AI Chess Opponent Final Report

Karim Zaher

1. Summary

For my project, I will be creating an AI chess opponent. Although plenty of AI chess opponents exist in software nowadays, there are very few that are able to play on a physical chess board. My project focuses on creating an interactive chess board that allows the AI opponent to call out its moves through lights that indicate where the user should move the AI's chess piece. This system is especially useful in instances where a user cannot interact with other players, either due to a pandemic or due to a lack of internet.

2. Introduction/background

When I was younger, I loved playing chess, especially on a physical chessboard. I had a chess application on my phone and computer, however, it did not feel as fun as playing with physical chess pieces. Physical chess boards make the game feel much more intense and immersive than a simulated chess game. The only problem with a physical chessboard is that when there is no other individual that is available to play a game of chess, one cannot play chess on his or her own. If there was a way to play chess with a chess AI on a physical chessboard, people would still be able to feel immersed in the game even when there is no human opponent to play against.

In today's Covid-19 era, people are unable to be in the same room together. So when an individual wants to practice playing chess on a chessboard with another player, they are unable to. With an AI chess opponent, the individual does not have to worry about making contact with another person and risk getting sick in the process.

3. System Design

Everything is stored within a large storage compartment that is located underneath the chessboard. This storage compartment is typically used to store chess pieces, however, using it as a storage compartment for my hardware allows the end product to look clean. It also removes the need to use multiple breadboards or extension wires to get things connected to the raspberry pi if the part is located far away from the pi. To prevent any overheating from within the box, I created a massive hole on the side of the box to allow for continuous airflow. I also created another hold to allow wires to run through the compartment without getting in the way of the product. The black chess pieces were also covered in ripped up pieces of labels. This was done

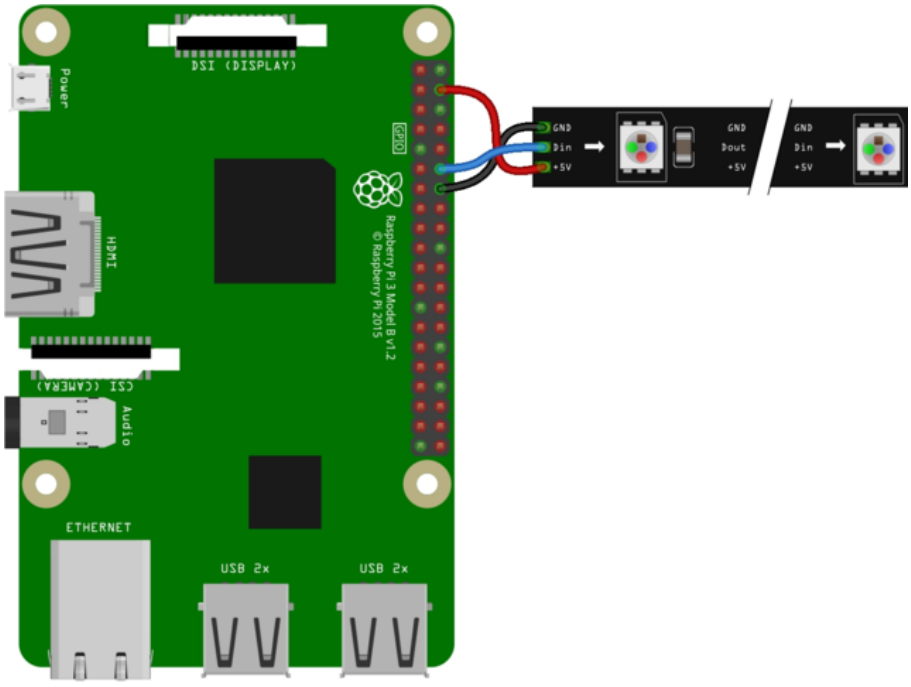
because of the Pi Camera's low resolution which prevented the camera from being able to differentiate between black (chess pieces) and brown (grid color) pixels.

3.1 Hardware

Excluding the Raspberry Pi, this section covers the hardware components that are used in this project.

