A0: AutoPuzzlr

Authors: Andrew Conduff, Connor Maggio, Aneek Mukherjee: Electrical and Computer Engineering, Carnegie Mellon University

***Abstract*—AutoPuzzlr is an unobtrusive automatic puzzle-solving system that guides users as they work through a puzzle to speed up the lengthy process. We are using modern technology to allow our users to reap the mental benefits of solving puzzles while reducing the time committed and easing the difficulty. We utilize CV to make the product easy to use for puzzlers today. This is a niche and relatively untouched space commercially and no competing technology can claim the features that AutoPuzzlr will have, so we are pioneering a much more advanced and capable product for passionate puzzle solvers!**

***Index Terms*—Computer Vision, Feature Matching, Hand-tracking, Point Detection, Puzzle, PyGame, Solving**

I. Introduction

AutoPuzzlr is a project that is designed to help a user complete a puzzle using modern technology and an intuitive user interface for the user to interact with. Solving puzzles offers mental and physical benefits, but sometimes these benefits can be difficult to reap, as puzzles are a significant time commitment and can be very difficult. Given the niche area, there are few other competing technologies, and no other technology claims to be able to handle the same set of features this project is able to boast. This project can claim a more technologically advanced and unobtrusive solution, using CV to take user input and provide a suggested piece location. The goals of this project are to be able to guide a user to build a puzzle through a touch interface on the physical puzzle itself. AutoPuzzlr will detect a user pointing at a piece for 3 seconds within 300ms and display where that piece should go in the larger puzzle within 4 seconds for any piece within the workspace. This will be achieved through the use of just a Logitech C920S webcam and a laptop computer. This project achieves an 82% accuracy for piece placement within 1.5 inches of the final piece placement, and runs for 1.4 seconds on average for an individual piece.

II. Design Requirements

Our original design requirements were revised due to COVID-19 making our project remote, so the revised requirements are detailed in this section. For future reference, we have included the original requirements in italics and mentioned the changes. Our high-level user requirements are as follows:

- End-to-end suggestion latency: 4 seconds to provide a suggestion to the user
- Suggestion Precision: 1 inch between piece's suggested and actual location (*originally 0.5in*)
- Suggestion Accuracy: 80% of the time the piece will satisfy the precision requirement (*originally 90%*)

We understand that we need to be able to account for some errors in the environment itself and that no computer vision code will be perfect so we thought that having a 80% suggestion accuracy would be a high enough placement accuracy score such that the user can rely on it, but if the piece does not fall within that 1 inch radius of where that piece should go, then that would be considered an inaccurate placement of a piece. The requirements were changed to reflect the decrease in accuracy from removing the Leap Motion controller from the hand-tracking system due to COVID-19. Our design is robust for detailed pieces, but given that some puzzles have similar textures across wide swaths of the picture, there will be a circle of confidence of where that piece could be that will grow larger across similar areas. (e.g. in a puzzle with a lot of open sky with a lot of blank sky-blue pieces, then our accuracy will likely be a lot lower than a puzzle piece with a specific detail on it.) We chose a 4 second design time because of the limitations of technology and the algorithms we are using. We decided that 4 seconds was a tight enough constraint such that it would still feel intuitive and useful to a user, but gave us enough time to compute where these pieces should go.

We have outlined that there is a 4 second response time between user input (point) and system displaying its solution to the user. We have further subdivided this into the following list:

- Point Detection: 500 milliseconds from actual point held for 3 seconds to point being recognized by CV system (*originally 50ms*)
- Piece Identification: 300 milliseconds from point being recognized to identified piece (*originally 500ms)*
- Piece Matching: 3 seconds from identified piece to suggested location returned to back-end
- Response Latency: 50 milliseconds from a user tap notification or returned coordinates in the back-end a response graphic will be displayed.

Our timing requirements are derived from estimates of computation power required for the algorithms that we are using. The point detection requirement changed significantly due to the detection being entirely CV-based, instead of using the optimized and production-ready Leap Motion controller. However, we were able to reduce the piece identification time since both pieces were now in the same module and this reduced

the overhead of passing data across a local socket as our original system was designed.

Our hardware performance metrics are as follows:

- Camera Field of View: 100% of our ~24”x24” workspace
- Camera Sensor: 12+ MP resolution and good color identification

These requirements are based off of the size of the puzzle and we have an upper limit of about 20” by 20” for our maximum puzzle size, and 200 pieces for maximum puzzle pieces. We came upon the 12+ MP resolution camera as this should be sufficient given the distance of the camera as well as the average size of puzzle pieces. We have also referred to previous projects using cameras that tested a few options at the scale we’re looking at and found 12+MP to be sufficient.

III. Architecture and/or Principle of Operation

Our system architecture is described visually in Fig. 1 and at the end of this document. Our design has changed significantly since the design report, including the removal of a number of physical components, which is detailed below. These changes are largely due to being unable to work on-campus and in-person due to COVID-19.

Our system was originally composed of a PVC frame containing all of our components and defining the workspace for the user. This frame is no longer part of our project, as our hardware components have been revised to be just the webcam and laptop computer. These components were chosen to satisfy the user interaction requirements we set and we discuss the selection process in the following Section (IV).

The software system is broken down into 3 libraries functioning like microservices - the piece solving system, the finger and point detection, and the user interface. The former two components each represent a core system operation in the user’s interaction cycle with the system. We have removed the animations builder in our original design, as we are utilizing the laptop screen for output rather than a projector.

