Team E8: Smart Poker Table

Authors: **Brandon Hung**: Carnegie Mellon University;

Zongpeng Yu: Electrical and Computer Engineering, Carnegie Mellon University;

Patrick Kollman: Electrical and Computer Engineering, Carnegie Mellon University

***Abstract*—The current standard for live poker uses the outdated method of human estimation to track valuable information about the game state, such as the overall pot size and bet sizes. This practice often leads to slower playing times and shifts a player's focus from the crucial decision making processes involved in playing towards bookkeeping, which ultimately takes away from the playing experience. We propose a system to track and display important game elements for real-life poker players. Motivated by online poker, our system will track individual bet sizes, the overall pot size, and player order through a device controlled by a Raspberry Pi. The device will include a computer vision system to scan stacks, an actuation system to direct the camera to the correct location, and a real-time display to show this information to the players. The game state will be controlled by a dealer through an intuitive UI. Our goal is to provide a simple device that a dealer can use to provide an online poker-like experience to in-person games, improving overall gameplay quality.**

***Index Terms*—Computer Vision, Mechatronics, Microcontrollers, Motor Control, Python, Servo, Systems, User Interface**

I. Introduction

The rules of poker state that all table information should be available to players at all times. Table information is crucial to competitive poker players for making informed decisions during a game. For example, knowing the stack size of other players is very important when choosing to either call, raise, or fold. If an opponent has more money than themself, they may want to rethink about raising the bet size. If the opponent has less money than themself, it may be an opportune time to step on the pedal and raise. In a real life poker match, if a player wanted to know the size of another player's stack, they would need to ask the casino dealer to physically count that player's stack chip by chip. This method is becoming outdated given the efficiency of online poker games.

Online poker games have revolutionized the poker environment by constantly displaying all table information directly to the players. In an online poker game, players can always view the pot size, player stack sizes, bet sizes, and whose turn it is to act during a round. This is the way poker is meant to be played, with all table information readily available. The goal of the Smart Poker Table is to display the same information provided in an online poker game to real life poker players. With a maximal update time of 10 seconds, the Smart Poker Table will successfully imitate the online poker environment for real life casino poker players.

II. Design Requirements

Our goal with this project is to create an environment that creates a smooth play experience like one found in online poker games; as a result, the most important requirement is providing the players and dealer with real-time, accurate information. Since there is no objective measurement of real-time speed as defined in a poker game, we have chosen a maximal update time of 10 seconds (allotting for turning, image capturing and computer vision processing, and updating the display). In person, a round of poker with a full table takes anywhere from 30 seconds to 10 minutes. Averaging round lengths gives us 5 minutes and 15 seconds; with a full table of eight, our device will require 8 * 4 * 10 = 320 seconds per round, or about 5 minutes and 20 seconds a round in the worst case scenario. An update speed bounded by 10 seconds will allow our device to keep pace with standard live poker games in the worst case.

Our metric for the accuracy of estimating the pot size will be within +/- 10% of the actual pot size. This number may seem quite low, but it is sufficient in the context of the problem. At the end of the game, the winner's payout is based on the physical chips on the table --- not our estimated pot value. Instead, the main purpose of the pot estimation is to help players make better betting choices; a rough idea of the pot size works well for this purpose, which is why 90-110% estimates are just as useful as 100% accurate ones.

Our device is intended to be easy to use and require only 5 minutes to learn for the average poker dealer. This represents a way to metric the ease of use since end users ultimately want something which works as out of the box as possible. While this is admittedly an arbitrary value, it provides a good guideline to follow on the eventual simplification/streamlining of the design. As for the computer vision, we would like the image capture and processing pipeline to occur within 3 seconds. This will provide us ample time to meet our maximal update time listed in the first paragraph. Turning the camera to the correct position using a servo will need to occur within 5 seconds to fit alongside our computer vision processing time and maximal update time requirements.

Here, we'll address the more physical requirements of the system. The servo will need to turn within a 4.735 degrees of our reference stack reading position. This translates to a worst case scenario (when the stacks are 48 inches away) of +/- 4 inches of deviation from the horizontal line of our stack reading position (arctan(4/48) = 4.735 degrees). Anything

beyond this point will result in the camera pointing in the wrong direction and being unable to read the stack, which will result in a completely incorrect bet size. Finally, our camera will need a minimum resolution of 63x460 pixels. This is derived from our window size of 48x48 square inches and a chip area of 1.53x0.209 square inches, with the formula $axisResolution = 2\,x\,FOV\,/\,minSize$. Plugging in the values 2*(48/1.53) and 2 * (48 /0.209) gives us a minimum bound of 62.74 and 459.33 pixels in the x and y directions, respectively.

Finally, we want our internal game state to perfectly match the current state of the game in real life. These two must be exactly the same or else synchronization issues will occur in the software.

