Team A2 – Virtual Whiteboard

Authors: Alan Song, Andrew Huang, Brian Lane: Electrical and Computer Engineering, Carnegie Mellon University

***Abstract*—A system capable of allowing a user to control their computer cursor from a distance to access a web browser and navigate to several different pages. The system will allow for a touchless touchscreen experience where a user can use just their hands. State of the art systems are either too expensive or require the user to be very close to the screen for functionality. Our system will allow for a user to control their device from a comfortable distance using just a camera and code that can run on their computer.**

***Index Terms*—Calibration, Computer Vision, Cursor Location Transform, Gesture Recognition, Hand Detection, Neural Network, OS Interface, Pose Detection**

I. Introduction

The Virtual Whiteboard originated from the initial idea of developing something similar to Tony Stark's Iron Man Suit user interface, where he just uses his hands to control a bunch of different things on his user interface without direct contact. However, we expanded on this idea of a touchless touchscreen and thought of how it might be practical in the real world. As students who likely spend lots of time on computers, there are a lot of downsides with prolonged computer usage including damage to the eyes from being too close to the screen and harm to the body from sitting too long. The Virtual Whiteboard which allows for any user to control their computer cursor from a distance with hand motion and gestures would allow for these problems to be mitigated since they can now stay standing and will not be close to their screens. The touchless aspect also has a sanitary benefit in this time where the pandemic is still an issue, since people who might have to use the same public computer can do so without transmitting germs. Additionally, this system would allow for students or teachers to give presentations or lectures naturally in a classroom environment while making the experience more interactive and engaging.

The most important requirements for this system are the distance at which the system is functional and making the entire experience very smooth for the user. The first requirement can be directly quantified, and we have decided on trying to make our system functional for users that are between 3 feet and 12 feet from their screen. The 3 feet minimum distance is because the average arm length is around 3 feet, so if a user is within this distance they could just reach out and use a normal touchscreen. The 12 feet maximum distance is the length of an average classroom at CMU (not lecture hall) and is also the maximum distance at which a user can comfortably see their cursor on an average 20-inch monitor. The smooth user experience will be expanded more upon in the design requirements, but we want to enable the user to simulate the capabilities of a mouse with their hands by using different gestures for mouse clicks.

II. Design Requirements

The smooth user experience can be broken up into three quantitative categories.

A. Latency

One of the key aspects of a smooth experience is a user making a gesture or a motion and seeing the result of it shown immediately on screen. This will be accomplished by making our design meet as low of a latency as possible. We have decided to strive for achieving a 50 ms latency for our system. This is equivalent to 20 Hz or 20 frames per second, which means the user's hand image is captured 20 times per second and the cursor on screen should update its position following user input at this frequency. This latency will not create noticeable lag to the average human and should make the system feel like it is instantly responsive.

B. Gesture recognition accuracy

When using a mouse or a touchscreen, a user wants a click to be registered as a click 100% of the time. When using any device, the user would desire that their inputs are properly detected all the time. However even then, it is natural for users to have to click multiple times with a mouse or to tap repeatedly on a touchscreen to guarantee their input goes through. We have decided to aim for less than 10% gesture recognition error in our system. To put this into perspective, for every 10 clicks a user tries to input through hand gestures, we would guarantee that they must possibly repeat a gesture only one extra time for successful detection. As an extension of gesture recognition, we want to ensure that users only need to repeat a gesture at most one extra time to execute their intended mouse command.

C. Cursor precision

Our system ultimately controls the computer's cursor, which allows for the user to interact with objects on the screen. For a standard resolution of 1920x1080 pixels, the smallest area that a user would have to click on is 15 pixels wide, which is the "exit" button for a tab in a web browser or the width of the scrollbar. For all other objects on screen, there is a larger area for the object to be interacted with the cursor. We want our system to be able to track user hand motion with an error of around 15 pixels so that the user will never misclick because of a system error, but only due to human error.