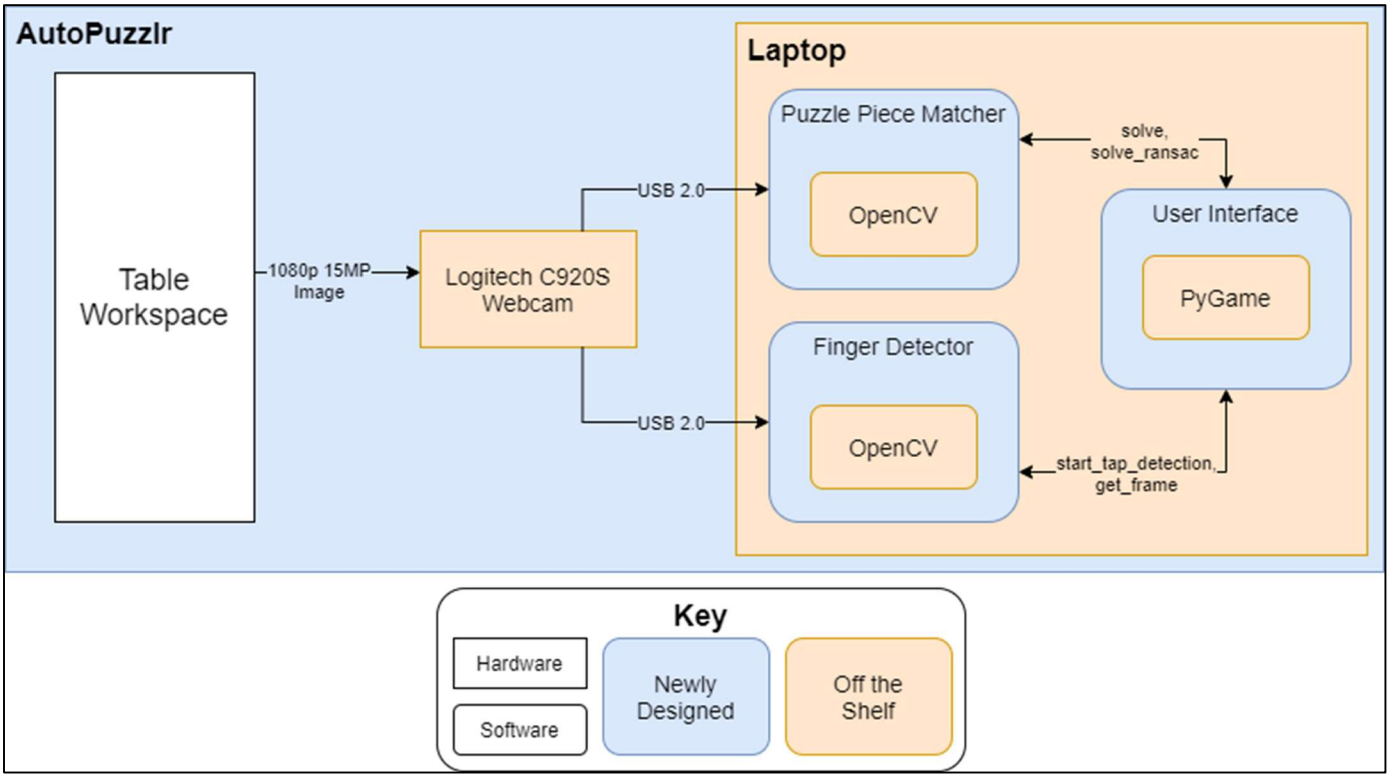
Fig. 1. A complete block diagram of out system architecture

The user’s points are first recognized by the point detection system, which is triggered by the user interface to start looking for points at the appropriate stage. The system reports back the webcam frame for display, and the point coordinates and cropped image of the puzzle piece when a point is detected.

Then, the piece matching system takes the cropped image of the puzzle piece from the user interface and identifies the piece, runs its feature and orientation matching algorithm, and reports the location of the suggested final location, as an image of the puzzle solution and coordinates of the guess, to the user interface to display.

Finally, the user interface orchestrates the above operations according to the interaction cycle of the user and takes in input so that the user can move from piece to piece as they solve the puzzle.

Our system is architected in this way so that there is clear separation of responsibilities between components and a high level of possible parallelization in the development process, since there is a clear API between otherwise independent components.



IV. Design Trade Studies

Our project has undergone significant changes in physical components due to COVID-19. Some of the research discussed in this section are for components that are no longer part of our project, but for future reference we have chosen to include them. Removed components will be listed as such.

A. Physical Components

Our final project requires just a camera and computer, and the projector, frame, and Leap Motion controller have been removed. We have specific functions for each of these components and carefully considered our options.

1) Frame (Removed)

Our frame was designed keeping in mind that it needs to be portable, adjustable, and relatively cheap. We decided to use 1.5" diameter Schedule 40 PVC due to its sturdiness, light weight, and price compared to wood. We decided on a 1.5" diameter because of its rigidity and reasonable weight. We will need to drill screws into some of the PVC connections in order to mount the projector. This diameter leaves room for this.

In addition, PVC piping has reliable connectors that include sliding components. We may need to adjust the heights of our projector or cameras in our prototyping efforts. Using PVC makes this easily possible.

Our design is susceptible to toppling over from the weight of the projector. We considered preventing this by adding a counterweight to the base of the frame or simply adding an extending pipe to the base. Adding an extension would have an effect on the user's workspace. Thus, a counterweight was a better option.

2) Surface

Originally, we designed the system to cover the work surface in Duvetyne fabric to improve the Leap Motion controller's performance. As this component has now been removed, we no longer require that particular fabric, but we do require a consistently colored background where the user's hand and puzzle pieces will stand out to improve the performance of the finger detection system.

3) Camera

We considered a few options for our camera. Initially, we were debating whether to use a DSLR, a smart phone camera, or a personal web camera. Personal web cameras are the best choice for our project's scope. That is to say, a DSLR camera or a smartphone have far too many functions that would go unused for us.

The web camera we decided to use is the Logitech C920S. In choosing this, we mainly considered camera sensor size, megapixel count, and price. Thanks to CMU's IDeATe and ECE departments we had lending access to a couple models including Logitech's C615 and C920 webcams, and Quickcam's Pro 9000 model. While each camera had a similar sensor size, the C920 has the largest megapixel count at 15 MP. Upon further testing, we decided that we would require this resolution to ensure our image recognition requirements. Past projects that have used OpenCV have succeeded using this camera. The C920S we purchased is an updated model of the C920 we tested and has the same specifications.

4) Projector (Removed)

Our projector was chosen based on size, throw ratio, lumens, and price. We would have liked to use a mini projector in order to cut down on weight, however they tended to have a low number of lumens. Since our product will be used in the light, we required at least 2500 lumens. In addition, we were looking for a high throw ratio in order to meet our requirements of projecting onto a 20" by 20" puzzle within a distance of 4 ft. The Apeman M7 mini and the Epson Powerlite 1776W were the only models that met the lumens and price specifications, as the Apeman was quite affordable and the Epson Powerlite was already owned by the ECE department. These and other projectors considered are shown in Fig. 2. Upon testing, we found that the Apeman projector projected a very wide screen that would limit us down to around 16" puzzle heights. The Epson Powerlite 1776W was the clear choice moving forward.

5) Computer

Our computation currently is being developed on a quad core i7 with a 2.6 GHz processor. Our system will run within the requirements on the average laptop computer, running an i5 with a 1.7 GHz processor. We originally planned to use a Raspberry Pi 4 Model B computer, but since the frame and other hardware components were removed, there was little benefit to moving computation to the Pi. This also allowed us to continue to use the laptop screen as an output rather than adding a screen to the Pi.

B. Piece Matching

Our piece matching relies heavily on a pipeline of image processing techniques, however we need to balance both the speed of these applications as well as how robust they are, and there is still a lot of tweaking to be done as we get the parts we need and see how everything translates computationally to the Pi.

1) Technologies Used

We have decided to use the Python version of OpenCV version 4.2.0, as it was the most up to date version at the start of our project and the OpenCV community is vast and helpful in case we ran into any problems.