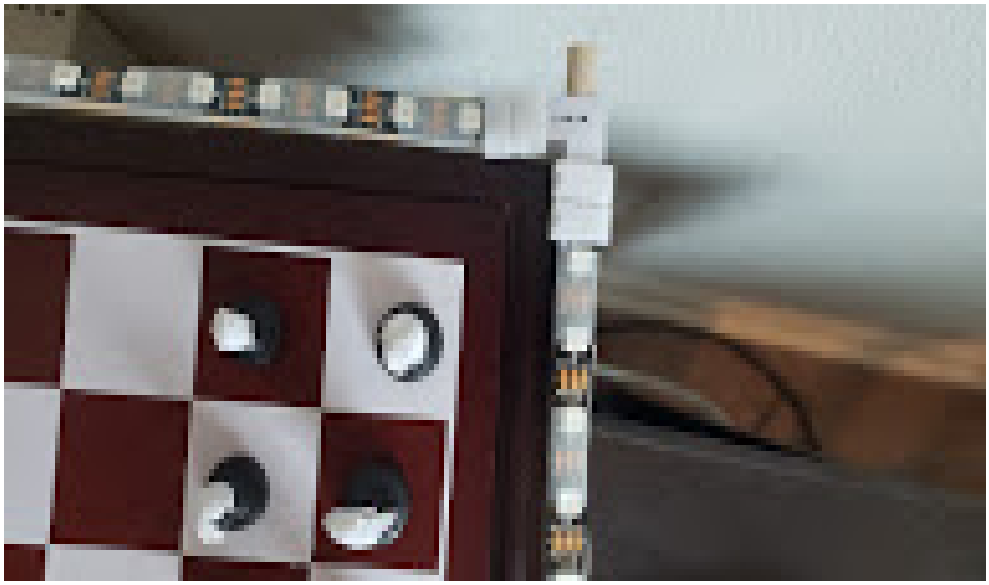
3.1.1 WS2812B LED Strip

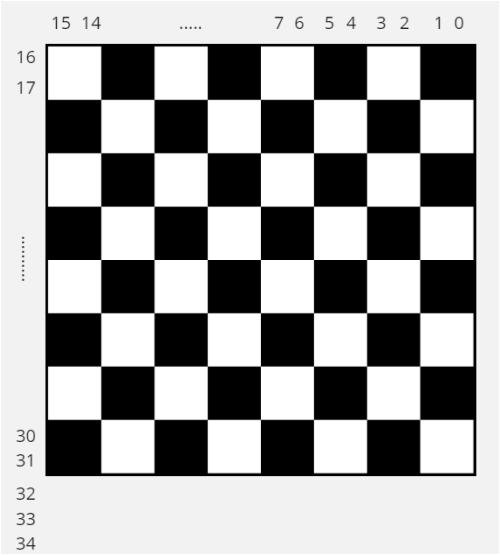
Initially, I planned on using multiple RGB LED light bulbs to indicate what AI chess piece should be moved and the position of its new location. However, I felt as though the number of wires that I would have to use was terrifying in both the sense of debugging and plain aesthetic. After submitting my one-pager, my project was approved, however, I was told that LED lightbulbs have been a tough challenge before. Seeing as to how massive this project was, I realized that lightbulbs should have to be the least of my concerns, so I went out to find a better alternative. This is when I found out about WS2812B RGB LED strips. These LED strips are not only wired through a PCB, each LED is individually programmable.



While looking at LED strips, I realized that the shortest length I could find was much longer than the chess board itself. This was an issue, especially in terms of design. In addition, only one WS2812B LED strip can be connected to the raspberry pi at a time. I was able to find a WS2812B LED strip that can be cut and reattached at

certain parts of the strip. However, they needed to be reconnected somehow. After more research, I found connectors. I purchased L-connectors, 45-degree connectors, so that the cut LED strip could easily curve around the edge of the chess board.





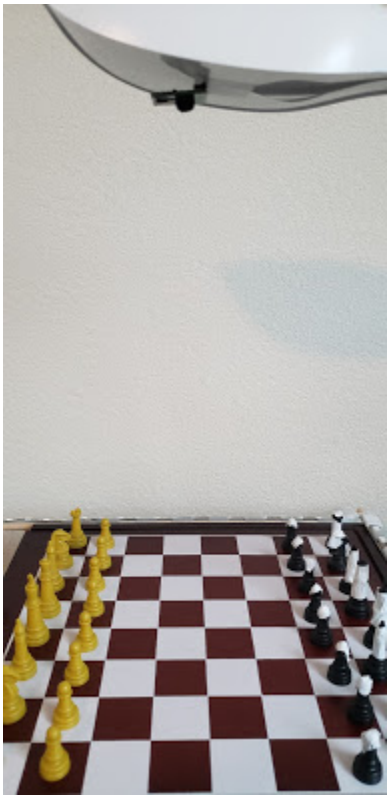
In total, there are 35 LEDs used in my project. The first 32 are used to indicate where the AI opponent's chess piece should go. This is done by using 4 LEDs per grid position. For example, the position (0,0) uses 2 LEDs 14 and 15 on the X-axis and 16 and 17 on the Y-axis. LEDs that light up red shows what piece needs to be moved while LEDs that light up green show where the piece must be moved. The formulas below show how the LED indices are calculated:

- redX = 15 - currentPos_x * 2
- redY = 16 + currentPos_y * 2
- greenX = 14 - newPos_x * 2
- greenY = 17 + newPos_y * 2

The remaining 3 LEDs (32, 33, and 34) are feedback LEDs. They let the user know what is going on. Their significance is further explained in Section 3.1.3.

3.1.2 Pi Camera

To keep track of the chess pieces, I use a raspberry pi camera module. The camera is attached to an old lamp with a flexible neck. That way, not only do I save on any additional spending on a stand, I also have the ability to quickly and easily adjust the camera's position until I get an optimal view of the chess board.

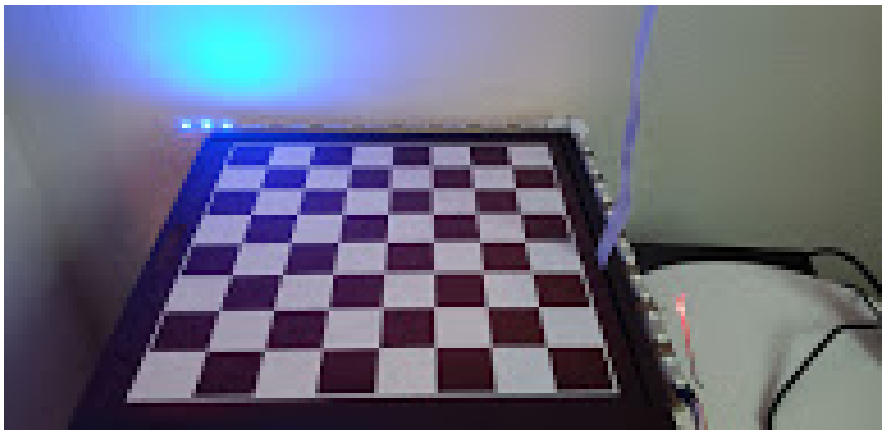


3.1.3 Button, Buzzer, and 3 Feedback LEDs

This product only takes 1 user-controlled input which comes from the button. The button is used to let the Raspberry Pi know three different things depending on what is going on:

1. If it is the start of the game and all three LEDs are on, pressing the button lets the Pi know that the chess board is completely cleared out.
2. If only the last LED on the light strip is on, pressing the button lets the Pi know that all pieces have been placed on the chess board.
3. If only the middle LED is on, which indicates that it is the user's turn, pressing the button lets the Pi know that the user has moved their piece into a new position.
4. If it is the AI opponent's turn, pressing the button lets the Pi know that the opponent's piece has been moved to its designated location.

The buzzer lets the user know that the Pi has moved onto a different stage.



