Chess Teacher

Authors: Michael Cai, Joseph Chang, Jee Woong Choi: Electrical and Computer Engineering, Carnegie Mellon University

***Abstract*–A system capable of teaching people how to play chess on a real chessboard. The system is able to play chess against a human being and also recommend moves to give them optimal moves to help them improve their performance. The system involves a real chessboard with a camera that takes image when pressed a button to detect the moves of the chess pieces on a computer. Then, it validates the move and also recommends moves on the FPGA, communicating through UART with the computer.**

***Index Terms*—Artificial Intelligence, Chess, Computer Vision, Design, FPGA, HPS, OpenCV, PyGame, Stockfish, UART**

1 INTRODUCTION

The Chess Teacher is motivated from learning how to play chess in this pandemic and the chess boom after the Netflix's historical drama "The Queen's Gambit" became famous. In the COVID era, it is hard for people to learn how to play chess from a human coach or play over the board with others. We wanted a way to solve this issue when people are desperate to learn how to play chess after the boom from the Netflix drama. Our group aims to create a Computer Vision based Chess Artificial Intelligence (AI) which can compete against a human player and teach various moves while a user is playing the game.

Computer Vision based Chess AI can give the following advantages. First, people can save money from the in-person tutoring. Because the Chess Teacher will recommend various moves and play against the human player, the user does not have to learn from a human coach which can cost quite a lot. In addition, our approach definitely helps social distancing in the pandemic since people do not have to meet each other to play chess. Most importantly, users can play with a physical chessboard. There are chess games available online, but users often feel like they are not playing chess. They feel like they are playing a computer game. However, because our Computer Vision chessboard detector can detect changes on the board, users can play with an actual chessboard which will help them feel like they are in competitive settings such as tournaments.

Our goal for this project is to let people enjoy playing and learning chess with physical chessboards and pieces without any help from other people. In order to achieve this goal, we need to detect the board accurately and efficiently through computer vision algorithms. The computer of the Chess Teacher must detect changes on the board within 400 ms with 99 percent accuracy. The FPGA of the Chess Teacher should be able to generate all possible moves in 500ms. The UART communication between the FPGA and the computer must have a latency of less than 1 second with 100 percent data accuracy. The user interface of the Chess Teacher should be able to display the board correctly 100 percent of the time.

2 DESIGN REQUIREMENTS

2.1 Move Detection

The main requirement of the move detection is split into two parts: the accuracy and the processing time. The accuracy of detecting the board will be done through a scene of 40 unique moves. The reason behind 40 unique moves is that an average of moves per chess game is 40 moves. To test the accuracy, after 40 unique moves, we will have a visual confirmation of where the chess piece moved to on a physical board and also print out what the software thought to make sure both of them indicate the same position of the chessboard. To achieve 100 percent accuracy, we want to ensure none of the moves are detected incorrectly, since not detecting the move will have a significantly negative impact on the user experience especially messing up with the clock system. To measure the processing time, the software will have print statements to indicate how long it took to detect the move in milliseconds after receiving two frames.

2.2 Legal Move Generation

For the main requirements of the legal move generation, we want to ensure 100 percent correctness on all generated moves. We tested the correctness of legal move generations on 10 unique board states. On these 10 unique board states, we will check all the valid moves by going through each piece and see if all the moves have been generated correctly. This is part of the FPGA subsystem and hence, we will provide a hardware test bench which provides the proper inputs and checks the outputs for correctness. We also have a latency requirement for this move generation. We want the overall system to run very quickly and given that we are using the FPGA to accelerate stockfish, the legal move generation should take 50ms at the most.

2.3 Communication Protocol

The main requirements of the communication protocol are accuracy and speed. We want to ensure that each of the packets we send is not corrupted and 100 percent correct. We also want to ensure that the speed of the Transfer

Protocol to be under 1 second. This is important because communication will be the main bottleneck of the FPGA subsystem. The user should not experience any large latency or noticeable lag after making their move.

2.4 User Interface

The main requirement of the user interface is its correctness and accessibility. The interface should show the correct state of the board. The correctness is tested on visual confirmation on whether the board and the interface matches. In addition, we wanted the user interface to be easily accessible to the users. Users shouldn't have trouble understanding how to play the entire game. So, we focused on making the interface intuitive and easy for the users.

3 ARCHITECTURE OVERVIEW

The overall architecture is divided into two main subsystems. The computer vision software, and the game logic software/hardware system. These subsystems are further divided into the following components:

- OpenCV
- UART Communication Protocol
- PyGame
- FPGA Custom Hardware
- Stockfish AI

We will now describe in detail these components and interconnects and demonstrate how they fit into our overall architecture. For a full diagram of how all components work and connect, see the appendix.

3.1 OpenCV

We are using OpenCV for our Computer Vision part of the project, which provides an optimized Computer Vision library and tools. Since OpenCV supports Python, it easily soothes into our program which is mainly based on Python. OpenCV provides conversion to HSV scale, chess corner detection and blob detection. We use all of the functions mentioned to reduce the development cycle drastically and also tune them carefully to achieve our target accuracy. The OpenCV takes in input from the frames obtained from the camera to do all the computer vision work necessary.

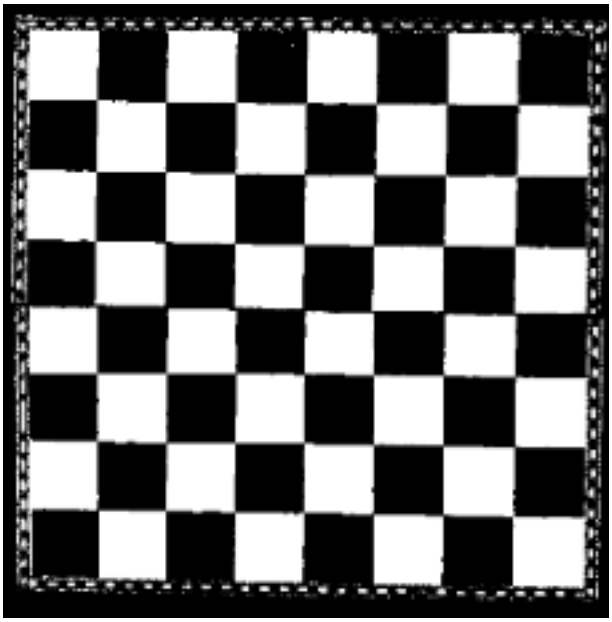

Figure 1: Chessboard image converted into HSV space and black and white

3.2 UART Communication Protocol

UART is used to communicate between the PC and FPGA. We chose UART over other serial communication protocols such as I2C or SPI because of the ease of implementation. The FPGA has a dedicated UART to microUSB port, thus allowing an easier time to implement the serial communication protocol. I2C or SPI would require use of general purpose IO pins and possibly a very specific cable to connect to the PC. Moreover, the UART protocol is estimated to fulfill the communication latency requirements as demonstrated in equation 1. We are also able to use the python library PySerial on both the FPGA and the PC to further reduce the burden of implementing a complex communication system.

