Gammix: Mixed-Reality Multiplayer Board Games

Brandon Liu, Eric Chien

Advisor: Lin Zhong

A Senior Project

Submitted to the Department of Computer Science

Yale University

In Partial Fulfillment of the Requirements

for the B.S. in Computer Science

May 2022

Abstract

Many of us have played games like chess online. The online versions of multiplayer board games are fun, but lack the tactile experience of physically moving game pieces and seeing the board and gameplay live in 3-D space. But what if you could still play with others online, but use the real-life, physical board game in a straightforward way? To explore this question, for our senior project we designed and built a system called Gammix for such mixed-reality multiplayer board game experiences. Gammix brings the offline board games experience online by using a robotic arm to recreate opponents' actions on your physical board. Each move is captured using a standard webcam, interpreted using computer vision, then sent over the internet and replicated using a robotic arm on the opponent's boards. We implemented Gammix and demonstrated its feasibility for the board game Settlers of Catan, using a standard USB webcam and Trossen PincherX-100 robotic arm. We make use of Python, OpenCV, and the ROS MoveIt framework, as well as Google Cloud Pub/Sub for communication over the internet. We use classical computer vision techniques, including thresholding, Canny edge detection, the watershed algorithm, and Harris corner detection to detect the board and subsequent moves. We also designed our own representation of the board state, to translate coordinates from the camera frame to coordinates in the robotic arm frame. The system works end-to-end: after one player makes their moves on the board and ends their turn, Gammix detects their moves locally, then issues the commands over the internet for the opponents' robot arms to replicate them remotely.

Acknowledgments

We would like to acknowledge the professors and advisors who have helped us get to this point today. Thank you to DUS Richard Yang, senior class advisor, Prof. Drago Radev, and CPSC 490 coordinator Inyoung Shin for organizing and making the senior project possible. We'd also like to acknowledge Prof. Marynel Vazquez, Man-Ki Yoon, Prof. Aaron Dollar, and Andrew Morgan for their advice early on in our senior project. We express our gratitude to our early professors in computer systems, Prof. Abhishek Bhattacharjee, Prof. Zhong Shao, Prof. Rajit Manohar, and the late Prof. Stan Eisenstat, for providing us with the fundamentals of computer science and ample opportunity over challenging semesters for the two of us to work together and become friends. In spring 2020, after everyone went home due to the pandemic, the two of us stayed in contact, attending Prof. Abhishek's CPSC 323 lectures and bouncing ideas off of each other for the psets. There was a distinct moment, perhaps over Zoom, when the both of us simultaneously realized that it was seeing each other continue to work hard in 323 that pushed each of us individually to continue challenging ourselves (and not just take the easy route, given that it was all pass/fail during remote classes). In that way, we found a recursive loop of motivation, and we certainly have each other to thank!

Lastly, we would like to say our sincere thanks to our advisor Prof. Lin Zhong for being an especially thoughtful, patient, and supportive professor and mentor. Brandon would like to thank Prof. Zhong in particular for the opportunity to work with him in the Efficient Computer Lab in the summer of 2020 after his internship was canceled due to the pandemic. That summer proved an especially formative experience in his computer science education. We appreciate Prof. Zhong's time, advice, and ideas — like combining computer vision with a robotic arm for our project — which we ended up very happy about as the process and results were exciting to us and we both learned a great deal.

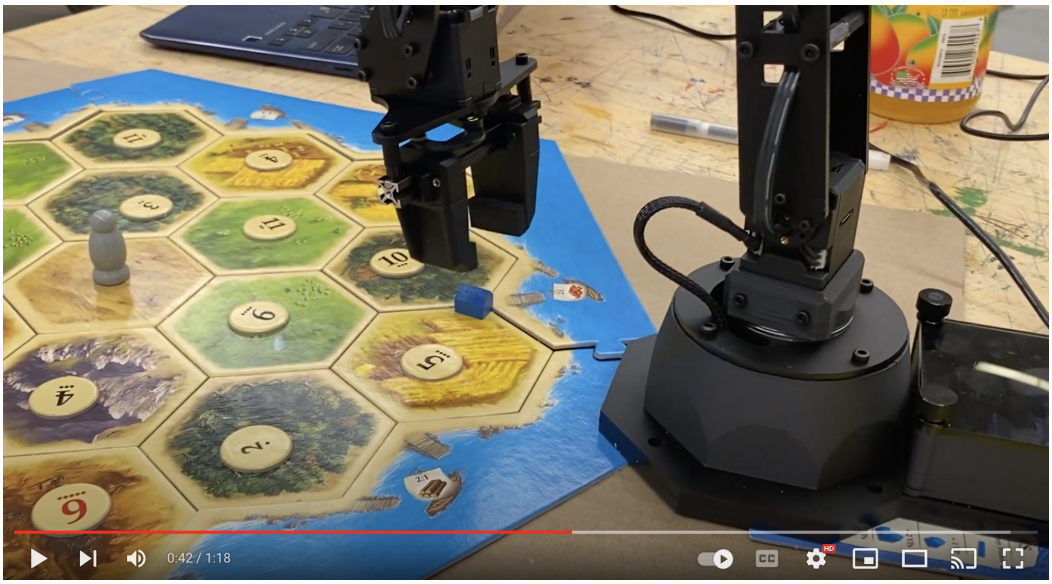
Introduction and demonstration

Gammix brings the offline board games experience online by using a robotic arm to recreate opponents' actions on your physical board. Each move is captured using a standard webcam, interpreted using computer vision, then sent over the internet and replicated using a robotic arm on the opponent's boards. This system differs from our original proposal. Originally, we proposed using a projector to display an interface showing the opponents' moves atop the board, thus allowing the player to easily update the board state based on it. Instead, we have designed and built our final project around a robotic arm as the actuator, such that the opponents' moves are carried out automatically. This way, gameplay is not interrupted by the player having to update the board based on their opponents' moves, as it is done automatically, so the overall experience is much more seamless.

There were multiple key challenges and open questions at our project's outset: 1) whether a single standard webcam could accurately detect the board game state, and what computer vision algorithms would help 2) how best to represent the board state for multiplayer game purposes, 3) how to design a software architecture to synchronize new actions with potentially

multiple remote opponents, and 4) how to translate pixel locations from the webcam frame to the robot arm frame so that the arm knew where to move a piece and could actually do so when it received commands. We were able to solve all of these challenges by the end of the semester and will detail how we came to our specific solutions in this report.

We implemented Gammix and demonstrated its feasibility for the board game Settlers of Catan, using a standard USB webcam and Trossen PincherX-100 robotic arm. The system works end-to-end: after the first player makes their moves on the board and ends their turn, Gammix detects their moves locally, then issues the commands over the internet for the opponents' robot arms to replicate them remotely. Linked in the below screenshot is a short video to demonstrate this process.[1]



In the first few seconds, you can also see an image of all vertices detected and represented atop the image of the board. This was a key goal of our computer vision pipeline, or move detection module, which Brandon designed and implemented. Eric was responsible for the programming and providing an interface for the robotic arm to accept moves. Both of us collaborated throughout and iterated frequently on our software design, APIs, and choice of data representation, to connect our two halves of the system. Our system works for an arbitrary number of turns, as well as an arbitrary number of pieces placed per turn, and currently supports placing settlements. Placing roads, upgrading to cities, and moving the robber are left for further work, due to time constraints in one semester. However, the key challenges of detecting, representing, and actuating moves have been addressed, making implementing these additional pieces relatively straightforward after understanding our design.