All 3 feedback LEDs enabled

3.2 Software

Libraries used:

- Pygame: For simulation and board set-up
- gpiozero: For buzzer and button functionality
- OpenCV: For computer vision and image processing

3.2.1 Chess Board Simulation

Using an 8x8 matrix, each piece on the board is kept track of using the Piece class which keeps track of the piece's color, position, and whether it's the piece's first move or not (useful for pawns since they move differently on their first move). Then 6 subclasses are used to keep a log of the possible moves that the piece can make.

These subclasses and their possible moveset are:

1. Pawn class:

    a. Hardcoded it so that the pawn can only move up to 2 steps right at the start, then 1 step after that. If there is a piece in its way, no matter the color, it cannot move forward. If there is a black piece in its diagonals, it can take the piece's position.
2. Rook class
    a. For the rook's algorithm, it checks to see whether there were any pieces blocking its x path or its y path. If there was a piece with its opposite color in its way, it could move to that piece's position and capture the piece. If it was a similar color to it instead, it could only move to the position right before that piece's position.
3. Bishop class
    a. The bishop functioned similarly to the rook algorithm, however, it took diagonal movements into account.
4. Knight class
    a. Knights move by taking 1 step in one direction and 2 in another. The direction does not matter as long as it is not off of the board. It cannot move to a spot if there is a piece equal to its color on that spot.
5. Queen class
    a. Combined both the bishop and rook algorithms to create the queen. This saved me a ton of time with having to reimplement something that I have already implemented
6. King class
    a. Can take 1 step in any direction unless there is either a white piece in its way or if it is in the path of an opposing team's movements.

The chess AI algorithm will use the minimax algorithm, which works to minimize the worst case. This algorithm uses a tree to look at every possible scenario within a certain number of rounds which depends on the set depth of the tree. In this case, this project had a set depth of 80. Each node on the tree is then assigned a value based on what pieces could potentially get lost during that round. This is done by assigning each chess piece a weight. Then, the worst case is calculated by adding up all of the weights of the pieces that could be lost. Finally, the minimax algorithm decides what move will cause the least amount of casualties. This causes the AI to play a more defensive role, however, it also makes it tougher to capture the opponent’s king. The aforementioned weights are listed below along with any reason for the set weight:

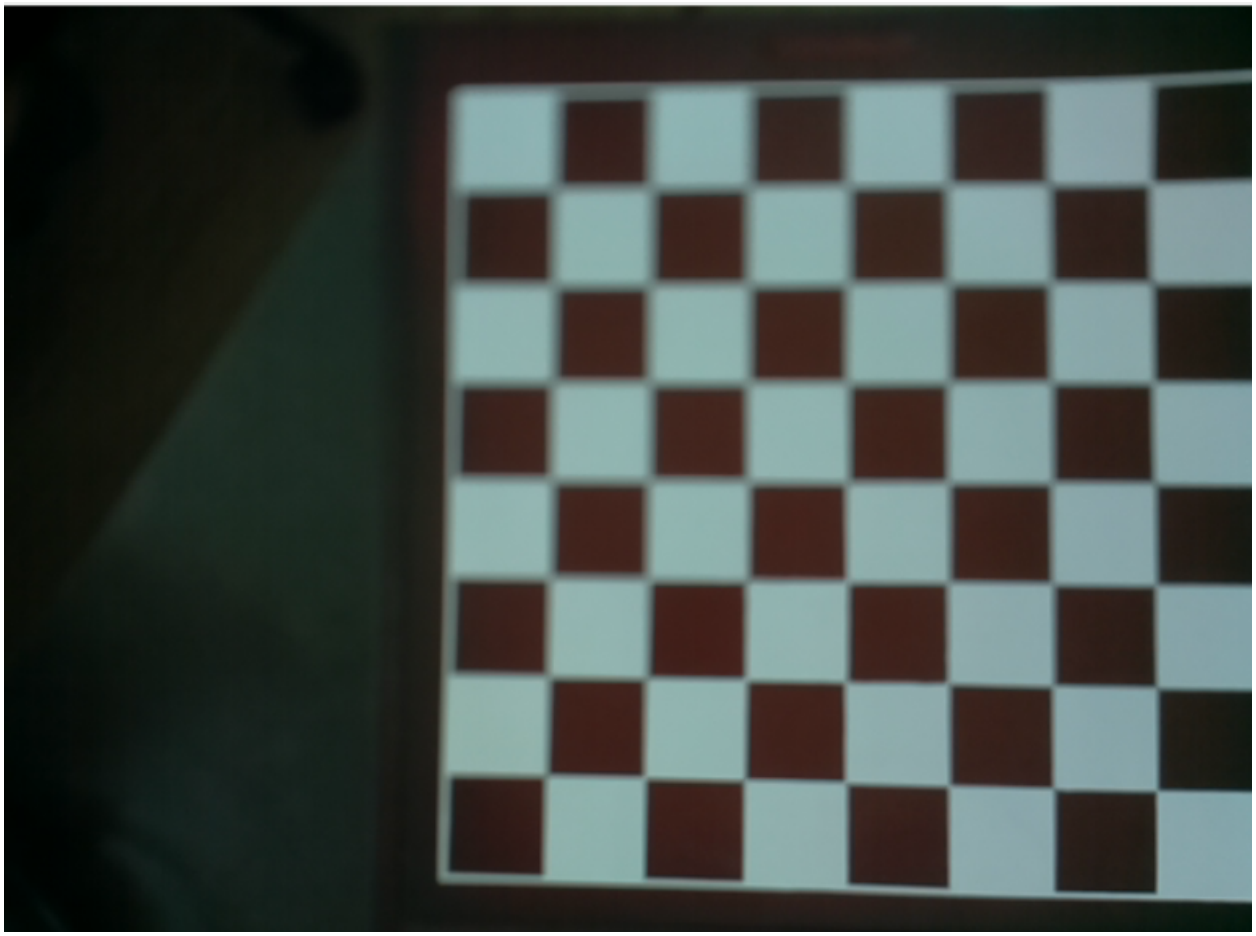
1. Pawn = 10
    a. Not the most important piece in the game.
2. Knight = 30
    a. Slightly more important than the pawn.
3. Bishop = 30

    a. Slightly more important than the pawn.
4. Rook = 50
    a. I gave this a higher score because I personally use rooks to checkmate my opponents whenever I play.
5. Queen = 650
    a. The queen is definitely the most important piece in the game, however, if she could be sacrificed for a good play, then it is worth losing her.
6. King = 900
    a. The king cannot be lost in any case. Losing the king means losing the game, which is not what the AI is supposed to do. Therefore, it should avoid situations in which the king is at most risk.