III. Architecture and/or Principle of Operation

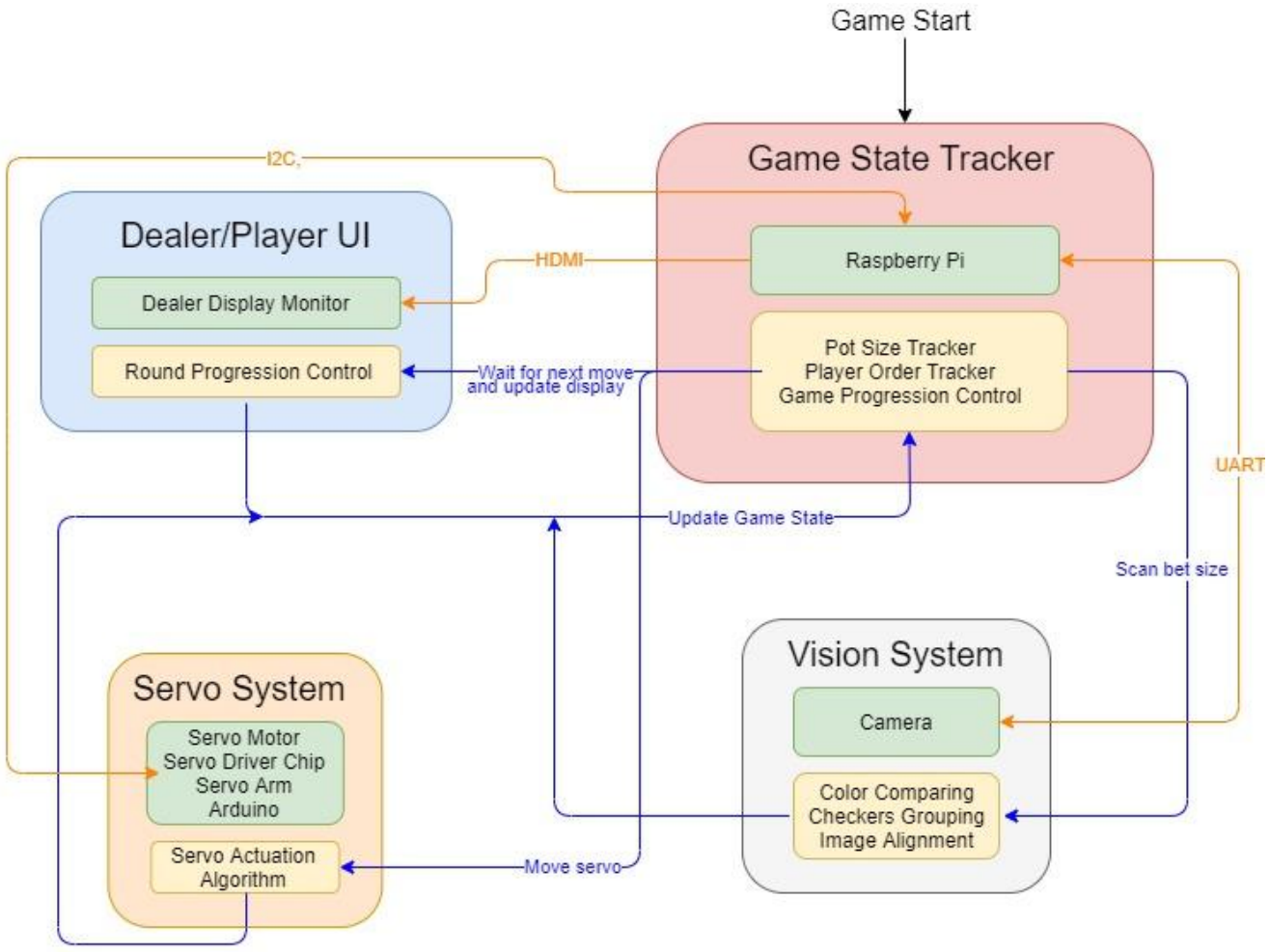


Fig. 1 Overall System Architecture

Broadly speaking, we can divide our project into four main subsystems: the dealer/player UI, the game state tracker, the servo system, and the vision system. Since our design presentation, we have decided that the vision and servo systems were disconnected enough to warrant their own subsystems and separated them from our original hardware subsystem. In the diagram above, each system has hardware component(s) in its green box and software component(s) within its yellow box. The orange arrows show the connections between different hardware components, and the blue arrows represent the conditions and order in which software functions are called. In addition to the block diagram above, we have provided two more block diagrams at the end of the page. The Software Block Diagram details specifically the interplay between the dealer UI and the game state tracker, while the Communications Block Diagram describes in detail the protocols used to communicate between different devices/drivers.

When a game is started, the game state tracker is initialized. The game state tracker (or GST, for short) then initializes the Dealer/Player UI, the Vision System, and the Servo System. Control is then handed to the Dealer UI, which waits for the dealer's input to start a round or to configure the system. Once the dealer starts a round, the "Play Round" loop in the Software Block Diagram begins. During this loop, the Dealer UI collects information regarding players folding and/or betting and transmits it to the GST, which then hands over control to the appropriate subsystem. If a player folds, the GST removes them from the order tracking and proceeds to the Dealer UI. If a player raises or calls, the GST will first proceed to the Servo System. Once the Servo System has moved to the appropriate position, control is returned to the

GST which in turn invokes the Vision System. After the Vision System collects the stack size information, it transmits the information back to the GST to update the state and repeat the cycle. For a pseudocode example of this process, please refer to Fig. 12.

The hardware architecture consists of a Raspberry Pi, a camera, a display monitor, a servo motor / servo driver. The Raspberry Pi serves as the main controller of the system, to which the peripherals are connected. To send and receive image data, the Pi communicates with the camera over UART. To actuate the servo with positional feedback, the Pi uses I2C to communicate with the servo driver which in turn uses PWM to drive the servo itself. The monitor is updated over HDMI as the game progresses. Most of the design is abstracted behind libraries, allowing us to focus more on the algorithms than connecting the subsystems.

IV. Design Trade Studies

A. *Stack Scanning: Computer Vision vs RFID*

One of the biggest design decisions we made as part of our design process was choosing to go with computer vision instead of RFID for the stack height reading. The most prominent reason for this change is feasibility. While we have little actual data on the feasibility of scanning a stack of chips using RFID, feedback from the professors generally suggested it was a nigh impossible task to achieve. While there are a few examples of RFID being used to scan items quickly (RFID Blog, 2020), our problem is fundamentally harder due occlusion from RFID tags. This is less of a problem when scanning items such as the clothes mentioned in the above article, but can prove to be a strong issue when dealing with stacked playing chips. Even the most sophisticated leaders in RFID technology can struggle with scanning thin stacks of more than 25 chips (RFID Journal, 2013). In contrast, computer vision can scan chips using fairly simple methods if properly structured. In the end, we decided to go with a tradeoff of a method that is more likely to provide a guarantee of being able to scan stack heights to some accuracy over experimenting with a relatively unknown technology potentially unable to fulfill our requirements.

B. *Servo Subsystem: Continuous vs Regular*

Another design decision we made was choosing between a standard 180 degree servo and a continuous rotation servo. A 180 degree servo tends to be fairly positionally accurate and easy to use; the ability to simply write an angle to the motor is built into the driver chip we initially wanted to use. Meanwhile, a continuous rotation servo provides 360 degrees of rotation but is more complex in terms of control. Additionally, its feedback is not readable by a Raspberry Pi so using this component requires introducing an Arduino to communicate between the motor and the Pi. Our device placement on the poker table (see Fig. 14, Model of Smart Poker Table Setup) also depends on what motor we choose, as the range of rotation limits how far we can place the device from the dealer. In the end, we chose to go with continuous servo. We initially chose the continuous servo because it allowed us to place the device in the middle of the table and close to the players --- thereby reducing the difficulty of scanning stacks. We believed this would translate into an increased accuracy, making the continuous servo well worth the extra complexity involved with implementation. In the end, we chose to only use a rotation of 180 degrees, and to place the servo at the edge of the table. We had no time to order the non-continuous servo, so we just stuck with the servo we had.

C. *Hardware Components Consolidated vs Distributed*

Deciding between a placement of components in which parts were distributed versus a placement that consolidated the parts in one area actually proved to be quite difficult. Distributed components allow us to hide the less visually appealing electrical components. This would create a prettier setup which is less distracting from a standard poker environment. However, this method creates complex wiring schemes that increase both the chance of failure (i.e a wire getting cut) and the difficulty of setup and troubleshooting (since devices must be individually repositioned and inspected). On the other hand, consolidating the various components as much as possible allows us to shorten wires and more quickly identify any hardware issues that crop up during gameplay --- but doing so involves more mechanical design and construction, as seen in the rendering of figure 3. Our choice to go with consolidated components is based on the decision to invest more in design, as we believe that a good design preemptively mitigates many risks associated with possible system failure later on.

D. *Computer Vision: Blob Detect vs Checker Counting*

The most major trade-offs we made were whether to use the blob detection algorithm or the checker counting algorithm for scanning stack sizes. The blob detection algorithm is a straightforward, easy to implement algorithm which simply matches a color to a blob and finds the minimum bounding box around said color blob. However, as seen in table 1, its performance was far less accurate. This is because no matter what color space representation we used (HSV, RGB, or LAB), we found that lighting conditions had a noticeable effect on color interpreted by the algorithm. Tuning the algorithm to match colors and values properly, as well as getting it to avoid counting random pixels as part of the stack, ended up being extremely difficult; as a result, the algorithm interpreted stack heights wildly incorrectly. On the other hand, the checkers counting algorithm (also in table 1) was more accurate but extremely complex to implement. So, we had to make the tradeoff of time and complexity versus accuracy. In the end, we chose to implement the checker counting algorithm. While it was not sufficient to get us to our desired metric, it still performed far better than blob detection.