2) Thresholding

We have tested many methods, but for now we are planning on using OpenCV's built in THRESH_BINARY_INV method for thresholding, in combination with THRESH_OTSU, in order to increase the confidence of our thresholding. (See labeled picture #2 in Fig. 3). We felt like this gave us the best combination of background filtering as well as the ability to

Projector	Size (depth, width, height)	Throw Ratio	Resolution	Lumens	Price
Epson Powerlite 1776W	210 x 292 x 44 mm	1.25	1280 x 800	3000	$0 (from dept.)
Insignia Slim-line Pico	145 x 79 x 23 mm	X	854 x 480	50	$160 ($70 used)
Viewsonic M1 mini	104 x 110 x 27 mm	0.83	854 x 480	50	$70
PIQO mini projector	57 x 57 x 57 mm	X	1920 x 1080	200	$27
Apeman M7 Mini	102 x 102 x 102 mm	1.2	854 x 480	4000	$70
AAXA P4X Pico	141 x 71 x 31 mm	2.18	854 x 480	175	$200 ($70 used)
BenQ MX631ST	287 x 232 x 114 mm	1.08	1024 x 768	3200	($445 used)
NEC NP-M353WS	368 x 292 x 135 mm	0.45	1280 x 800	3500	$936

Fig. 2. A table of various projectors we considered

clearly choose what part of the image is the puzzle piece. Fig. 3 picture 4 preserves the picture, but that is not important for this step of our algorithm. Once we have this piece separated, we are able to extract only those pixel values and consequently detect features across them.

3) Feature Detection

While we could go with methods like SIFT and SURF we didn't feel like it was necessary as they are not open source in all cases. While we had originally intended to use the ORB feature detector, we decided to pivot to the BRISK feature detection method as it ended up offering us better performance for detecting features, as it found key points more easily and reliably, as well as providing more reliable matches when using RANSAC. RANSAC was the method we chose to match the key points between images. RANSAC stands for Random Sample Consensus, which chooses a random amount of these key points and matches them between images (the piece image and the overall puzzle image) and attempts to find the strongest match between the two.

4) Background Cloth

We had decided to use Duvetyne fabric as the base of our operations as compared to table surfaces, but this fabric did not come in time, as COVID forced us to change how we handled the background for the project, so we had to forego this portion of the project.

C. Finger Detection

We wanted user interaction with our service to be as close to the reality of solving a puzzle, so we are utilizing the webcam to track the user's hand and fingertip and detect a point in the workspace. This removes a hardware layer between users and their puzzles and allows us to get user input in an unobtrusive way. Some key design decisions for this subsystem included hand and fingertip identification methods, API design, and the surface of the workspace. These are covered in Sections IV.C.1-4. Originally, we planned to use the Leap Motion controller to identify users' hands and taps in the workspace, but this option was removed due to COVID-19. However, some of our research into the usage of the controller is included here for future reference, in Sections IV.C.5-9.

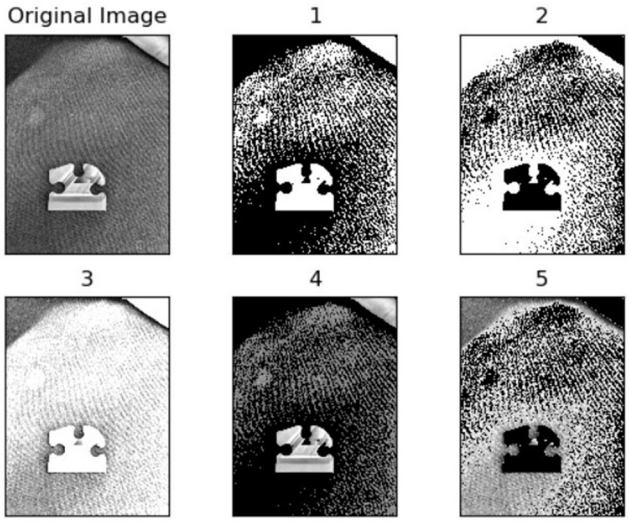


Fig. 3. An image of a puzzle piece after various thresholding operations

1) Technologies Used

As in the previous section, we have decided to use the Python version of OpenCV version 4.2.0. Refer to Section IV.B.1 for further details.

2) Hand & Finger Identification

The first step in point detection is identifying the user's hand in the camera image. We considered a few options for doing this, chief among them using CV techniques like finding contours and image masking via a calibrated color histogram, and using an ML model for identifying hands. First, we considered the ML option. However, some research into current performance of hand detection in video streams ran at around 21 fps on an average laptop, and couldn't identify fingertips, just hands in general. We found one research paper on fingertip detection in particular, but as we were making this decision well into our project due to COVID-19, we decided it would take too much of our remaining development time to try to implement. Our other option was using a purely CV-based approach. This involved calibrating a color histogram, masking the image using the histogram to pull out the hand, and then finding the largest contour and the farthest point from its center to locate the hand and fingertip respectively. We decided this approach was better for our circumstances, as it used common OpenCV operations and we were able to use online resources and articles to learn more about these tasks.

3) API Design

Since this finger detection runs in Python 3, it can be integrated with the other modules as an object. Therefore, its API is defined such that it can be instantiated and interacted with via calling functions rather than waiting for event-based messages, as in the previous implementation (see Section IV.C.7). We decided to have the Finger Detection module provide the entirety of the webcam GUI in the user interface via a function, so that the user interface could simply draw UI elements around the webcam stream and not have to interact with that image itself.

4) Work Surface

While we no longer needed to specifically use Duvetyne fabric (see Section IV.C.9), we decided to require a consistently colored background different from your skin color from the user, so that the image masking would properly detect the hand and allow for the contours and fingertip to be properly detected.

5) SDK (Leap Motion)

The Leap Motion controller was originally designed for using as a touchscreen/keyboard replacement or creating a virtual reality control surface. However, the company has since pivoted toward Virtual Reality applications and their recent libraries are exclusively for Unity/Unreal Engine. To use the latest SDK with a Python program, we would have to spin up a VR project on the Raspberry Pi. Therefore, we are utilizing an older Python SDK rather than the latest version so that we can avoid the unnecessary and significant computation overhead of running a VR project.

6) Technologies Used (Leap Motion)

Our decision to use the older SDK constrained us to

Python 2.7 for interacting with the SDK and controller. However, we wanted to take advantage of the modern features and performant libraries available to us in Python 3.8.1, so we decided to separate the entire Tap Detection & Localization component into a separate process that communicates with the back-end via local socket. This decision allows us to separate the Python 2.7 code while still maintaining minimal latency by using local sockets. We decided that since these modules were part of the same program and would be running concurrently, we will be setting up the socket on initialization and closing it on teardown, which will greatly simplify development.

7) API Design (Leap Motion)

Since we are communicating with the back-end via local socket, we needed to define an API so that we can minimize data transferred and have a clear understanding of how both sides will communicate to support parallel development. We decided to have a call-and-response style of system, where the backend sends a trigger, and the service responds with the data on the following detected tap.