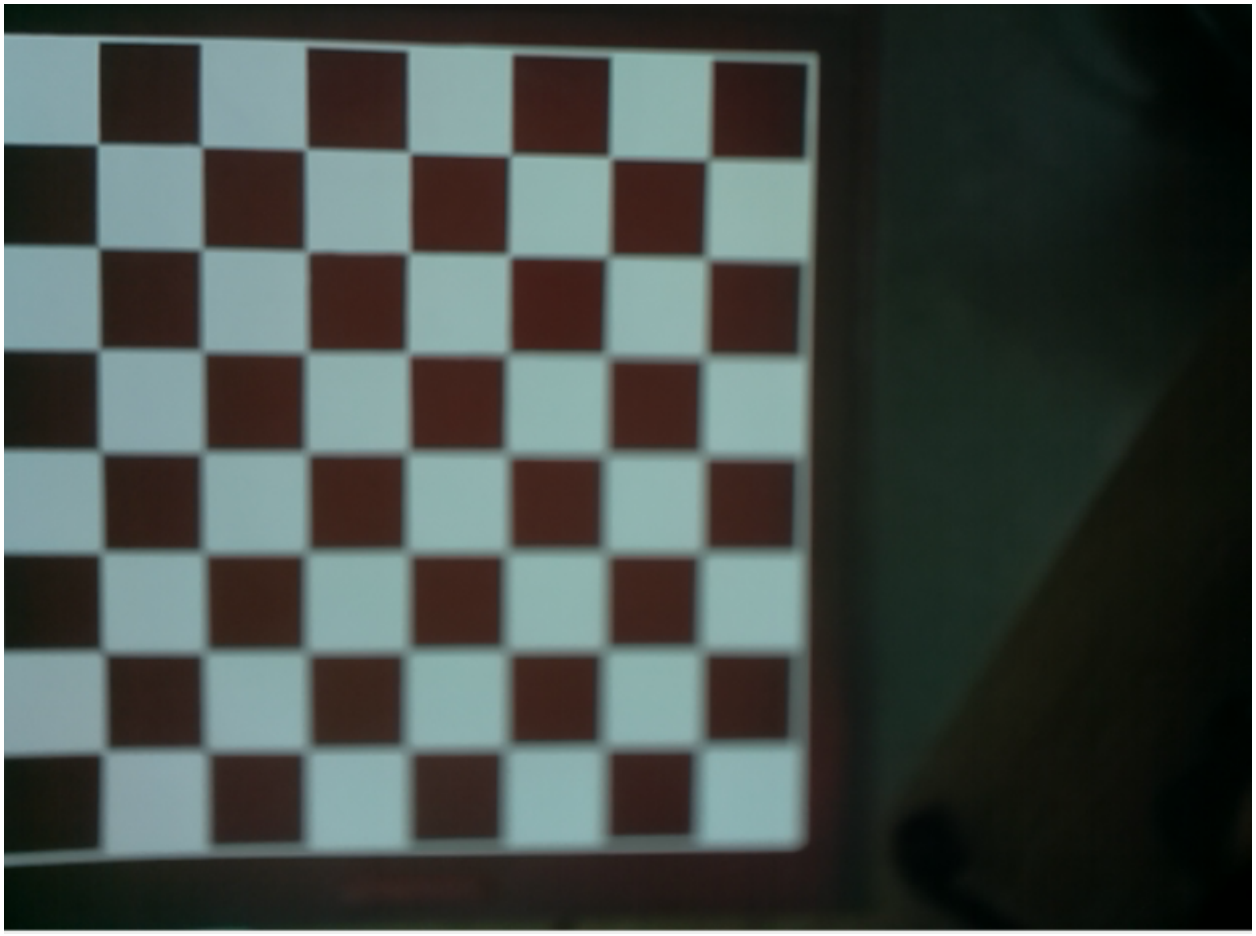
Once an ideal move is found, the system lights up two LEDs based on the position of the piece on the chess location matrix. The move-piece LED will be represented by a red light and the new position will be represented by a green light. If the piece is moved to the incorrect position, the incorrect position will flash a red light and the correct position will flash a green light until the piece is moved.

