0]
        if (x1 == x2):
            m = 10000
        else: m = abs((y2-y1)/(x2-x1))
        # print(x1, y1, x2, y2, m)
        if (m < .5): # Horizontal
            new_lines_h.append((y1+y2)/2)
        elif (m>35): # Vertical
            new_lines_v.append((x1+x2)/2)
        else:
            cv.line(warped_img, (x1,y1), (x2,y2), (255,100,50), 1)

    pts = []
    for hline in new_lines_h:
        for vline in new_lines_v:
            pts.append([hline, vline])

    return pts, warped_img

def hough_pts(img1):

    width = int(img1.shape[1] * 0.2)
    # print(width)
    height = int(img1.shape[0] * 0.2)
    # print(height)
    img1 = cv.resize(img1, (width, height))


    image = cv.cvtColor(img1, cv.COLOR_RGB2GRAY)
    d, img = cv.threshold(image, 140, 255, cv.THRESH_BINARY)

    num_labels, labels, stats, centroids=
cv.connectedComponentsWithStats(img, 8, cv.CV_32S)

    # Skip label 0, as it is (usually) the background - start at one to find
largest object (the board)
    m=1
    for i in range(2,num_labels-1):
        if stats[m][cv.CC_STAT_AREA] < stats[i][cv.CC_STAT_AREA]:
            m = i


    contours, hier = cv.findContours(img, cv.RETR_LIST,
cv.CHAIN_APPROX_SIMPLE)
    a=contours[0]
    for contour in contours:
        if cv.contourArea(contour) > cv.contourArea(a):
            a = contour
```

```
    #Find corners 2
    br = 0 #bottom right is max sum
    tl = 1000000 # top left is min sum
    tr = 0 # top right is smallest difference
    bl = 1000000 # bottom left is largest difference
    brp = [0,0]
    trp = [0,0]
    tlp = [0,0]
    trp = [0,0]
    for point in a:
        # print(point)
        sum = point[0][0] + point[0][1]
        diff = point[0][0] - point[0][1]
        if sum > br:
            br = sum
            brp = point[0]
        if sum < tl:
            tl = sum
            tlp = point[0]
        if diff > tr:
            tr = diff
            trp = point[0]
        if diff < bl:
            bl = diff
            blp = point[0]

    rect = np.array(((tlp[0]+5, tlp[1]+5), (trp[0]-5, trp[1]+5), (brp[0]-5,
brp[1]-5), (blp[0]+5, blp[1]-5)), dtype="float32") # The plus/minus 5 are to
exclude the direct borders
    # rect = order_points(np.array([tlp, trp, blp, brp], dtype="float32"))
    width = 200
    height = 200
    dst = np.array([[0,0], [width-1,0], [width-1,height-1], [0,height-
1]],dtype="float32")
    M = cv.getPerspectiveTransform(rect,dst)
    warped_img = cv.warpPerspective(image, M, (width, height))

    edges = cv.Canny(warped_img, 100, 150)

    # lines = cv.HoughLinesP(edges, 1, np.pi/180, 80, minLineLength=40,
maxLineGap=70)
    lines = cv.HoughLinesP(edges, 1, np.pi/180, 60, minLineLength=40,
maxLineGap=70)
    # lines = find_chessboard_lines(lines)
    pts, warped_img = find_intersections(lines, warped_img)
    pts = remove_duplicates(pts)

    # need to remove pts that do not fit grid somehow...

    # side_length = find_med_smallest_dist_between_pts(pts)
```

```
    # now that I have the side length, I can find neighbors if they are
roughly that distance up / right / left / down
    # ex the right neighbor should be around x + side +/- error,
    # also can remove points if they have no neighbors that are roughly the
correct distances
    # once I have neighbors I can follow the grid to get squares (somehow!?)



    for pt in pts:
        # print(pt)
        cv.circle(warped_img, (int(pt[1]), int(pt[0])), 3, (0,255,0), 1)

    for line in lines:
        # x1, y1, x2, y2, l, p, n = line
        x1, y1, x2, y2 = line[0]

        # print(line)
        cv.line(warped_img, (x1,y1), (x2,y2), (0,0,255), 1)

    return warped_img


# testing on multiple images:

def load_images():
    images = []
    cnt = 1
    for filename in os.listdir('board_images/'):
        img = cv.imread(os.path.join('board_images/',filename))
        print("loading image: ", cnt, " file: ", filename)
        cnt+=1
        if img is not None:
            images.append(img)
    return images


def hough_images(images):
    cnt = 1
    warped_imgs = []
    for image in images:
        print("warping image ", cnt)
        cnt+=1
        # try:
        warped = hough_pts(image)
        warped_imgs.append(warped)
        # except (Exception):
            # print("error warping image ", cnt)

    return warped_imgs

images = load_images()
warped_images = hough_images(images=images)
```

```
# print("warped", len(warped_images))
# print("normal", len(images))


def plot_figs(images, hough_images):

    split_images = [images[i:i+4] for i in range(0, len(images), 4)]
    split_hough = [hough_images[i:i+4] for i in range(0, len(hough_images),
4)]

    # cnt = 1
    fig_num = 1
    c = 0
    for i in range(len(split_hough)):
        fig = plt.figure()
        plt.tight_layout()
        cnt = 1
        for j in range(len(split_hough[i])):
            fig.add_subplot(len(split_images[i]), 2, cnt)
            plt.imshow(split_images[i][j])
            plt.axis('off')
            plt.title("Original #" + str(fig_num))

            fig.add_subplot(len(split_hough[i]), 2, cnt+1)
            plt.imshow(split_hough[i][j])
            plt.axis('off')
            plt.title("Hough_result #" + str(fig_num))

            cnt+=2
            fig_num+=1

        c+=1
        # plt.savefig('~/Desktop/figures_for_chess_report/hough_example' +
str(c) + '.png')



plot_figs(images, warped_images)
plt.show()

# fig = plt.figure()
# l = len(warped_images)
# for i in range(len(warped_images)):

#     fig.add_subplot(l, 2, 2*i+1)
#     plt.imshow(images[i])
#     plt.axis('off')
#     plt.title("Original #" + str(i+1))

#     fig.add_subplot(l, 2, 2*i+2)
#     plt.imshow(warped_images[i])
#     plt.axis('off')
```

```
#     plt.title("Hough_result #" + str(i+1))

# plt.show()









# warped_img = hough_pts(img1)

# plt.imshow(warped_img)
# plt.axis("off")
# plt.title("Hough Lines")

# plt.show()

# win = Tk()
# win.geometry("1280x900")

# label = Label(win)
# label.grid(row=0, column=0)
# cap = cv.VideoCapture(0)


# def show_camera():

#     cv2image = cv.cvtColor(cap.read()[1], cv.COLOR_BGR2RGB)

#     cv2image = hough_pts(cv2image)


#     img = Image.fromarray(cv2image)

#     imgtk = ImageTk.PhotoImage(image=img)
#     label.imgtk = imgtk
#     label.configure(image=imgtk)

#     label.after(20, show_camera)

# show_camera()
# win.mainloop()
```