Blokus: A socially distanced board game

Author: Nadine Bao, Jonathan Nee, Aria Zhang: Electrical and Computer Engineering, Carnegie Mellon University

***Abstract*—A system capable of supporting the gameplay of a real-time game of Blokus, with participants not being physically together. We want to allow participants to be able to enjoy the physical aspects of the board game even when participants may not be able to be physically present in the same space. The users primarily interact with a custom-designed Blokus board, with LEDs lighting up when pieces are placed. A camera with computer vision is used to track newly placed pieces and managed through software and a web server to communicate changes between boards.**

***Index Terms*—AWS, Blokus, Color Detection, Computer Vision, LEDs, Low Latency, NeoPixel, OpenCV, Arduino Uno, Social Distancing, Web Server**

I. Introduction

Our group is targeting the problem of being able to play board games physically in person during Covid 19 times, specifically focusing on one board game, Blokus. Given that social distancing is desirable, it may no longer be feasible to play board games in-person with friends. Yet, board games often do not translate well when played in a virtual environment, because much of the appeal comes from the physical aspect of being able to physically touch pieces and move pieces on the board. Despite the advent of popular online board game sites like Board Game Arena and Tabletopia, board games feel different without the physical interactions with the board itself. Our group aims to implement a socially distanced form of the board game, Blokus, where players can physically place pieces and interact with the board as per normal, but without needing the other players to be in the same physical space playing the game. The board game Blokus is a strategy game with transparent, Tetris-shaped colored pieces. Pieces are placed touching at least one corner of your pieces already on the board, without lying adjacent to any of your pieces. Based on the usual game of Blokus, our implementation must be able to support 4 players at a time, playing this board game together. Pieces are placed on your own board, tracked using a camera. Then, using computer vision and software to handle detection and state tracking, information is sent through a web server to the other players' boards. There, the local software will get the respective piece LEDs to light up the corresponding squares on the grid. We will also ensure that only valid moves are allowed and implement the ability to resume the game later. To simulate the real-time component of board games, the key goal of the project is to be able to implement this with minimal latency, updating board state within 150ms of a new piece being placed.

II. Design Requirements

The following design requirements have been set up to meet the functionalities of the board game, as well as to manage the real-time aspect of the board game itself.

A. Overall System Requirements

There are two main overall goals of the system. First, accuracy in piece updating in terms of board state across the entire system. Second, minimizing overall latency of the system.

- *Accuracy* – Given a piece placed on one board, the same piece's coordinates should light up the LEDs on all boards, assuming it is a valid piece placed, and not do anything if the piece placed is invalid.
- *Latency* – Given a valid piece placed on one board, the same piece's coordinates should light up the LEDs on all boards within 150ms. This 150ms is further broken down between the computer vision portion and client portion, the server portion, as well as the LED portion. This 150ms was set as it is said that this is the minimum possible latency to be perceptible by the human eye [12]. This latency requirement is important so that quality of the original gameplay is maintained.

B. Computer Vision and Client Requirements

When a piece is placed, the first task is to figure out the coordinates of the new piece on the board quickly and determine if the piece is valid or not. This is handled by the computer vision and the client component.

- *Video Stream* – Given an input video stream, to be able to correctly identify the coordinates with error rate of <1%. This will be split into 3 incremental tests. First a color identification test, where we test images being able to pick up a specific color with different lighting conditions. Given an input image and a specific color to filter (only testing for the 4 players' colors), be able to pick out the given colors on the image, under various lighting conditions. Second, a coordinate identification test, where we test images being able to output the correct coordinates of a certain color. Given an input image and a specific color to filter (only testing for the 4 players' colors), be able to pick out the given colors on the image, as well as the corners of the Blokus game board, and then correctly determine which coordinates have pieces placed on them, under various lighting conditions. Third, the video test where we have the input be a video stream. Given an input video stream and a specific color to filter (only testing for the 4 players' colors), be able to get images from the video stream, pick out the given colors on the image, as well as the corners of the Blokus board game, and then correctly determine which coordinates have pieces placed on them, under various lighting conditions. When a new piece has been placed on the video stream, be able to correctly update the coordinates that have pieces placed on them.

- *Client Software Requirements* – Given the coordinates of all the tiles on the board with a particular color, be able to identify the new piece that was placed as well as identify if the piece and position are valid. In addition, have a mechanism to keep track of player's turns to facilitate communication with the game session server as well as the Arduino. This will be split up into two tests. First, given a video stream parsed into coordinates by the CV, be able to determine the new piece that was placed and whether the position of the new piece is valid. Second, ensure that the piece validation process time is <1ms, which is negligible in the overall latency of the system.
- *Performance Requirement* – Given a new piece is placed, be able to detect the new piece within 130ms, from the time that the player's hand has left the board. This requirement is necessary since for any processing to begin, a clear view of the board is required to detect the changes and to know what should be updated. This average will be taken using a software timer, across the various functions.

C. *Web Server Requirements*

After the client software determines the coordinates of the new piece that the player has placed, this information must be passed to the web server so that it can be forwarded to the opponents' client software which is responsible for reflecting the piece on their physical boards. The web server is also responsible for allowing players to create game sessions and join other sessions. An additional functionality that the web server needs to have is the ability to save the state of the board which allows players to save and resume games later.

- *Server Testing* – Allow multiple games to occur concurrently. When a player creates a new game session, reflect that information to all the other clients in the game lobby. Also, when a player joins a particular game session, reflect the increase in current player count to other clients in the lobby. Also display the list of saved games the player has. If a game needs to be restored, fetch the list of pieces associated with the game session and replay them to all the players in the game to restore the board state of all the players.
- *Performance Requirement* – Given a piece, be able to send it to the web server and to the other players' local software within 40ms [2]. This is the round-trip time, inclusive of sending the information to the game session server and from the game session server to the other players, averaged using a software timer.

D. *LED Requirements*

After the Arduino receives the coordinates and color of the new piece, the corresponding LEDs need to light up.

- *LED Testing* – Provided with coordinates and corresponding colors, the LEDs matching the given coordinates should light up in the colors assigned. Visually inspect that the state of LEDs changes as expected.
- *Performance Requirement* – Based on the coordinates and their corresponding colors sent from the client software, the Arduino Uno should change the state of the LEDs to fulfill the request from client software. A software timer will be used to determine the execution time of the Arduino software, which includes writing to the output pin, over 10 executions. The performance requirement for the Arduino Uno and LEDs should fall under 2ms, meaning it is required to be an insignificant part of our overall latency.

III. Architecture and Principle of Operation

The overall architecture of our project is split into three parts, the computer vision software, the game software, and the custom Blokus board with hardware components, as shown in Fig. 1 on the next page.

Each player will have a Blokus board that is fitted with 400 LEDs and driven by an Arduino Uno as well as a Logitech Camera mounted directly above the board. The game client software will be running locally on the player's PC, which includes the computer vision software. The client software as a whole is responsible for processing input either from the computer vision thread or the GUI and validating those inputs before communicating the player's piece to the game session thread on the central server which is hosted on AWS using EC2. The central server acts as the portal of communication between the players. The central server is composed of a game lobby manager thread and game session threads. The game lobby manager thread accepts connections from clients and then spawns game session threads to handle individual games. The threads that are part of the central server also have access to a database to store and retrieve game and player information. Fig. 1 on the next page shows the flow of the game as described.

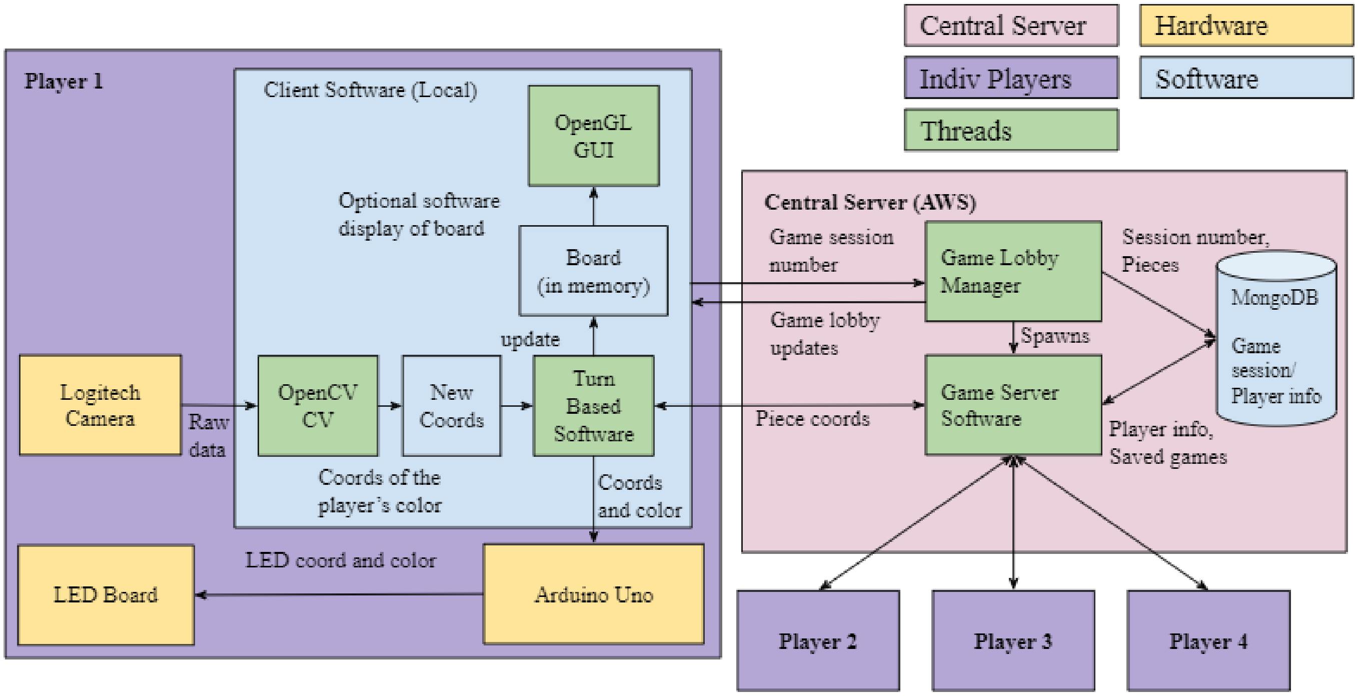


Fig. 1. Overview of the system architecture

When it is a player's turn, they can place down a piece of their color onto the board. The camera will be continuously streaming a live video of the board to the game software. The client software has two main threads, one that handles the computer vision and the other that handles the game logic. The CV thread continuously captures images from the live video and processes them to determine the coordinate of each tile that houses a piece of the player's color. This information is updated in the shared memory between the CV thread and the game logic thread. To prevent race conditions between the two threads, synchronization variables will be used to prevent race conditions such as stopping the game logic thread from reading the shared memory when the CV thread is in the middle of updating the locations.