We considered alternatives, such as constantly running detection and reporting every found tap, but this approach required the back-end to continuously monitor the received messages to make sure that it responded to the tap that the user intended to make when they are prompted by the system to tap a piece, rather than any randomly recognized taps made while computation on a previous piece was occurring. It made the most sense to only report a detected tap when the system was expecting one.

8) Orientation (Leap Motion)

Due to the original design goals of the Leap Motion controller, the software was optimized for using the controller face-up on a surface, tracking hands *above* it. Further research also showed that this older SDK included optimizations for tracking *palms*, since it assumed the upward-facing orientation. This orientation is not possible for our purposes, since the user will be tapping puzzle pieces directly on the work surface rather than tapping the air above them, so we experimented with the hand tracking under viable orientations - mainly, mounted overhead facing down, and mounted on a vertical column and facing sideways.

We utilized software from Leap Motion to view the actual IR camera inputs to make these observations. We realized immediately that the tracking was extremely poor when oriented sideways, as most of the hand is blocked and only the side of a palm is visible to the camera. This proved to be nearly impossible for the tracking software to recognize, with the palm-tracking optimizations enabled or disabled. Overhead tracking was better, but the effective range was far lower than claimed in the data sheet. We were able to track hands up to approximately 10”, while the datasheet claimed two feet. One issue was certainly that our use case necessitated that the controller track the backs of users’ hands rather than the palms, but another issue was the work surface itself, which we discuss in the following section. Regardless, the downward-facing orientation is clearly the better option, and that is the orientation we decided to move forward with.

9) Work Surface (Leap Motion)

We had determined that the downward-facing orientation was our best option, but we hadn’t yet been able to reach the performance levels we were looking for and were promised in the device specifications. We continued to use the visualization tool and compared performance between the down-facing and standard, upward-facing orientations. We observed that the primary difference in the images between the orientations was the contrast of the hands against the background in the camera input, as shown in Fig. 4.A and Fig. 4.C. The surface below the hands was reflecting IR light back into the camera and washing out the image when the controller was down-facing, but there was no such reflection in the up-facing orientation and any hands in-frame were clearly contrasted against the background.

We tested this hypothesis by holding the controller 6’ above the ground and tracking hands 4’-5’ above the ground. The distance to the ground would ensure that little to no IR light was reflected back into the camera and would accurately simulate the upward-facing orientation. With that setup, we were able to achieve the 2’ tracking distance that we were looking for, and looking at the image in the visualization tool, shown in Fig. 4.B, confirmed our hypothesis that the reflected IR light from a nearby surface was causing the tracking issues.

We researched mitigations and discovered that an IR absorbent material covering the workspace would increase the contrast of the users’ hands and still allow them to work on a surface rather than 4’ in air. We found a few options, including Aktar foil and Duvetyne fabric. Aktar was an order of magnitude more expensive than Duvetyne ($199 and $17 for comparable amounts, respectively), so we decided to use a Duvetyne sheet to cover the work surface under the frame and to provide the necessary contrast for the Leap Motion controller.

D. User Interface

The user interface both displayed instructions and information to the user, and orchestrated the two other software components. The key design decisions we made to fit our requirements and the requirements of the other services was the choice of language, and choice of framework used.

1) Language Used

Taking into account the skills and experience of the team, our primary language options are Python and C/C++. There are definite performance benefits to using C/C++, since it is a

A B C

Fig. 4. Images of the camera input from the Leap Motion controller

compiled language. However, we decided to use Python 3.8.1 (the latest version) for the bulk of this project because it is much more familiar to us and thus allows for a higher speed of development. Additionally, the OpenCV implementation in Python uses compiled C++ under-the-hood, so we are able to utilize the benefits of compiled OpenCV and use Python.

2) Framework Used

We considered a few Python UI frameworks, including PyGame, Kivy, and Tkinter. However, we eliminated Kivy since our team had experience with PyGame and Tkinter previously, but not Kivy. We then researched integrating OpenCV images and video streams with Tkinter and PyGame, and decided that the high-level constructs in PyGame, like surfaces and buttons, were more advantageous than Tkinter. We decided to move forward using PyGame 1.9.6.

V. System Description

Unlike previous sections, where we have chosen to include research and discussion from our design report even in cases where it is no longer utilized in our project due to COVID-19, Section V details our project as it is in its final form. All components and discussion in this section describe the revised project.

A. Physical Components

1) Work Surface

The work surface must be a consistent color and texture, different from the user's hand. This is primarily to improve the accuracy of the finger detection system, since it relies on a color histogram and image masking to identify the region of interest of the hand.

2) Computer

Our software will run on an average laptop computer, running at minimum an i5 processor with a 1.7 GHz clock speed, and we have also tested with computers running i7 processors with clock speeds from 2 – 2.6 GHz.

3) Camera

The Logitech C920S webcam provides the video stream of the workspace to the user. It must be mounted above the workspace, facing down.

B. Piece Matching

Our computer vision relies heavily on a pipeline of image processing techniques.

1) The Tap

As previously mentioned, we had planned to use a Leap Motion controller to detect hand gestures, however due to COVID, we transitioned to using the Logitech C920S webcam and more heavily leverage the computer vision aspects of our project. Our project now leverages vision-based gesture detection using OpenCV further discussed in V.C. This tap is now being leveraged in OpenCV and it discussed in the next section. However, we can use the point of the finger that is identified in our gesture recognition pipeline as the point that we then can capture the image from. Having this point is not only helpful for the piece identification portion of our project, but it also gives us the ability to provide a seamless user interface and experience.

2) Segmentation

Once we have resolved the user's tap location, through our software back-end's coordinate re-mapping, we are able to determine the exact piece that is tapped by the user, and segment that piece out away from the background. We are then able to send this piece extraction image to the feature extraction portion of the pipeline.

3) Feature Extraction

Once we have the piece, we are looking to match isolated, we can run our feature extraction method to try to find parts of the puzzle piece with notable features like corners and edges, and map them to that piece. Features are extracted using the BRISK feature detector.

4) Feature Matching

Once we have these features, we are able to match them with the features extracted from the puzzle's final image. From there we will perform different confidence checks to ensure that the algorithm is confident that it has found the right piece. These features are also fairly rotationally robust, meaning when the user taps a piece that piece will not have to be in the correct orientation for it to be recognized as the correct piece by our algorithm.

These features are matched using RANSAC, which has been discussed earlier. Once a match has been found, the system will display a rectangle around the part of the final puzzle that should contain the piece. These graphics are discussed further in Section V.D. A problem to note, is that since these pieces rely on key points to match the pieces in the puzzle, if there is a piece that is particularly "bland" or is lacking much texture, key points are not easily extracted. Figs. 5 and 6 show RANSAC and the final piece solution image.