E. *User Interface: Combined or Separate*

A minor tradeoff we encountered early on was whether we should separate our dealer and player UI screens. Initially, we felt that the dealer might have information they would want to

keep hidden from the rest of the players and intended to separate them. However, the limitations of the Pygame library [5] prevented us from keeping these windows separated. While deciding between using a different library or using one window for both UIs, we concluded that there isn't any information the dealer needs to keep from the players and elected to have the dealer UI and player UI use the same display for simplicity.

F. Controlling Overall System Variables

Our Computer Vision algorithm is very dependent on color temperature and chip distances. To fix the issue of chip distances being inaccurate, we mandated bet positioning for each player on the poker table. On the Smart Poker Table, there are blue strips of tape indicating where each player should place their bet. Each strip of tape is the same distance from the camera. In addition, we maintained a constant color temperature by placing a high intensity white lamp over the poker table. This allowed us to see consistent color detection from our CV algorithm. For an example of this setup, please refer to figure 13.

V. System Description

A. Subsystem: Computer Vision

The computer vision subsystem includes a USB camera, an algorithm which takes a picture and scans the stack size in the captured photo, and an algorithm to calibrate the chip sizes and mapping of the value of a stack to its color. The code for the stack size scanning algorithm is given in figure 2.

First, the algorithm takes a photo. Then, it looks for white checkers using the intensity of the image. Once it has found all the white checkers, it clusters them into groups based on a modified version of k-nearest neighbors. It then calculates a minimum bounding box across each group of checkers to estimate the warped stack height. Then, a line is fit through the bottom of each bounding box using the least squares function from numpy [7] to estimate the rotation of the image. The inverse rotation is applied to fix any tilt; from there, the algorithm runs again to compute and aggregate the checkers. The final resulting bounding boxes are used as the height of the stack in pixels; dividing by the average chip height gives the number of chips in the stack. Finally, the algorithm uses k-means provided by OpenCV [6] to find the dominant color in each image and matches it to the color-value mapping produced in calibration for a value. The height of a single stack is multiplied by the value of a single chip to get the total value for that stack, with each stack value being summed to calculate the total bet size.

```python
def stack_values(dat, rectList, cut_frame, debug=False):
    totVal = 0
    # Want to match the values of the colors to the right stack
    if debug:
        cv2.namedWindow("debug")

    listOfColors = [dat.colorAssociation[key] for key in dat.colorAssociation]
    listOfIntensities = [dat.intensities[key] for key in dat.intensities]
    # weight = 100
    similarRange = 25
    for i in range(len(dat.values)):
        # First, check the stack value
        x1 = rectList[i][0]
        x2 = rectList[i][1]
        y1 = rectList[i][2]
        y2 = rectList[i][3]
        height = y2 - y1
        section = cut_frame[y1:y2, x1:x2]
        secColor = get_color_dominant(section)
        var = np.linalg.norm(secColor - np.mean(secColor))
        inten = intensity(secColor)
        secColor = bgr2hsv(secColor)
        # Find the closest remaining color
        minError = 10000000
        # minInten = minError
        minIndex = -1
        ind = 0
        # print("Color Comp: ", secColor)
        for color in listOfColors:
            intColor = listOfIntensities[ind]
            tempError = np.linalg.norm(secColor - color)

            if var <= similarRange:
                tempError = np.abs(inten - intColor)

            if tempError < minError:
                minIndex = ind
                minError = tempError
            ind += 1
        chipVal = dat.values[minIndex]
        chipH = dat.chipHeight
        value = chipVal * np.round(height / chipH)
        if debug:
            print("Detected Color: ", secColor)
            print("intensity: ",  inten)
            print("Closest Color: ", dat.colorAssociation[minIndex])
            print("Stack Height: ", height)
            print("Value: ", dat.values[minIndex])
            print("Estimated Height: ", np.round(height / chipH))
            print("Total Value: ", value)
            while True:
                cv2.imshow("debug", section)
                k = cv2.waitKey(1) & 0xFF
                if k == 13:
                    break
        totVal += value
    return totVal
```

Fig. 2 Stack Scanning Algorithm Code

The mechanical assembly is shown in Figure 4, while the rendering is given in Figure 3. The camera is mounted to a 3D printed rod with a ¼"-20 thread. The other end of the rod is mounted to the servo horn. The servo, Arduino, and Raspberry Pi are arranged on the box-like mounting assembly to keep the components close to one another.

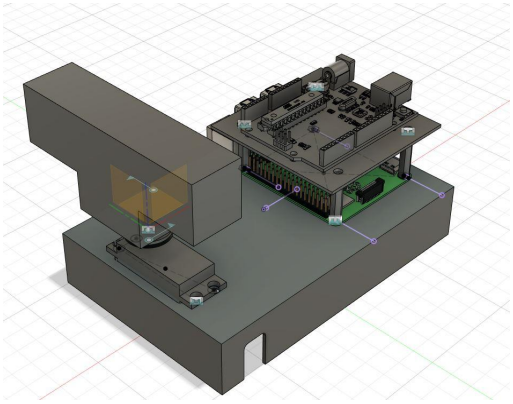


Fig. 3 Servo and Camera Rendering

B. Subsystem: Servo

The servo subsystem includes a 360 continuous rotation servo with feedback from Adafruit, 6 volt power supply, Arduino Uno, and Raspberry Pi.

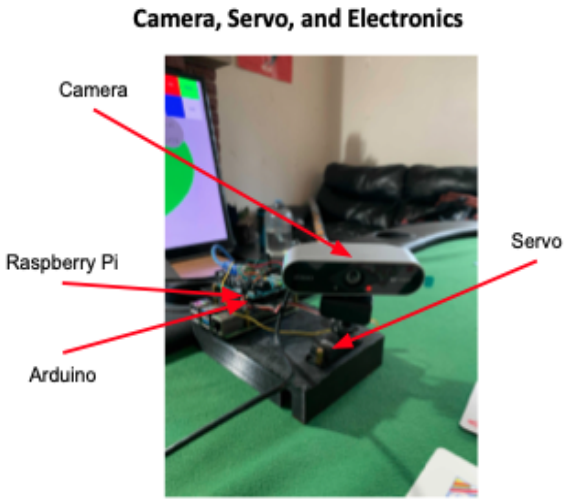


Fig. 4 Integrated Assembly

To start, we connect 360 servo to arduino. The reason why we didn't connect servo directly to Raspberry Pi is because the Pi lacks the capability to read PWM from a GPIO. However, the Arduino could do that because it has PWM pins designed to receive feedback, so not only are we able to get the 360 continuous servo rotating with different RPMs, we could also set the angle where we want the servo to rotate to. The library we are using to actually control the servo angle is the Parallax-FeedBack-360-Servo-Control-Library-4-Arduino [4].