On the game logic software side, there are two states that the software can be in. When it is the current player's turn, the game logic software will be continuously polling for piece location updated by the CV thread. When there is a new coordinates update, the game logic software will determine the piece type and piece location. After that, the game logic software will make sure that the move itself is valid. If the move is invalid, the software will keep polling for new updates from the CV thread. If the position of the new piece is valid, the coordinates of the piece will be sent to the central server using TCP.

The game session thread will take the coordinates of the new piece and store them in memory as well as forward that piece to all the other players in the game. If a player in the game chooses to leave before the game is over, the list of pieces placed during the game session will be persisted into a database (MongoDB). And the session id of the game will be stored under the player information for each of the players that were in the game. This allows the saved game to be displayed to the players that were part of the game, and furthermore allows those players to rejoin the session later.

During an opponent's turn, the client software listens for a message from the game session thread. When the client software reads a message from the server, it will send the coordinates and the color of the piece to the Arduino via serial communication.

When the Arduino receives a message through the serial port, the Arduino will translate these coordinates into a linear LED address which it will drive with the designated color which was also passed via serial communication. Thus, on each of the players' boards, the coordinates of the piece that the player placed will be lit up.

The player's turn then ends, and the next player's client software will begin checking for their new piece, and this process repeats until the game ends.

IV. Design Trade Studies

A. *Detecting Pieces and Conveying Moves to Players*

The main requirement of our Blokus design is to be able to detect pieces that have been placed, and then to visually convey to the player what pieces opponents have placed. Blokus will support up to four players, meaning that the Blokus board must be able to display the moves of four players represented by four different colors. For example, if the player's color is red, then the opponents' colors will be yellow, blue, and green. There were 5 methods that we considered for piece detection, and 2 for conveying opponent moves. We will discuss the methods for conveying piece detection first, and the pros of cons of each have been summarized in Fig. 2 below.

Alternatives	Sensors	PCB	RFID Tagging	Camera +CV
Size (board space)	1	4	2	10
Power Consumption	4	6	8	10
Cost	5	10	6	9
Time (detection)	8	10	8	5
Time (production)	7	1	7	10
Able to handle conveying moves as well	5	10	5	5
Total	30	37	36	48

Fig. 2. Piece Detection Comparison, 10 is best, 1 is worst

For our metrics, we only consider the amount of board space it takes up, as external space is irrelevant to keeping the feel of the original board game, and the main concern is that the Blokus board itself has limited space to work with. The first method that we considered was using sensors. There were two types of sensors that we considered. In both cases, we used sensors as a threshold detection mechanism, effectively using the sensors as a switch to determine if there was a piece on top of a tile on the game board. The first were hall effect sensors. These sensors measure the magnitude of a magnetic field, and its output voltage is proportional to the magnetic field strength through it. This approach would require each piece to be manipulated with tiny magnets to be able to trigger the sensor when on the board. When a piece is placed, the magnitude of the magnetic field nearby increases and triggers the sensor. The second were light sensors. These sensors measure the magnitude of light coming through. When a piece is placed, it blocks off part of the light coming through, as the pieces are not totally transparent, and using the same concept of threshold detection, will trigger the sensor. The main drawback of sensors is the amount of space they would need, especially given that they are done individually, and so would need to be hooked up properly for all 400 cells. The second method was designing our own PCB. This was a very appealing option, because it would not only be able to handle piece detection but also help convey player moves at the same time, given that we could mount LEDs and sensors within the dimensions desired, handling everything at the same time. It would also provide the fastest detection time, which is very important for our project, given a key metric is low latency. The main drawback for a PCB though is the fact that we are currently in Covid 19 as we speak, and production time has drastically increased. Designing a PCB within the United States is not of the best quality and would still take time and designing a PCB and shipping it to China to be produced would take up to several months. In comparison to all the other items, which are readily available and can be bought and delivered within a couple of days, there is a longer turnaround time involved with a PCB. In addition, unlike the other options, there is a possibility that the design itself had errors, or if the mask on the PCB board was not done properly. Given the turnaround time and risk involved, it seemed like a relatively unsafe option to manage. The third method was RFID tagging. This would involve tagging each piece with an RFID tag and having an RFID reader detector attached to the board. By detecting the distance to the reader, the piece can be unique identified. This approach was good, but like the sensors, required modifying pieces and was costly as well. The last approach involved a camera and computer vision. In this approach, a camera would send a video feed to a computer, where computer vision would be used to process the image of the board and help detect the coordinates of those pieces and identify the newly placed piece. This approach's main drawback was the time for detection.

We will now discuss the 2 methods for conveying player moves. The first method that was considered was using Augmented Reality in which the player would view the Blokus board through their smartphone camera to see where the opponents' pieces are. We found this method to be unsuitable for our project because it did not align with our main goal: to allow use of a physical game board while playing Blokus remotely. With Augmented Reality, players would not be able to get the hands-on board game experience we intend to provide through this project. This would make the board game less seamless, and less desirable to the players. From this, we concluded that the method used to convey opponent moves needed to be part of the Blokus board such that players would not need to use an external device, such as a smartphone or laptop, to play the game.

The second method that was considered was using individual LEDs for each tile that would be lit according to the players' color when a piece was played. This meant that players would be able to immediately see squares physically on the board where opponents have placed pieces and does not take away too much from the physical component of the board game. The only downside to this approach was that we need to be careful that the LEDs do not bleed into other tiles as it could be confusing to the player and confusing for the piece detection portion of our project given that we went for the computer vision approach.

Ultimately, we decided on using computer vision for piece detection, and using LEDs for conveying moves. These methods best fit our needs and based on our analysis above, were the most suitable for this project.

B. Computer Vision Software

Since the latency of our system is of critical importance, we did a trade study between C++ versus Python OpenCV libraries and functions to see which would be able to more quickly process images to get the desired outputs. Based on research, we expected C++ to be faster, as Python's OpenCV basically does the same functions as in C++ with a wrapper to provide the ease of use that Python offers. We did a quick test on a test image and timed how long it would take to isolate red, as shown in Fig. 3 below.

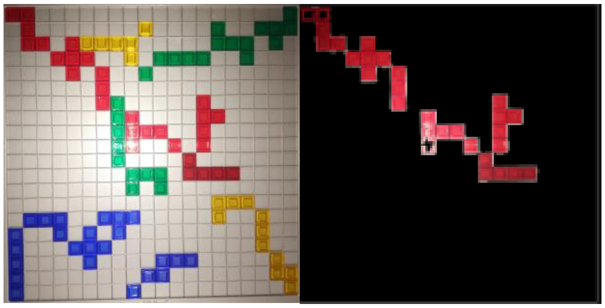


Fig. 3. Test image and desired output for software trade study

The code we used applied a HSV mask, since that is the ideal color scheme to do color manipulation on, then a morphological transform and dilation to isolate a range of pixel values and smoothen out the resultant pixels found. We timed how long it took from start to finish and averaged out the readings to avoid skewing the results due to abnormalities.

We found that implementing this in Python took 0.05s while it only took 0.007s in C++, using the equivalent function calls and similar timing mechanisms. As such, this confirmed our hypothesis, and justified us using C++ for the computer vision part of our project, since our project was concerned with latency as a key metric.

C. Computer Vision Method (Resizing)

We also did a design trade study on different computer vision methods to achieve the low latency requirement we wanted. Notably, we had a choice in whether we wanted to resize the image we got. Resizing the image immediately incurs a latency delay, while not resizing the image incurs additional latency delays throughout processing the image, as every additional instance of any function that must loop through the image will take longer as there are more pixels involved. For instance, changing the image to HSV color scheme, filtering out colors, and looping through the image at the end to check if a given grid cell is filled or not will all incur additional latency costs. Fig. 4 shows the latency of the various main steps of the computer vision portion, as well as how they differ by resizing versus without resizing. We concluded that the latency cost of resizing the image was too high, and that the gain in terms of reduced latency in the other functions were insufficient to justify the latency cost of resizing and that not resizing the image would give us a lower overall latency. As such, we ultimately decided not to resize the image.

Note that the values in the table are rough estimates, and the range provided shows the variation. The numbers in each row do not sum up totally to the whole, because in different iterations there are different fluctuations for various reasons, and so in this case the whole is not the sum of its parts, even though each component was individually measured. For the smallest activities, we were careful to consider the latency involved in even setting up the software timer and ensured that we kept all comparisons constant between the two. In addition, while the range is large, in general the average latency tended to be in the lower end of the latency range.

Activity	Resizing Latency (ms)	No Resizing Latency (ms)
Read new image	1 (negligible)	1 (negligible)
Resizing	30-80	-
Change to HSV	10-30	16-40
Filter color of corners	3-10	12-24
Morphological transform	3-5	3-5
Find contours	1-3	3-4
Find bounding rectangles	<1 (negligible)	<1 (negligible)
Calculate grid dimensions	<1 (negligible)	<1 (negligible)
Filter color of player's color	9-15	8-30
Calculate if each cell is of player's color	5-10	3-7
Total latency range	80-240	60-120

Fig. 4. Resizing Comparison

D. Computer Vision Method (Grid Detection)

Another computer vision method we had to analyze were the different methods we had to do grid detection. There are two possible simple ways to do this. The first method involves detecting the grid from scratch, and there are many well known ways to do this [16]. Such methods use the difference in colors between the grid and the surroundings to detect the gridlines. The problem with such methods that we faced was caused by two issues, the first is that the Blokus board's gridlines within the board itself were thin and gray, and the second is that pieces placed covered the gridlines themselves. This caused two problems. First, even on a bare board, we could not detect the grid just by setting a color range, because in different lighting conditions, what would be the grid lines might easily be part of grid cells in other lighting conditions. This is still resolvable if we use color contrast to isolate the grid lines. The second and larger problem is that when pieces were placed over the gridlines, the gridlines were no longer detectable, as the grey color was no longer detectable from the image. This problem can be seen from Fig. 3 above as well. This is, however, a problem that can be mitigated, if we assume that the board state will not change over the course of the game, and we calibrate the board position across the video stream based on fixed grid lines from the start of a new game or before resuming a saved game. There are inherent downsides to this, however, in that the board itself is not stuck to any surface, and so if by accident the board gets shifted during gameplay piece detection would become impossible. The second method to do grid detection was simply by placing colored squares at the corners of the grid, demarcating the corners of the grid, and searching for these colors instead. The method we used here was to start from the center of the image and find the outermost positions in the image where those colors were detected. This is better than the previous solution in that practically speaking it is a more robust solution: the board might get moved from time to time, and this is a method to detect the grid whenever an image is being processed, that would help to deal with such issues. For this reason, we chose this second option instead. That said, there are still downsides to such a method, for instance, the surroundings cannot be of the same color as that of the colors used to demarcate the grid. We believe this is a smaller problem than the one discussed earlier, because we have perfect freedom in choosing a color that is likely rare to see on playing surfaces such as a table or the floor, and so by avoiding common colors like white or black such an issue can be mitigated.