3.2.2 Physical Board Mapping

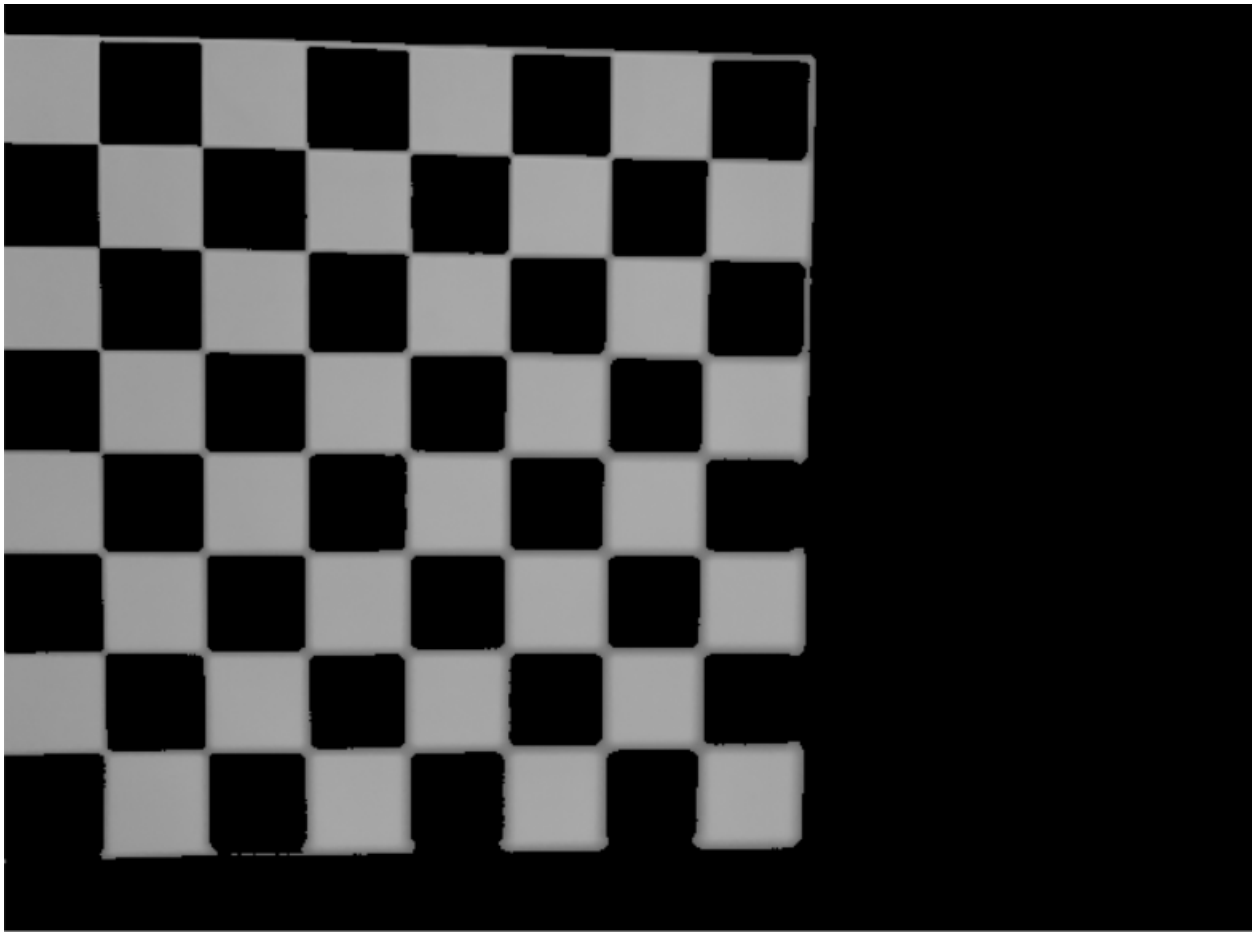
The physical board is mapped using the OpenCV library. First, a picture of the board is taken using the Pi Camera. The image is then flipped so that traversing the image is much quicker during analysis. Next, the image is converted into a binary image -- black and white -- to remove noise from awkward lighting in a room. Since the chess board is a repeating pattern of brown and white squares, the board is much more differentiable as a binary image.