In order to make the servo rotate to the correct position, the Arduino will need to receive a signal from the Raspberry Pi containing the angle to turn to. Once received, the Arduino will then begin spinning the servo. After it has reached the target position, the Arduino will transmit a confirmation signal to the Raspberry Pi, allowing the GST to call the Vision Subsystem. For the communication protocol, we will use USB, and we will import the python serial library on the Raspberry Pi and use the builtin serial function for Arduino.

```
//servo control code snippet on Arduino
void loop(){
  int angle;
  if(Serial.available()){
    String data =
Serial.readStringUntil('\n');
    angle = data.toInt();
    servo.rotate(angle, 4);
    Serial.println(angle);
  }
}

// Pi to Arduino Serial (python)
import serial
ser = serial.Serial('/dev/ttyUSB0', 9600)
string = str(Playerlist[currP].angle)
ser.write(string)
```

Fig 5. Code Snippet for Servo Subsystem

The complete servo subsystem diagram is in Fig. 21 under Section *Servo Subsystem Diagram*.

C. Subsystem: Dealer UI

The Dealer UI has two functions: system configuration and game state updates. The Dealer UI is meant to be used by non-engineers so it is designed to be very simple and user friendly. It consists of simple controls to calibrate the system and control the game state.

System configuration can occur in between any round, but usually happens at the very beginning of a game. With system configuration, the dealer can enter the chip configuration screen, the player addition and removal screen, the player stack sizes screen, and choose when the game begins. These updates are then relayed to the GST.

Fig. 6 System Configuration Interface

In Fig. 7, the player addition and removal screen is shown. The dealer can click on the various circles to add players to that location on the table. Clicking again on that circle will remove the player from the game. The dealer can also input the name of the player when prompted.

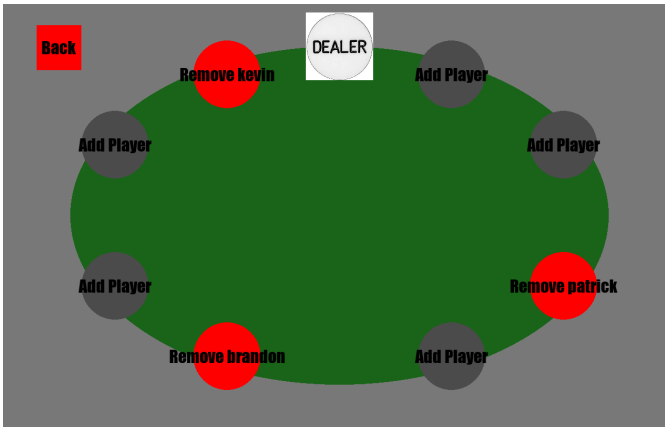


Fig. 7 Player Addition and Removal Screen

In Fig. 8, the player stack sizes screen is shown. Here, the dealer can edit the stack sizes of the current players in the game. Each player starts with a default of $200, but the dealer can edit that number to be anything the player wants.

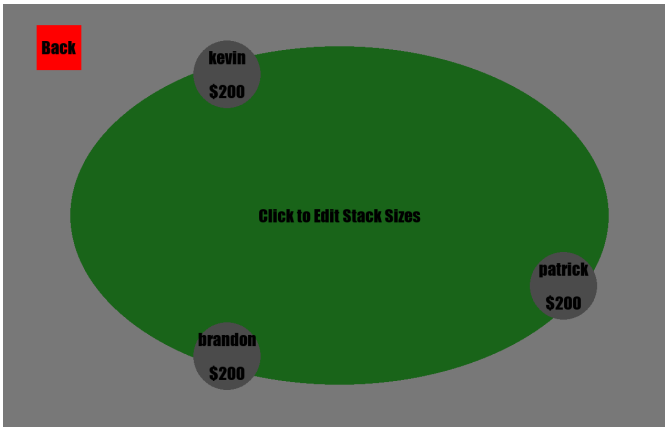


Fig. 8 Player Stack Sizes Screen

In Fig. 9, the chip configuration screen is shown. In our poker game, we allow for 5 different chip types and their values can be edited here. In addition, the big blind value for the game can be edited here as well. There is no function for editing the small blind because it will always be half of the big blind. In the top right corner of the screen there is a calibrate button which begins the Computer Vision calibration routine.

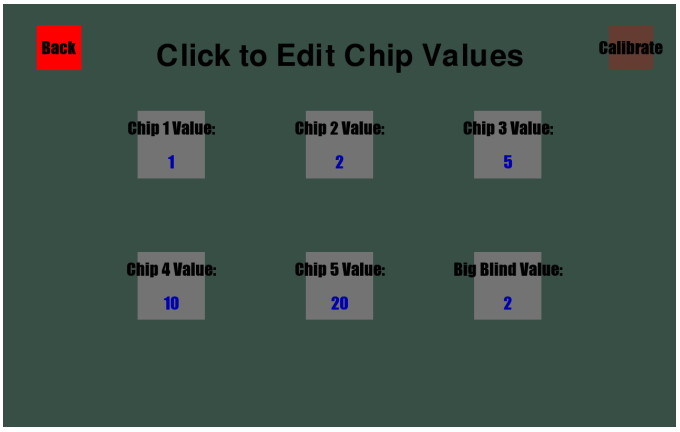


Fig. 9 Chip Configuration Screen

In Fig. 10, the game screen is shown. The player highlighted in light green is currently in their turn to act. The small blind position is denoted with a blue, small blind chip image and the big blind position is denoted with a yellow, big lind chip image. The size of the pot and current round of the game (preflop, flop, turn, river) is displayed in the middle of the screen. In the top right corner, the dealer has 4 options to choose for each player: fold, check, raise, call. When the dealer makes these updates, they are relayed to the GST and the game state is updated.

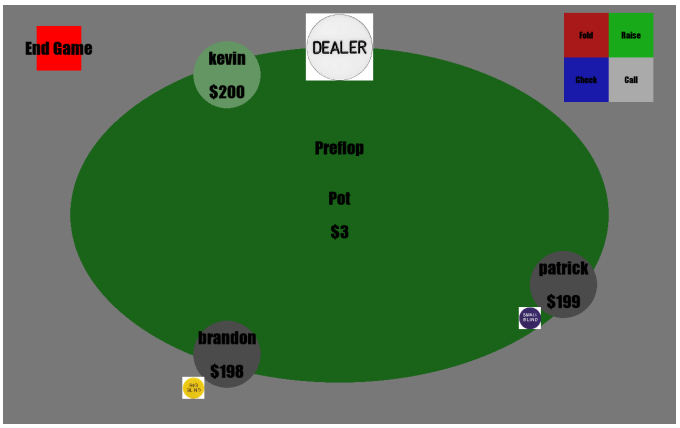


Fig. 10 Game Screen

D. Subsystem: Player UI

The Player UI is a graphical user interface for the players at the table. The purpose of the Player UI is to display all relevant table information to the players during the game. The player UI communicates with the GST and retrieves all the necessary information from its modules. Essentially, the player UI is displaying the same game screen shown in Fig. 10. This screen is separate from the Dealer UI game screen, but they are both displaying the same thing. We will connect a monitor to the Raspberry Pi via an HDMI cable to display this graphical interface.