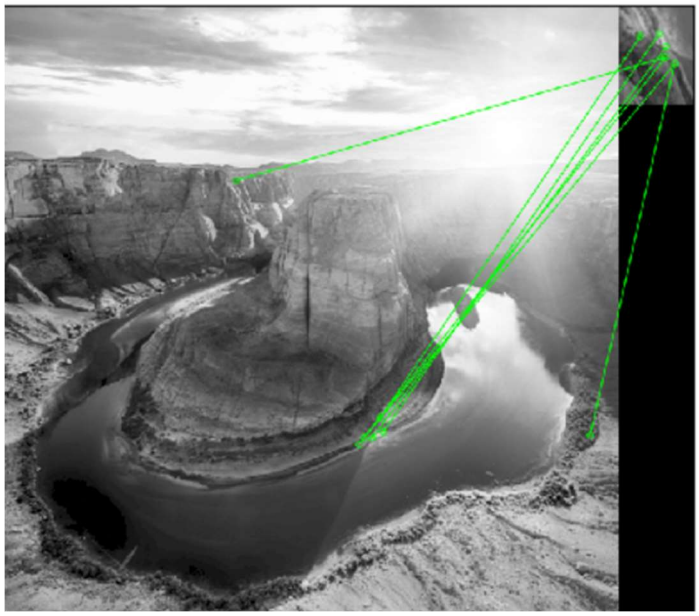


Fig. 5. A display of how RANSAC is matching points, and that it is able to place a rotated version of the piece. Due to RANSAC's consensus information, it is able to show that the piece should be placed correctly, despite other key points being incorrectly matched.

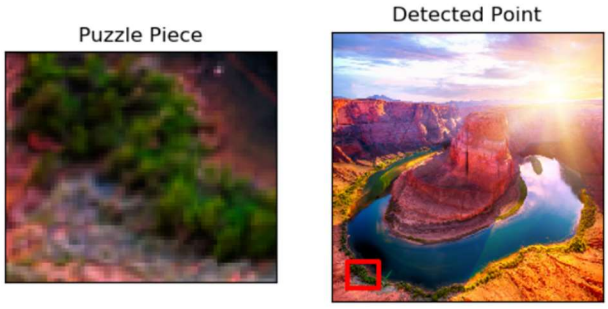


Fig. 6. Below: A demonstration of how the piece matching visualization is shown.

6) *No-Key-Point Matching*

In the case that there are not enough key points to match the piece, there is a backup method that is on average much slower. This method involves template matching the piece in different rotations until it can approximate the best fit. This is used as a last case scenario, as most pieces of will have some features that can be extracted, however, in some puzzles with solid color pieces (such as those found in sky pieces) this backup step is vital. This is shown in Fig. 7.

C. Finger Detection

The finger detection component is an object that the user interface instantiates and interacts with. It is instantiated when the user interface is started. It opens the webcam and offers 2 functions for the user interface to call. The function *get_frame* requires a Boolean value (if the calibrate key input is pressed), and returns a tuple of point coordinates, cropped piece image, and webcam frame image. The function returns the webcam frame image with the drawings on it, and if point detection is running and a point is detected, it returns the coordinates and the cropped puzzle piece at the location of the point. The second function *start_tap_detection* simply sets the flag for point detection to start, and must be called for every successive point. The name is a holdover from the initial development of this component, which detects taps using the Leap Motion controller. The following sections describes the important operations in the image processing pipeline used in this component.

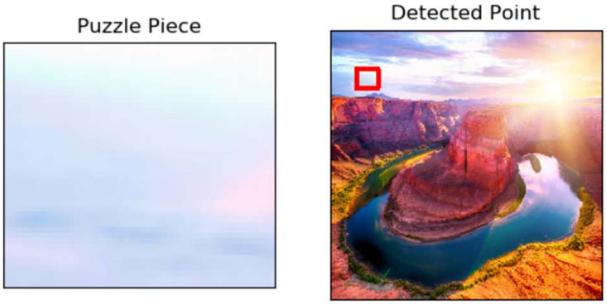


Fig. 7. A demonstration of the backup pipeline working as it is able to place the piece without any detected key points.

1) Calibration

The first step is calibrating the system to recognize the user's hand by creating a color histogram. The frame displays 9 rectangles at the center, which outline the 9 regions from which color is sampled to create the histogram. When the user presses the key to calibrate, the system samples the pixels from each of these regions. Then, using OpenCV functions *cv2.calcHist* and *cv2.normalize* (using the type *cv2.NORM_MINMAX*), the histogram is created. Now, every successive frame can be sampled using the color histogram to identify regions of skin where the hand is located.

2) Locating the Hand

On frames after calibration, the system is tracking the hand and fingertip. This is done through a series of operations on the original webcam image. Using the OpenCV function *cv2.calcBackProject*, the system segments out the image using the color histogram to only the regions with the right colors. Then, after some filtering and thresholding to smooth the image (using *cv2.THRESH_BINARY*), the system uses *cv2.bitwiseAnd* to mask the original image using the calculated threshold, leaving behind just the areas of the image with the user's hand.

Some accuracy issues in the system can manifest in this step when the background of the image includes many regions with similar colors to the user's hand. This can create a contour with a much larger area than the hand. This can be mitigated with a consistently colored background different than the user's hand, which is one of AutoPuzzlr's requirements for use.

3) Identifying the Fingertip

Once the hand is identified, the system identifies all the contours in the image. Ideally, there should only be one, as everything but the regions of the hand should have been masked out. However, since it is possible that lighting conditions may make it hard to distinguish some other regions of the image, the system then looks for the contour with the largest area, which should, in most cases, identify the single contour making up the hand in the image. This is done using OpenCV functions *cv2.findContours* and *cv2.contourArea*. Finally, the system identifies the centroid, hull, and convexity defects of the identified contour, using the OpenCV functions *cv2.moments*, *cv2.convexHull*, and *cv2.convexityDefects*. Then, it identifies the convexity defect farthest from the center, which is likely to be the farthest extended fingertip of the hand.

Some accuracy issues in the system can manifest in this step, in particular when significant portions of the arm are in frame. The arm is likely to be included in the masked image, so it is possible that the bottom of the frame where the arm ends is farther from the centroid then the extended fingertip. This can be mitigated in scenarios where it occurs by recommending that the user where long sleeves, but at the scale and workspace dimensions AutoPuzzlr is generally used in, it is unlikely to occur.

4) Identifying a Held Point

Once the fingertip's location is determined, AutoPuzzlr adds it to a list of stored locations. As the frame rate of the

application (and therefore the rate of calls to *get_frame*) is controlled by PyGame, the system caps the length of this list to correspond to the number of frames in 3 seconds. Then, for each frame where it is looking for a point, it checks if 90% of the stored fingertip locations are within a certain radius of the current fingertip location. We use this percentage and radius to control for small movement and any inaccuracies in the pipeline which would otherwise void a valid point. If the stored list satisfies the percentage and tap radius, then we recognize a point. Finally, we crop the image of the puzzle piece from right beyond the tip of the fingertip by calculating the vector between the hand's center and the fingertip to get the direction of the point, and extend it to locate the area being pointed to. Fig. 8 shows the full webcam frame, including the center of the hand, the identified fingertip and stored locations, and the rectangle that is cropped when a point is detected. Finally, the component returns the coordinates, the cropped image from the tip of the fingertip, and the full webcam frame to be displayed to the user in the UI. If no point is found, it simply returns the frame.