Assumption 1: 921,600 baud rate
Assumption 2: 20,000 bits / board state
20,000/921,600 = 0.022s communication latency

3.3 PyGame

PyGame is a cross platform set of Python modules designed for writing video games. A Chess game written with PyGame will mainly handle the user interface part of the game. The user interface will show the status of the board, moves that the Chess Artificial Intelligence makes, and recommendation of moves for the user.

3.4 FPGA Custom Hardware

The FPGA custom hardware will be computing the legal move set for any given board state. This is the compute heavy portion of any chess engine because of the sheer number of squares and possible moves for any given square.

However, the task is easily parallelizable by generating legal moves for all squares at the same time. The FPGA excels at this task as it can generate application specific hardware for each square on the chessboard which allows the legal move generation to happen in parallel for each square. Furthermore, the FPGA consists of a hard-system processor (HPS) which allows the chess engine (described in section 3.5) to be closely integrated with the FPGA's application specific integrated circuit. This reduces any additional communication latency.

3.5 Stockfish AI

The chess engine we are using is Stockfish, which is an open source chess engine. The majority of the engine code will run on the FPGA's Hard Processor System (HPS). Move generation is repurposed to make use of the custom FPGA logic. This system used Stockfish 8, an older version of the engine. This is because of some compatibility issues due to the Arm Cortex CPU and somewhat dated operating system which ran on it. We chose Stockfish because it is the most known open source engine, and as it is open source we may modify its code base to implement and make use of our FPGA fabric[8].

4 DESIGN TRADE STUDIES

4.1 Chessboard Corner Detection

We used the chessboard corner detection algorithm provided by OpenCV to retrieve chessboard corners' individual pixel points to keep track of the positions of pieces. Another option for us was to use an edge detection of the square tiles on the chessboard. Although the edge detection might be effective at first, the accuracy of the edge detection was poor compared to that of corner detection. Because there are so many edges on the chessboard which can possibly make a noise, a small error on edge detection could result on wrong positioning of the tiles and result in wrong positioning of the pieces. Hence, we decided to use chessboard corner detection which was more accurate than edge detection algorithm.

4.2 Move detection Algorithm

To detect the moves of the chess pieces, we had two choices. The first choice was to use template matching [7]. to find which pieces moved to where. From having two frames to compare to, we find each position of the pieces to find which moved to where. However, the template matching was not very accurate. In figure 3, you can see the template and in figure 4, you can see the whole chessboard to find the template. As you can see with human eyes, the template and the image does not really match in most of the pawns due to the angle of the camera and also because of having different backgrounds. As a result, the algorithm is also unable find any correlation from the image, and the template matching was not particularly useful.

Figure 3: An Image of a Pawn as a Template

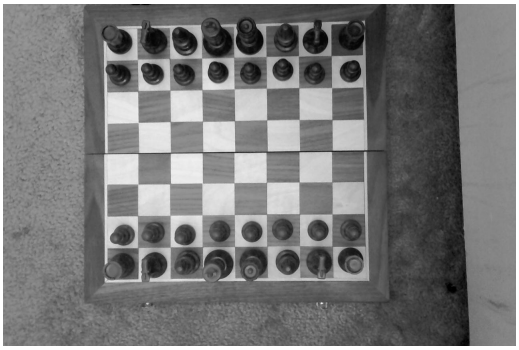

Figure 4: The Board to Match the Pawn

The other choice, using background subtraction algorithm [5], was effective. We used the background subtraction to find the original position of the chess piece and where it moved to. Since the camera does not move, only the pieces that have moved from the two frames that the system uses will be subtracted and will be represented as white circles because white pixels have a value of 255. Another reason that this is useful is because our camera has a top down view, so in most cases the subtraction of pieces will result in circles rather than random shapes. Having circles is especially useful for blob detection after the background subtraction step because the shapes are similar.

4.3 Turn Based Chess Game

To do a background subtraction algorithm on two frames, we could have chosen two approaches to take the frames. First approach is to do a real time analysis of the chessboard. We can capture multiple frames and try to find the best two frames which the chess system sees fit to subtract background on. However, though many applications use this approach, our system is just complicated by this design. Instead, we could use a turn based system, where the users click a button to capture each frame, similar to how chess players a push a button to indicate that they are done. Using this approach, the game becomes more realistic and also becomes more accurate to meet our design requirements.

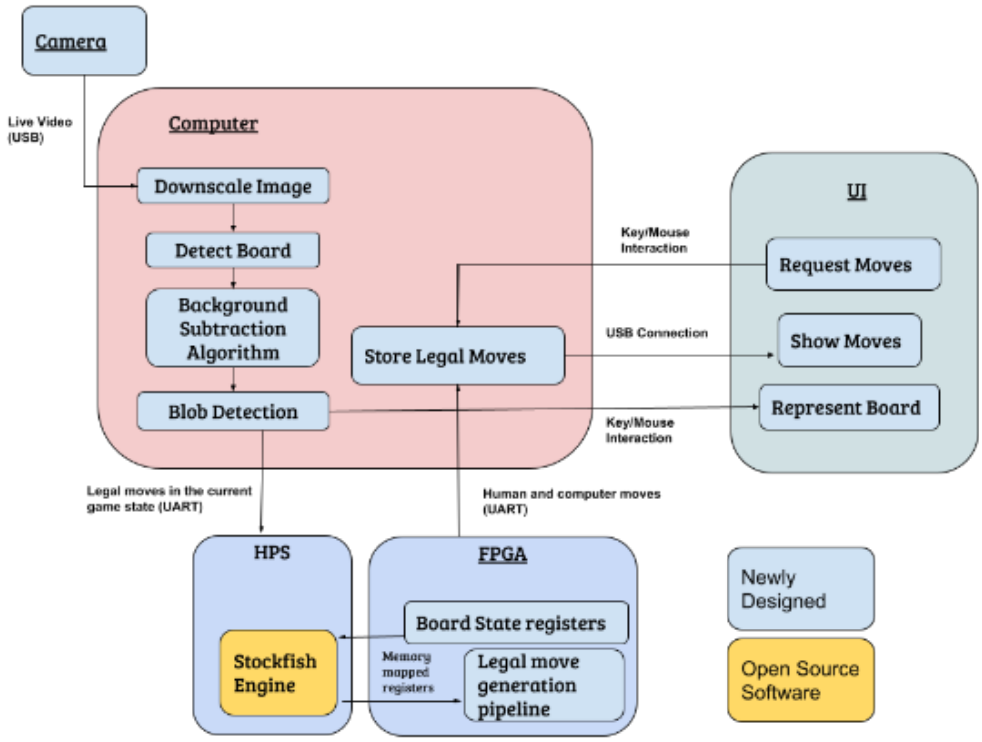

Figure 2: System Diagram

4.4 User Interface

We wanted to use Python to develop the User Interface because we are also using OpenCV which is written in Python, which makes it easier for the User Interface to interact with the Computer Vision algorithms. There are mainly two possible options for us to create the User Interface. First, we can use tkinter which is a built-in Graphical User Interface toolkit. However, tkinter only supports simple features, so we thought it would be a good idea to use a different toolkit. The second option was to use Pygame. Pygame is explicitly built for creating a game using Python programming language. Thus, it will be a better idea to use Pygame over tkinter to create the User Interface for our Chess game. [3]