III. Architecture and/or Principle of Operation

The block diagram in figure 1 represents the overall system architecture. The project is largely done in software with the two main hardware systems being a camera and a laptop. The camera will provide the primary input to the system in the form of image data. We purchased our camera and will not be building our own camera for this project. The camera will continuously feed this image data into the laptop and more specifically the calibration and pose estimation blocks, which are both part of a hand detection module. The rest of the system is software that runs entirely on the laptop. The laptop is also something that we own, and we will not be building a laptop for this project.

A. Hand Detection

The hand detection module includes the calibration and pose estimation blocks in Fig. 1. The module takes in images which include the user and possible other objects in the background and isolates and identifies the user's hand. The hand data shows up as data points in the hand detection module and will be converted into coordinate points to be sent to other parts of the system. In the calibration block, the user will map out their range of motion by moving one of their hands in a circle to detect the largest range in which the user's hand can move. The range of motion will be sent into the cursor location transform which is part of the OS interface. The pose estimation block will continually send coordinate information about where the hand is located into both the cursor location transform and the gesture recognition module. Although the pose estimation block could potentially be used to determine gestures from the different points mapped onto the user's hand, we decided to just send landmark coordinates that were normalized to the bottom of the user's palm into the gesture recognition module instead. This is different from the design report where we planned on sending a cropped image of the hand with landmarks into the gesture recognition module instead of numerical coordinates. The pose estimation block was off-the-shelf while the calibration block was developed by us.

B. Gesture Recognition

The gesture detection module will take in normalized hand landmark coordinates to determine what gesture the user is making. This is different from the design report where we originally wanted to use a zoomed/enhanced image of the user's hand to determine what gesture the user is making. The change from inputting images to inputting numerical coordinates allowed us to develop a simpler and faster model for classifying hand gestures. The gesture recognition module uses a neural network to detect the user's hand gesture among a dataset of 26 different hand gestures, of which we only need five. The neural network will directly convert the landmark coordinate input into an integer output that represents the gesture detected. This gesture integer will be fed directly into the OS interface. The gesture recognition module was developed entirely by us, although the dataset used to train includes off-the-shelf images as well as images we took ourselves.