5) Conclusions

Overall, despite the changes to the user input system due to COVID-19, we were able to implement a CV-based system that hit our revised specific requirements and still matched the original end-to-end requirement. The point identification takes between 250-300ms with an average of ~282ms, which is well within our revised requirement of 500ms. The piece identification is far faster, at an average ~10ms rather than our revised estimate of 300ms, because everything ran in the same system and we took a simpler approach to piece identification than we originally expected. Thus, our overall average time of identifying a point and puzzle piece, 292ms, is well within our original end-to-end requirement of 550ms.

D. User Interface

AutoPuzzlr's user interface is created in PyGame, and consists of 3 primary screens displayed to the user. The user interface also orchestrates the 2 other components to get information and trigger operations at the appropriate times.

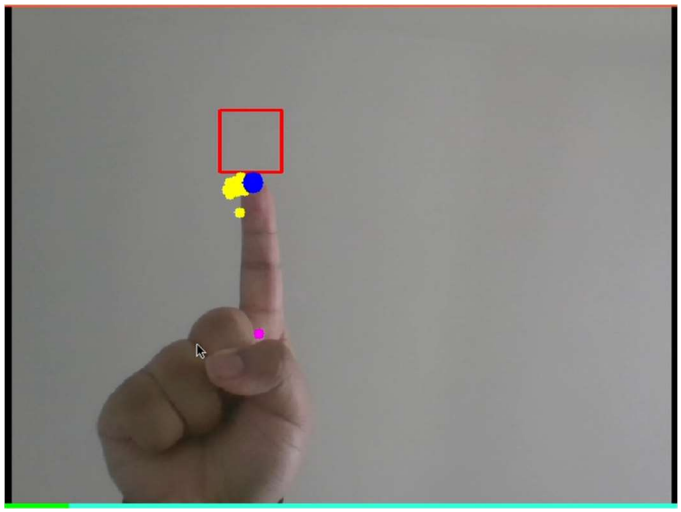


Fig. 8. The drawn-on webcam display showing the identified center, fingertip, stored locations, and rectangle of the cropped image

1) Title Screen

The title screen simply presents AutoPuzzlr's name and provides a jumping off point to the user experience.

2) Setup Screen

The first step in using AutoPuzzlr is providing the completed puzzle as a solution image, so the setup screen provides the options for doing that to the user. They can take a picture with the webcam or they can upload an image from their computer. We recommend they upload an image from their computer, as this allows them to crop the image and make sure the lighting is even.

3) Solving Screen

The main screen that users will spend the most time on is the solving screen, shown in Fig. 9. This shows the webcam's video stream with the fingertip, hand, and tracked locations displayed on it in the same way as Fig. 8 does. It displays instructions for the user in the top bar, and shows all the pieces they have solved in the bottom bar. Once a tap is recognized, an overlay appears showing the piece's location within the larger puzzle solution provided in the previous screen. This is shown in Fig. 10.

The display loop for this screen also orchestrates the other two components. It calls *get_frame* and *start_tap_detection* on the finger detection component, and *solve* to kick off the piece matching pipeline as points are recognized and images are solved.

VI. Project Management

A. Schedule

Our complete schedule is expanded at the end of the document. In general, each team member has a task to complete every week. We were following a cycle of research, then into implementation-refinement cycle, and then into integration. After spring break, the COVID-19 lockdown caused a number of tasks to shift and you can see the changes reflected. This pushed integration farther back than we originally planned, as the entire UI had to be rewritten, and the Tap Detection component scrapped and Finger Detection component written.

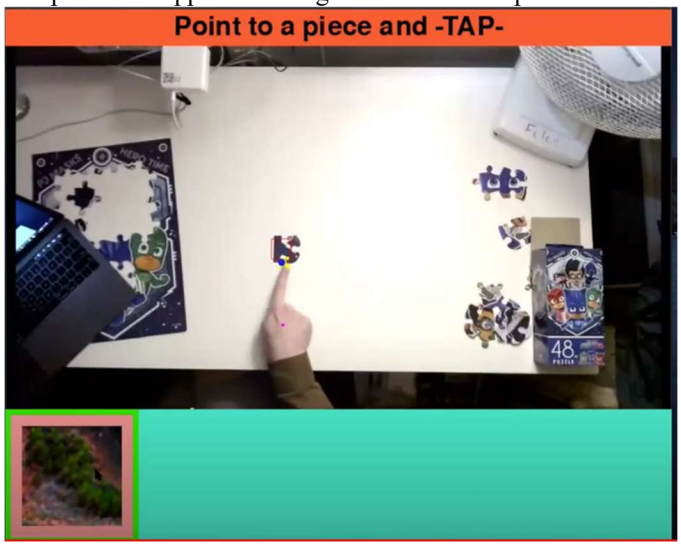


Fig. 9. AutoPuzzlr's primary solving screen

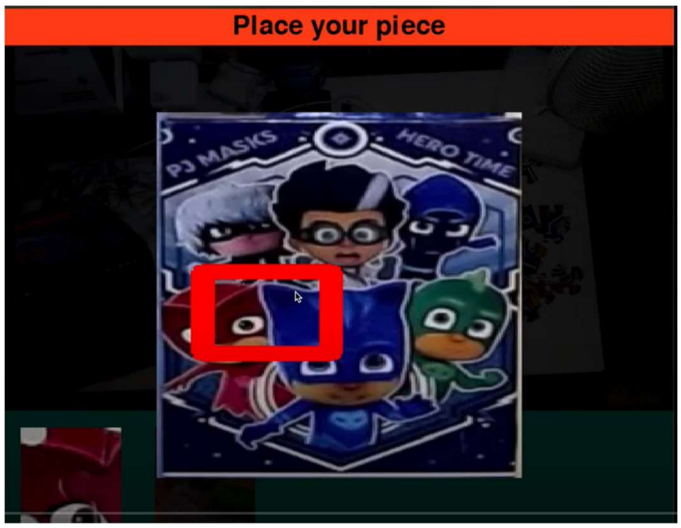


Fig. 10. AutoPuzzlr's piece solution screen

B. Team Member Responsibilities

Andrew took the lead on the Piece Matching aspects with a secondary responsibility in the integration of the software insofar as it is helpful for the piece matching code. Aneek took the lead on finger detection and shares a bulk of the integration of the project with Connor. Connor is heading the user interface, as well as helping Aneek with the integration of the user interface with the other software components.

C. Budget

We completed our project using less than half our budget. The complete Bill of Materials and their associated costs is included at the end of the report.