Initial image of the chess board

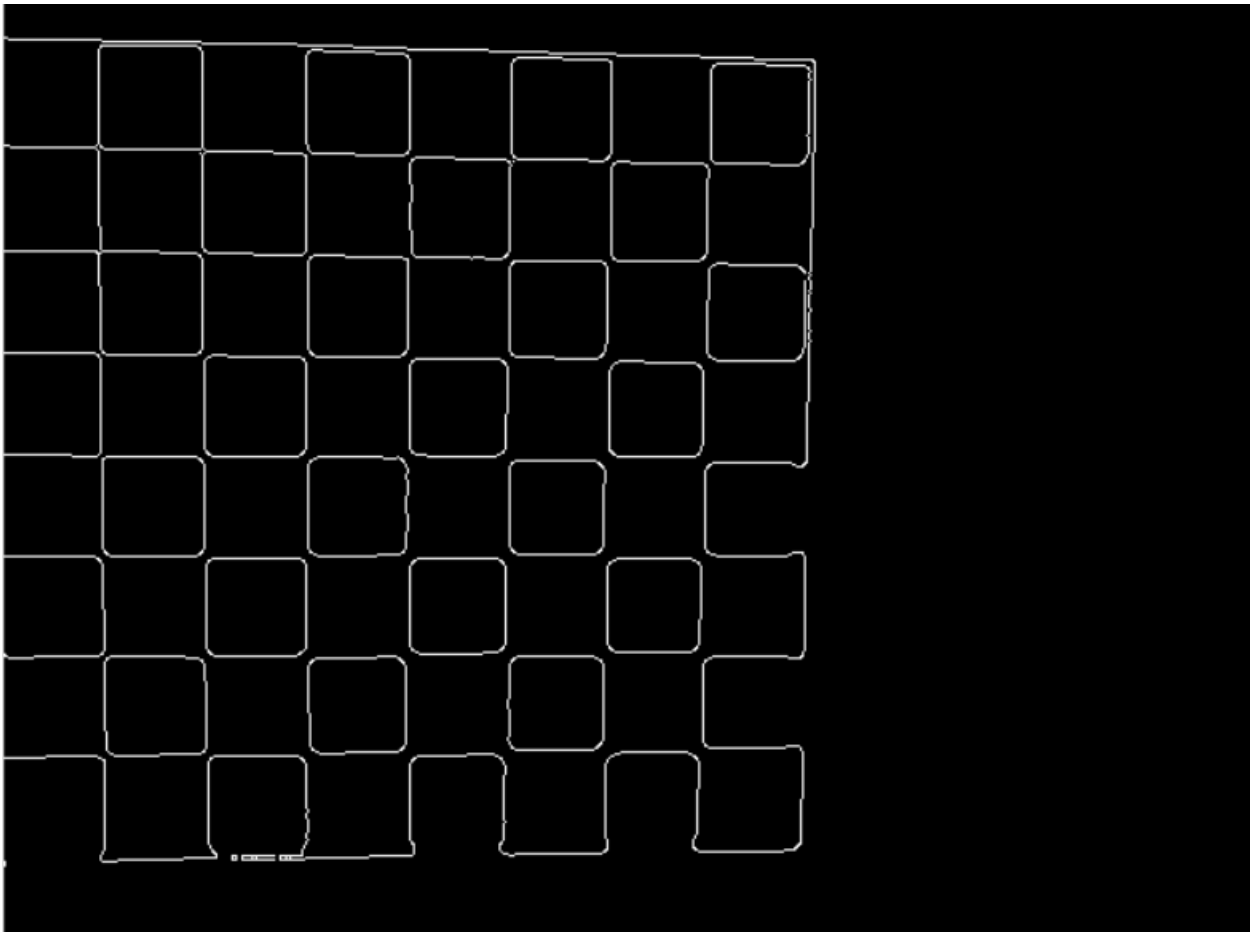


Flipped image of chess board

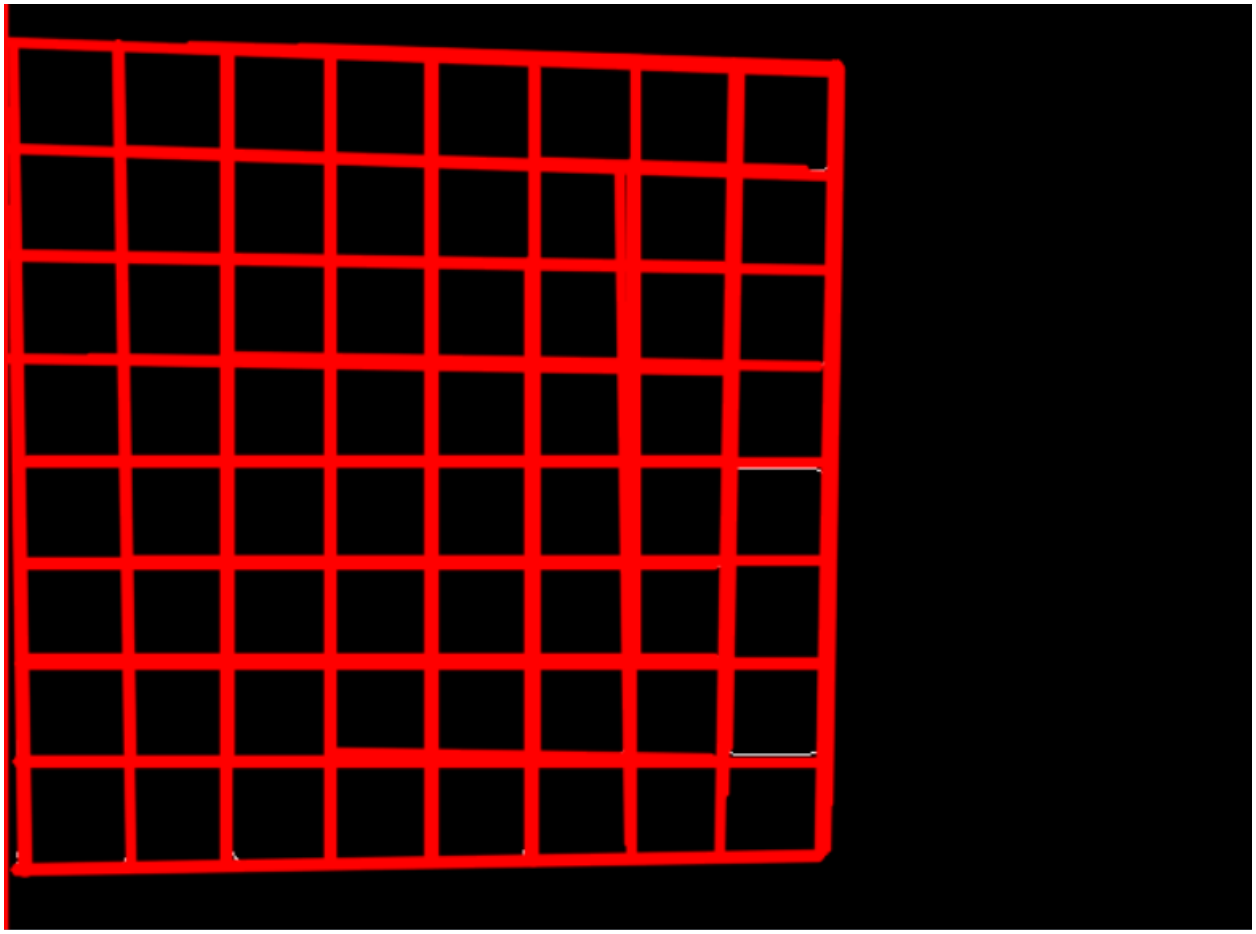


Binary image of the chess board

Next, the binary image is taken and its edges are traced using Canny Edge Detection. Canny edge detection creates a black image with a white outline to indicate detected edges. Finally, Hough Line Transform is performed on the image which detects potential points or lines and draws a solid red line over them. I did this because I could not optimize the Canny Edge Detection algorithm to detect every single edge on the board because the Pi Camera's low resolution was not able to detect the white outline that separated the outer grids on the chess board from the outer edge of the chess board. This caused multiple gaps to show up on the Canny image. Applying Hough Line Transform filled in these gaps to create a red and black outline of the chess board.