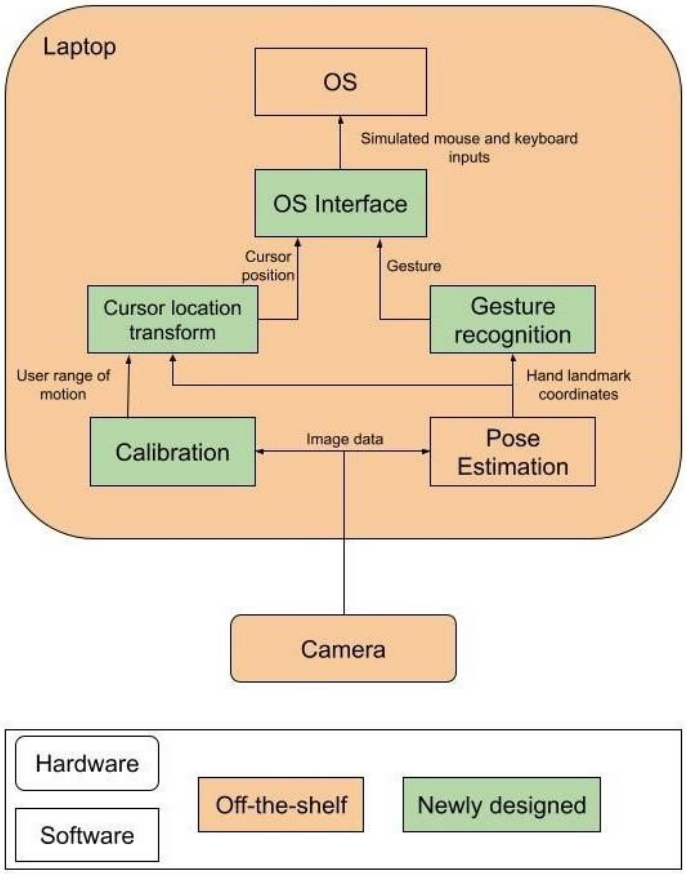

Fig. 1. Block diagram for entire system

C. OS Interface

The OS interface module includes both the cursor location transform and the OS interface. The cursor location transform is a type of calibration for the OS interface system. Since coordinates from the hand detection module will not correspond directly to coordinates in the OS to represent the screen, a transformation is needed to map motion of the user to motion of the cursor on screen. This way for users far from the screen they will not have to make super small and precise movements to move the cursor onto small on-screen objects. This scaling will also make the system's sensitivity like a regular mouse, where the user may have to pick up and re-adjust the mouse multiple times while using it. Once the calibration is done, the OS interface will move the cursor based on relative positioning, which is taking the difference in position between two time frames to determine how far to move. The gesture recognition module will provide the gesture which will be converted into a mouse action. The OS interface module will take the hand position, calculate where to move the cursor based on current and previous hand position, and perform certain mouse actions based on the inputted gesture integer. These simulated mouse movements and actions will be fed into the operating system of the laptop, allowing for direct control of the cursor through software.

IV. Design Trade Studies

A. Hand Detection Trade Studies

Multiple object detection methodologies have been used for hand tracking as the problem boils down to recognizing and sensing hand data. We considered multiple different solution approaches for hand detection: IMU, infrared sensors, ultrasonic sensors, computer vision, and Ultraleap Leap Motion Controller. Ultrasonic sensors eliminate the need for external sensors on the body, but usage of it would require complex calculations for extracting pose and location data of hand making it too difficult to work with for our use case especially given its low resolution of around 1 cm. Infrared sensors obtain sensor location with around a couple millimeters of resolution. However, the approach only gets the location of the sensors, and we would need to research and develop our own complex algorithms for pose and gesture estimation. IMU accelerometers are especially subject to drift, and small errors in measurements are exponentially multiplied during double integration for position estimation. We considered an external sensor as a calibration metric for position estimation as well as a Kalman filter, but both approaches seemed too complex given our problem statement and the effort to implement either was not outweighed by the benefits the IMU itself provided to our task. We also considered the use of an external hardware sensor called Ultraleap Leap Motion Controller that was geared specifically towards hand pose detection for AR games. While the accuracy was good at close distances, the most complex sensor the company provides has a max usage distance of around a meter; our solution statement requires around a three to fifteen feet operation distance. Thus, we ultimately decided on the use of computer vision for our detection algorithm for a multitude of reasons.

First and foremost, there has been by far the most work done in this area with regards to hand detection, so it simplifies our solution approach as well as more easily discretizes our task for hand recognition. Secondly, as we are using a neural network on the hand image for gesture recognition, the computer vision approach standardizes and feeds more seamlessly into our system pipeline. The specific library we'll be using is the MediaPipe body landmark recognition library [1]. The library includes a pretrained model that includes facial and hand detection algorithms on images fed through a camera and allows for us to tune parameters like detection confidence and maximum number of hands to detect. With respect to the Leap Motion Controller, a computer vision approach will allow us to increase the distance with which our user can operate provided our camera has a high enough resolution. To combat low confidence detection due to motion blur when a user moves their hand in the camera view, a camera with a higher frame rate refresh will be used as well. The high frame rate camera (60 frames per second) will also meet our requirement for 50 ms latency. We also decided on a webcam because it is likely that our users will have access to their own webcam that they can use with their own laptop to run our system. Practically, it is unlikely that users have professional cameras handy to use to get our system working on their computer.