[1] https://www.youtube.com/watch?v=z9rEOX2ND-E

Prior work

Catan-omous Dealer is an autonomous dealer system for distributing resource cards in the board game Settlers of Catan, built as a capstone project by Carnegie Mellon students.[2] It uses computer vision to detect and track the board state and an array of motorized card dealers to distribute resources. While Catan-omous Dealer is effective at detecting which players are adjacent to what resources, it doesn't identify the precise location of pieces in the webcam frame and translate that to the world frame, which is what Eric and I implemented together to solve the problem of replicating a detected move on the Catan board. We use the same first two algorithms (Canny edge detection and watershed) in our own computer vision pipeline, but take the project in a different direction by identifying vertices so that we could get coordinates. Moreover, Catan-omous dealer relied on modifying the game pieces and a specialized platform to set the board upon, while our system required no such modifications or custom hardware.

Another prior system that relates to our project is Catan Image Recognition, which also detected Catan's board state using computer vision with varying reliability but without any further game actions. As it was open source, we were able to reference a portion of their code for the isolation of the board from the background in our project.[3]

Other projects have programmed robotic arms to play Catan locally, as shown in a Kuka Robotics video with students in OTH Regensburg in Germany.[4] But, based on our research, no other prior work has used a robotic arm to play Catan with remote opponents, over the internet.

We were also inspired by the system built by engineer Shane Wighton that allowed him to play pool against other players online but on an actual pool table.[5] It did so by projecting an interface onto the pool table, and by employing a custom-built robotic pool cue that automatically lines up and makes the shot with the force and angle specified by the online player. While Wighton initially built the robotic pool cue and projector system as a way to guarantee that he sinks every shot algorithmically using software, the added benefit of his system enabling an online multiplayer experience on the physical table turned out to be an absolute highlight and a source of joy.

High-level software design

Our software design consists of two main modules that run on each player's local machine: one for move detection and one for move actuation, with message passing via a singular new_moves topic with a pub/sub model. The move detection module checks if it is the current player's turn, and if so it waits for the player to finish making their moves. Once the player has completed their turn, they press space, and the computer vision node will process the webcam images to detect the type and position of any new moves they made. This information is published to a topic called new_moves, to which all move detection and move actuation modules

[2] https://course.ece.cmu.edu/~ece549/spring16/team08/website/pages/design.html
[3] https://github.com/Vieja/Catan-Image-Recognition
[4] https://www.youtube.com/watch?v=oCQPWv_ky2c
[5] https://www.youtube.com/watch?v=vsTTXYxydOE

subscribe. Upon notification of new moves, the move actuation module on each machine checks if the moves are from an opponent, and if so, plans and executes each move by sending commands to its corresponding robotic arm. When the move detection module receives new moves, it ignores the move data itself and just updates its local turn counter.

We use the pub/sub architectural design pattern as it provides a straightforward way for multiple subscribers to be made aware of new moves and thus changes needed to the board state. Our design provides a minimal interface for turn-based multiplayer games to be implemented. In figure 1, we provide an overview of the Gammix architecture. We show the Gammix design's viability by implementing it for the popular board game Settlers of Catan. In this implementation, figure 1's rectangles correspond to Python programs, diamonds correspond to hardware, ovals correspond to topics, and arrows represent sequential actions/communication. Blue arrows represent publishing, and red subscribing. In figure 2, we show an example of how this architecture progresses over a turn for each player in a three-player game of Settlers of Catan.