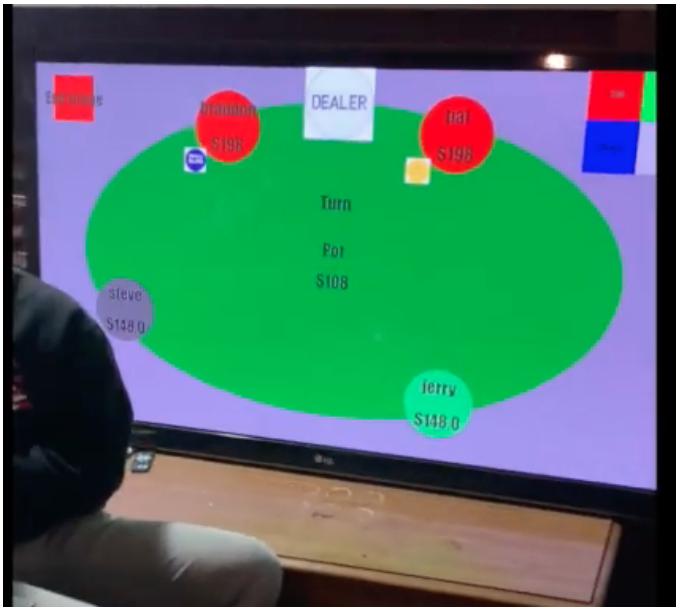


Fig. 11 Player UI

E. *Subsystem: Game State Tracker*

The Game State Tracker is the nucleus of our project. The role of the GST is to first and foremost to track the game state of the match, and relay this information to the UI. It's also responsible for the transfer control between the UI, Servo System, and Vision System.

Once initialization has occurred and a round has begun, the order in which the GST transfers power between subsystems proceeds as follows. First, the GST polls the Dealer UI for a player action. Given this action, the GST will update its internal game state then possibly transfer control to the Vision subsystem to update stack sizes, but then it will always need to transfer control to Servo subsystem to rotate to the next player. Then, the cycle repeats and GST will poll the Dealer UI for the next player's action. This flow within the software is depicted in Fig. 18 at the end of the document, and a pseudocode algorithm is given below:

```
GST.init()
dealerUI.init()
playerUI.init()
servo.init()
camera.init()
while !(exit):
    next_move = dealerUI.poll()
    GST.round_start = false
    switch next_move
        case add_or_remove
            dealerUI.add_remove_players()
            playerUI.update()
        case start_round
            Gst.round_start = true
        case calibrate_chip_colors
            cam.calibration_routine()
    while (GST.round_start):
        player = player_order.get_next()
        if player == null:
            GST.next_phase()
            continue
        input = dealerUI.poll()
        switch input
            case fold
                GST.remove(player)
            case bet
                servo.move(player)
                bet_size = cam.scan_stack()
                GST.pot_size += bet_size
        GST.update_player_order()
        dealerUI.render()
        playerUI.update()
    GST.calculate_payout()
    update_player_UI()
```

Fig. 12 Abstracted game state tracker pseudocode

F. *Overall System*

The overall system combines all the previous subsystems: the Dealer/Player UI, the Game State Tracker, the Servo subsystem, and the Computer Vision subsystem. The camera is placed in front of the dealer's seat, and rotates to the current player acting in the game. If the player raises, it's bet size is scanned and the GST will update the game state. These actions will need to be inputted by the dealer though the dealer UI. The servo will then rotate to the next player and the cycle will repeat until the round ends. At the end of the game, the dealer will be prompted to select the winner and that player will win the pot.

Fig. 13 Physical Smart Poker Table

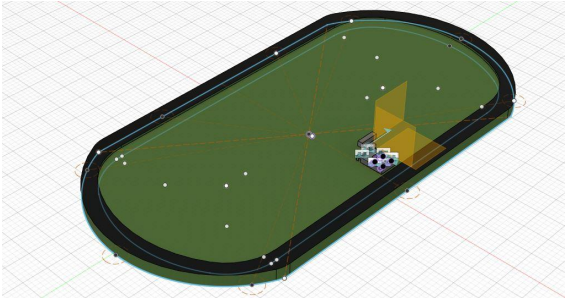


Fig. 14 Model of Smart Poker Table Setup

VI. Test and Validation

A. *Results for Servo Subsystem*

For testing and validation for the servo subsystem, we did two tests in order to meet our original specifications. The first metric is that servo has to rotate to the correct target within +/-4 degree offset. The second metric is that the camera rotates to a specific position within 5 seconds.

For both tests, we used the same inputs: the player angle that is defined in Player Class in the dealer software. We did 20 trials in total. For the angle test, we used a protractor to see what the difference is. For the timing test, we simply used a timer.

The result for the angle test is the average error was within 1 degree, which beats our expectation. For the timing test, on average, the servo was able to rotate to the correct position in 1 second, which also beats our expectation.

Servo Subsystem Test Result

Original Metrics	Test Inputs	Method	Result
Camera Angle offset +/-4 degrees	Player position sent by dealer SW through serial	Compare the result with the protractor; 20 trials	Average error was 1 degree from target
Camera rotate to spot with 5 secs	Player position sent by dealer SW through serial	Use a Timer (20 trials)	Average time within 1 seconds

Our measurements beat our original predictions by a huge margin. Originally, we were concerned about whether the servo could provide enough torque and support to rotate the camera. However, the testing shows that the power supply from Arduino itself is enough to provide enough torque. Another change we made was that we placed the servo at the edge of the table to not only mimic the player positions in the Player UI but also avoid the issue of cables getting tangled up while the servo rotates more than 360 degrees.

B. *Results for Computer Vision*

The tests we performed to measure the accuracy of the stack measuring algorithm involved placing a different amount of chips in front of the camera with individual values of 1, 2, 5, and 10. We ended up testing four different combined chip values of $46.00, $84.00, $150.00, and $210.00.

The performance difference between the original algorithm, which relied on blob detection (denoted by the B.D.), versus the method we created dubbed checker counting (abbreviated by C.C.), is presented in the table below:

Table 1. *Algorithm Comparison Test Result*

CV Algo	Stack Val.	Avg Error (%)	# Tests
C. C.	$46.00	6.49%	10
B. D.	$46.00	25.47%	7
C. C.	$84.00	8.68%	10
B. D.	$84.00	30.61%	7
C. C.	$150.00	18.48%	10
B. D.	$150.00	47.29%	9
C. C.	$210.00	22.46%	10
B. D.	$210.00	52.38%	9

Fig. 15: Results of Comparing Computer Vision Algorithm Performance

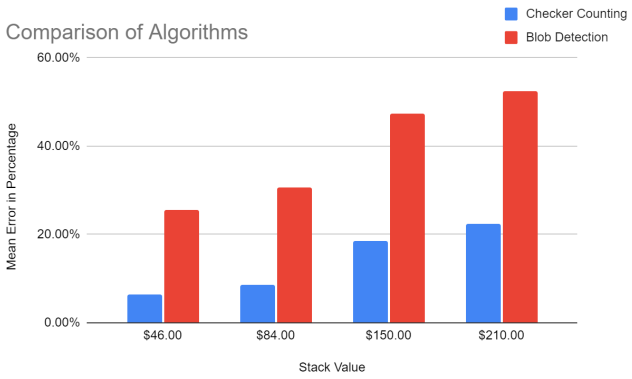


Fig. 16: Table 1 Data, Represented as Bar Chart

Note that we have a different number of tests for the B.D. algorithm since the data was taken at an earlier iteration of the code that we were unable to properly set up again. In any case, the data suggests that the rewritten algorithm performs better for each stack value and combination tested. The C.C. algorithm performs far better with small stack values than it does for large ones, with the error percentage growing faster than the stack value.