B. Gesture Recognition Trade Studies

Classification of user hand gestures from image data is a perfect fit for a machine learning approach. Finding a function to discriminate between the number of hand gestures required to meet user product specifications would not be feasible by hand. The requirements for this users' product specifications of this gesture recognition include quick training of our model and low inference computational latency, as well as high accuracy. Considered approaches included deep learning in the form of a deep convolutional neural network applied to raw image data or a simpler architecture neural network that would be provided feature data in the form of landmark coordinates output by our hand pose estimation. Both approaches would be implemented with a simple supervised learning approach employing stochastic gradient descent on a multiclass cross-entropy loss function. Unsupervised learning approaches were found unfit for this product, due to the extended learning time required, as well as the saturation of available appropriate datasets including images of various hand gestures.

The final design selected is the latter of the two options: the simple neural network trained on pose estimation data. This selection was made because the simplicity of the model would allow for a much shorter inference time, as well as a predicted higher accuracy than what could be achieved from raw image data that would include background pixels, pixel alpha differences from lighting, as well as overall higher noise. This meets the requirements for a smooth user experience by lessening overhead for latency and improving classification accuracy for gesture recognition accuracy.

C. OS Interface Trade Studies

The OS interface and cursor location transform will be implemented in Python. Python will allow for easy connections between our design components and easy transfer of information between modules. The control of the mouse through the OS will be done using the mouse library in Python [2]. Other libraries that could accomplish the same task of controlling the cursor include pywin32 and pyautogui [3][4]. All these libraries would fit the user product requirements since they would all be able to interface with and control all aspects of the cursor with minimal overhead. However, the mouse library has excellent user-friendly wrappers that are much easier to work with than the functions in the other libraries.

We chose to use a laptop rather than something like an RPi (Raspberry Pi) because we felt that this fit our user product requirements better. The goal of our Virtual Whiteboard is to allow the user to control their own computer from a distance using hand gestures. Having on the system run on an RPi may be sufficient and cheaper for demonstration purposes, but the goal is for users to be able to run the code on their own computers and use their own webcams to utilize the system on their own. Therefore, ensuring that the system can function on one of our laptops using a webcam is what we want to verify that our system fits what we want it to do.

D. Gesture Recognition Model Trade-Off

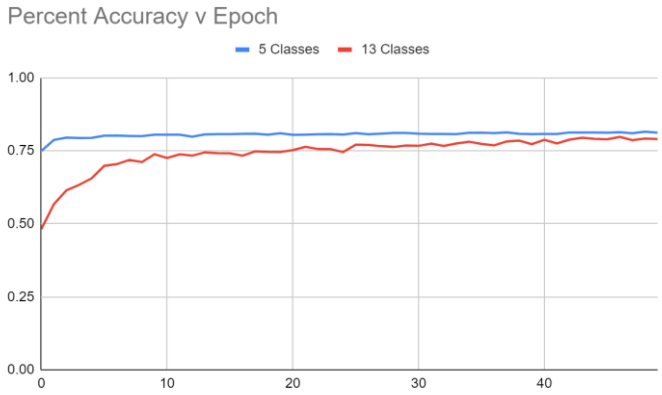

Fig. 2. Model accuracy vs epochs for 5 classes and 13 classes

One major trade-off for machine learning models is the relationship between verification accuracy and the number of classes used for classifying. There is an established relationship between the number of classifications and the accuracy of the model. As the number of classifications goes down, the accuracy of the model should increase. In our case, we started with a dataset of 26 possible gestures (13 for each hand), but since we only need to use 5 of these, we wanted to see if it would be beneficial to train a model for classifying gestures into only 5 classes per hand instead of the original 13. The results shown in Fig. 2 indicate that the model accuracy does indeed benefit from a smaller number of classifications. However, while accuracy does improve, this limits the expandability of the system to include more features in the future using more gestures. The improvement of the model allows up to get up to 88% validation accuracy, which is enough for us to meet our gesture accuracy requirements with some clever improvements in the OS interface implementation.