Figure 1. Gammix Architecture Diagram

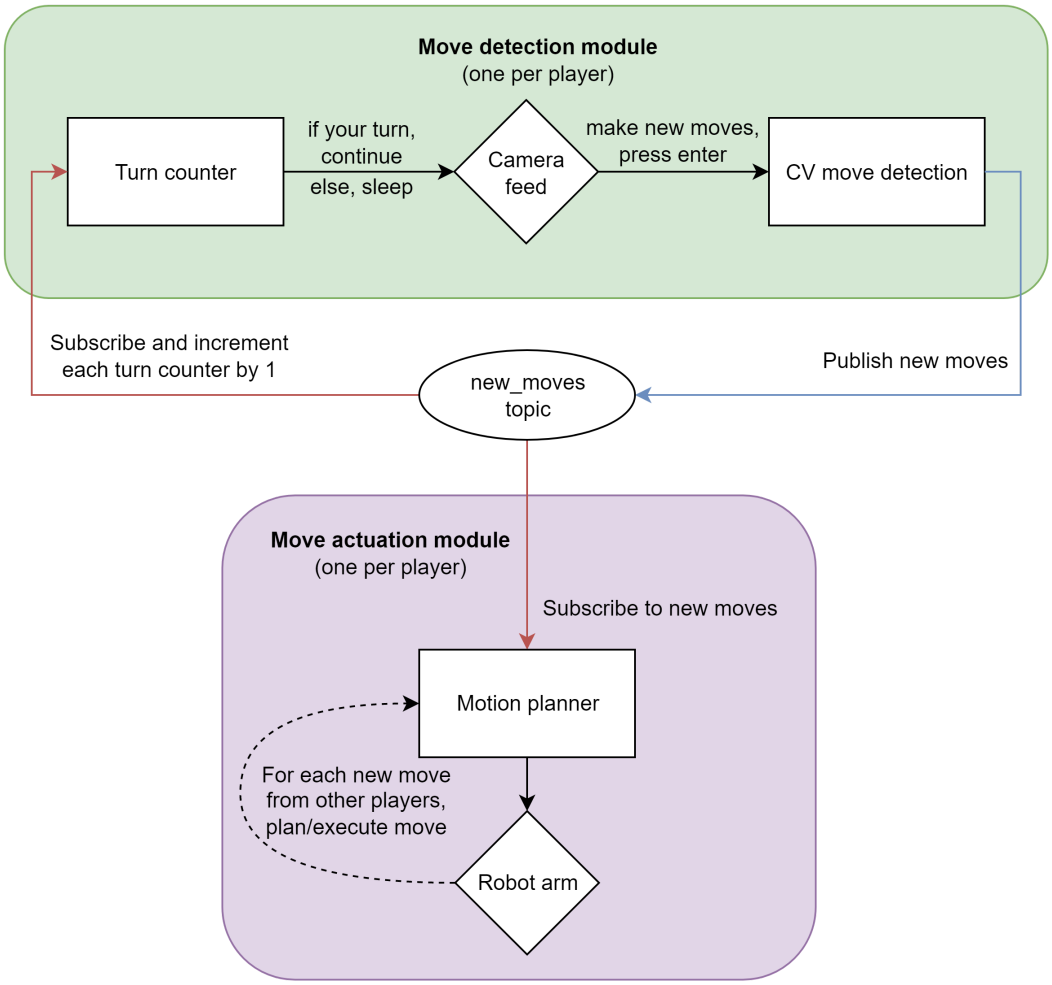
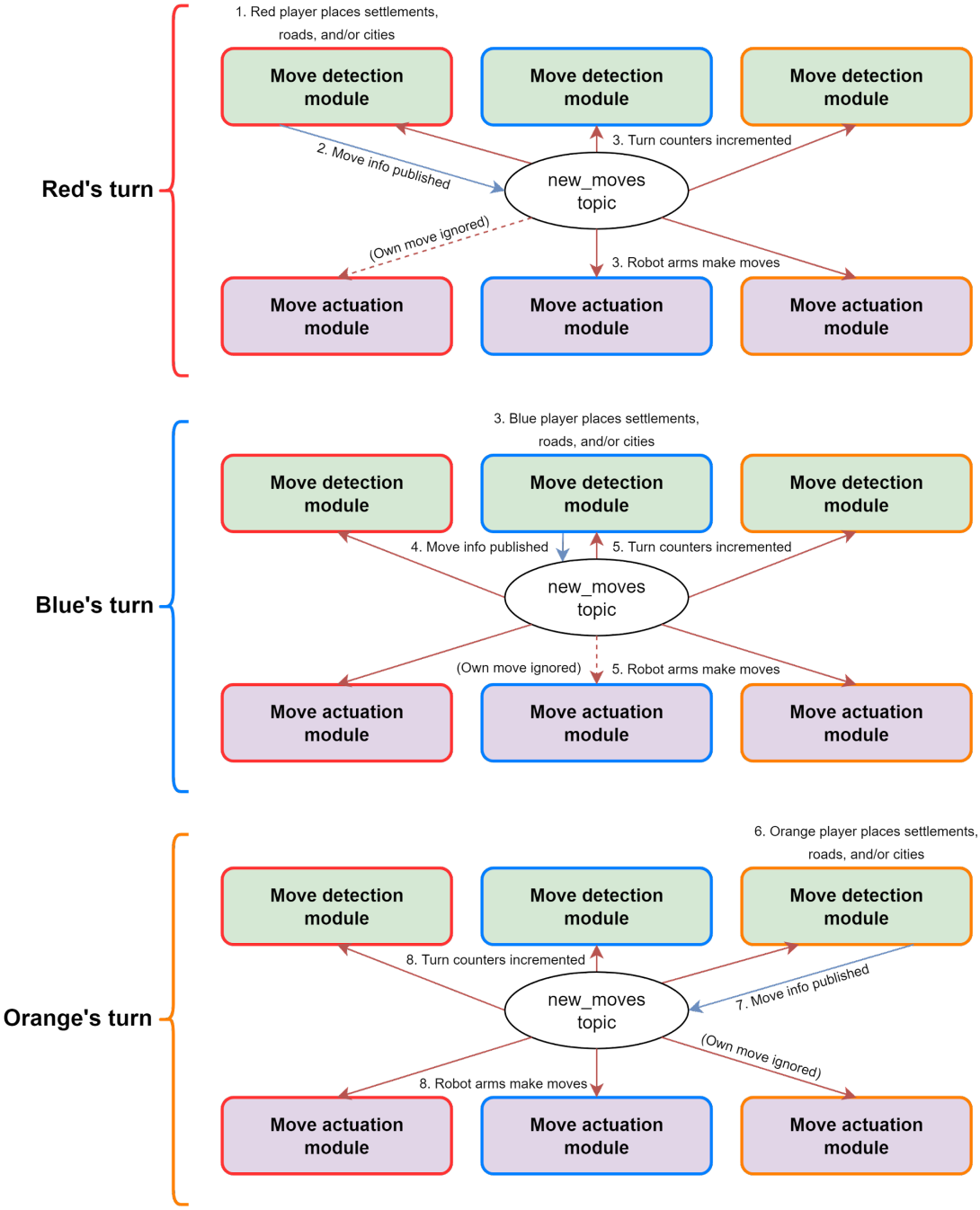


Figure 2. Three turns in a three-player game of Gammix Catan



Board state representation: Design

One of the central questions our project wanted to solve was how we could translate information of the pieces on a board from a camera to the information needed for a robot arm to

make those same moves on a different board. As both the reference frames of the camera and the robot arm are different — the camera frame holds a pixelated bird's eye view of the board, while the robot arm must use relative (x,y,z) offsets from the base in order to move objects — a clear system of synchronization between the two was needed. To accomplish this, we first needed to construct the representation of a board in both the frame of the camera as well as the robot arm. By having the concept of a board in each reference frame, we are able to create a framework to then understand what it means for a piece to be placed on the board. Thus, for each frame (besides the linking pub/sub frame) we have a preprocessing step to generate the board representation and an execution step that handles the translation of piece placement from camera to robot arm.

We began with the camera frame. Since we knew that pieces could only be placed on vertices and edges, we only needed to maintain a list of a finite number of points in an image. Our understanding of what it means for a piece to be placed would then simply be the idea of checking to see which vertex or edge point coincides with the point of a piece on the image. Thus, for the camera frame, we represent the board as a list of pixels containing all possible positions in which a piece could be placed. Similarly, with this board representation, to represent the placement of a piece, we simply detect which pixel the piece was placed on. The next important frame we will discuss is the robot frame. The robot arm needs to understand a similar list of vertices/edges as the camera, but rather than using pixel coordinates, the robot arm needs to understand a list of physical offsets representing each valid placement of a piece. By having a list of physical offsets we can utilize inverse kinematics to plan the movement of the arm to each position of the board. Thus, the board is represented as a list of offsets. Using this board representation, we represent the placement of a piece as a selection of the exact offsets needed to place a piece in the specified location.