4.5 Computer

While it may be easily thought that Raspberry Pi would be an appropriate choice for our choice of computer to run the computer vision algorithm, the Raspberry Pi was not able to handle the Pygame. Pygame is heavy for many reasons. It is constantly reading external resources, reading the events going on and also reading the position of the mouse constantly. As a result, the Raspbery Pi kept on crashing after not being able to handle the workload of Pygame. In metrics, the Computer Vision algorithm took 20 percent workload of the CPU while the Raspberry Pi took around 80 percent workload of the CPU that constantly made the Raspberry Pi crash. As a result, we decided to use the Macbook Air that we had readily available, which has same number of 4 i5 Intel cores running at 1.1 GHz compared to 4 Arm Cortex A72 cores running at 1.5GHz on the Raspberry Pi. Just by the numbers, it may seem the Arm Cortex should perform better, but the Intel cores actually perform better because of other factors such as instructions per cycle, better CPU cooling, and overclocking.

4.6 Computer Vision Software

For using OpenCV, we had two choices. We could have either used C++ or Python. C++ had its advantages. We had more fine grained control of the computer vision algorithm, which would decrease the execution time of the OpenCV algorithms. Python had other advantages. It has a fast development cycle, and since the Pygame was in Python, the integration with the UI would be seamless without having to compile C++ into machine code and integrate the code with the user interface application. Since the execution time of the OpenCV algorithm was relatively negligible because the algorithms we were running were not heavy, we chose Python as our software for a faster development cycle and better portability.

4.7 FPGA

We decided to use an FPGA for move generation. This was mainly inspired by previous chess engines which handled move generation on FPGA. Such engines include Hy-

dra[4] which was developed in 2004. We hope that running Stockfish on the integrated HPS system will ensure that latency costs are minimized and there is a total compute time decrease as compared to Stockfish's base move generation. FPGAs and custom logic appear to have some advantage in terms of computing legal moves of a given state. This is because the task is highly parallelizable over the 64 squares of the chessboard. Checking for the legal moves of each of the 64 squares requires the exact same logic and all the other squares. Hence, it makes sense to attempt to accelerate the move generation portion of the chess engine over an FPGA. Alternatively, we may instead run standard Stockfish13 on the PC. This may have reduced the communication latency needed because of the FPGA subsystem, but may have worse compute times overall when considering the legal moves of any given board state.

4.8 Terasic DE-10 Standard FPGA

We chose the DE-10 Standard mainly because of its IO capacities and its relatively simple layout and design process. The DE-10 Nano has no switches and only 2 push buttons. We figured there may be a possibility that we use FPGA IO to toggle different modes in Stockfish. The DE10-Standard has sufficient switches and push buttons to allow for this possibility[10]. We also considered Xilinx Kintex-7 boards but ultimately decided to use an Altera compatible board due to our experience with the Altera workflow as compared to Xilinx's Vivado software. The DE-10 Standard also had capabilities to display to an external monitor via VGA cable. This was important for debugging purposes as a graphical interface came in handy when working with the FPGA.

4.9 Stockfish

We chose to build on the Stockfish engine rather than other engines because Stockfish is open source and perhaps the most widely known and documented open source chess engine. Although it may not be the absolute best as it has lost to other engines such as Alpha-Zero[9] in computer chess tournaments, it has won a fair number of chess tournaments and was the most documented of open source chess engines. Another open source engine we could have used was Leela Chess Zero but we decided there was more documentation and popular usage around the stockfish engine.

5 SYSTEM DESCRIPTION

5.1 Image Processing and Chess System

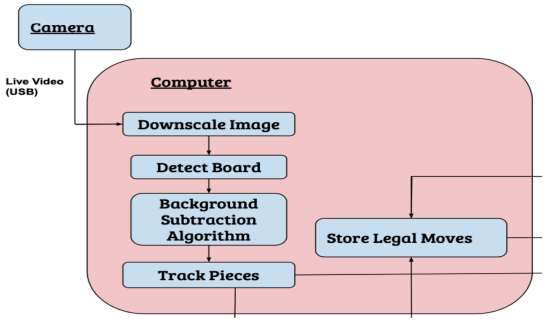

Figure 5: Image Processing Diagram

The computer starts the image processing through receiving images from the camera through a USB connection. The images received form the camera will be converted from a RGB image into a HSV space. This will allow the image to be less affected by glare which helps our computer vision algorithm to detect the board and pieces better. After converting into HSV space, the chessboard detection algorithm will detect the corners of the chessboard. The problem with this chessboard detection algorithm is that the order is non-deterministic. It will sometimes look for corners from the top left to bottom right, and it will sometimes look for corners from top right to bottom left, and etc. To solve this problem, we sort the corners so that the bottom left corner is the first corner on our list. Another problem we faced was that this chessboard detection algorithm can only detect the inner corners of the chessboard, so we had to use the 64 inner corners [2] to find the outer most corners. The computation behind getting the outer most corners was done by calculating the average width and height of the corners and using this information to predict the outermost corners. After getting the outermost corners, the number of corners become 81 corners.

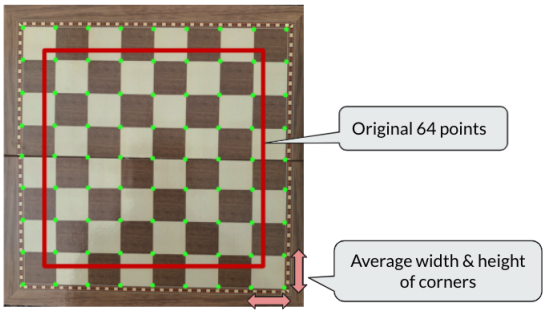

Figure 6: Corners Detected as Green Dots on a chessboard

By obtaining the coordinates of the 81 chessboard corners like on figure 6, we are able to get the corners of each square. Then, we place the pieces and map each of the squares to its pieces. When the game starts, the camera

will automatically save the frame that it is seeing at the moment which will serve as the initial frame. Then, after the white player makes a move, the player presses the button. This signifies an end of a turn, in which case the camera will save another frame at the end of each turn. From these two frames, we use the background subtraction algorithm on the board to see which pieces have moved by comparing two frames.