D. Risk Management

Our risk management has revolved around alternative computational methods, such as exchanging the Raspberry Pi for a laptop or AWS, and giving ourselves some slack. We gave ourselves plenty of budgetary room in case anything was to go wrong or we found out we would need more expensive parts or unexpected parts for any reason. We also gave ourselves the ability to forego the Leap Motion and go back with our fall back design which would involve more computer vision, where we designate a spot on the workspace for the user to place a piece for which then the computer vision pipeline is run. This turned out to be a great idea given that our plans were upended anyway, thus our conversion to a new project was fairly seamless given the COVID-19 situation. We took advantage of our plan to shift from the Raspberry Pi to a computer, and from the Leap Motion to a CV-based approach.

VII. Related Work

There is very little related work available commercially. We have seen some other attempts on physical puzzles, but they were only preliminary results and didn't end up working. We also found a simulation that was able to match pieces to a puzzle, but they had the pieces in the right orientation originally. Furthermore, the simulation was only working with virtual pieces which used perfect pieces, instead of the imperfections we are dealing with by using real world pieces.

VIII. Summary

1) Fulfillment of Design Requirements

AutoPuzzlr does meet the revised requirements we set out after the COVID-19 crisis changed our project, and most of our original design requirements as well. Overall, we were satisfied with our results, as we achieved most of our requirements. We were particularly pleased that removing the Leap Motion controller and revising our user input-related requirements did not lead to revising our high-level requirements, as we were able to satisfy our original requirement using the purely CV-based system.

For our high-level user requirements, we achieved:

Requirements	Result
End-to-End suggestion latency of 4 seconds	Avg. 1.4 sec, 1.2 – 4.8s
Suggestion precision of 1 inch	0-1.5in
Suggestion accuracy of 80%	82%

In terms of our CV-based component requirements, we achieved:

Requirements	Result
Tap Detection within 500ms	Avg, 282ms, 247 – 303ms
Piece Identification within 300ms	Avg. 10ms, 9 – 12ms
Piece Matching within 3 seconds	Avg. .86s, .3 – 4.5s
Response Latency within 50ms	Avg. 10ms

B. Future work

Some key improvements we would like to make include integrating some of the components that were removed due to COVID-19, including the physical frame, projector, and Leap Motion controller. In addition to these features, which were originally part of the design, we would like to expand this algorithm to not require a visual of the box front. This felt like it would be out of the scope of the time and budget constraints. Finally, we would like to explore more advanced CV techniques to scale this product up to puzzles larger than 20"x20" and 200 pieces.

C. Lessons Learned

We think that there is a lot that can be learned from accurately and realistically giving yourself a good schedule. We think that a lot can be learned by accurately tuning your values to grab the relevant images as well as making sure that you take into account your development platform early on into development. We learned a lot about how to be flexible due to uncertain circumstances, as well as realizing that most projects will rarely

go to plan and it is important to keep an open mind and always be able to make a plan B. We also learned that sometimes errors stem from not reading documentation thoroughly. (Namely, knowing that OpenCV does not adhere to the RGB standard)

REFERENCES

[1] Leap Motion Datasheet, https://www.ultraleap.com/datasheets/Leap_Motion_Controller_Datasheet.pdf

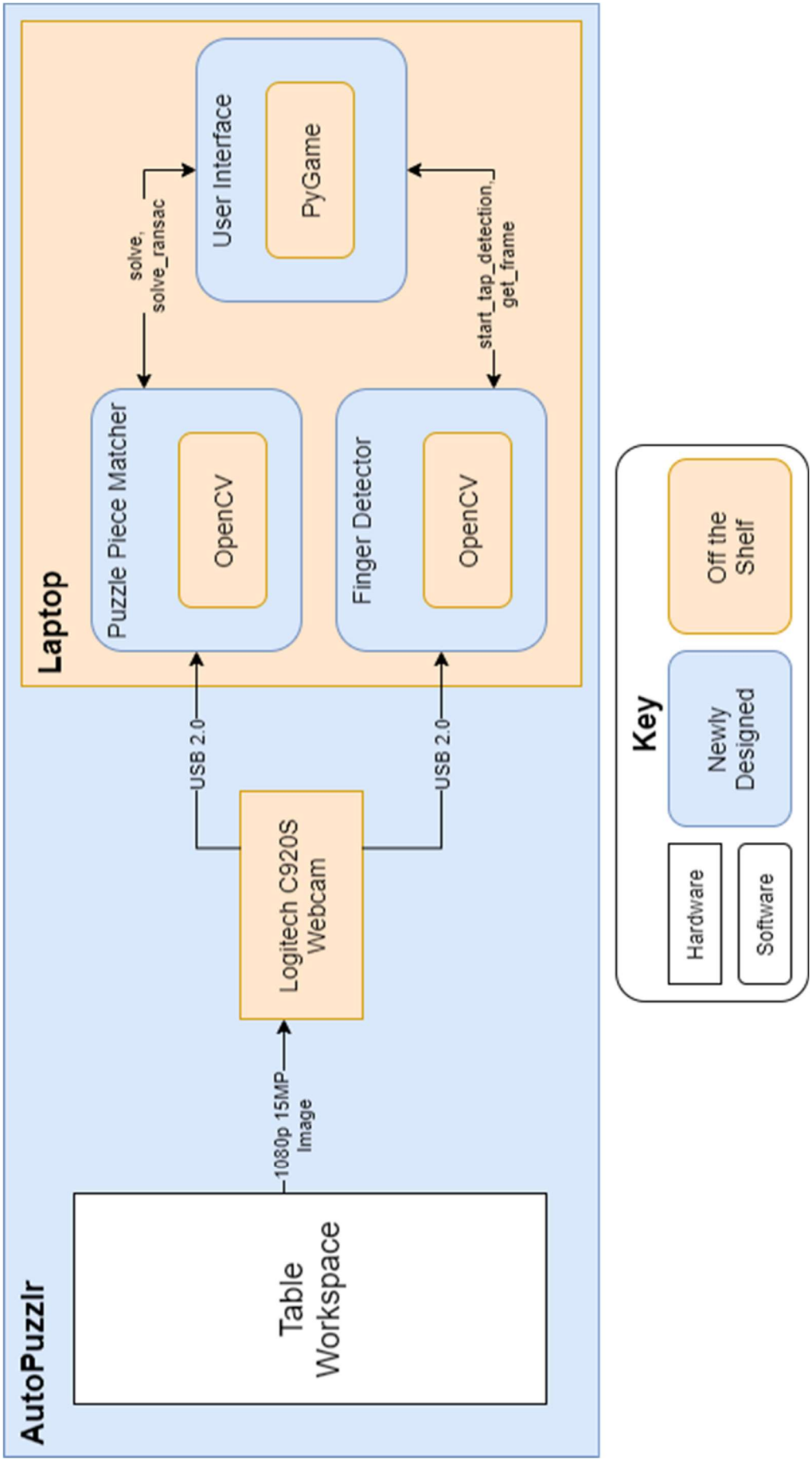
[2] OpenCV documentation, https://opencv.org/