E. *Process of Triggering Piece Storage*

A supported feature of this project is the ability to resume a certain game later. This involves storing the state of the board in the database when a player chooses to save the game. Originally, new pieces were supposed to be inserted into the database by the game session thread on receiving the piece. Due to feedback from the proposal presentation, more research was done to measure the latency of a database query. It was found that the latency overhead of inserting a piece into the database would be in the 10 to 20ms range. Hence, this approach was ruled out, as it would directly add to the latency of updating a piece.

Services like AWS Lambda and AWS SQS were also considered, and were initially part of the design presentation, but were ultimately removed. This is because both services are not provided in the free tier that students get and because the process of triggering a Lambda function would have been slower than accessing the database directly since Lambda functions require time to set up the execution environment. This contributes to overhead costs which would have been detrimental to the latency goals set for the round-trip time from client to server to client.

To keep the latency to a minimum, we concluded that updating the database on each new piece was simply unfeasible. Thus, we decided to have the board be saved in memory until the game has been saved, at which point the game server would persist the board state to the database.

F. *Web Server Choice*

The three cloud services that were considered for hosting the central server were Amazon Web Services (AWS), Microsoft Azure and Google Cloud Platform. Part of the reason why AWS was chosen was because it is one of the most comprehensive cloud platforms that currently exist and is widely used around the world. Also, the pricing for AWS services is very competitive compared to the other cloud services. They offer free tiers of service which some of the other cloud services providers do not.

The main reason why AWS was chosen was because it was the cloud provider that our team was most familiar with, making it easier for us to use, allowing us to be more productive rather than spending time experimenting with other cloud providers. The service we will be using from AWS is EC2. Some of the other services that were also considered include the Google Compute Engine, Azure Virtual Machines and IBM Cloud Virtual Servers which were all ruled out because they were more expensive. Overall, the decision to use AWS was mainly due to expenses and experience with the product [5][10].

G. *Database*

The database that was originally chosen was DynamoDB since it was assumed that it would be more compatible with the other AWS services that were being used, but ultimately the database chosen was MongoDB. This is because it was very easy to set up a cluster using MongoDB Atlas, a fully managed cloud database service that is very user friendly. Also, since our group has had experience with MongoDB prior, it seemed to be the best choice for the project.

H. *Display*

The display for the pieces being placed was generated using OpenGL and GLUT [15], however another choice for the display could have been through a web application. The web application would have made this project more accessible as that would have allowed users to access it through their web browser instead of having to use frameworks like OpenGL and GLUT. However, using a web application would have added additional latency through the number of requests that would have to be sent back and forth, and would be less efficient since it is written in languages like Javascript. Updates would take longer to appear on the web application versus the GUI which is written in C++ and is updated very frequently. Also, group members were not very familiar with web development so sticking with a C++ application using OpenGL was what the group ended up pursuing.

I. *Types of LEDs*

To display opponent moves, the team decided on constructing a 20 x 20 LED matrix display and installing the LED display under the Blokus board. To effectively display opponent moves, the 400 LEDs would need to be individually programmable/addressable and light up in at least four colors (red, green, blue, yellow). The following ways of constructing a LED matrix were considered:

- *Prebuilt LED Matrix Display* – A ready-made programmable LED matrix display would have been ideal for this project, as it would remove any need for soldering. Unfortunately, the spacing between LEDs on such displays available for purchase do not fit the necessary spacing requirements to fit under our Blokus board. Each square on the Blokus board is 1.5cm apart, center to center, which is much larger than most prebuilt LED Matrix Displays, which are meant for having more fine-grained color changing.
- *PCB LED Matrix* – Another consideration was to design a custom PCB with the necessary LEDs. The benefits of this method include being able to control the spacing of LEDs and eliminating the need for soldering. The end prototype will also be much neater

due to the lack of wires needed. As discussed earlier, this could also be integrated with sensors to handle piece detection at the same time. The main disadvantage of this method is that it would require a longer time to obtain a working product that could be used for testing. The time it would take for the manufacturer to make and send us the board and make any revisions would greatly affect the timeliness in which we could integrate and test the LED matrix with other components of the project. This method would be ideal in the situation where there is more time.

- *Individual RGB LEDs* – In this method, individual RGB LED diodes will be used to construct a circuit. There are not many benefits to this method besides the ability to control spacing. It is labor intensive and not the most cost effective as each LED costs over a dollar. Given that we need 400 lights, this method would consume over two-thirds of our 600-dollar budget.
- *RGB LED Strip* – The LED matrix can be constructed using RGB LED strips which contain individually programmable LEDs. The benefit of this method is that it costs less compared to buying 400 individual LEDs. It also removes a lot of circuit building work as the LEDs come wired together. Spacing is more difficult with this method as manufacturers only sell LEDs strips with around three variations in spacing; however, because the strips themselves are flexible, the position of each LED can be altered to some extent.

The team found that constructing the 20 x 20 LED matrix using an individually programmable RGB LED strip would be the most suitable for our needs. Given our time constraint, this method would allow fast fixes/revisions if necessary and is much less labor intensive compared to using individual LEDs.

J. Selecting LED Strips

Model (60 LEDs/m)	Power Consumption	Working Voltage	Price
WS2812B	18 Watts/m	DC5V	$51.76
WS2815	18 Watts/m	DC12V	$107.86

Fig. 5. LED Strips Comparison

When it came down to selecting the right model of LED strips to use for our project, we had a choice between WS2812B and WS2815, as shown in Fig. 5 above. Both models consist of individually programmable LEDs as required by our project and come in densities of LED, 60 LEDs/meter that would work with our spacing requirement (LEDs 1.5cm apart, center to center).

With regards to power consumption, the seller, BTF lighting, has both models listed at 18 Watts/m which indicates that 400 LED would use around 120 Watts. This may not be true in practice as some research online regarding power consumption of various LEDs indicate that the WS2812B consumes significantly less power than the WS2815B; in the 30 LEDs/m configuration on max brightness on all channels, 150 WS2812B LEDs consumed 13.65 Watts while 150 WS2815 LEDs consumed 20.184 Watts [14]. This likely indicates that WS2812B draws less power despite the measurements provided by the seller and thus comes at an advantage when it comes to saving power.

Another benefit the WS2812B has over the WS2815 is its cost. Purchasing 2 sets of 300 LEDs/m WS2812B to meet our 400 LEDs requirement costs half the amount it would take to purchase the same quantity of WS2815B (refer to table). However, the cheaper price is likely the result of the WS2812B's shortcomings compared to WS2815. With WS2812B, since the working voltage is DC5V, a single DC5V power supply would struggle to evenly light up a strip of 400 LEDs as there are not many DC5V power supplies for sale that have high power ratings. This means that to have 400 LEDs light up in addition to having the LEDs light up evenly, it would require multiple power supplies and power injections at multiple points along the strip. This adds bulky components to the matrix display setup and increases the cost due to the need to purchase multiple power supplies.

The voltage drop across the LEDs has more consequences in terms of even lighting when the power rating of the supply is low. This issue is fixed by using the WS2815 which is DC12V compatible. DC12V supplies usually come with a higher power rating; in this project, we managed to purchase a reasonably priced power supply rated at 360 Watts. This higher power rating allows us to not have to perform power injection, which saves us cost on purchasing multiple power supplies and makes for a neater build.

The team chose to use the WS2815 despite its higher power usage and cost because it was a safer and more compact solution that provided a better end product. With WS2812B, the power supplies used will have lower power ratings. This means that with such a long LED strip, the strip will likely be drawing near max power which can easily lead to overheating of the power supply. With the WS2815, the LED strip can draw power from a 360-Watt supply which prevents the issue of overdrawing power and provides leeway for when power draw increases due to modifications to the strip. Additionally, the cost of purchasing WS2815 with a compatible DC12V power supply would end up costing less than the cost of purchasing WS2812B with the necessary amount of DC5V power supplies.

K. Board Integration

One important aspect of this project is integrating the LEDs into the Blokus game board we purchased. This involves centering each of the 400 LEDs under each of the 400 squares. There were two ways by which we could make the LED lights under the Blokus game board visible to the player. One method was increasing the brightness of the LEDs such that they show through the board. While this method would have been aesthetically pleasing, we did not choose to do this due to risk of the LEDs overheating. The LEDs are very warm at high levels of brightness, so it did not seem safe to have them on high brightness levels while placing them in a confined space inside the Blokus game board. As a result, we opted to run the LEDs at low brightness (around 30 out of 255) and drill holes in each of the 400 squares on the Blokus game board so that the player can see which LEDs are lit. The holes would also allow for better air flow, which is important keeping LEDs cool especially when they are near each other.

Before drilling the holes, we needed to decide between using an electric drill or a hand drill. We ultimately decided on using a hand drill because it would have cost us less to purchase compared to buying an electric drill which costs around $50. While opting to use a hand drill has saved us more of our $600 budget, it did lead us to spending more time on board construction and integration as each hole had to be drilled by hand. We found that the tradeoff between time and cost was worth it as our budget was rather limited; the money remaining would be better off saved in case we needed to buy replacement parts.

Positioning the LEDs was another design challenge we had to face. Initially, we had planned to attach the LEDs directly on the inside of the Blokus game board; however, during construction we realized that adhesive did not stick particularly well to the plastic material of the Blokus game board and permanently fixing the LEDs to the game board using glue would make repairs very difficult to make. Hence, we instead attached the LEDs to a flat, square cardboard surface that would be placed underneath the Blokus game board. The carboard surface with LEDs can be easily detached from the gameboard anytime we needed to make repairs to the LED matrix. Fig. 6 below shows our completed LED board as well as an image of what our game board looks like.