The background subtraction subtracts the two frames. Then, it uses a threshold of 45 to set every pixel value lower than 45 to 0 and every pixel value greater than 45 to 255. As a result, we get the background as a black background with the moving piece as white circles to indicate where it moved from and where it moved to. An example of this result is shown on figure 7. Afterwards, we use the fact that the ones subtracted are circles to our advantage by using a blob detection[6]. Because blob detection algorithm only works on white backgrounds, we first invert the frame we obtained from the background subtraction algorithm. Then we find the new black circles through the blob detection. An example of this result is shown on figure 8. From the blobs that we identify, we get the coordinates of the original position of the piece and the position it moved to. From these positions, we look at our original 81 chess corners to find which square the coordinate belongs under. We find if the piece belongs to a square by comparing the top left and bottom right coordinates of the corners of each square of the chessboard. If the coordinate belong between the coordinates, then we locate the squares to the system. Then the system looks if any of the squares correspond to squares of any chess pieces. If it does, we know it is the original position that it used to be in. Then, the other square is the new position it moved to. The reason that we have to find this manually is because blob detection is not deterministic and can potentially find one circle before the other and vice versa.

After the moves are recognized, the board first communicates with the FPGA to validate the possible moves. If the move turns out to be illegal, it will communicate with the user interface to make the user interface show the user that the move made is an illegal move instead of allowing it on the board state. Otherwise, the board updates the board state with the new chess piece positions and communicates with the user interface to show the new board state.

Another feature that the chess system has is recognizing whose turn it is. This feature helps the chess game from figuring out certain edge cases. For example, if a piece takes another piece, it will show two circles from the background subtraction, but we are unable to figure out which piece took which piece because the the pieces are converted to black and white. By having turns, we know which one took the other one because one of them has to be a white piece, and the other one has to be a black piece. So we eliminate the other piece from the game in this way.

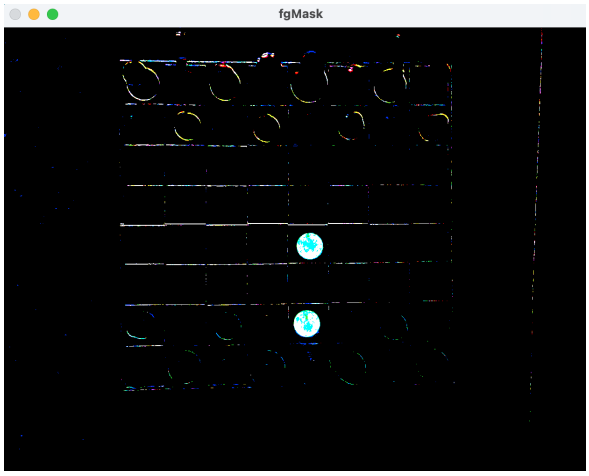

Figure 7: Frame Obtained from Background Subtraction

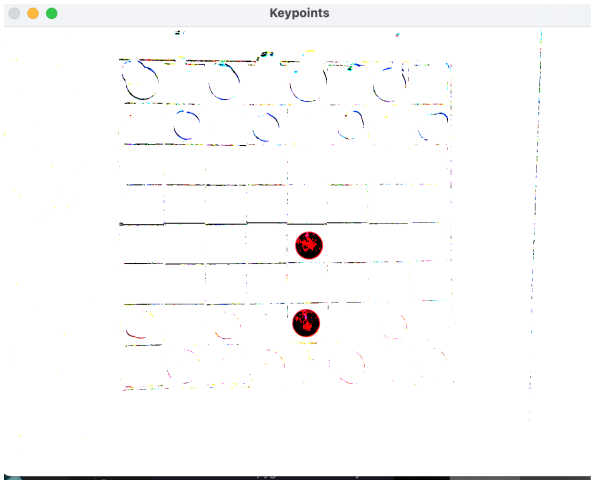

Figure 8: Blob Detection Locating Two Blobs with Red Circles

5.2 FPGA + Hard Processor System

The FPGA and Hard Processor System are used to generate the legal moves given a specific board state. The Hard Processor System (HPS) is tasked with running Stockfish, an open source chess engine/AI, with slight modifications to make use of the FPGA hardware. Custom logic will be described in SystemVerilog and synthesized onto the FPGA board. This custom logic will handle the legal move generation of a specific board state. This will aid stockfish to make faster decisions and also quickly verify whether or not the human user has made a legal move. The General algorithm of the custom logic is demonstrated in the two figures below (figures 9, 10):

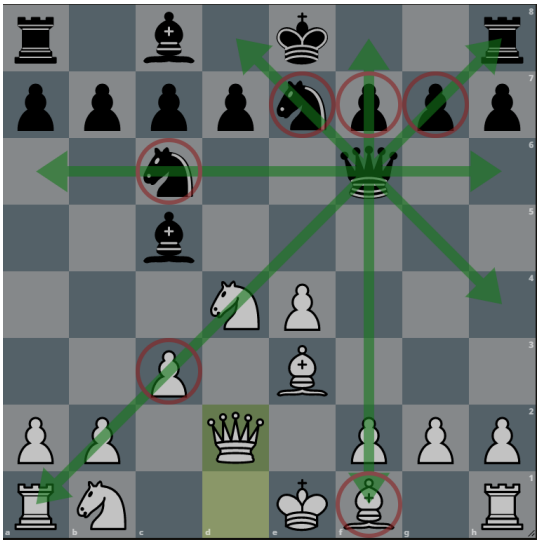

Figure 9: Standard Queen Logic

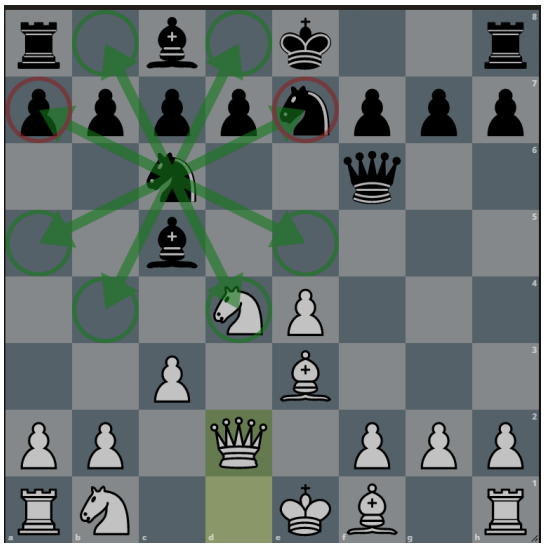

Figure 10: Standard Knight Logic

To summarize, one may think of the circuit as a grid of comparators. For the queen, it must compare itself to all squares which are on the same diagonals, same row, and same column. When it encounters a piece which is the same color as itself it must not move to that square or past it. When it encounters a piece which is a different color it may move to that square but not past it. The knight, on the other hand, has 8 set squares which it may move to. We simply check these 8 square to see if they are legal squares and if there is not a piece of the same color on that square. All other pieces move similar to a queen and will have some subset of the queen's move.