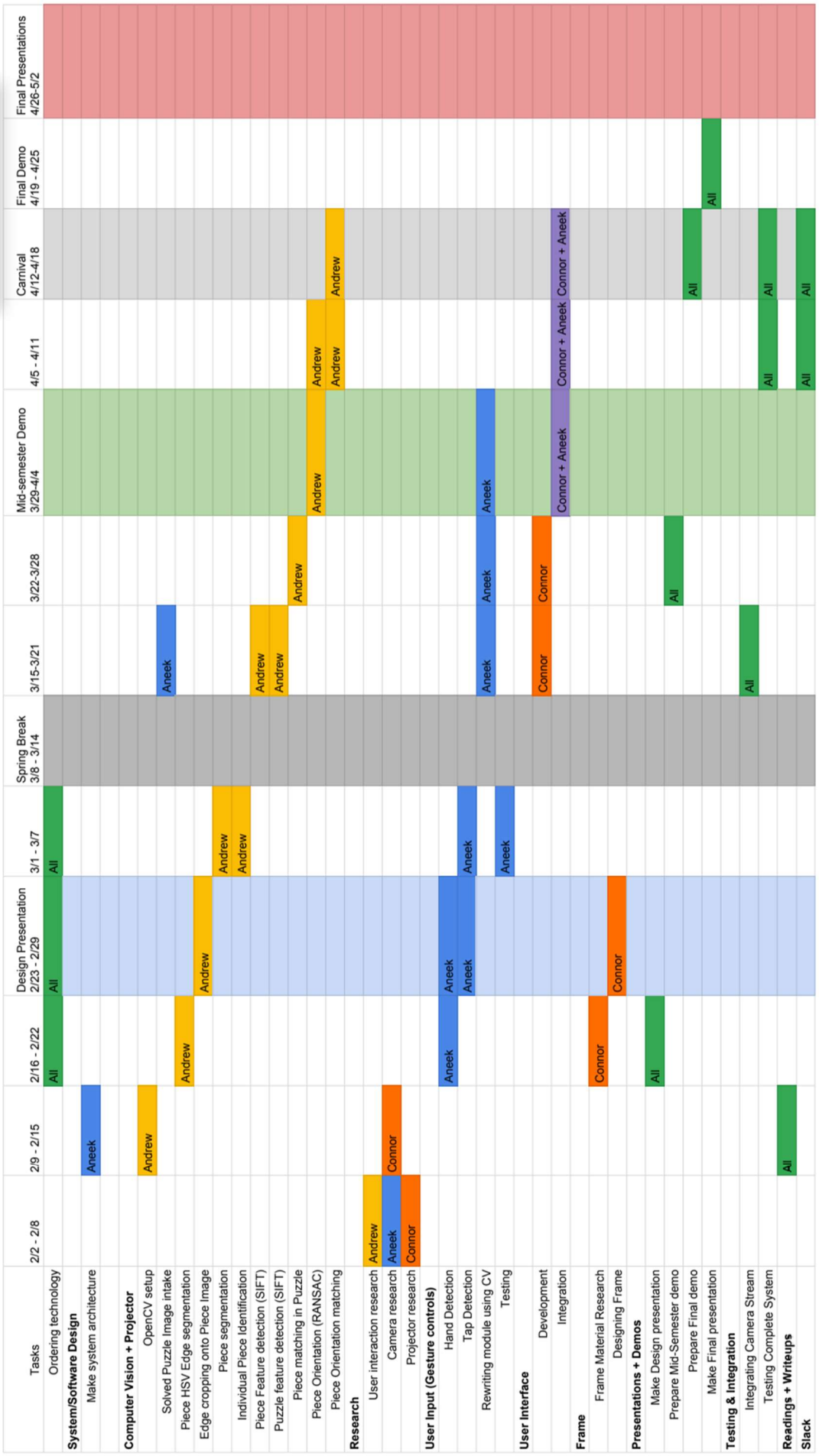
[3] Research paper on Leap Motion latency, https://pdfs.semanticscholar.org/3aab/d55945b1460620e78ff040e23a819f1523dc.pdf

[4] Team B9, 18-500 ECE Capstone team, Spring 2019, http://course.ece.cmu.edu/~ece500/projects/s19-teamb9/wp-content/uploads/sites/35/2019/05/Final-Report.pdf

[5] Finger Detection and Tracking using OpenCV and Python, https://dev.to/amarlearning/finger-detection-and-tracking-using-opencv-and-python-586m

[6] PyGame Documentation, https://www.pygame.org/docs/


AutoPuzzlr
Table Workspace
1080p 15MP Image
Logitech C920S Webcam
USB 2.0
USB 2.0
Laptop
Puzzle Piece Matcher
OpenCV
Finger Detector
OpenCV
User Interface
PyGame
solve, solve_ransac
start_tap_detection, get_frame
Key
Hardware
Software
Newly Designed
Off the Shelf


Tasks
2/2 - 2/8
2/9 - 2/15
2/16 - 2/22
Design Presentation 2/23 - 2/29
3/1 - 3/7
Spring Break 3/8 - 3/14
3/15-3/21
3/22-3/28
Mid-semester Demo 3/29-4/4
4/5 - 4/11
Carnival 4/12-4/18
Final Demo 4/19 - 4/25
Final Presentations 4/26-5/2
Ordering technology
System/Software Design
Make system architecture
Computer Vision + Projector
OpenCV setup
Solved Puzzle Image intake
Piece HSV Edge segmentation
Edge cropping onto Piece Image
Piece segmentation
Individual Piece Identification
Piece Feature detection (SIFT)
Puzzle feature detection (SIFT)
Piece matching in Puzzle
Piece Orientation (RANSAC)
Piece Orientation matching
Research
User interaction research
Camera research
Projector research
User Input (Gesture controls)
Hand Detection
Tap Detection
Rewriting module using CV
Testing
User Interface
Development
Integration
Frame
Frame Material Research
Designing Frame
Presentations + Demos
Make Design presentation
Prepare Mid-Semester demo
Prepare Final demo
Make Final presentation
Testing & Integration
Integrating Camera Stream
Testing Complete System
Readings + Writeups
Slack
All
Aneek
Andrew
Connor
Connor + Aneek
Connor + Aneek
Connor + Aneek

Item	Cost	Description	Status
Epson Powerlite 1776W Projector	0.00	Borrowed from ECE dept.	Did not use
Logitech C920 Webcam	69.99	Purchased on Amazon	Arrived; Used
Raspberry Pi 4 Model B	73.11	Purchased on Adafruit	Arrived; Did not use
Duvetyne Sheet	22.68	Purchased via independent distributor	Arrived; Did not use
Leap Motion controller	106.64	Purchased on Adafruit	Arrived; Did not use
Total	272.42		