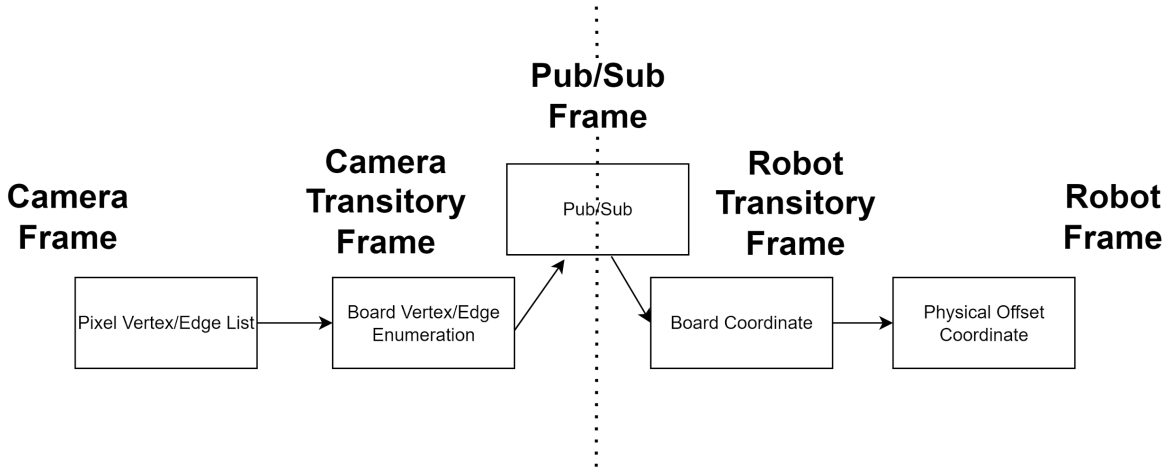
Now that we have established the concept of a board for both the camera and robot arm, we now need a concrete way to translate from one board representation to another. So, we need a series of transitory frames. Let’s start with a camera transitory frame. While our vertex/edge pixel list from the camera frame now provides us the power to represent the important features of the board, we still have no idea how each vertex/edge relates to each other. As in, given a pixel vertex/edge, we have no idea if this represents a vertex/edge at the top of the board, leftmost bottom corner of the board, middle of the board, or any other positional information that the robot arm placing the piece would need to understand. Thus, we create a board representation enumerating the vertices/edges on the board with a translation dictionary converting a pixel to a specified point in the robot arm frame. To represent the placement of a piece, we find the nearest valid board position from the given pixel where the piece was detected.

Similar to the camera, we also need a robot transitory frame. This frame represents the board as another sort of enumeration of points on the board so that we can map board positions to physical offsets, and we can have a notion of the ordering of the physical offsets on the board. Note that even though both transitory frames utilize board positions, the representation of the board positions can and in our case does vary between the two frames. This is done for

convenience purposes which will be explained later in the report. But what this does mean is that the robot transitory frame has an extra role of translating from camera board position to robot board position. The notion of placing a piece is thus the translation of camera board position to robot board position and then subsequently translating robot position to robot offset to represent where the piece would be placed relative to the robot arm. Finally, to connect the two systems together, we have a pub/sub frame that simply acts as a messenger from the camera transitory frame to the robot transitory frame.

Therefore, we have described the major components of our 5 stage board and piece representation pipeline. Let's recap and put it all together. Before any piece processing begins, both the camera and robot frames must do preprocessing to generate their board representation lists. The camera transitory frame must also generate the pixel to board enumeration dictionary while the robot transitory frame must generate dictionaries for both the camera board enumeration to robot board enumeration and robot board enumeration to physical offset. After the preprocessing is finished, we can demonstrate the process of translating the position of a piece from placement to robot arm movement. To begin, a piece is placed on the board. The camera frame then calculates the pixel of the piece and determines the vertex/edge that the piece is covering by comparing it against the vertex/edge pixel list. The camera transitory frame then takes the pixel and converts it to a vertex/edge enumeration which is sent by pub/sub to the robot transitory frame. This frame converts the enumeration to a robot board coordinate and subsequently converts the robot board coordinate to a physical offset. Finally, the robot utilizes the physical offset to move a piece to the same position as captured by the camera.

Figure 3. Board and piece state representation pipeline



Board state representation: Implementation

The code for our implementation is hosted at: https://github.com/RollingPeanuts/senior_project

- /src/ directory contains the source code python scripts for our implementation
- /utility/ directory contains helpful debugging scripts

- start_arm.sh is a script to run the startup bash commands to start the robot arm

Camera frame: Computer vision pipeline for vertex detection

1. Board detection

The vertex detection computer vision pipeline is run once at the start of the game and is used to determine the possible locations of settlements and cities on the board. We begin by first isolating the game board from the raw webcam image to reduce the data we have to further process. We do so by isolating the background by first finding the specific Catan blue on the border of the board, a relatively unique color within the frame, using thresholding on the hue channel of the image. Then, we find the second largest contour from the remaining image data. As the blue color spans the outline of the board, the first largest contour would be the entire board's hexagon, while the second-largest is the hex tile grid, as shown above. Then we use a bitwise_and operation to cut out the background using the second-largest contour as a mask. This step is taken from a prior open-source Catan Image Recognition project.[6]

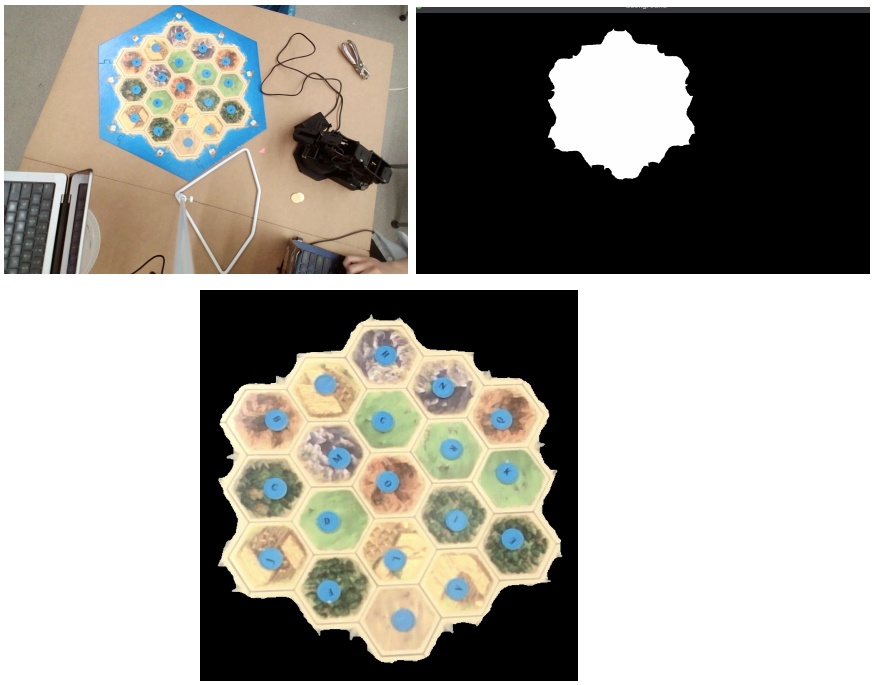


Figure 4. Isolating the rough outline of the hex grid from the background.

2. Canny edge detection

Next, we convert our isolated hex grid image to greyscale and run the Canny edge detection algorithm on it. To reduce noise and ensure the hexagonal grid edges are fully connected, we then run two morphological transformations to dilate and then erode the image

[6] https://github.com/Vieja/Catan-Image-Recognition

shape. The parameters for Canny edge detection and for the two morphological transformations were found by trial and error until the cleanest edge detection was generated.