5.3 User Interface

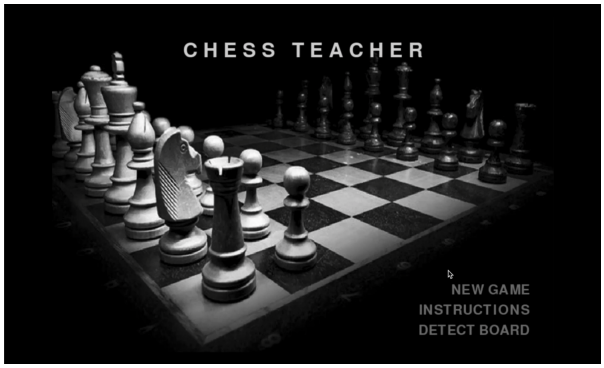

Figure 11: Main User Interface when Chess Teacher is started

User Interface is mainly written in Pygame, which allows the user to interact with the overall system. The figure 11 above is the interface when the user first starts Chess Teacher. There are mainly three options on the main page: start the chess game, see instructions, and detect board. After reading the instructions on how to play Chess Teacher, user must detect the board first without any pieces placed on the board. The camera that the user is using should have a top-down view in order for the easier detection of the chessboard. When the user interface shows that the board was detected successfully, user can start placing chess pieces on the board and start the game whenever the user is done setting up the initial chess game set up.

When the game starts, the user has two options. First, the user can place the piece by him or herself, or the user can press the light bulb button [1] see a recommendation of move that the AI is recommending. The figure 12 below shows a recommendation of move with a dotted circle which the AI is suggesting. This is to help user learn various moves from the recommendations that the AI sends. Once the AI's turn is over and the User Interface shows the AI's move, the user needs to move the piece for the AI so that the new board representation can be recognized by the Computer Vision Algorithm. And, this process will be repeated until the game is over. Each user and the AI gets 5 minute of total time to make moves in a single game, and the timer for each of them are shown on the top-right corner of the user interface.

Figure 12: User Interface when the Game is played

6 TEST & VALIDATION

6.1 Move Detection

As stated in the design requirements, there were two main criteria to the move detection: accuracy and latency. The accuracy we aimed for was 100 percent accurate from 40 frames because the user should have a good experience in each game. The latency we aimed for was less than 400ms for the user to have a low-latency experience with the game.

The accuracy of 100 percent was achieved under perfect conditions. By perfect conditions, there are mainly two requirements to be met. The first requirement is to make sure the lighting is appropriate. The lighting source needs to be ideally on top of the chessboard. If the lighting source is not from the top of the chessboard, then the chess pieces cast a shadow on the chessboard. Then, when moving the chess piece, the shadow carries over with the chess piece. In these cases, the chess system occasionally detect the move of the shadow rather than the actual move, which is adjacent to the actual move. The second requirement is for the board to not have the same color as the chessboard. This problem was initially a major problem for us because the camera was not able to distinguish the board color from the color of the chess pieces. Hence, we have resorted to using red and blue pieces for our chess game, which corresponds to white and black pieces respectively.

There are some parameters that is tuned in order to meet the requirement of 100 percent. The first parameter is the threshold of converting the frame we obtain from the background subtraction algorithm to binary values. After subtracting two frames, in order to convert them to binary values, we had to threshold the difference. After trying different values, we found 45 to be the best value as the threshold. 45 keeps the most of the subtraction of the two frames. If the value is too low, then there is too much noise to be handled by the blob detection algorithm. If the value is too high, then the differences that the background subtraction is just turned into black pixels. This situation is particularly true for red pieces moving to green squares. Since the values of the red and green values do not differ by much, the change is subtle for the camera to recognize. The next figures show the different results from different thresholds. Figure 13 shows the different out comes of different thresholds during the conversion to binary values. The top left image is the outcome of threshold at 10. As you can see, there is too much noise, and the white circles are hard to identify. The top right image is the out come of threshold at 45. This is the final threshold we went for with the best accuracy. The difference is barely cut off which can be handled by the blob detection. The bottom left image is threshold at 60. Here, we start to see slight problems, the middle part where the top part of the pawn is shown is slightly turned to black most likely due to glare. In addition, most some parts of the pawn are cut off as well. The blob detection sometimes does not handle this well, so we did not use this threshold. On 32 of the opening moves, it was not able to detect 4 times, resulting in 87.5 percent accuracy. The bottom right image is threshold at 100. As you can see, almost all of the difference are cut off, so we did not use this threshold.

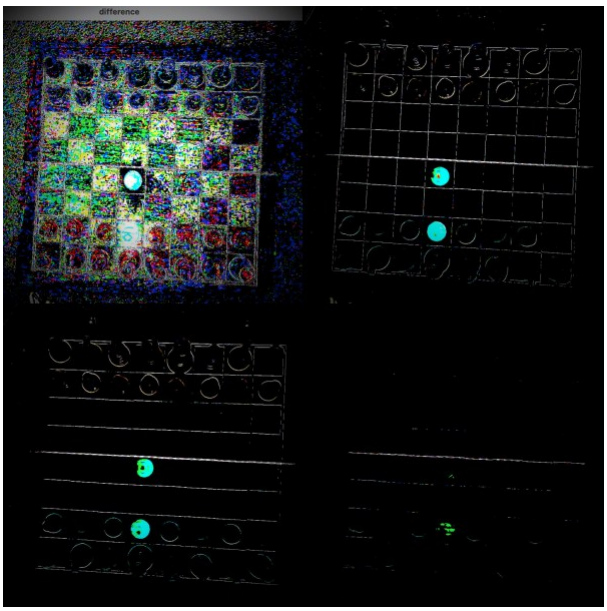

Figure 13: Four different thresholds

Another parameter that is tuned is the circularity. The circularity is critical for chess pieces such as a knight. Other than the two knights, every other chess piece has a circle shape. However, from a top down view, the knight has its back of head and the front of the head sticking out to abandon its circle shape. To account for this, we have tested few different parameters for minimum circularity parameter of the blob detection. Figure 14 shows the accuracy of move detection from the different values of minimum circularity. The accuracy of move detection was tested on 32 different board states. We can see from the trend of the graph that the accuracy is improved as the minimum circularity parameter is reduced. The error in the move detection were mostly attributed the knight pieces not being able to be detected.

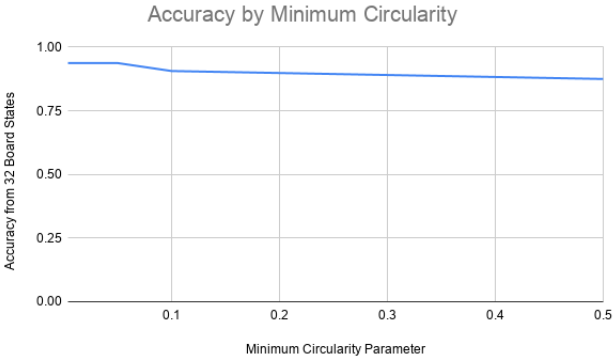

Figure 14: Circularity vs Accuracy

The last parameter that was tuned is the convexity parameter. This parameter was tuned after the circularity parameter because the circularity parameter alone could not reach. The chess pieces sometimes can have abnormal convexity due to the angle that the camera is viewing in. Hence, different convexity parameters helped accommodate for these cases. Changing the convexity parameter also helped fixing detecting the knight pieces that were hard to detect before. Figure 15 shows the accuracy of move detection from the different values of minimum convexity.