Canny image of the chess board



Hough Line Transform of the Canny image

Once the Hough Line Transform image is obtained, it is traversed to find the coordinates of each line. Starting from the top-left pixel, I make my way across the image until I hit a red pixel. Once that happens, I keep track of the red pixels until I hit a black pixel. When I hit a black pixel, I travel 25 more pixels across. If no other red pixel is detected, I store the position of the last red pixel. That is how I get the vertical lines' position. For the horizontal line, after I hit a red pixel, I keep track of the number of red pixels that I am hitting in a row. If that number goes over 100, it is a horizontal line and I store its position. Repeatedly doing this process gets the position of every single line. Now I know the bounds of each individual grid on a chessboard. In the instance where I ran the demo, I obtained the following coordinates:

X: [11, 60, 115, 170, 224, 277, 331, 386, 440]
Y: [31, 86, 139, 193, 240, 305, 363, 431, 477]

Assuming that the chessboard does not get moved around, I am able to instantly know where to look on an actual image to find a specific grid position. For instance, if I wanted to check the grid position (0, 0), I could examine the positions 11 < x < 60 and 31 <

y < 86. In more general terms, to check the grid position (a, b), I could examine positions X[a] < x < X[a+1] and Y[b] < y < Y[b+1].

3.2.3 Player Movement Tracking

The technique above is how the player's movement is tracked throughout the game. An image of the board is taken before the player makes a move. Once the player does make a move, they are requested to press a button. Pressing the button takes another picture of the board. Then the two images are traversed at the same time, one grid at a time. Each pixel contained within a grid is compared to the other image's grid equivalent grid. The percent change of each pixel between the two images is calculated and stored in an array. Once each grid is checked, an algorithm to find the top 2 largest percent changes in an array is run. These 2 grids represent the piece's initial position and its updated position. Initially, I created an algorithm that found 2 percent changes greater than 30% and used those 2 grids, however, shadows that the pieces cast created noise in the image that was caught and recorded as percent change. This can be seen in the image below

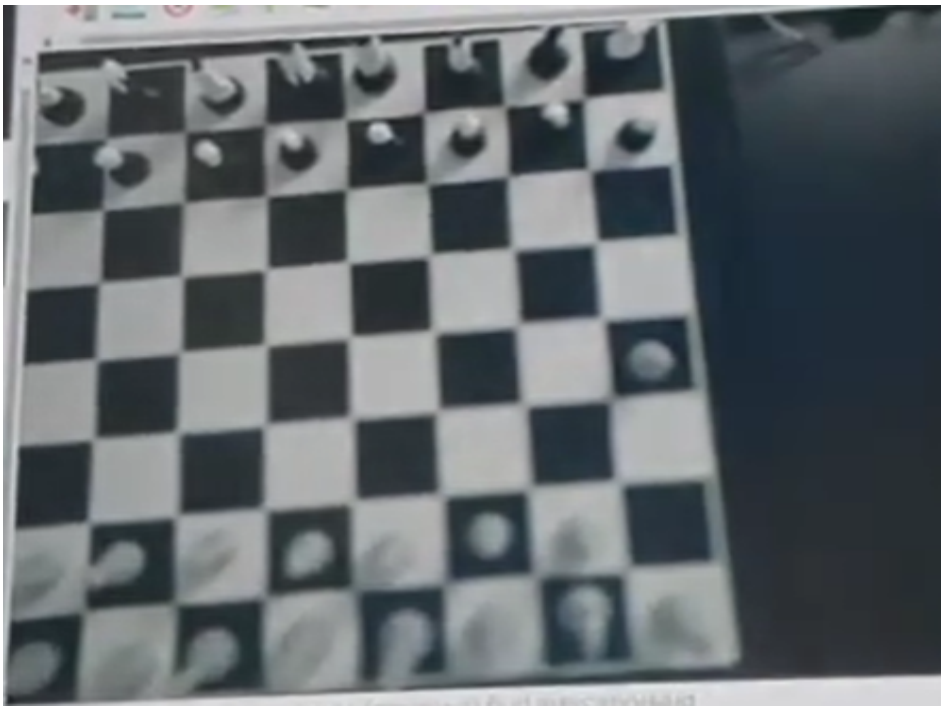
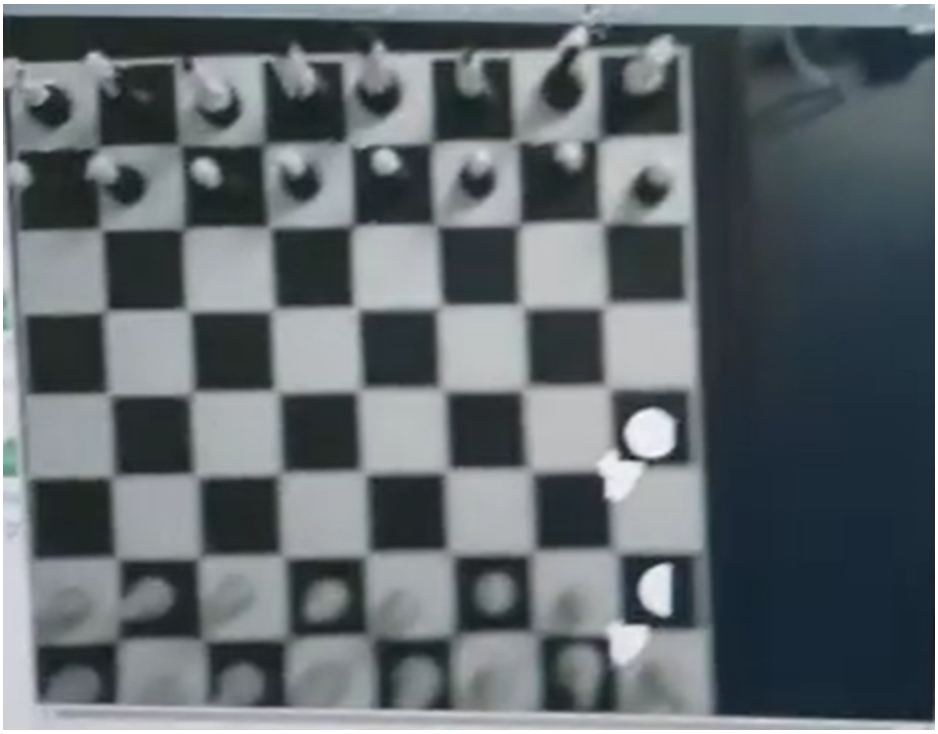