The averaged mean error when measuring stacks, in percentage, is presented below. Given the tests were measured on different stack heights, we have elected to leave out standard deviation as it would not represent the proper statistic here.

Table 2. *Stack Scanning Test Results*

Algorithm Type	Mean Stack Error
C. C.	17.38%
B. D.	44.56%

Fig. 17: Overall Mean Stack Error Measured by Algorithm

This figure reaffirms the notion that the checker counting algorithm performs better than the blob detection algorithm.

Our initial requirement was to have the measured stack value be within +/- 10% of the real stack value. Based on our results, we did not achieve our metric as intended. Although the algorithm could meet the requirements fairly consistently for stacks with a small value, it was not able to do so for larger stack values.

There were several challenges that made this problem very non-trivial. The first issue we ran into was different chip sizes. People do not place stacks at exactly the same locations, so some stacks appear smaller or larger in the image. While this can normally be handled by computing a homography between the corrected image and a reference image, since each stack was in reality not placed in a line. This caused the transformed images to have unusual warping detrimental to stack counting. Moreover, a monocular camera on a non-translating platform cannot get depth without a reference; as a result, it was fairly difficult to estimate the depths of each stack. To simplify this problem, we made the assumption that all chips were roughly the same distance away and used an averaged chip height and width amongst the different values for chips. This resulted in some stacks being measured as larger or smaller than they were in reality.

Perhaps the biggest issue we ran into was finding the correct chip height in pixels for a single chip. Most of the algorithm hinged on a fairly good estimate of chip height. The checkers were expected to be around the chip height, and the stack height in pixels would be divided by the chip height in pixels to calculate the number of chips in a stack. Acquiring this automatically was impossible; as a result, finding it became part of the calibration process. The problem is that calibrations were done by hand. Due to the tilts in the calibration images (which could not be fixed until a chip height was inputted), it is quite difficult to select the correct chip height. As a result, this uncertainty is baked into the system. As far as we are aware, there is not an easy way to resolve this issue.

C. Results for Software

The main requirement for the software components was that the GST and the Dealer UI would have a 100% accurate simulation of a poker game. This metric just means that there are no logical errors with the gameflow of the poker game. After robust gameplay testing and intentionally trying to break the software, no bugs were found. The Dealer UI acts as it is supposed to and the poker logic stays consistent and holds true. Therefore, our software succeeded in passing this test and met the 100% accurate simulation metric.

In addition, we also required our Dealer UI to be easily learnable by non-engineers. The Dealer UI will be handled by casino employees or recreational poker players who should be assumed to not have previous software engineering experience. This is why we required the Dealer UI to be learnable in under 5 minutes. After introducing the software to 5, non-software engineering individuals, the average time to learn and master the Dealer UI was 2 minutes and 55 seconds. As a result, the Dealer UI passed the requirement of being learnable in under 5 minutes.

D. Results for Overall System

Our main requirement for the overall system was an update time of 10 seconds. These 10 seconds include the camera scanning chips, the servo rotating the camera, and GST updating the display. Our overall system shattered this requirement, and consistently updated in approximately 1 second per update.

The other requirement for our overall system was that previous subsystems metrics hold for the overall system. The software metrics and servo metrics held constant, but there were a few deviations with the Computer Vision subsystem. As discussed in the design trade offs, the Computer Vision algorithm is very dependent on the positioning of chips and lighting in the room during the scanning routine. As a result, it is more difficult to control these variables when playing a real poker game versus when controlling these variables in a subsystem testing environment. When chip positioning and lightning were held constant during overall system testing, the previous subsystem metrics for CV were held.

VII. Project Management

A. Schedule

We organized our schedule into 3 phases: Brainstorming/Designing, Development, and Finalization. Right now we are nearing the end of the Brainstorming/Designing phase. The remaining tasks in this phase is to finalize our design choices in this document, and wait on the shipment of outstandings part orders.

The next phase will be the Development phase. Once each team member receives the parts they need to begin working, this phase will ramp up and include the bulk of our work throughout the semester. The Development phase entails setting up our Raspberry Pi environment, and developing each subsystem of our project within this Raspberry Pi environment. At the end of the development phase, we left time to thoroughly test each subsystem before the Finalization phase.

The Finalization phase could also be called the Integration phase. In this phase, our team will integrate each subsystem together and develop a final product. Our team will mainly be working virtually so we need to be very conscious about how we integrate our subsystems and develop interconnections. We believe this task isn't the least bit trivial, so we left a good portion of time at the end of semester to make sure this phase goes smoothly.

A full diagram of our schedule is depicted below in Figure 19, under the *Schedule* section.

B. Team Member Responsibilities

Each team member has more or less taken ownership of the subsystem they specialize in. Brandon has most background with Computer Vision, so he has taken the lead on developing

the algorithm for scanning stacks. Zongpeng has the most background on hardware, so he has taken the lead on developing the Servo subsystem and writing the firmware for the Servo. Patrick has the most background on software development, so he has taken the lead on writing the Dealer User Interface and the Player User Interface..

The overarching subsystem that the entire team will be working on together is the Game State Tracker. The GST is the central part of the project which transfers control from subsystem to subsystem, so it is necessary that each team member contributes significantly to its architecture and development.

Lastly, the entire team is responsible for the overall system construction. This entails putting all the subsystems together and developing a product that will be user ready and presentable to the stakeholders/professors. The construction of the overall system will require knowledge of individual subsystem interconnections, so it is necessary that each team member contributes significantly to its production.

C. Budget

Our Bill of Materials can be found in Figure 20.

D. Risk Management

Component	Risk
Servo Motor	Motor provides enough torque; power supply; servo doesn't burn; servo burned
Computer Vision	Lighting changes, image noise
Software	Bugs and undefined behavior
Hardware	Successful hardware integration

For servo, to mitigate the risk, we could purchase additional hardware. For instance, if the servo burns out we could purchase another one; if the power supply is too small to provide enough torque, we could purchase a higher wattage power supply. These risk mitigation plans will work because we currently only used ⅓ of our budget, giving us room to buy replacements.

On the computer vision side, there are several ways we can mitigate inaccuracies in scanning. One way is to implement image pre-processing techniques such as blurring or closing to reduce the amount of noise. Another way is through the calibration routine. By having it accessible at the beginning of every round, the dealer has the ability to counter changes in lighting or chip color. While these methods aren't a guaranteed way to fix a catastrophic failure, we believe they are suitable enough to mitigate the smaller risks associated with the CV.