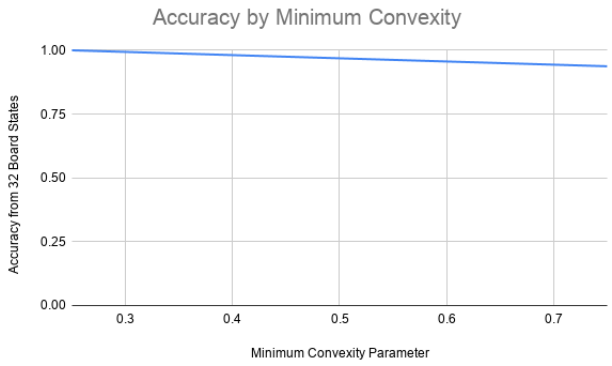

Figure 15: Convexity vs Accuracy

After the parameters were tuned, 40 board states were tested as mentioned on the design requirements. This was tested on a normal game up to 40 turns and seeing if they worked every time. We were able to achieve 100 percent accuracy.

The goal of having move detection latency under 400ms was achieved by mainly using our own background subtraction algorithm. The OpenCV algorithm has steps such as Gaussian Blur to remove noise in a real-world scenario, but since our chessboard only takes in two frames and does not have much noise to our frames, we get rid of steps such as this. Hence, we only find the difference between the pixels of the two frames, get the absolute values of the frames and make the pixel values binary to only find the useful information. We were able to achieve 22ms average in total, which was tested on 40 board states and were averaged. The timing was from the start of the background subtraction algorithm to the end of blob detection algorithm.

6.2 UART Communication and FPGA subsystem

UART serial communication ends up being the bottleneck latency of our system. On average over the course of 15 tests where packets of 4 bytes (the average chess move encoded in long algebraic form) were sent from PC to FPGA and a return packet was sent back stating whether the move was legal or illegal. We calculated this time using python's time library. By simply adding a call to python's time function we can measure the time between sending the packet using PySerial and receiving the return packet using PySerial. Over the course of these 15 tests we got an average round-trip-time of 542ms. A possible bottleneck for the communication is that after sending a packet from the PC, the PC must not immediately read the packet it had just sent out. This means we must use a sleep of half a second which ends up being the majority of the time taken in our round-trip-time. The actual move generation took just 3-4 ms and was measured in a similar fashion on the FPGA simply using python's time library to measure the time to compute the result.

6.3 User Interface

We tested whether the UI could correctly represent various board states. This test was conducted on 20 board states, and we confirmed that the user interface correctly represents the board states 100 percent of the time. Specifically, we tested multiple scenarios of the board states. First, we tested whether the user interface is behaving correctly on regular moves. Second, we also confirmed the case when a piece takes another piece. Third, we prevented users from cheating to make multiple moves at a single time and making a invalid move with a piece by showing "Illegal Move" message to the users. Thus, by testing with multiple scenarios, we have confirmed that the user interface can represent boards in different cases.

7 PROJECT MANAGEMENT

7.1 Schedule

There were slight changes to our schedule from the previous version. In particular, we were able to finish our individual components to start integrating earlier than what we proposed in the previous schedule. In addition, the integration actually took longer than what we expected it to take because there were some details that we did not really account for would take longer than what we expected. For the integration, everyone worked together, but Joseph and Drew mainly worked on integrating the computer vision component and the user interface together, while Michael worked on the integration between the computer and the FPGA. The full schedule that we adhered to is included at end.

7.2 Team Member Responsibilities

Michael was in charge of the FPGA and HPS subsystem. This includes pipe-lining the legal move generation, integrating Stockfish code to utilize the FPGA custom logic, and integrating UART communication with the PC. The FPGA custom logic will be designed in SystemVerilog while modifications to Stockfish will be in C++.

Joseph was in charge of the chess system being able to use the computer vision algorithms correctly and detecting moves. This involves implementing background subtraction algorithm, choosing the right parameters for the

blob detection as well as testing them to have high accuracy, making sure the chess system is able to see the board correctly from the moves it is detecting and testing the robustness of chessboard system.

While collaborating with Joseph on computer vision algorithms, Jee Woong implemented the user interface of the Chess game. User interface of the game, written in Pygame, is the main communication program for the user, which allows the user to interact with the program, see recommendations of moves generated by the Chess AI, and moves for the Chess AI.

7.3 Budget

Table 1 is the list of materials we have used for our product. Most of the materials were used for the product only, but for some of them, we did not end up using them in our product because the purpose was for all of us to work parallel on the product since we were working from different places. We eventaully ended up buying a main chessboard and pieces to avoid having the issue of chessboard and the chess pieces having the same color. We have budgeted $346.33 for our complete product.

7.4 Risk Management

The risk factors we had in our project can be divided into two parts. The edge cases in a chess system that we could encounter and the integration of the subsystems.

There were three main edge cases in our chess system. The first edge case was not being able to detect the chess pieces on the board because they would have the same color as the board. We were able to handle this problem by using red and blue chess pieces instead of white and black chess pieces. Even though the red pieces were still similar to the black squares of the chessboard, we were able to resolve this risk by finding an ideal threshold after our background subtraction to separate the piece from the square it is on. Another edge case was not being able to detect knights through blob detection because they were did not have circle shapes from a top down view compared to other pieces which did. These pieces were able to be detected from having ideal values for the circularity and the convexity parameter of the blob detection. The last edge case was detecting the correct move when pieces are taken. If a piece takes another piece, we were not sure how to figure out which piece took which because the pieces were merely circles from the view of the camera. To handle this risk, we used the fact that we know whose turn it is to know which piece took which. We know the positions of each piece, so we know which piece is the white piece and which piece is the black piece. Based on the turn the system recognizes, it will make the piece that is appropriate take the other piece.

There were two big risk factors in integrating our chess system. The first risk factor was integrating all the algorithms through a chess system. Detecting corners of the chessboard and the blob detection being able to detect the moving piece is one thing, but it is another thing to coordinate them together. We need to identify which piece moved through the background subtraction algorithm and communicate with the board detection algorithm to figure out where it moved to. We need to set up a chess system in our code to coordinate all of the algorithms. To mitigate this risk, we were able to finish our subsystems earlier to have enough time for coordination. The other risk factor was integration of all the hardware components. The FPGA and PC must communicate with each other and ensure that this communication happens without large latency or lost packets. We have also mitigated this risk by beginning integration as soon as possible and working on components that can be integrated earlier on in our schedule. These details are specifically listed in the Gantt chart and schedule section.