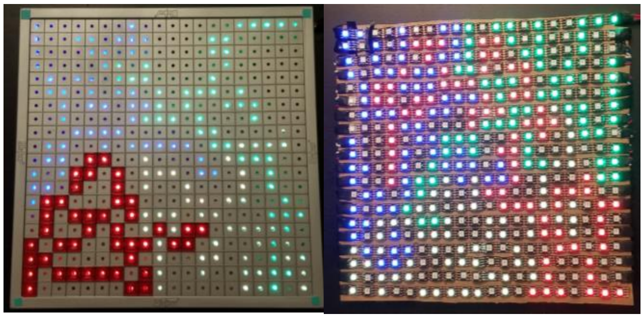


Fig. 6. LED Strips

V. System Description

A. Computer Vision

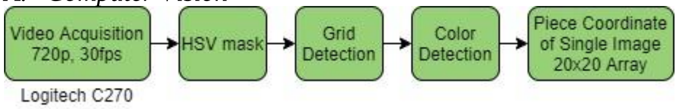

Video Acquisition 720p, 30fps
Logitech C270
HSV mask
Grid Detection
Color Detection
Piece Coordinate of Single Image 20x20 Array

Fig. 7. Computer Vision Schematic

Fig. 7 above shows the process for the computer vision portion. The first stage of each turn is the player's move, and this portion of the code was written in C++, as discussed in Section B in the design trade studies, using the OpenCV library [6][9]. We use a Logitech C270 camera to maintain a live video feed of the board. When it is the player's turn, for each frame received, we first apply a HSV mask, since that is the ideal color scheme to do color manipulation on. We then apply a morphological transform and dilation to isolate a range of pixel values and smoothen out the resultant pixels found. We do this twice, once for the grid detection, by searching for bounding rectangles in the image, and then again for color detection of a specific player's color, depending on which player the board is being used for. Thereafter, we plot out which pixels correspond to which coordinates on the original board and calculate all grid coordinates with pieces that belong to the player in a 20x20 array. The computer vision portion then passes all the coordinates belonging to that player to the turn-based software. This process continuously happens regardless whether it is the player's turn. The turn-based software handles processing of the coordinates sent by the CV portion only when it is that player's turn. This means that players can make their move in advance before their turn if they so choose, and such moves, if valid, will be picked up immediately when their turn starts. Fig. 8 below shows an image of this processing being done, on the player who is using the red pieces. The left image shows the instantaneous image as captured by the Logitech camera, while the right image shows the processed perceived grid lines as well as all the pieces detected from the given image on the left.

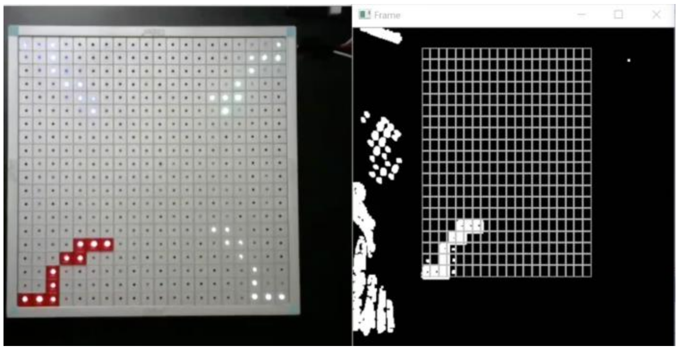


Fig. 8. Video Stream (left) and Computer Vision Post Processing (Right) isolating the color red

B. Game Software

The client software is composed of five threads with the bulk of the work being done on the game logic thread, the computer vision thread, and the GUI thread. The other two threads help read input from standard in as well as read messages from the game session thread. The software is written in C++ and uses the OpenGL and GLUT frameworks. A tuple hash template was also used which can be credited to Leo Goodstadt[8].

- *Game Logic*

There are two main parts of the game logic thread: turn recognition and piece validation.

  - *Turn-based logic*

The game logic thread can be in one of two states, player turn or opponent turn. During opponents' turns, the software constantly listens for messages sent by the game session thread running on the central server. When the game session thread sends the coordinates of a piece to the client, the client forwards those coordinates to the Arduino along with the color of the piece through serial communication. The Arduino then drives the LEDs to reflect the piece that was just placed by the opponent.

The other state the software can be in is the player turn state. During this state, the software continuously checks the memory it shares with the CV thread. If there are new coordinates, the client software finds which coordinates belong to the new piece placed. After isolating the new piece coordinates, the piece must be validated. If the piece is invalid, i.e. the player does not have a piece left in their inventory that matches the new piece shape or if the location of the piece is invalid then the client software will wait for another update from the CV thread and attempt to validate again until eventually a valid piece is played. When a valid piece is placed, the coordinates of that piece are updated in the copy of the board on the client side and then the coordinates of the new piece is sent to the game session thread via TCP as well as the Arduino through serial communication. Note that when the client software detects a new valid piece, that piece is finalized, and the player can no longer remove that piece from the board. The group debated on allowing the user to amend the piece they placed by adding an undo button, but any amending processes would just add an unnecessary delay to the system, and since our goal was to minimize the latency, that idea was scrapped.

○ Move Validation

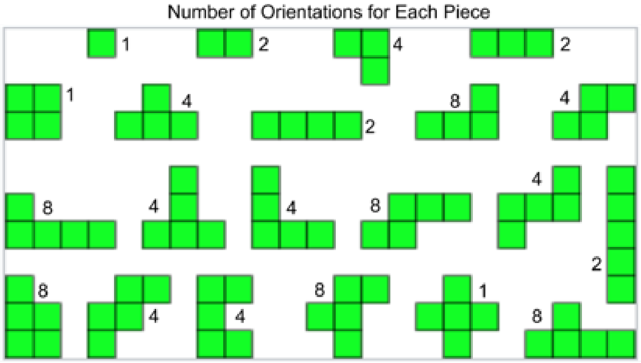


Fig. 9. Player’s starting inventory of pieces and the number of orientations for each piece

Pieces in software are represented as a list of coordinates that can either have length 1, 2, 3, 4, or 5. Each piece can have between 1 and 8 orientations. For example, the piece of size 1 only has 1 orientation while some of the pieces in Fig. 9 above have up to 8 orientations. Thus, each orientation needs to be considered when trying to determine what piece was just placed on the board. While there are only 21 Blokus pieces, when orientation is accounted for there are about 91 different piece orientations to search through when trying to determine the piece that was placed, as shown in Fig. 9 above. Generating and checking through 91 different pieces is not a trivial amount of work, especially when our design is focused on latency as a key metric. To speed up the processing, before the game starts the client software will read in the 21 standard Blokus pieces from a csv file and then generate all possible orientations for those pieces and place them in an unordered map that links the normalized piece coordinates to the piece type (a number from 0 to 20). This allows the client software during the game to perform on average a O(1) look up for piece type after first normalizing the coordinates it reads from the CV thread.

Since the CV software returns coordinates on the board, before trying to determine what piece those coordinates represent, the coordinates need to be normalized first. The coordinates (row, col) received from the CV thread are normalized by finding the minimum row and the minimum column out of all the coordinates for that piece. Then each coordinate is normalized by subtracting the row by the minimum row and subtracting the column by the minimum column. The coordinates also need to be sorted by the smallest row and if coordinates are on the same row, the tiebreaker is the coordinate with the smaller column. An example representation of a normalized piece is shown in Fig. 10 below. This normalization process is critical because coordinates from the board cannot be directly passed into the unordered map as one piece could have a multitude of offsets.

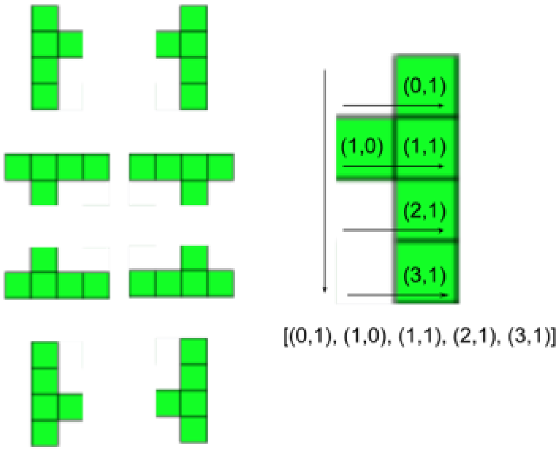


Fig. 10. Multiple orientation of pieces as well as example of normalized piece representation

If the piece is not in the map that means the piece is not an actual piece. This could mean the player might have just tried to place multiple pieces on the board during one turn. If the piece is in the map, then the piece type is extracted (there are 21 unique values for pieces) and checked against what remaining pieces are in the player’s inventory. If the piece is not in the inventory, then that means the player has tried to place a duplicate piece that was already placed before, which is an invalid move.

While the piece itself could be valid, the location where the piece was placed may not be. One of the rules for placing pieces in Blokus is the first piece placed for each color must be placed on one of the board’s four corners. This rule can easily be checked based on the coordinates passed by the CV thread on each player’s first turn. Another rule is that each piece must be placed so that it touches at least one piece of the same color corner to corner. Edges of pieces of the same color are also not allowed to touch. This validation is done by looping through each coordinate of the piece being placed and checking all four edges as well as the four corners of the coordinate for what colors are at its edges and corners. To make sure no pieces of the same color are at non corner positions, and that there is at least one corner where there is a piece of the same color and that there isn’t already a color at the coordinate.

- *GUI*

The GUI was created using OpenGL as well as GLUT, the OpenGL Utility Toolkit. The GUI was originally created to facilitate debugging as it was useful to have a way of displaying the state of the in-memory board; however, it also became another possible way of playing the game. If a player does not have access to a camera and Blokus game board, they can use the GUI as a way of inputting pieces onto the board. Using the up and down arrows, users can select the piece that they want to place, and the “1”, “2”, “3”, “4” number keys allow the user to rotate the piece. The “enter” key can be used to reflect the piece across the y-axis and by clicking, the player can place the piece if it is a valid move. The GUI is also used to detect user inputs such as saving which users can do by pressing the “S” key and giving up which can be signaled by pressing the “L” key. Fig. 11 below shows an example of what the GUI looks like when a game is ongoing.