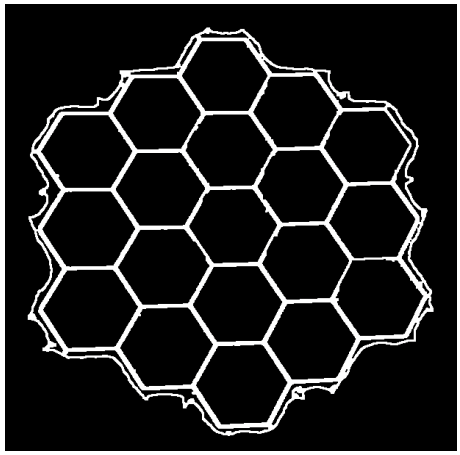


Figure 5. After running Canny edge detection and morphological transformations.

3. Watershed algorithm

After having isolated the edges as best as possible using Canny, we must further clean up the image, particularly the exterior edge artifacts, or else the subsequent step of corner detection will return false positives. We can accomplish this by using the watershed algorithm, which takes in a set of markers and the input image. Watershed treats the input image's pixels as local elevation, or topography, and the markers then "flood" the valleys, moving outwards until valleys of different markers meet one another. This is useful for image segmentation, and in our case, more cleanly isolating the individual hexes within our grid. The input image we use is the output of our canny edge detection.

For the markers, we take the circular number tiles, as they lie in the center of each hex. To make their detection extremely simple, we perform the same isolation of the blue color as in the background removal step but now applied to just the board image. This is especially convenient in a game of Catan as before every game begins, the board is set up by placing the circular number tiles blue side up before they are flipped when the game begins. After running the isolation code on the blue color of the board, we then get a set of markers that we are certain to be inside of their respective hexes. After running watershed, we get a clean image of the hexagons, correctly segmented from one another.

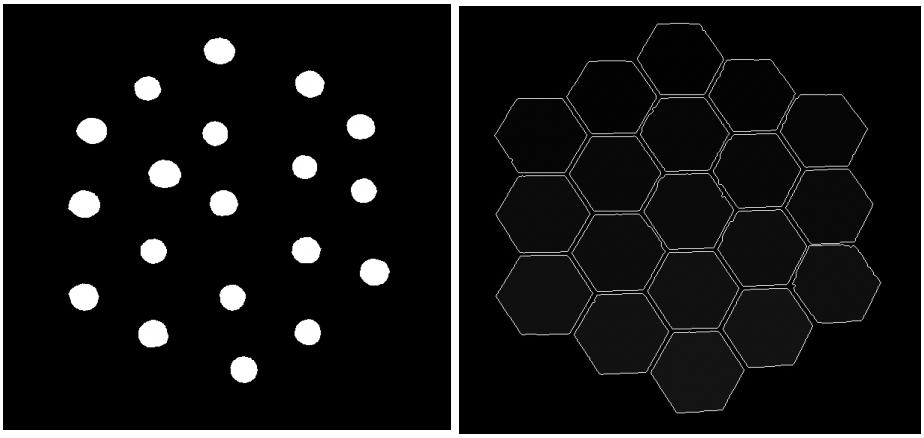


Figure 6. Left: Markers taken from the blue side of Catan's circular number tiles. Right: Output of running the watershed algorithm on the Canny edges and the above markers.

4. Harris corner detection

Next we perform Harris corner detection to find the probabilities that any given pixel is a corner, dilate the results, and mark the pixels that surpass a set threshold as white to show the vertices roughly. Then we can simplify these rough vertex markings. To do so, iterate over each contour and calculate the centroid. Now, we have the locations of all vertices on the game board.

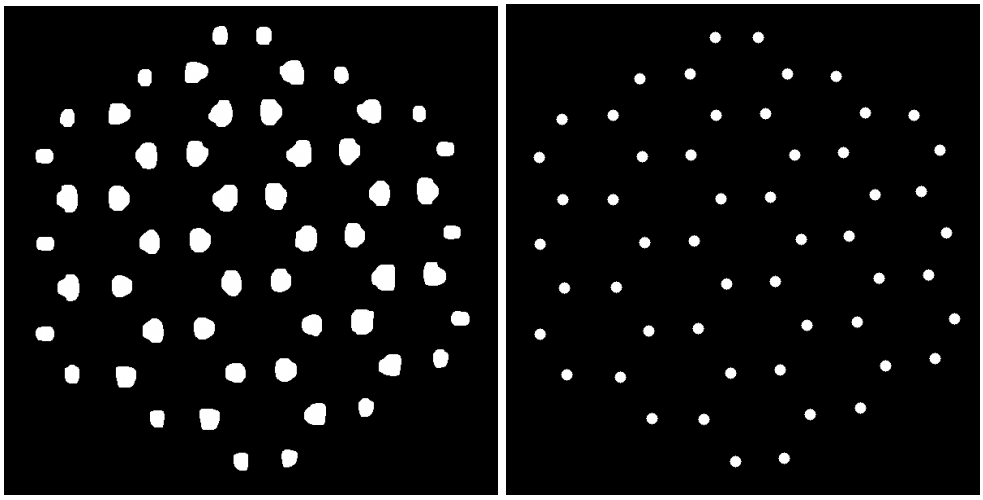


Figure 7. Left: rough vertices found after Harris corner detection. Right: Simplified vertices after finding the centroid of each contour.

Notes on the CV pipeline

It is important to note that this step was particularly challenging due to changes in lighting and angle leading to differences in the output of various computer vision algorithms. We tried other techniques including Hough transform for line detection, preprocessing using thresholding, Gaussian blurring, or bilateral filters.We ultimately found success in working at a

consistent time and place (in the CEID, after sunset to have consistent lighting) allowed us to see progress and ultimately fine tune and achieve a successful computer vision pipeline.

While it may be tempting to assume that we could simplify the algorithm by always detecting two neighboring fixed points on the board (like detecting the pixel coordinate of the lowest vertex of the board) to determine the translation from pixel distance to physical distance to extrapolate all other pixel coordinates of the board, we need to account for the potential camera angle skew that is very likely to exist in the image. Thus, while the hexagons on the board may be regular, the image may not represent the hexagons as regular, so hexagons closer to the camera may have a larger side length than hexagons further away, creating potential error and limiting the ability of the application to generalize the board representation problem.

Camera transitory frame: Board enumeration

At this point, we are given an unordered list of pixels corresponding to all of the various edges and vertices filtered out from the image. With this list, we can spatially understand where each point is located relative to each other, however, we need a little bit of outside information in order to understand how the points overlay onto the board. That is, we need at least one fixed point with known relative positioning of both the board and pixel list. To accomplish this, we orient the board such that the top and bottom edges of each hexagon in the hexagon grid form horizontal parallel lines relative to the image captured by the camera.