Unfortunately, there isn't much we can do to mitigate undefined behavior in the software other than rigorous testing. The best way we can deal with a crash is to crash gracefully and ensure the program restarts quickly. That being said, rigorous testing should allow us to catch and eliminate enough of the bugs to where this won't be an issue, especially given how little access an end user has to the underlying software subsystems.

One approach to hardware risk mitigation is simply to have multiple components ready for replacement; this way, if something fails we can quickly replace it. Combining this with a thorough electrical schematic and reading of each data sheet should allow us to wire up the system without likely risks of failure.

VIII. Ethical Issues

Our product is a smart poker table that uses computer vision to track bet sizes and displays pot sizes and stack size onto a monitor in a casino. The main source of errors comes from Computer Vision, and there are a couple implications that come with using a camera and computer vision in a casino.

The first implication is that the computer vision algorithm is not sophisticated enough to be 100% reliable. One thing that could cause computer vision to fail is lighting. One of our constraints is that we have to keep the lighting source consistent, which is easy when casinos are indoor. However, once we move outside, we can't guarantee that we will have constant light temperatures. Another constraint we have is that players need to position chips at designated areas. Players can manipulate their stack and exploit this constraint in order to trick computer vision algorithms to their advantage.

Moreover, once the computer vision algorithm fails, it will reflect incorrect information on the display, which could be misleading to players, cause mayhem, and slow down the game, which ultimately will affect both the casino's profit and the player's profit. In a situation where incorrect information is displayed, poker players' experiences and casinos' reputations and revenue streams will both be adversely affected. This may result in legal issues where players sue the casino or the casino sues our team for inaccuracies in technology.

Another ethical issue that comes with the Smart Poker Table is privacy. Using a camera for the smart poker table will mean players at the table and individuals walking around the table will be caught on camera. This threatens these individuals' privacies. Therefore, casinos will have to get the permission of poker players and other individuals in the casino in order to use the camera at the tables. This means casinos will not only need to attain additional clearances and permits in order to use our product, they also have to dedicate additional cost to security so that people's privacy is protected.

Glossary of Acronyms

- CV - Computer Vision
- GST - Game State Tracker
- RPi - Raspberry Pi
- C.C - Checker Counter
- B.D. - Blob Detection
- UI - User interface
- Fig - Figure

References

[1] RFID Blog. (2020, June 11). UNIQLO announced the introduction of RFID tags in 3000 stores worldwide during the year. rfidcard.com. https://rfidcard.com/uniqlo-announced-the-introduction-of-rfid-tags-in-3000-stores-worldwide-during-the-year/

[2] RFID Journal. (2013, March 10). Can I Scan Multiple RFID Tags Simultaneously When They Are Kept in Alignment? rfidjournal.com. https://www.rfidjournal.com/question/can-i-scan-multiple-rfid-tags-simultaneously-when-they-are-kept-in-alignment

[3] Poker Live News. https://pokerlivenews.com/free-online-poker-games/

[4] Parallax-FeedBack-360-Servo-Control-Library-4-Arduino https://github.com/HyodaKazuaki/Parallax-FeedBack-360-Servo-Control-Library-4-Arduino

[5] Pygame Documentation https://www.pygame.org/docs/

[6] OpenCV Documentation https://docs.opencv.org/master/

[7] Numpy Documentation https://numpy.org/doc/stable/user/index.html

Software Block Diagram

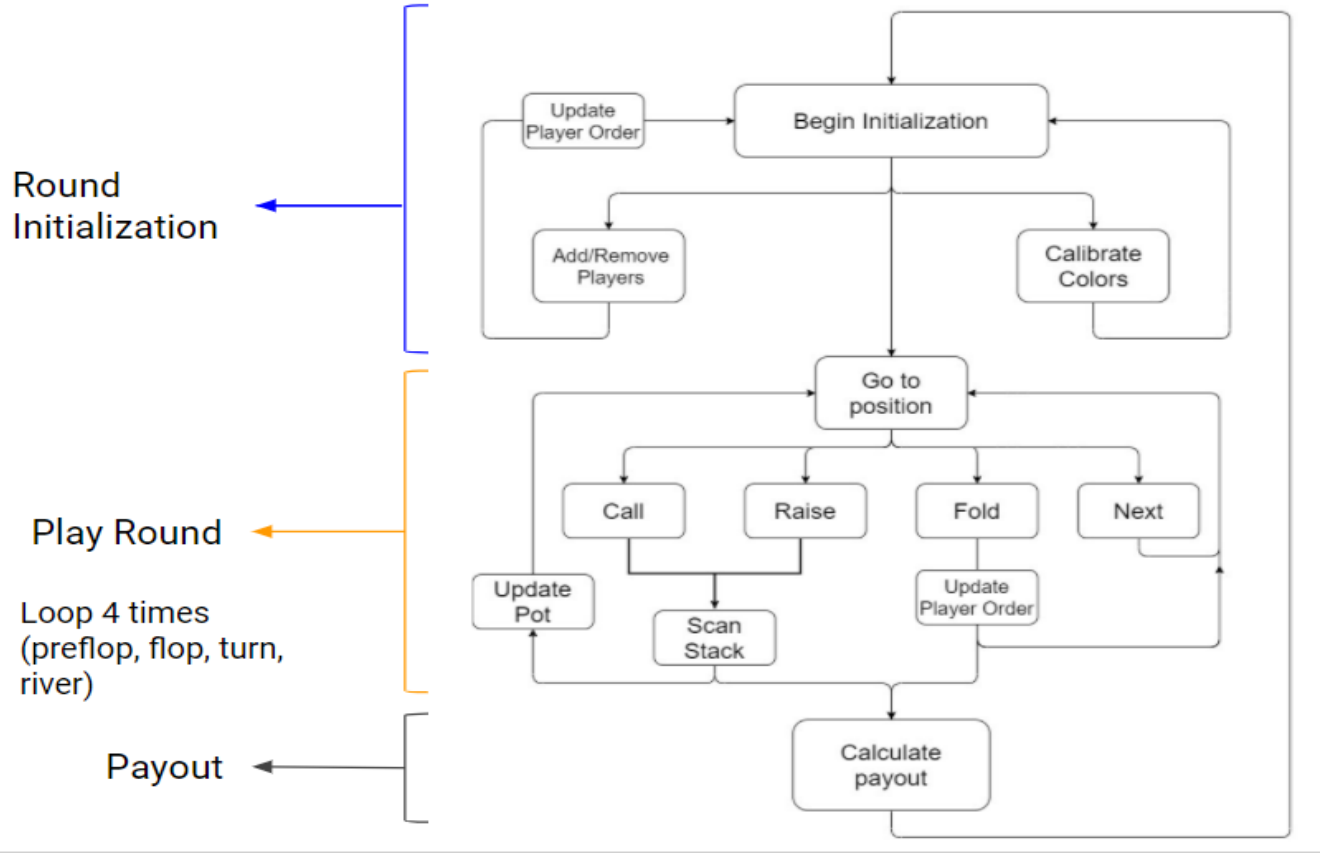


Fig. 18 Software Block Diagram

SCHEDULE

Fig. 19 Schedule

BUDGET

Item	Quantity	Unit Price	Total Price	Link to Item
Raspberry Pi 4B	2	61.5	123	https://www.amazon.com/Raspbe
IFROO Webcam	2	21.99	43.98	https://www.amazon.com/IFROO-
16 Channel Servo Driver	1	14.95	23.26	https://www.digikey.com/en/produ
Servo	1	27.99	40	https://www.adafruit.com/product/
Power Strip	1	9.88	9.88	https://www.amazon.com/GE-Prot
Power supply adaptor	1	12.99	12.99	https://www.amazon.com/Adapter
			0	
			0	Overall Total 253.11