8 ETHICAL ISSUES

An ethical issue that could arise from our product is teaching people wrong chess moves. If our product does not work as intended, people are going to learn chess in a wrong way, and our product is going to make people have poor chess skills. Not only that, it will waste people's time and money using our product. To mitigate this issue, we need to make sure our product has great accuracy in communicating with the Stockfish engine system and displaying the moves. The system should also be ideally reviewed by professional chess players to make sure the chess moves are optimal.

Another ethical issue that could arise from our product is that people could lose their jobs. In particular, chess tutors could lose their jobs because our product is cheaper than their tutoring expenses. To mitigate this, we could hire chess tutors to improve our product. The chess tutors have experience in tutoring for a long time. Their experience could be valuable in knowing what each person needs to learn at a certain level of experience. Hence, the tutors should be hired to improve the features of our product.

The last ethical issue we have found is violating people's privacy with camera. Since our product has a camera and a computer, with a computer's networking capabilities, our clients' privacy could be violated by a malicious insider who develops our product and adds a capability on the product to share the video with the malicious insider. To mitigate this product, we should remove all networking capabilities in our product. Without networking capabilities, the product would not be able to share anything. In addition, our product does not require networking capabilities because all the software could be pre-installed. We could also have a switch to turn on and off networking capabilities for a possible software patch.

Table 1: Bill of materials

Description	Manufacturer	Quantity	Cost @	Total
DE10-Standard FPGA	Terasic	1	ECE Owned	$0.00
Macbook Air 2020	Apple	1	Personally Owned	$0.00
Chess Sets	JAOK	3	$25.43	$76.29
Webcam	Logitech	3	$42.39	$127.17
Webcam Stand	Pipishell	3	$23.31	$69.93
Black and White chessboard	ASNEY	1	$35.99	$35.99
Red and Blue Chess Pieces	House of Staunton	1	$36.95	$36.95
				$346.33

9 RELATED WORK

There are similar projects to ours, but none of them seem to be exactly the same as ours. One example we found was DecodeChess. DecodeChess is a virtual chess teacher that goes in depth about recommending what the best move would be in each of the situations. In comparison to ours, this environment does not give a realistic environment, but it does give more feedback than our game because it also provides reasoning behind each move. Another example is a Fall 2020 capstone project from the Blokus group. They simulated a virtual environment for playing Blokus, which is also kind of similar to our project in that they detect the moves using a camera to upload it to a computer. The difference with ours is that our product goes against artificial intelligence while the Blokus group used a player to player interaction when playing the game.

10 SUMMARY

As we have shown, we believe we achieved the design specification we originally planned. A small limitation we have is that the AI can only recommend one move at a time. If we had more time to implement our project, we would have made our system to recommend various moves in a given turn. It would have also been cool if we had some way of automation to have the moves of the artificial intelligence move on its own.

10.1 Future Work

There are two directions for expanding our product. The first approach builds on our approach, and the other approach changes the whole subsystems of our product.

The first approach is to build upon our approach to make a p2p multiplayer chess game. Allowing users to compete with other users on the web application would have made out Chess Teacher much more interesting since users can use their skills learned from Chess Teacher to compete with others.

The second approach is getting rid of the computer vision aspect of our product. For this product to be commercialized, we would want the environment get in the way of the functionality of our product. Our whole setup of our product is hard. We need a camera directly on top of the chessboard, with having pieces that are not the same color as the chessboard. The environment could also be a factor in the performance of our product. If there is no lighting, our product would simply not work. If we abandoned the whole computer vision aspect of the project and simply used magnets with sensors to show which piece moved to where, it could have ignored the environment fully. I believe this could be a better approach as it removes the whole necessity of having a monitor and a computer too which makes the product cheaper.

Another possible approach would be to fully integrate the system onto FPGA. Building a UI which the FPGA can run would be the main task as well as implementing some features such a full chess engine on FPGA fabric similar to that of Hydra as referenced earlier.

10.2 Lessons Learned

We learned that system integration is often the most difficult part of the project. Although certain features were difficult to implement on their own, integrating all features together tended to be the most difficult task. Hence, we recommend for any future projects that more time is left to integration of all components.

Glossary of Acronyms

- HSP - Hard System Processor, an on FPGA CPU tightly integrated with FPGA custom fabric.
- HSV – Hue-Saturation-Value, type of color presentation.
- P2P – Peer to Peer, meaning computer systems of peers connected through the internet

References

[1] URL: https://pythonprogramming.net/pygame-buttons-part-1-button-rectangle/.

[2] *Camera Calibration¶*. URL: https://opencv-python-tutroals.readthedocs.io/en/latest/py_tutorials/py_calib3d/py_calibration/py_calibration.html.

[3] Fsmosca. *fsmosca/Python-Easy-Chess-GUI*. URL: https://github.com/fsmosca/Python-Easy-Chess-GUI.

[4] Ulf Lorenz et al. "Hydra: Report and Technical Overview". In: *ICGA Journal* 42.2-3 (Jan. 2020), 132 – 151.

[5] OpenCV. *Background Subtraction*. URL: https://docs.opencv.org/3.4/d1/dc5/tutorial_background_subtraction.html.

[6] OpenCV. *Blob Detector*. URL: https://docs.opencv.org/3.4/d0/d7a/classcv_1_1SimpleBlobDetector.html.

[7] OpenCV. *Template Matching*. URL: https://docs.opencv.org/master/d4/dc6/tutorial_py_template_matching.html.

[8] Tord Romstad, Marco Costalba, and Joona Kiiski. *Stockfish 13*. URL: https://stockfishchess.org/.

[9] David Silver et al. "A general reinforcement learning algorithm that masters chess, shogi, and Go through self-play". In: *Science* 362.6419 (2018), pp. 1140–1144. ISSN: 0036-8075. DOI: 10.1126/science.aar6404. URL: https://science.sciencemag.org/content/362/6419/1140.

[10] Terasic Technologies. *SoC Platform - Cyclone - DE10-Standard*. URL: https://www.terasic.com.tw/cgi-bin/page/archive.pl?Language=English&CategoryNo=205&No=1081&PartNo=1.