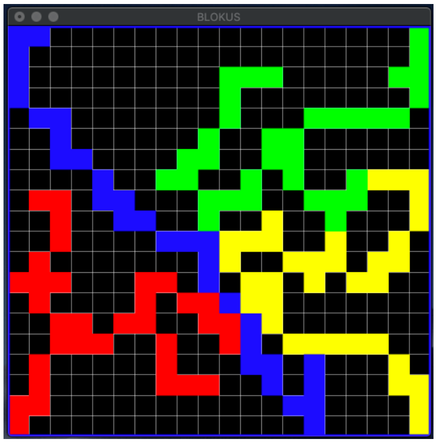
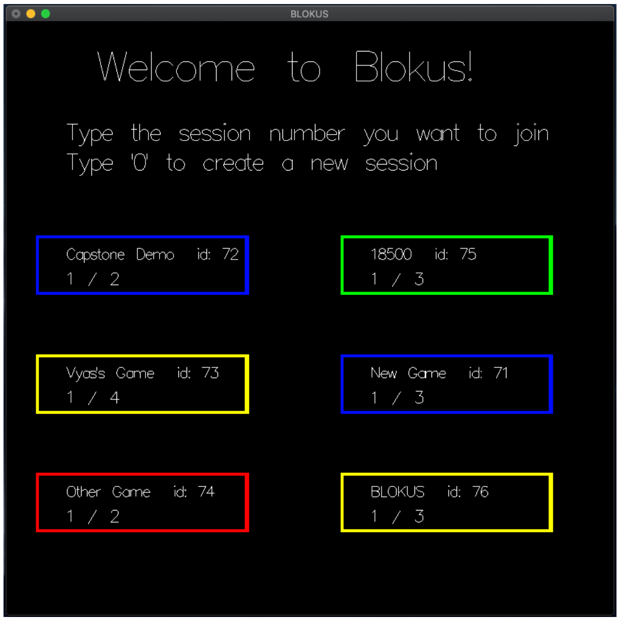


Fig. 11. Image of the GUI midgame

The other primary responsibility of the GUI is to display the open game sessions. When a client connects to the game lobby manager that is running on the central server, the client receives information about the current ongoing game sessions. Up to six of these game sessions are displayed to the user on the GUI, the displayed information about the game session include the name of the session, the session id, the current number of players that have joined the session and the total number of players needed for the session. This allows the user to see what game sessions are available for them to join. An image of the lobby GUI display is shown in Fig. 12.



Welcome to Blokus!

Type the session number you want to join
Type '0' to create a new session

Capstone Demo id: 72
1 / 2

18500 id: 75
1 / 3

Vyas's Game id: 73
1 / 4

New Game id: 71
1 / 3

Other Game id: 74
1 / 2

BLOKUS id: 76
1 / 3

Fig. 12. Image of the GUI midgame

- *CV Communication*

The computer vision thread communicates with the game logic thread through shared memory. Delving deeper into the implementation, the shared memory is a vector of coordinates protected by a mutex and a variable called update_num. On the other end, the game logic thread also has a local variable called current_update_num. Update_num and current_update_num are both initialized to 0. When the CV thread calculates a new batch of coordinates for a frame of the live feed, the coordinates are not stored directly into shared memory. Instead, after the CV thread finishes compiling a list of new coordinates it grabs the mutex and then wipes the old coordinates in the vector and copies the new coordinates into shared memory. Before unlocking the mutex, the CV thread also increments the update_num. The game logic thread, when it is the player’s turn, continuously loops and checks the update_num and compares it against its current_update_num. Spin waiting around updated_num should not be an issue since we are using a multi core processor. If update_num > current_update_num then that means there are new piece coordinates in shared memory, so at that point the game logic thread will then start attempting to lock the mutex. When it does, it can then copy the coordinates out of shared memory into a local vector and set its current_update_num equal to the update_num. Then the mutex will be unlocked and the game logic thread can proceed with validating the new piece placed.

C. *Server Software*

The server software runs on an EC2 instance and is composed of one main game lobby manager thread and multiple game session threads depending on the number of ongoing game sessions. An additional thread is also used to constantly accept connections from clients.

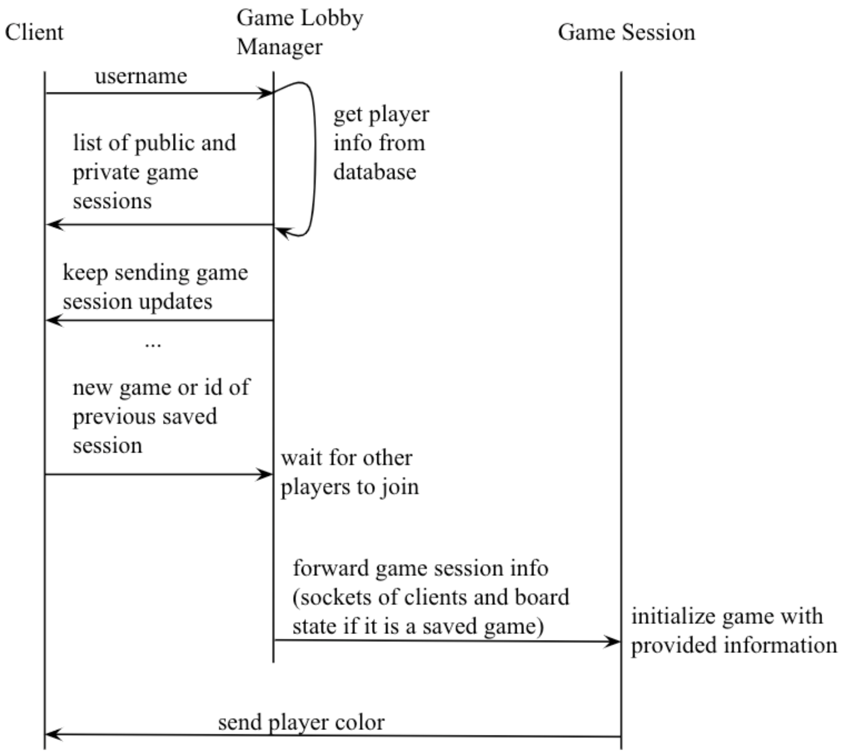

Client
Game Lobby Manager
Game Session
username
get player info from database
list of public and private game sessions
keep sending game session updates
...
new game or id of previous saved session
wait for other players to join
forward game session info (sockets of clients and board state if it is a saved game)
initialize game with provided information
send player color

Fig. 13. Image of the GUI midgame

- *Game Lobby Manager*

To join a game session and start playing the game, the client software must first establish communication with the game lobby management thread. The client communicates with the game lobby manager through TCP. A thread is constantly listening for new connections from clients. After receiving a new connection, the thread creates a new player object and adds the socket descriptor to the object and places it in a queue protected by a mutex for the game lobby manager thread to process.

The client software sends the game lobby manager its username and the game lobby management server. It then queries the player information collection in the MongoDB Capstone database to see if the username exists. If the username exists, the game lobby manager fills the player object with the player's id as well as a list of saved games that it got from the query to the player information collection. If the username does not exist, then the game lobby manager then creates a new entry in the database for the user.

The game lobby manager maintains a map of public game sessions that any player can request to join. Information saved about the public game sessions include the name of the session, the session id which is a unique number, the number of current players that have joined the session, and the total number of players that need to join the session before it can begin. The total number is customizable and can range from 2 to 4 depending on how many players the user creating the session wants to play with. When players connect with the game lobby manager, the manager also adds their profile to the active player map which maps the player id to information about the player, including their username and a list of game session ids. This list contains the game session ids of the games that the player has saved for later and is used to display the option of joining saved games to only the players that were originally in the game when it was saved.

Another responsibility that the game lobby manager has is sending game session updates to the clients that have connected with the game lobby manager and have not yet picked a game session to join. Those clients will receive updates when a new game session is created, or a player decides to join a game session. Using the map of active players, the game manager lobby can keep track of which players are in-game and which players are in-lobby and should receive updates. When a game session update is triggered, the game lobby manager sends a message in this form: <session_id>:<session_name>:<current_num_players>:<total_player_num>/ <session_id>:<session_name>:<current_num_players>:<total_player_num>/... of all the games that they can join to all the players marked as in-lobby. Note that while all players will get information about public games, some players will get additional game session information if they have previously saved games. This information is then parsed and reflected to the client's GUI.

When the game lobby manager receives a request from a client to join a particular game session, the manager marks the player as in-game and updates the current number of players count that is associated with a game session. When that number reaches the total number of players the game session is removed, and the game lobby manager spawns a game session thread and passes along the information of the players that have joined the session. That game session thread can then initialize the game based on the information provided. This flow is shown in Fig. 13 above.

- *Game Sessions*

The main purpose of the game session thread is to forward along piece information from one client to all the other clients in the game. The first thing that the game session server does is send each player the color that they will be. The color is dependent on the order that the players joined the game except when restarting a saved game. In that case the color of each player is the original colors of the players when the game was saved. By sending the colors over to the client software, the client software then knows that the game is starting and then transitions to the replay state. In the replay state, the client software listens for pieces sent over by the server and does not allow inputs from the user until the game session server sends over an end of replay packet.

If the game session is not a saved session, then the game session thread will immediately send over the end of replay packet to the players. Otherwise, the game session thread will use the list of pieces associated with the saved game session which is stored in the game session collection in the database. The ordering in this list is the order that the pieces were originally placed. Thus, a slight artificial delay is added between the game session thread sending replay pieces over to the clients so that the clients get a nice replay of the order that pieces were placed originally which helps them get a refresher on the game before it begins again. Part of the game of Blokus is trying to counter the moves of the opponent, as such, it is important to know the order of the pieces that were placed, and it would be too jarring if the pieces all showed up at once.

When the replay stage ends, the game session thread starts listening for a piece from the player whose turn it is. The server then forwards the piece to all the other players in the game. It also appends the piece to the list of pieces that it maintains. Along with pieces, players can also send over two other types of packets to signal either a save request or a give up request. To end a game each of the players must send a give up request in which case the game session server will then exit the loop and free the piece list that it has maintained since the game ended without any player saving. If the game session that ended was a saved game, the game session thread will also remove the game session id from each of the player's saved game session list because the game has then been officially finished. On the other hand, only one save request is needed before the game session saves the game. For the game to be saved, the game session id must be saved along with a list of player ids and the list of pieces. Also, on each of the player profiles the current game session must be appended if it is not already in the list.

D. *LED Matrix Display*

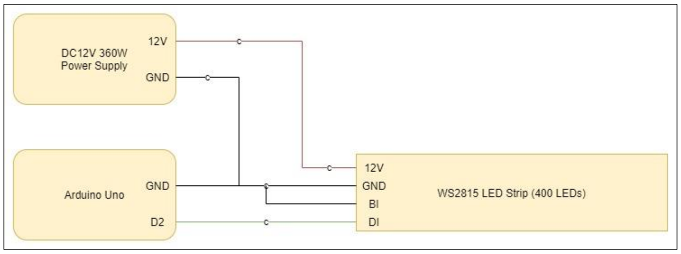


Fig. 14. LED Matrix Display Schematic

The LED matrix display used to display opponents' moves to the player is constructed from three main components: a DC12V 360W power supply, an Arduino Uno, and a WS2815 individually programmable RGB LED strip, as shown in Fig. 14 above. This matrix display is installed under the Blokus board and indicates the pieces played by opponents of three different colors.