Fig. 20 Bill of Materials

Servo Subsystem diagram

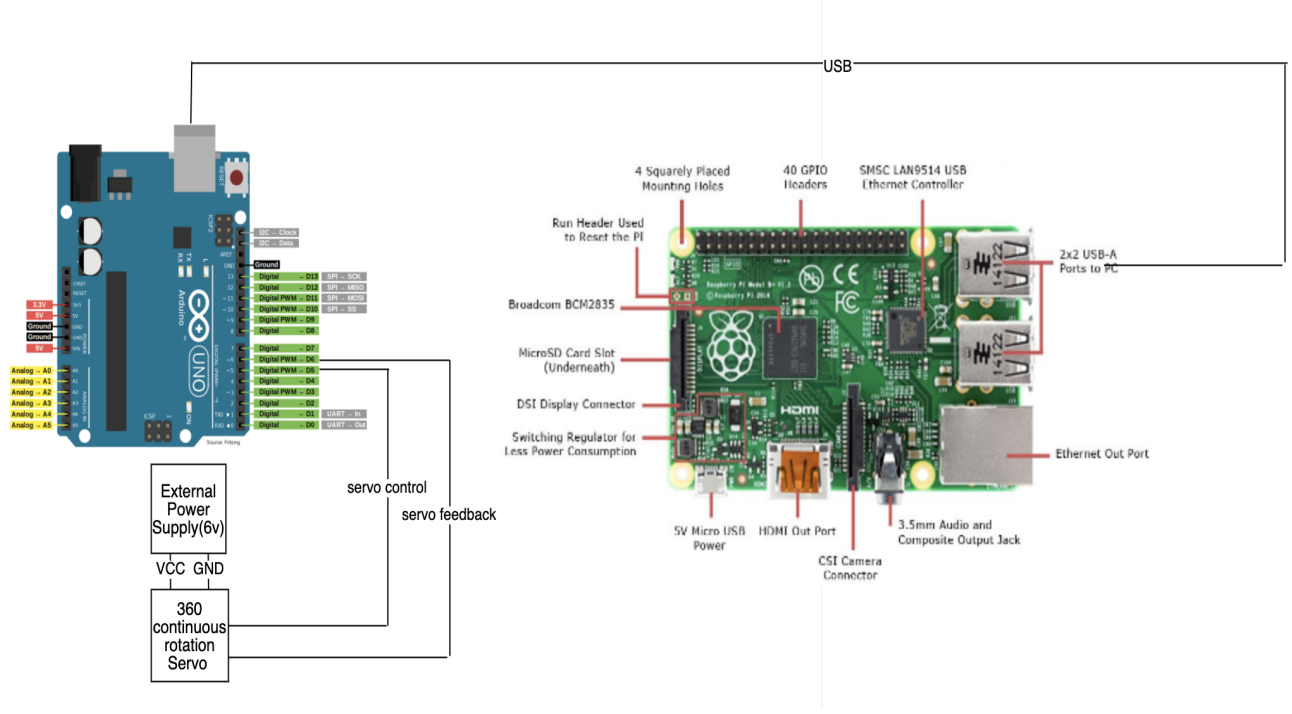
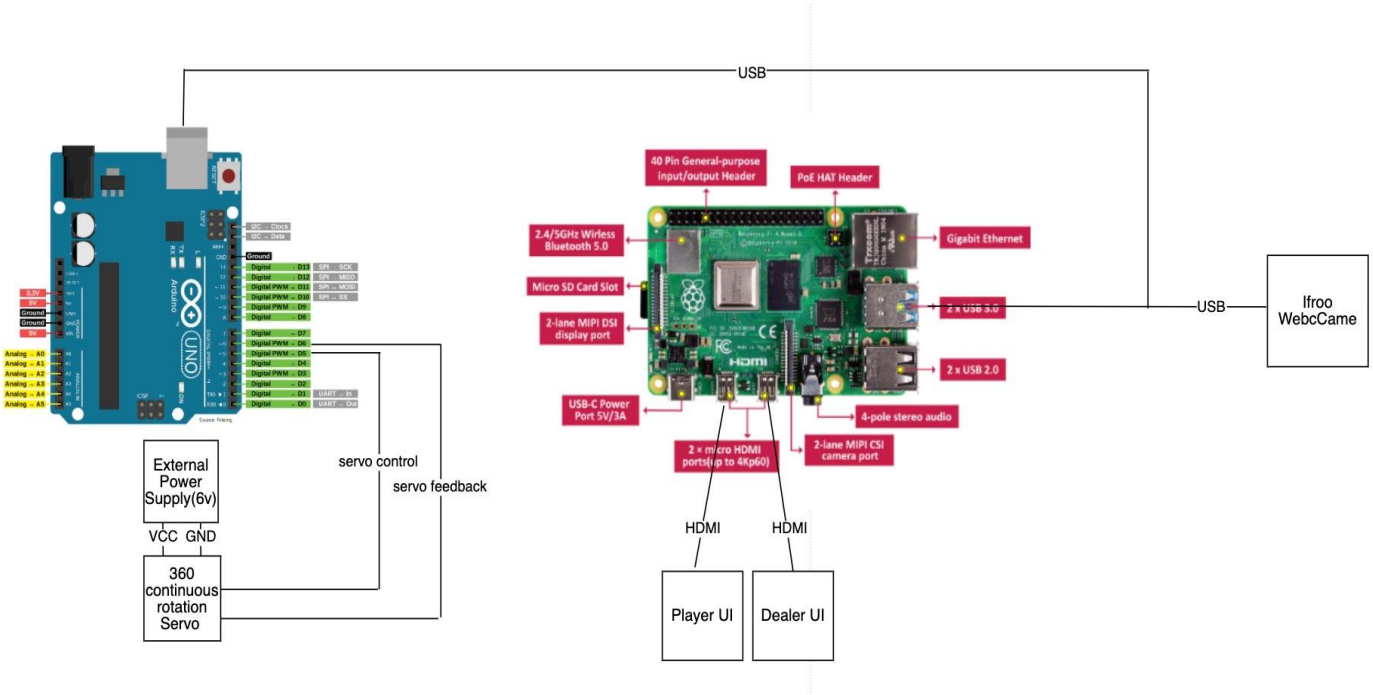


Fig. 21 Servo Subsystem Diagram



Electrical Schematic

Fig. 22 Electrical Schematic