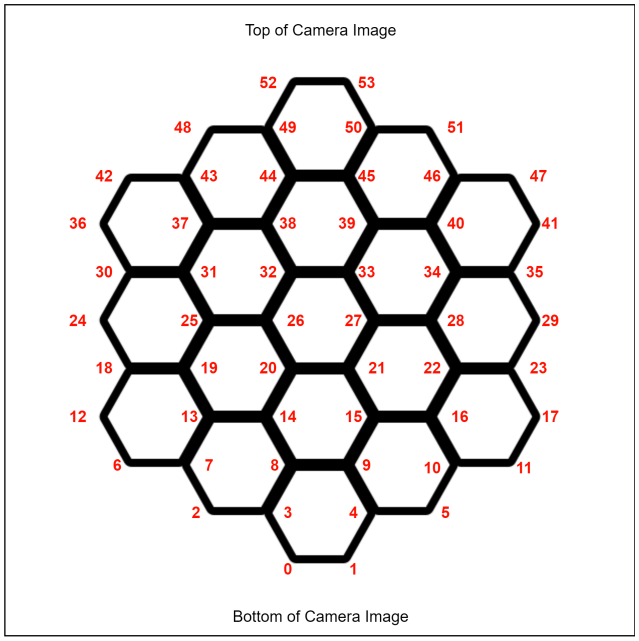


Figure 8. Vertex enumeration

This provides us a simple enumeration of vertices by ordering them bottom to top with left to right handling tie-breakers as shown in figure 4. Now naturally, we cannot always expect that our orientation of the board would always be perfectly parallel to the camera orientation, so it's possible for slight deviations to occur like the 0th vertex being slightly higher than the 1st

vertex. Therefore, we can't simply sort by y coordinate and tie break on x coordinate. Instead, to have a degree of fault tolerance, we devise an algorithm for numbering by enumerating one row at a time. Notice that the first two rows have 2 vertices and 4 vertices respectively, the last two rows have 4 vertices and 2 vertices respectively and all other rows have 6 vertices. We first sort our list of vertices by lowest y coordinate. Then for each row, we take the first n vertices of the list (where n is the number of vertices in the row) and go from lowest to highest x coordinate to give each vertex an enumeration. We then remove all vertices processed from the list. This ultimately deals with any minor tilting the board may have. It does create issues when the list of edges and vertices received from the camera is slightly incorrect (missing vertices/edges or falsely identifying points that aren't really vertex/edge) as we are simply taking the lowest 54 pixels in our list without any error correction. But, such correction code is currently outside the scope of our project and would be future work that could be implemented in a future iteration of the project.

We can employ a similar system for creating an enumeration for the edges. Now, we have two distinct dictionaries, one containing pixel to vertex number translation and another containing pixel to edge number translation. We can effectively combine both dictionaries into one, making sure to add an additional field to each value stating whether the pixel-to-point translation yields a vertex or edge. This completes the preprocessing step of this frame.

Move detection

The move detection code is relatively straightforward. After the CV vertex detection pipeline completes, the move detection module enters a loop that tracks two frames, before and after. The before frame is captured at the start of the players' turn, and and after the player is done placing pieces on their board, they press space and the after frame is captured. At that point, we can find the differences in the image to detect where pieces have been place. We isolate the board image on both before and after, subtract the two images, convert the difference to grayscale, and threshold the result to return reflect just the actual pieces placed. Then, we can iterate through each contour found, knowing that it corresponds to a piece that has been placed, and find the center of its corresponding bounding rectangle.

We now need to find the nearest vertex to the piece pixel. We can simply use the distance function $d = \sqrt{(y_2 - y_1)^2 + (x_2 - x_1)^2}$. We find the distance of the piece pixel to the pixel coordinate of each point and can run a linear time comparison to find the closest point. With the closest point determined, we can send the vertex/edge enumeration through pub/sub to the robot arm to process.

One potential optimization that could be addressed as future work would be to use a k-d tree for the nearest neighbor search. While the space complexity would be O(n log n), we could reduce our search to log(n) (as opposed to O(n)), an improvement in other games should there be a large number of piece locations. That being said, realistically the operation of the application

would most likely be bottlenecked by the physical movement of the robot arm much more than the calculation of the distance of the piece to other points.

Pub/sub

The pub/sub component of the pipeline can be simply imagined as a message delivery system across the internet between a camera handler to the robot arm handler of each other player. As we utilized the google cloud pub/sub API, the code on our end was rather simplistic and just consisted of ensuring that the proper authentication keys and topic names were used as well as any of the other setup steps required. The implementation for the publisher in the camera module was thus just converting the point enumeration to a string and then to a byte to be published to the cloud. For the subscribers in the robot arm module, we utilized a callback function to ack the message and read the message to unpack piece location points. Finally, our message to be sent over follows the format: "<piece_type> <board_enum>, <piece_type2> <board_enum2>" where piece_type is a string representing the type of piece to pick up and board_enum is an integer representing the position to place the piece. Each action is a collection of two values separated by a space, piece_type, and board_enum, and each action is delimited by a ','. Thus, our program supports an indefinite number of actions being sent in one command. Currently, piece_type can be any string, but it's non-functional and we don't currently have the infrastructure set up to differentiate between piece types. This is another feature that would be interesting to implement in the future.

Robot transitory frame / board coordinates

The central idea behind our robot board coordinate system is that it simply takes a single calibrated point with a known offset and known position on the board to determine the required offset of all other points on the board. Unlike the camera frame, there is no notion of perspective skew muddling our sense of distance, rather we can just use the fact that we have a regular hexagon grid to calculate the distance between two arbitrary points on the board. Thus, in designing a robot board coordinate system, we wanted to create an intuitive representation of the board in which we would easily be able to determine the distance between any two points on the board. In creating the coordinate system, we leaned heavily on the fact that we were working with regular hexagons. There was clear uniformity in the vertex and edge distribution of the hexagon grid, along both horizontal and vertical axes. For this reason, along with the fact that the offsets required by the robot arm required (x,y,z) offset values, we decided to pursue a cartesian coordinate system using a square-like grid system.

The first decision we made that helped to constrain our design was to set the orientation of the board relative to the robot arm to be the exact same as the orientation of the board relative to the camera. That is, the same two vertices at the bottom of a camera image are the two closest points to the robot arm. By enforcing this orientation, we not only simplify the conversion process between camera board position and robot board position but also create a uniform

horizontal axis to work with. Now any consecutive two points in the same column (same x coordinate) are a uniform distance apart, let this be a distance x. If we draw horizontal lines for each vertex and edge point in the hexagon grid, we are left with lines that are each x/2 distance apart. This forms the basis of the row numbering system for our square grid overlay. Finally, since we know that there is a horizontal symmetry in the hexagon grid, we set the center horizontal line to be row 0. With this setup we now have two important properties, for any four points a=(x, y), b=(x, y+1), c = (u, v) and d = (u, v+1), dist(a,b) = dist(c,d); also for any two points a= (x, y), b = (x, -y), dist(a, (x, 0)) = dist(b, (x, 0). This effectively means that to get the y component difference between two points, we simply take ((y2 - y1) * hexagon_constant).

This same framework can be applied to the x-axis as well, although the conversion is not as perfectly nice. To begin, we once again take the vertical line crossing the center of the horizontal grid and set it to 0 to enforce the symmetry. If we draw a vertical line for every vertex and edge, we see that every fifth line is uniform with each other, but the four lines in between are not uniform with each other. Thus, to determine the distance between two points, we calculate how many uniform 4 line widths are between the two points and add a custom mathematically calculated offset depending on the remaining non-uniform distance. Figure 5 shows the complete board coordinate mapping in action. This then establishes a cohesive representation of the board that allows us to conveniently calculate the relative distance between two points. Thus, once we have a single calibrated point, we can calculate the specific offsets relative to the calibrated point of the point we want to put a piece at.