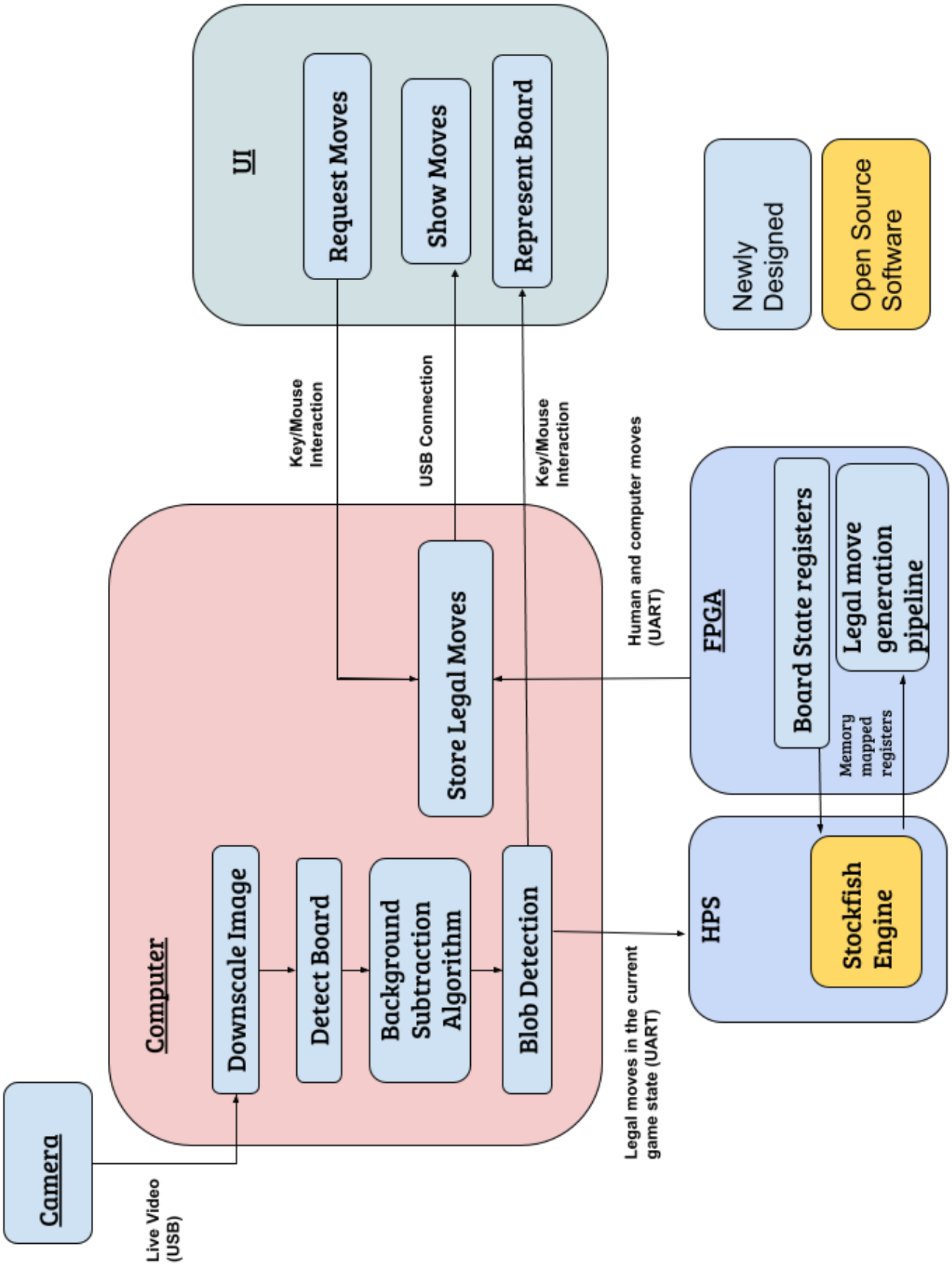

Figure 16: Block Diagram

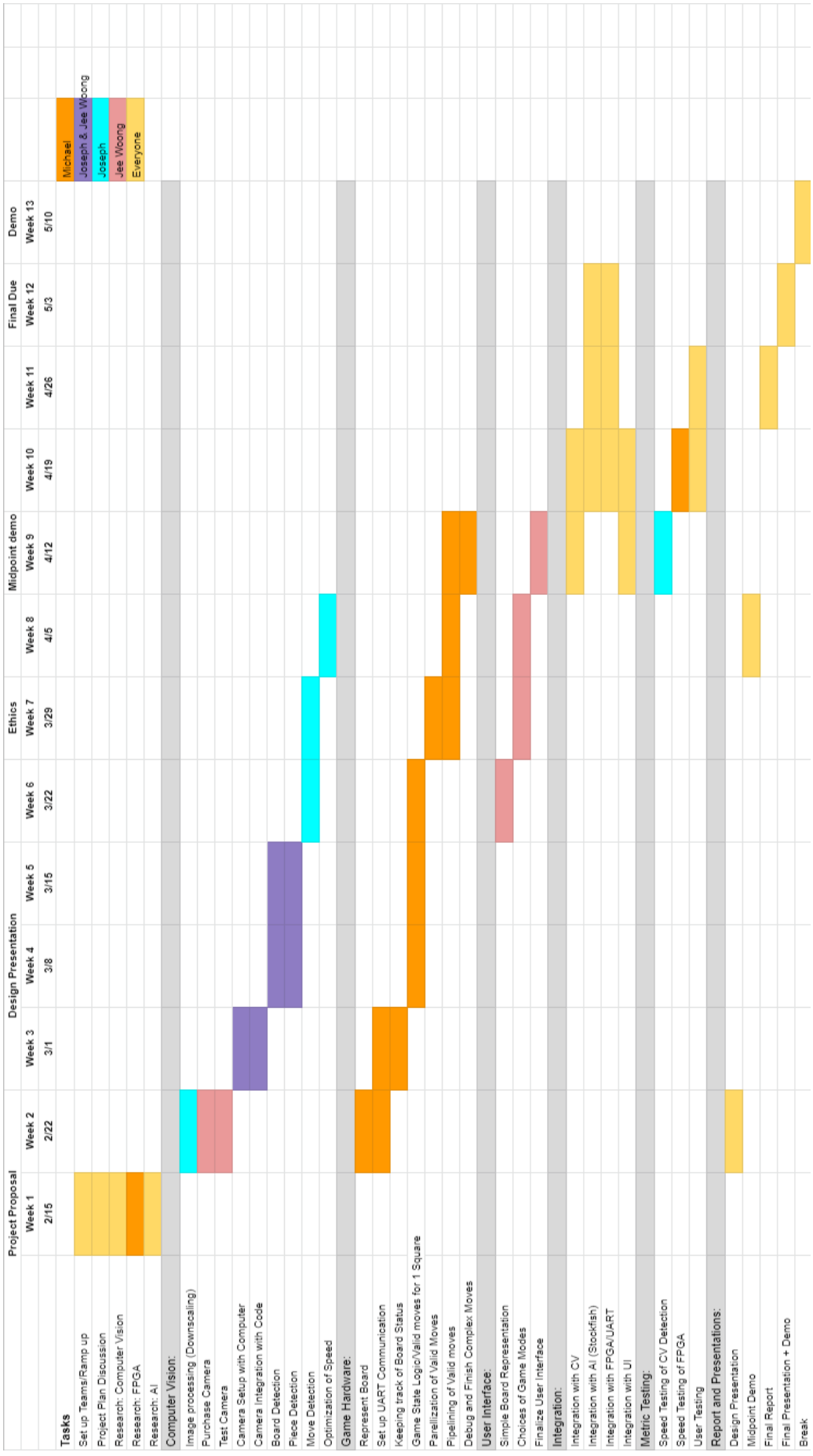

Figure 17: Gantt Chart