Image of the board after a piece is moved

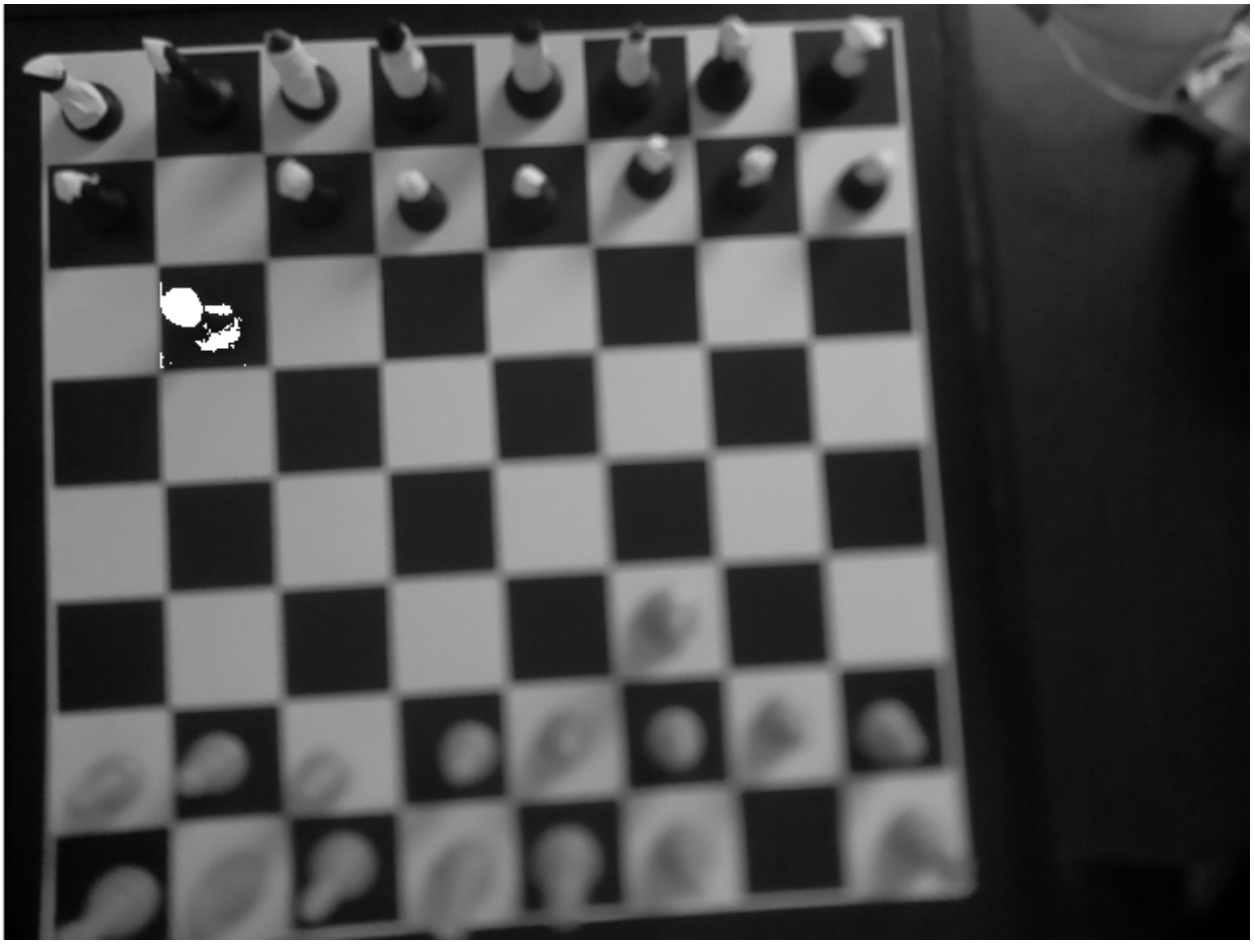


Detected pixel changes are marked in white. The chess piece's shadow is also detected.

Finally, the two grid positions are sent to another algorithm that figures out which of the two grid positions contains the piece and which grid position is the new position.

3.2.4 Opponent Movement Verification

To make sure that the player is not cheating or misplacing the opponent's piece, before the LEDs light up to indicate what the AI played, a picture of the board is taken. Once the opponent's position is marked with the LEDs, the player is instructed to move the piece to its new position and press the button. Pressing the button will temporarily turn off the LEDs then take a picture of the board again. The LEDs are turned off to make sure that the colorful lights that spread onto the board do not make the image taken messier than it needs to be. Once both the 'before' and 'after' pictures are taken, they are compared. However, instead of traversing through every single grid position, only the new grid position is checked. If there is a noticeable difference detected, the software is updated and the game continues, or else the 3 indicator LEDs flash red to indicate that the piece was either misplaced or not detected. This repeats until the user placed the piece in the correct position.



Only the designated destination grid is checked

4. BOM

Item	Price
WS2812B LED Strip	$10.88
L-shaped light Connector	$9.99
Pi-Camera	$13.99
Raspberry Pi	Already own
Camera stand	Already own
Chessboard	Already own

5V Battery	Already own
Wires and resistors	Already own
Total:	**$34.86 [Free shipping]**
Arrival Date:	**03/22/2021 [5-day shipping]**

Request for reimbursement:
No reimbursement is necessary. I will be keeping my own project.

5. Tentative work schedule

Week	**Plan**
Week 1	● Purchase any additional materials that I may need ● Complete hardware wiring ● Have a basic understanding of what I need to do on the software side
Week 2	● Implement additional hardware that may have arrived late or needed to be replaced ● Begin working on the chess algorithm using a software GUI ● Use OpenCV on software GUI to get a better understanding of OpenCV
Week 3	● Polish chess algorithm ● Implement physical camera into system using OpenCV and raspberry pi camera
Week 4	● Polish any additional touches