It is expected that the LED strip will draw at a maximum around 120 Watts from the power supply, when all color channels are at full brightness. However, the LEDs will likely not be used at maximum brightness due to the proximity of the player to the Blokus board; LEDs that are too bright will hurt the player's eyes. The power supply takes in 110V (via power cable plugged into the wall) and has a 12V output which is connected to the 12V input of the LED strip. The LED strip's ground (GND) is connected to the power supply's ground as well as the Arduino Uno's ground. The data input (DI) of the LED is connected to D2 of the Arduino Uno. The backup input (BI) is connected to ground; each individual LED on the strip has a backup input that is connected to the data input of the previous LED such that if one LED in the chain were to malfunction, it would not affect LEDs connected to it.

The Arudino Uno receives from the client software via serial communication (using the SerialPort Library[17]) the coordinate and color of the square on the board that needs to be lit up. This information will be used in conjunction with the software written in the Arduino IDE using the NeoPixel[1] Library to send the necessary output through the D2 output to light up the corresponding square. The NeoPixel Library was selected because it is a widely used and well documented LED strip control library. The Arduino Uno can perform digital writes to D2 within a few microseconds, so it does not contribute significantly to our overall latency.

VI. Project Management

A. *Schedule*

Refer to the last page for our Gantt chart. Our schedule breaks the timeline down into mostly 4-day chunks, which we believe is necessary to achieve a mini milestone. While we had originally built-in slack into our Gantt chart nearer the end to give ourselves time to manage any integration issues, we ended up needing most of the slack time due to additional complexities during integration and small fixes that we needed once we got the game up and running. Ultimately, we mostly stayed on track and managed to complete everything we expected to achieve for the project, most notably meeting our low latency requirement. Additionally, we are all mainly helping with integration together because it would be too challenging to try and integrate each other's parts individually. The team members working on each part relevant to the integration worked together to assist in the integration of the relevant portions.

B. *Team Member Responsibilities*

Aria is responsible for the turn-based game portion of the client software as well as the web communication portion of this project. Her tasks include the following: researching the most suitable web services needed to communicate player moves to allow remote gameplay and writing software that handles interactions between players which consists of receiving moves from opponents, communicating moves to opponents, and checking for valid/invalid moves.

Jonathan is responsible for the player move detection through computer vision. His tasks consist of the following: setting up the camera, researching OpenCV library, and writing the computer vision portion of the client software to find all coordinates of squares that are in the players' color.

Nadine is responsible for the LED matrix display that indicates players' moves on the Blokus board. Her tasks include assembling the components for the LED matrix display, writing Arduino software needed to control the LEDs, and integrating the LED matrix into the Blokus game board.

To ensure smooth interfacing between the three portions of the project, Aria and Nadine will work together on the serial communication between the client software and the Arduino. Jonathan and Aria will work together on the communication of Blokus board state between the computer vision and turn based portion of client software.

C. *Budget*

Our group has spent a total of $269.76 so far (including the cameras we are buying this week). Fig. 15 shows our AWS Usage and Fig. 16 shows our overall cost, broken down by the various components.

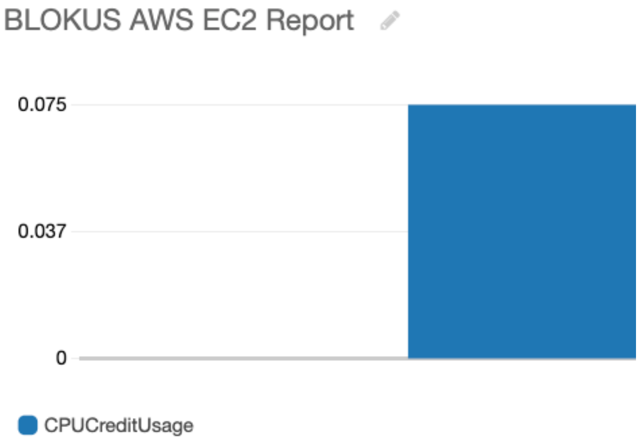


Fig. 15. AWS Usage

Purchases	**Costs**
2x WS2815 LED Strips (Ip30, 16.4ft, 300 LEDs)	$107.86
DC12V 30A Power Supply	$20.12
Power Cable	$10.30
Arduino Uno w/ USB Cable	$30[1]
2x Blokus Board Game	$49.98
2x Logitech Webcam	$81.50
Amazon Web Services Credit	$0.75 [1][2]
Hand Drill	$12.71
Ethernet to USB Dongle	$13.90[3]
Total Cost Spent on Project:	$296.37[4]

Fig. 16. Cost breakdown of various components

D. *Risk Management*

One risk that the team identified was that it will be challenging to meet latency goals of <150ms. This latency is the sum of computer vision execution time, turn based software execution time, web communication latency, and Arduino digital pin output latency. The team was most concerned about the computer vision processing time as preliminary tests showed that filtering and transforming one image for color detection takes 7 milliseconds in OpenCV C++. As this is only a portion of the computer vision client software and one of many images that need to be processed every turn, our latency may not hit the target goal of 150ms. The team will mitigate this risk by examining the execution time of each stage of the computer vision color and coordinate detection process and determine in which ways a stage can be sped up if needed.

Another challenge that the team may encounter is external factors affecting the accuracy of piece detection. The accuracy of detection could possibly vary depending on the lighting condition of the room. Accuracy could also differ between using a Blokus board with or without LEDs installed. Hence, it would be necessary to test the accuracy of piece detection in various lighting conditions and with different states (color, on/off, brightness) of the LEDs. This would help the team determine the optimal conditions for piece detection as well as adjust the computer vision software to be effective in a larger variety of environments. Another external factor that would affect the accuracy of piece detection would be hands and

[1] This part came from our inventory.
[2] Equivalent to 0.075 credits

[3] Bought for the Raspberry Pi, which we did not use as we swapped to an Arduino Uno instead.
[4] Theoretical Total Cost (including parts from inventory): $327.12

external objects that may come in between the camera and the Blokus board. To mitigate the effects such conditions would have on piece detection accuracy, resampling of the Blokus board will be performed.

One other challenge the team faces is the timeliness of the LED matrix display construction. Without the display constructed and integrated into the Blokus board, it will not be possible to test computer vision software on the final state of a board, which is a Blokus board equipped with working LEDs. To mitigate the effects of such risks, the team will prioritize LED matrix display construction after the design review such that it can be available for computer vision and gameplay testing.

VII. System validation and results

A. Overall System Results

Component	Req.	Testing Method	Result
End-to-end Latency	150ms	End to end latency from piece placed to LED being lit Avg time of individual components, using software timers	CV: 82ms (60-120ms) Server: 28ms (23-37ms) Arduino: 2ms (1.26-2ms) Total: ~112ms
Accuracy	<99%	Accuracy of piece placed on one board to LED lighting up on opponent board, including invalid moves Measured from one board to lighting up on board with LEDs	20/20 tests passed, accuracy of 100%.

Fig. 17. Overall System Results

Our overall system results have to do with end-to-end latency and accuracy. We discuss the various components of end-to-end latency more in the sections below, but it is summarized for the entire system in this section as well. In addition, overall accuracy of the system was measured by playing 10 games from one board to the opponent's board and ensuring that the piece placed on the first board lit up the respective LED colors on the opponent's board when the move was valid, and to not do anything otherwise. In addition, this also checked that the turn-based system worked as expected and the player turn did not advance when the move was invalid.

As shown in Fig. 17 above, our results meet our requirements that we set out to achieve. Most notably, we invited a tester to try out the entire game for himself, to great success.

B. Computer Vision Results

There was a total of 2 requirements for the computer vision component, for latency and for accuracy. For accuracy, these tests were done incrementally, as described earlier in Section II. Given reasonable lighting conditions, we tested that the computer vision portion could correctly determine the coordinates of the piece placed. Our tests showed that our coordinate identification works under various lighting conditions and passed all the different tests we tried. We had a test for latency as well, and here there was the greatest variance among all the components for latency. Our measurements reflect a range of 60ms to 120ms, with an average of 82ms over 100 timed runs. The range is slightly wide, with occasional measurements taking longer. We have tried to time the individual components within the CV code, but there is a decent variance, notably in how long it takes to convert the image to HSV color scheme, which we have discussed and broken down in Section IV when discussing resizing. These results are summarized in Fig. 18 below.

Component	Req.	Testing Method	Result
Color + Coordinate Identification Video Test	100% pass	Given piece placed in video stream, to correctly determine coordinates of a certain color and send set of 20x20 coordinates to client piece detection software	20/20 tests
Latency	<130 ms	Timing from when image is received to coordinates being sent to client piece detection (100 trials)	Avg: 82ms (60ms-120ms)

Fig. 18. CV Validation Results

C. CV to Client Piece Detection and Move Validation Accuracy

The functionality of the video stream to valid piece detection was tested by playing 10 games with one client that was using computer vision to get player input with the three other players using the GUI as input. During each of the CV client's turns, the user always initially tried to play an invalid move such as trying to place multiple pieces, overlap pieces with turned on LEDs, and playing illegal moves. After checking to make sure that the invalid move does not register, the player then moves the piece to a valid location, and it is checked that the LEDs under the piece light up which signals that the valid move is recognized by the client software. The validation and piece detection process behaved exactly as expected for each game.

D. Move Validation Latency

Most of the processing time needed to be allotted to the computer vision and client to server communication, so it was crucial for the move validation process to take a negligible amount of time which our group defined as sub 1ms. Through the optimizations described in the game logic section of the paper, the move validation latency was brought down to an average of 0.047ms with a maximum of 0.073ms and a minimum of 0.035ms. This measurement was done by timing 100 trials of how long it took for the GUI to receive a mouse input to when the piece would display on the screen. This is well under the goal of 1ms which meant that the move validation latency was negligible as desired.

E. Client To Server To Client Latency

For piece information to be sent to other clients in the game, the piece must first be sent to the central server which then relays the piece to the other players in the games. Thus, it must be ensured that the client to server to client transmission time takes a minimal amount of time so that placed pieces can appear on other players' boards with minimal delay.