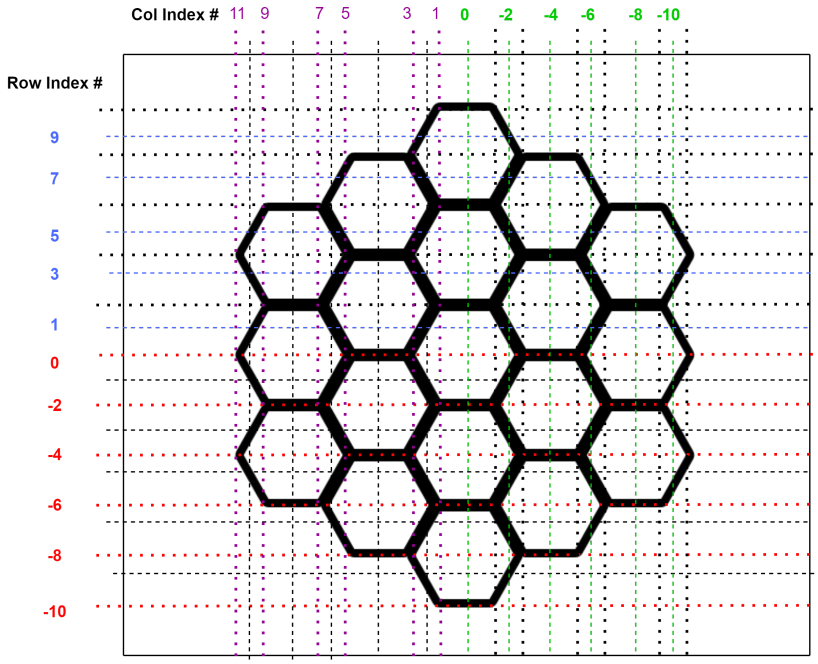


Figure 9. Board Coordinates Mapping

One potential alternative to the grid-like coordinate system we developed for the hexagon grid would be a three coordinate system representing the different vectors a point on the graph

can take to reach another point. Such a system is shown in figure 6. This system provides a more natural and uniform way to calculate distances between points on the hexagon grid by the use of three constant vectors. However, this system is less intuitive for our own spatial understanding of where two points lie relative to each other. For example, point (1,1,1) is 1 vertical unit above point (1, 0, 0). Ultimately we opted to use the more common square grid system as we valued having a more intuitive representation system for users despite the more complex logic needed for distance calculations internally.

The other piece of conversion required by the robot transitory frame is the conversion from camera board coordinate to robot board coordinate. The implementation of this

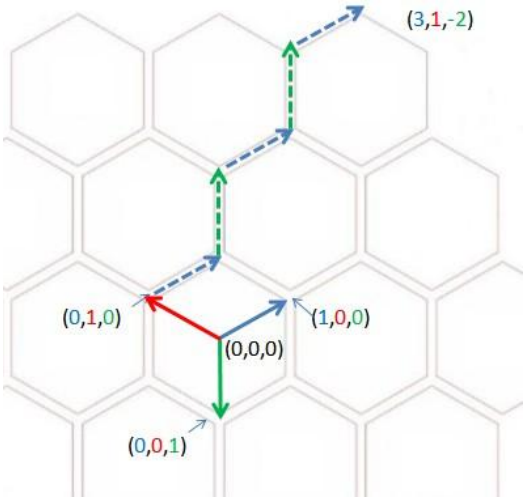


Figure 10. Alternative Board Coordinate Mapping

conversion utilizes the distribution of vertices/edges for each row to determine the specific row that a vertex/edge enumeration yields. We then calculate the vertex/edge offset in the row, if the specified vertex is the first vertex of the row, the offset is 0, while if it's the 3rd vertex of the row, its offset is 2. From there we use the knowledge of col number distributions for each row to determine the specific column number.

With these two conversion processes, we are now able to convert from camera board enumeration to the specified offset of where a piece should be. In our implementation, we are not saving any representation of the board, but rather doing the calculations every time they are needed. Thus, no explicit preprocessing step is required for this frame. When executing the program and trying to find the offset of a specific board enumeration, we simply run the appropriate conversion functions.

Robot arm frame

One step we only briefly touched upon in the previous frame was how we determined the calibrated point connecting robot board coordinates with physical offsets. Since we have no sensor for us to detect this information for us, we decided to allow the user to do the calibration step themselves. At the bootup of the arm and start of our program, we code a specific coordinate

on the board and a best guess offset for that coordinate and ask for user input to make subtle changes to the positioning of the robot arm to reach that coordinate and confirm the positioning once they are satisfied. We then record this new position as the calibrated base point to base our calculations on for the rest of the board. This concludes the preprocessing board representation step of the robot arm frame.

Besides the coding of the calibration, we simply used the available API (which handled the inverse kinematics for us) of the robot arm to handle the grabbing of objects using the physical offset values from the arm. For every offset calculated, we run the procedure of hovering over the object, coming down towards the object and picking it up, hovering up again, hovering over the drop position, coming down towards the drop position and dropping the object, hovering up, then going back to sleep position. After every pickup and drop-off motion, we ensure the arm first rises vertically out of the way of the board to ensure that nothing is knocked over on the board before continuing with the remaining execution.

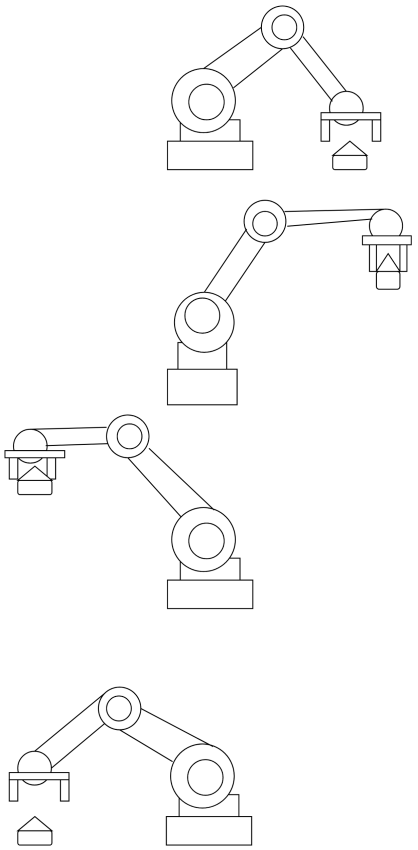


Figure 11. Robot arm pickup/drop procedure

Setup and procedure

Our physical setup was as follows: one Catan board was fitted with an overhead USB webcam (no specific model requirements) connected to a laptop running on MacBook running our the CV code (mode detection module) overlooking the grid of the board and another Catan

board was set up with a Trossen Robotics PincherX-100 robot arm connected to another laptop connected to Ubuntu on the edge of the Catan board. The operating system of the laptop running the CV code should not matter, but the laptop running the robot arm code must be Linux. We ensured that the two boards had the same orientation relative to the camera and robot arm.