The goal for this portion of the project was to have the client to server to client latency be sub 40ms. Through 200 trials of round-trip time testing, the average latency was measured to be 28.4ms with the maximum latency of 37.4ms and a minimum latency of 23.5ms. This means that the goal of being sub 40ms was reached, and it can be noted that even the maximum latency was smaller than the goal. This allowed additional time to be allotted to other parts of the system.

F. Concurrent Games

One of the goals for the server side of the software was to allow multiple games to occur concurrently. This was tested by spawning multiple clients and having them all play game sessions concurrently. During testing 9 simultaneous game sessions were created, with each session having two players. This testing was limited by the number of clients that can be spawned locally on one machine. The server was functioning as normal with 9 concurrent games and was still sending game session lobby updates in a timely manner. Pieces were also showing up almost simultaneously on the opponent boards. It would have been interesting to stress test the server more, but the local resources were not adequate to do such a thing since each client was running a GUI using OpenGL that was updating very frequently. Nonetheless, this testing showed that the server meets the goal of allowing multiple games to be played at the same time just like an actual dedicated game server. Fig. 19 below shows an example of three concurrent game sessions in progress.

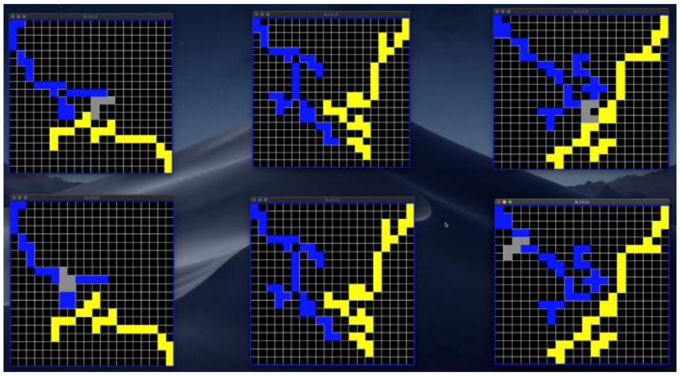


Fig. 19. Example of Three Concurrent Game Sessions

G. Game Session Display Accuracy

The game session display was first tested by spawning two client processes that both stay connected to the game session manager thread throughout the testing process without entering any games. Then six more threads were spawned, and each created a new game session with different names. The GUI of the two initial clients is checked to ensure that each game session is displayed correctly and arrived in a timely manner. Then all the clients are killed, since the number of clients that can be spawned is limited by local resources. After all the clients are killed, the process is repeated five more times. For each trial done, the two clients always displayed the correct game sessions, and they were always displayed in a timely manner.

The second test checked that when other players joined a game session, the number of current players in the session updated to the correct amount. Also, it checked that when the current number of players reached the total players, the game session is no longer displayed. Two client processes were initially spawned and stayed connected to the game session manager throughout the testing process without entering any games. Then four other clients are spawned and one of those clients creates a new game session with a capacity of four players. Then the other three clients join the game, and when they do, the GUI of the two initial clients are monitored to make sure the current number of players is updated correctly. Finally, when the last of the three clients join the game session, the GUIs of the original clients are checked to make sure that the game session has disappeared. Then all the client processes were killed, and this process was repeated three more times.

H. Saved Game Replay Accuracy

The functionality of the replay was first tested by playing the game with four clients. At some point during the game, the state of the board is recorded and then one of the players initiates a save. Then the players all disconnect and reconnect to the game lobby manager and they all join the saved game. After the replay finishes the state of the board is compared against the state of the board that was recorded. This process was repeated ten times and each time the boards were identical. A summary of the client and software validation results are shown in Fig. 20 and Fig. 21 below.

Component	Req.	Testing Method	Result
CV to Client Piece Detection and Move Validation Accuracy	100% pass	Given piece placed, local client to determine validity Placed a piece down, and measured if software correctly responded to valid/invalid move	10/10 tests
Valid Move Verification Latency	<1ms	Timing when GUI received mouse input to displaying on screen (100 trials)	Avg: 0.047ms Max: 0.073ms Min: 0.035ms

Fig. 20. Client Software Validation Results

Component	Req.	Testing Method	Result
Client To Server To Client Latency	Less than 40ms	Sending messages from client to server to client (200 trials)	Average: 28.3ms Max: 37.4ms Min: 23.5ms
Concurrent Games	More than 3 at once	Number of concurrent games (Note: this was limited by the resources on the local machine doing the testing, not by the server)	9 concurr-ent games
Game Session Display Accuracy	100% pass	Each game session is displayed correctly and arrives in a timely manner	5/5 tests

Fig. 21. Server Software Validation Results

I. Arduino Software Execution Time

Because this project emphasizes low latency, we needed to make sure that the execution time for the Arduino software, which would handle inputs from the client software and output signals to change the state of the LEDs, would be insignificant relative to our overall latency requirement.

The Arduino software takes in 15 bytes which indicate the color and coordinates (row and column) of 5 squares (max number of squares a Blokus piece can have). Based on this information, the software will determine which LED index needs to be changed and the color it needs to be changed to. Then it will write to D2, the digital output pin on the Arduino Uno connected to the digital input (DI) of our LED matrix to change the state of the LEDs. The average execution time of this software over 10 tests is around 0.78ms, making it a very small portion of our overall latency.

J. LED Behavior Accuracy

To determine if the LEDs are behaving as expected, we preformed over 20 visual tests. These tests were simply done by having the client software send the color and coordinates of Blokus pieces to the Arduino examining that the respective LEDs lit up in given color. We passed visual inspection for 20 of the 20 tests we had planned to do. The LEDs behavior is accurate and as intended. A summary of the LED verification results is shown in Fig. 22 below.

Component	Req.	Testing Method	Result
Arduino Code Execution Time	<1ms	Time necessary to: Parse row, col, color Calculate LED index Determine RGB values	~.78 ms 10 timed code execut-ion
Accurate LEDs Behavior	100% pass	Expected color and coordinates of each piece Given a color and coordinate, to light up that respective LED that color	20/20 visual tests passed

Fig. 22. LED Validation Results

VIII. Related Work

There were many sources of inspiration for this project. Other capstone groups in the past have attempted to create a physical representation of a board game that can communicate with other players that are not in the same physical location. For example, the Spring 2019 Capstone group Team AC created a Smart Chess board [13] that allows players to play chess with each other from far away on a physical board. It lights up to show legal and illegal moves by using lights and sensors under a transparent chess board. While our team was inspired by the project, the approaches that this team took was not feasible for the board game that was chosen. Another capstone team during Spring 2016 created a project called Catan-omous Dealer which featured an autonomous dealer for distributing resource cards for the board game Settlers of Catan. While the aim of their project was very different from this project, we were inspired by their use of computer vision to keep track of board state to determine what cards to deal to each person.

Another product that is like our project is Mind Sport International’s Scrabble board [7] that uses RFID chips to detect tiles and transmit tile information to other players in the game. This allows players to instantly see what words and scores players have. However, to purchase one of these game boards, each player would have to spend about $31,800 for this sophisticated technology. In comparison, our budget is to be under $600 where possible. Given the similar idea to be able to transmit game information in real-time, at a fraction of the course, we believe our prototype is a cost-effective solution with minimal loss in terms of latency that does not really impede the game flow.

A direction that the team was considering was an AR version of Blokus so that no physical pieces are needed at all. While we decided to pivot away from this idea due to some of the limitations, we felt came with using AR technology, there are products out there that focus on bringing AR to the board game genre. One of the most prominent of these products is the Tilt Five, which uses a game board, AR glasses and a wand controller. When players put on the AR glasses, they can see animated pieces as 3D holograms on the table. Interacting with these pieces are done using the wand. Due to the complex nature of all these technologies and the limited time capstone imposes, our team decided against a similar approach since it seemed outside the scope of capstone. However, this approach is an interesting take on the board game genre that we would like to explore in the future.

IX. Summary

The team managed to reach the broad goals of the project and created a product that we believe is well-suited for the purpose of playing a physically distanced game of Blokus. We managed to meet the latency requirement we set for ourselves, and we believe that this project is a good base to use to adapt other similar board games as well. The current latency is a low bound already, given the need to process an image as well as to send it over the server to other players. In addition, even when playing the game using our board, the latency difference is not noticeable, and the additional delay does not detract from the playing experience.

In the future, using the framework we developed during capstone, all grid-based board games can be adopted to use similar technologies using the hardware, CV, and infrastructure we created. There is also opportunity for this project to be shifted into the realm of augmented reality as that is what the state of board games seems to be heading toward.

We had several lessons learnt throughout this project. The first was that unexpected issues can and will arise during integration. For our project, we stumbled heavily during integration, particularly when dealing with Mac and Windows differences and libraries in the tools we used. A second issue was always making sure we had backup components. This was the main reason for our switch from a Raspberry Pi to an Arduino. This was also vital in ensuring that our LED board was buildable, as the LEDs were prone to breaking. Another issue we encountered was compatibility. There is always a tradeoff to be made between ease of use and performance. In our case, we were optimizing heavily for latency, but the result of that was libraries that were harder to use or where documentation was not as clear. One instance of this was the documentation between OpenCV in Python and in C++, and another instance was using the C++ MongoDB driver instead of the python MongoDB driver which would have been easier to set up. If we had less time for this project, perhaps we could have gone the well documented path and to use more compatible technologies and sacrifice some latency in the process. Lastly, planning is great, but it is impossible to plan for every single contingency plan and foresee every possible problem. And so, it is important to also learn when to just dive into the problem. This was especially true for the LED board construction. There was a lot of worry initially about the LEDs and how a lit LED might bleed into the surrounding squares, but I think the images show that the bleeding is minimal (Refer to Fig. 6 above).

Lastly, if future capstone teams are considering attempting to create a physical board game that can be played while players are in different locations, this team would recommend them consider all the possible approaches discussed in the trade discussion portion of the paper for creating the hardware for the board itself. CV may not be the best way to take if the tiles on the board are very large, or if there are not that many tiles. In addition, if teams are very good at designing PCBs, that might be an interesting approach to take as well. Overall, future teams should make sure that the approach they take fits the board they are trying to emulate.

References

[1] Adafruit. (n.d.) Adafruit NeoPixel Überguide. Retrieved from https://learn.adafruit.com/adafruit-neopixel-uberguide/arduino-library-use

[2] AWS Latency Rivenes, Logan. (2016, April 28). “Optimizing Latency and Bandwidth for AWS Traffic.” *Amazon*, AWS, aws.amazon.com/blogs/startups/optimizing-latency-and-bandwidth-for-aws-traffic/

[3] Blokus Game Guide wikiHow Staff Editor. (2020, April 2). “How to Play Blokus.” *WikiHow*, www.wikihow.com/Play-Blokus.