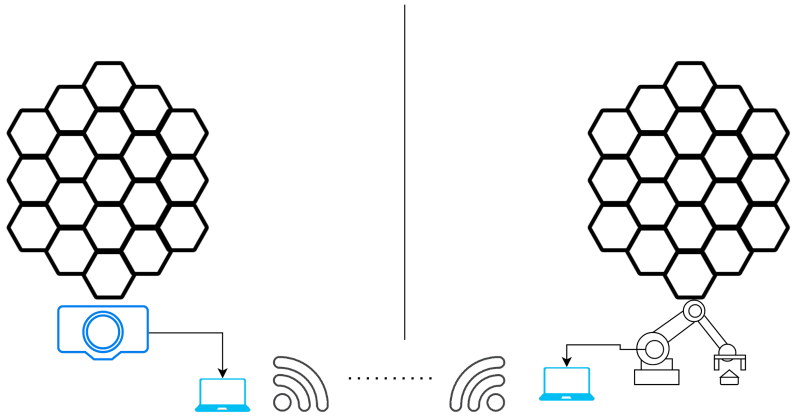


Figure 12. Physical Project Setup

Our software setup involved following the software setup detailed by Trossen Robotics to interface with the arm. Depending on the arm chosen, the arm id bash variable may need to be changed in start_arm.sh. Furthermore, the global variables for the arm type and best guess coordinates for the calibration code would need to be updated in src/actuation.py. Furthermore, a google cloud pub/sub account needed to be set up and both computers needed the proper authentication to access the proper channels. Thus, the global variables for the pub/sub ids need to be adjusted in both src/actuation_module.py and src/detecion_module.py. Now, before running any scripts, make sure the Catan board is set up with a circular number chips blue side up on each hexagon. This is so that the marker detection code can correctly detect its markers before running the watershed algorithm. Otherwise, the hex grid and corner detection will likely not run correctly. After the vertices have been found, however, the circular number chips may be flipped over as in normal gameplay.

To run all of the code after all of the software and hardware have been set up: on the computer connected to the camera, simply run

$ python3 src/detection_module.py

If no video frame seems to pop up, or the video frame is all black, the cv2.VideoCapture() line may need to be updated to point to the correct USB port number connected to the webcam. After running the script, a few images should appear, showing the analysis of the board based on our algorithms. It is highly likely that upon first running the script, the results will be off and not quite the same as the results we got. This is due to the finicky nature of openCV where the light conditions, as well as the camera orientation, play a big role in the ability of the OpenCV script to properly do its job. If the results of the analysis are unsatisfactory, make sure the camera can completely see the board and as directly overhead as possible. Furthermore, the threshold values of the code might need to be adjusted and played around to fine-tune the result. For reference, our setup was run on a workbench in the Yale CEID after the sun had set, thus our values are

tuned towards that environment. If all goes well, the OpenCV script should now be working properly. At this point, the script should prompt the user to press space to take a picture of the board in its initial state (the photo may take a second to take, don't click the spacebar twice until after waiting a few seconds). Now place pieces on the board and click the spacebar. The pieces should be detected by the camera and a message should be sent to the cloud. (Note after taking the first picture of the board, try not to move anything on the board other than adding pieces, otherwise the difference detection algorithm may detect false positives). Now, we have completed the work that needs to be done by the camera.

To run the code for the computer connected to the robot arm, run:

$ ./start_arm.sh src/actuation_module.py

The start arm shell script will handle the initial setup of the arm and start the actuation_module.py script after the arm has been initialized. After running the actuation_module script or if the script is closed out, the arm does not need to be reinitialized with the start_arm.sh script (unless the newly opened console running rviz is closed), rather the script can be run again by simply calling

$ python3 src/acutation_module.py

The first thing the actuation_module.py should prompt the user to do is to calibrate the arm to hover over a specific board coordinate. After calibration is complete, the robot will be ready to receive pub/sub messages from the google cloud. If at any point the arm doesn't complete a motion and outputs an error to the console, it's most likely that the arm was unable to reach the specified coordinate. Either the coordinates need to be readjusted, or the arm is simply physically unable to reach the position and orientation specified (the pitch also affects the ability of the arm to reach a specified position). If all things go well, both the arm and robot should be connected and working now!

Further work

Our project succeeded in establishing a proof of concept of the main architecture required to detect a piece on a board and have a robot effectively take a piece and move the same piece to the exact same position on another board. But, to do so, we had only implemented a bare minimum set of rules and scenarios specific to the Catan game. A number of further work would center around furthering the ability of our architecture to play a real game of Catan. For one, we would want to expand the piece detection algorithm to also consider the color of the object to decide which player's piece to place. Furthermore, we would like to determine an algorithmic way to determine the type of piece placed on the board (whether it is a city, settlement, or road), we imagine this could be handled by either taking the relative sizes of each piece to act as a way to differentiate the pieces or to train a machine-learning algorithm to detect the piece type. The machine learning approach can also be another approach to detecting the pieces on a board in a way that is not sensitive to slight perturbations on the board after taking the before picture for the difference detector.

Another feature that we have drafted up an algorithm for but have not yet implemented would be edge detection and translation. The idea for implementing this feature would be to iterate over the vertices, and to take the midpoint between it and it's neighbors to determine the center of each edge of the Catan board. After detecting all of the edges, we can then create edge enumerations and finish the translation from edge enumeration to board coordinate. Based on our vertex enumeration shown in Figure 8, the corresponding edges would be [(1,2),(3,4),(5,6),(7,8),(9,10),(11,12),(14,15),(16,17),(1,4),(2,5),(3,8),(4,9),(5,10),(6,11),(7,13),(8,14),(9,15),(10,16),(11,17),(12,18)].

Another nice improvement involves setting up the infrastructure for multiple players playing the game. Because of resource constraints, we only had one arm to work with and thus could not create a multiplayer setup. We have our envisioned infrastructure for how we could create a system to support multiplayer as described earlier in the report, but we would like to actually implement the code required to handle turns, cohesion between the various cameras and arms, as well as deal with the potential issues that may arise as a result of the camera needing to also filter out the robotic arm when processing the board. It would also be nice to test transferring the computer vision and robot arm handling code to a Raspberry pi or other smaller processing device so that there we can more fully integrate our system and avoid needing to have a computer for every single device for every single player.

Lastly, we recommend to future students or anyone looking to take up this project to purchase or build a different robotic arm. The range of this arm unfortunately limited our moves to only vertices 0, 1, 2, 3, 4, 5, 8, and 9. A robotic arm that moves overhead should solve this.[7]

[7] https://medium.com/@pacogarcia3/computer-vision-pick-and-place-for-toys-using-raspberry-pi-and-arduino-68a04874c039