[4] DB Query Time K. R. (2020, April 16). 11 Things You Wish You Knew Before Starting with DynamoDB. Retrieved October 17, 2020, from https://blog.yugabyte.com/11-things-you-wish-you-knew-before-starting-with-dynamodb/

[5] EC2 “Amazon EC2 Alternatives & Competitors.” *G2*, G2, www.g2.com/products/amazon-ec2/competitors/alternatives.

[6] Fernando, S. (n.d.). Install OpenCV with Visual Studio. Retrieved from https://www.opencv-srf.com/2017/11/install-opencv-with-visual-studio.html

[7] Fincher, J. (2015, May 02). The world's most high-tech (and expensive) Scrabble board. Retrieved October 17, 2020, from https://newatlas.com/worlds-most-high-tech-and-expensive-scrabble-board/25097/

[8] Goodstadt, L. (2011, Aug 18) “Generic Hash for Tuples in unordered_map / unordered_set.” Stack Overflow. Retrieved from https://stackoverflow.com/questions/7110301/generic-hash-for-tuples-in-unordered-map-unordered-set

[9] Hachcham, A. (2020, April 21). Install and configure OpenCV-4.2.0 in Windows 10-VC++. Retrieved from https://towardsdatascience.com/install-and-configure-opencv-4-2-0-in-windows-10-vc-d132c52063a1

[10] Harvey, Cynthia, and Andy Patrizio. (2020, March 17). Pros vs cons of cloud computing resources “AWS vs. Azure vs. Google: Cloud Comparison.” Datamation, www.datamation.com/cloud-computing/aws-vs-azure-vs-google-cloud-comparison.html.

[11] Holographic Tabletop Gaming. (n.d.). Retrieved October 17, 2020, from https://www.tiltfive.com/

[12] Pubnub (2020, May 08). How Fast Is Realtime? Human Perception and Technology. Retrieved from https://www.pubnub.com/blog/how-fast-is-realtime-human-perception-and-technology/

[13] Team AC: Smart Chess board. (2019, May 06). Retrieved October 17, 2020, from http://course.ece.cmu.edu/~ece500/projects/s19-teamac/

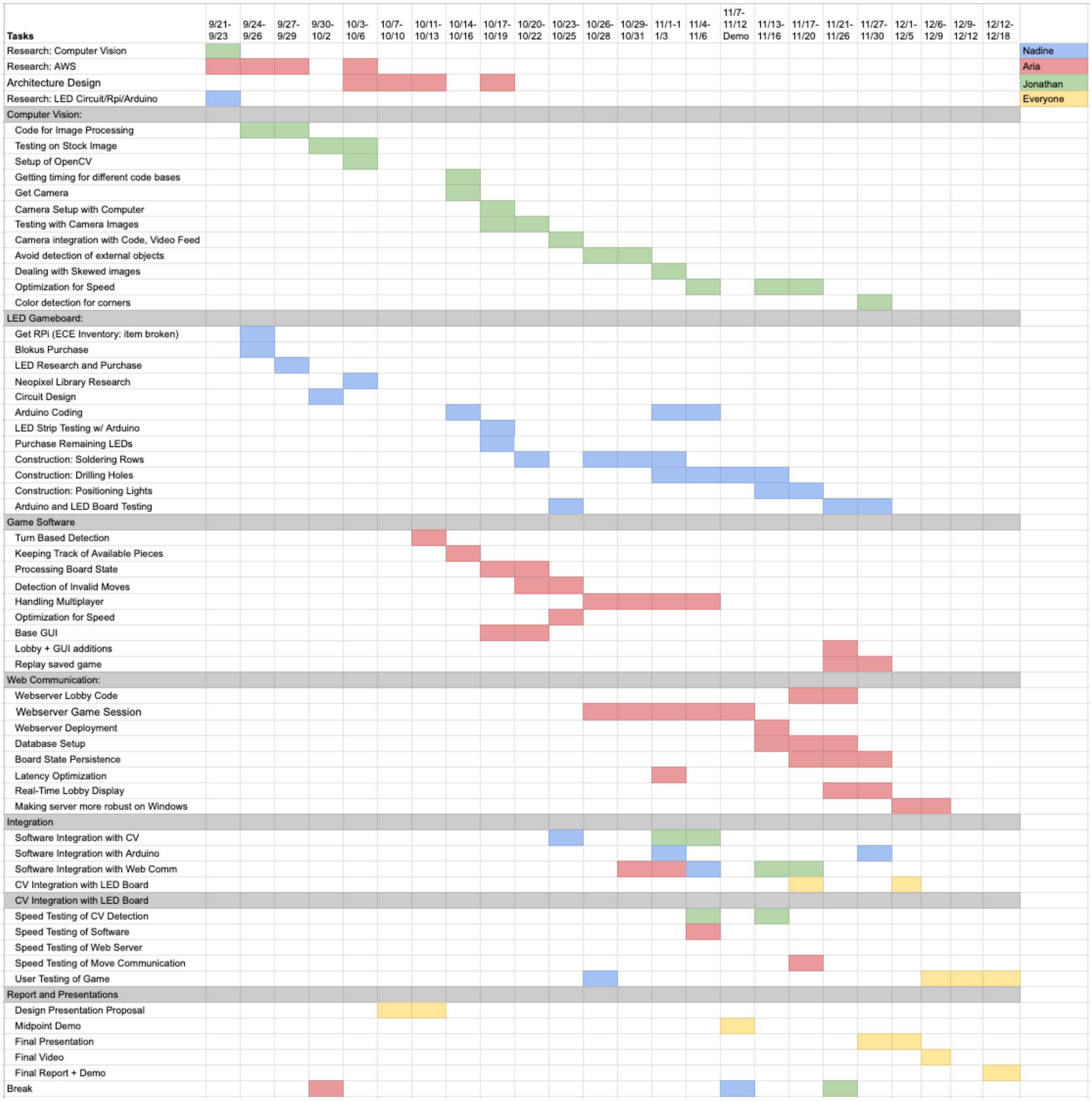
[14] The Hook Up. (2019, August 05). The COMPLETE guide to selecting individually addressable LED strips. Retrieved October 17, 2020, from http://www.thesmarthomehookup.com/the-complete-guide-to-selecting-individually-addressable-led-strips/

[15] The Khronos Group. (n.d.). The Industry's Foundation for High Performance Graphics. Retrieved from https://www.opengl.org/

[16] Watson, R. (2020, May 10). Multiple Color Detection in Real-Time using Python-OpenCV. Retrieved from https://www.geeksforgeeks.org/multiple-color-detection-in-real-time-using-python-opencv/

[17] ZainUIMustafa. (2018, July 16). Connect-And-Use-Arduino-via-Cpp-Software-Made-In-Any-IDE. Retrieved from https://github.com/ZainUlMustafa/Connect-And-Use-Arduino-via-Cpp-Software-Made-In-Any-IDE

APPENDIX A



Tasks	9/21-9/23	9/24-9/26	9/27-9/29	9/30-10/2	10/3-10/6	10/7-10/10	10/11-10/13	10/14-10/16	10/17-10/19	10/20-10/22	10/23-10/25	10/26-10/28	10/29-10/31	11/1-11/3	11/4-11/6	11/7-11/12 Demo	11/13-11/16	11/17-11/20	11/21-11/26	11/27-11/30	12/1-12/5	12/6-12/9	12/9-12/12	12/12-12/18	
Research: Computer Vision	🟩																								🟦 Nadine
Research: AWS	🟥	🟥	🟥		🟥																				🟥 Aria
Architecture Design					🟥	🟥	🟥		🟥																🟩 Jonathan
Research: LED Circuit/Rpi/Arduino	🟦																								🟨 Everyone
Computer Vision:																									
Code for Image Processing		🟩	🟩																						
Testing on Stock Image				🟩	🟩																				
Setup of OpenCV					🟩																				
Getting timing for different code bases								🟩																	
Get Camera								🟩																	
Camera Setup with Computer									🟩																
Testing with Camera Images									🟩	🟩															
Camera integration with Code, Video Feed											🟩														
Avoid detection of external objects												🟩	🟩												
Dealing with Skewed images														🟩											
Optimization for Speed															🟩		🟩	🟩							
Color detection for corners																				🟩					
LED Gameboard:																									
Get RPi (ECE Inventory: item broken)		🟦																							
Blokus Purchase		🟦																							
LED Research and Purchase			🟦																						
Neopixel Library Research					🟦																				
Circuit Design				🟦																					
Arduino Coding								🟦						🟦	🟦										
LED Strip Testing w/ Arduino									🟦																
Purchase Remaining LEDs									🟦																
Construction: Soldering Rows										🟦		🟦	🟦	🟦											
Construction: Drilling Holes														🟦	🟦	🟦	🟦								
Construction: Positioning Lights																	🟦	🟦							
Arduino and LED Board Testing											🟦								🟦	🟦					
Game Software																									
Turn Based Detection							🟥																		
Keeping Track of Available Pieces								🟥																	
Processing Board State									🟥	🟥															
Detection of Invalid Moves										🟥	🟥														
Handling Multiplayer												🟥	🟥	🟥	🟥										
Optimization for Speed											🟥														
Base GUI									🟥	🟥															
Lobby + GUI additions																			🟥						
Replay saved game																			🟥	🟥					
Web Communication:																									
Webserver Lobby Code																		🟥	🟥						
Webserver Game Session												🟥	🟥	🟥	🟥	🟥									
Webserver Deployment																	🟥								
Database Setup																	🟥	🟥	🟥						
Board State Persistence																		🟥	🟥	🟥					
Latency Optimization														🟥											
Real-Time Lobby Display																			🟥	🟥					
Making server more robust on Windows																					🟥	🟥			
Integration																									
Software Integration with CV											🟦			🟩	🟩										
Software Integration with Arduino														🟦						🟦					
Software Integration with Web Comm													🟥	🟥	🟦		🟩	🟩							
CV Integration with LED Board																		🟨			🟨				
CV Integration with LED Board																									
Speed Testing of CV Detection															🟩		🟩								
Speed Testing of Software															🟥										
Speed Testing of Web Server																									
Speed Testing of Move Communication																		🟥							
User Testing of Game												🟦										🟨	🟨	🟨	
Report and Presentations																									
Design Presentation Proposal						🟨	🟨																		
Midpoint Demo																🟨									
Final Presentation																				🟨	🟨				
Final Video																						🟨			
Final Report + Demo																								🟨	
Break				🟥												🟦			🟩						

Fig. 23. Gantt Chart