E. Latency vs Accuracy Trade-Off

The two biggest and most important requirements that we could trade off for our system would be latency and accuracy. We found that in our system, the hand detection module and the OS interface did not have tunable parameters that would trade off between latency and accuracy. Instead, the gesture recognition model was the most likely candidate for any evaluation in this department. With the trade-off done in section D with different classification numbers, it turns out that having the same machine learning model structure keeps latency constant, regardless of how many classes are used for classification. We then thought to experiment with different model architecture and developed a deeper model that would sacrifice speed for accuracy. Unfortunately, this model turned out to yield lower validation accuracy than our simpler shallow model, so the result of our brief trade-off analysis here was to stick with our original model.

F. Use Case Distance vs Accuracy Trade-Off

While we wanted our system to be fully operational between 3 feet and 12 feet of distance from the camera, we want to see what the ideal distance to use our system is. At variable

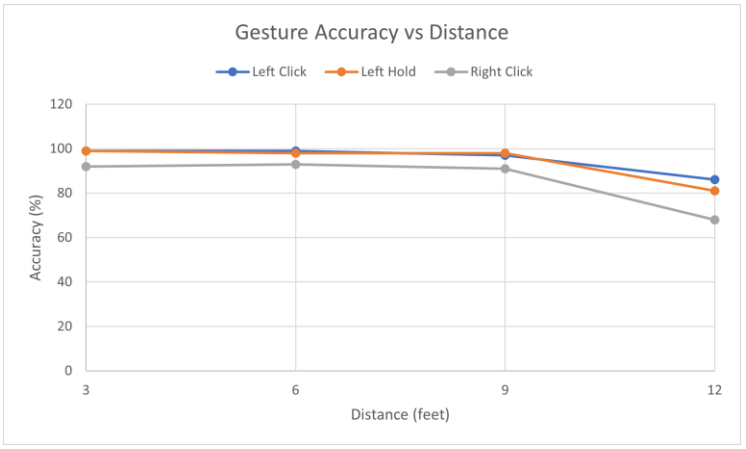

Fig. 3. Gesture accuracy vs distance for 100 trials of mouse commands

distances, latency and cursor precision did not vary much, however gesture accuracy in practice did. The process of obtaining the gesture accuracy will be expanded on in the testing and validation section. As seen in Fig. 3, gesture accuracy is very high and consistent between 3 feet and 9 feet, however there is a definitive drop at 12 feet. It is also worth noting that right click detection accuracy was consistently lower than the left click and left hold + drag accuracies. While we wanted to isolate an ideal distance for using the system, any distance between 3 and 9 feet proved to have nearly perfect left clicking capabilities, while the system did seem to struggle with accuracy at our maximum of 12 feet.

V. System Description

A. Hand Detection

To get accurate live-time hand detection, we used the Mediapipe library which is a crowd sourced package that produces several detection models for different human structures such as hands, face, and body pose. The hand detector we used scans an input image for sub-images that look like hands and tries to map a set of 21 landmarks corresponding to 21 major hand points, as seen in Fig. 4 and Fig. 5. If the detector found a sub-image of the camera frame that looks like a hand, it returns these 21 landmarks in the form of a 21x2 matrix where the i^{th} 2-dimensional vector of the matrix corresponds to the x, y pixel location of the i^{th} landmark.

When we tested the package, MediaPipe was accurate to a reasonable extent. The Mediapipe hand detector streamlines our solution approach since it allows us to vectorize our data making our model simpler and easier to fine tune and train. In using Mediapipe, we were able to avoid using a convolutional neural net which would likely have slowed down our train time depending on our kernel size and increased latency on our entire system since the model would have more inputs. We applied the hand detector to our entire train image set into a csv file where the i^{th} csv row corresponds to the hand detector landmarks of the i^{th} image.

Fig. 4. Image of Andrew with landmarks on his hand

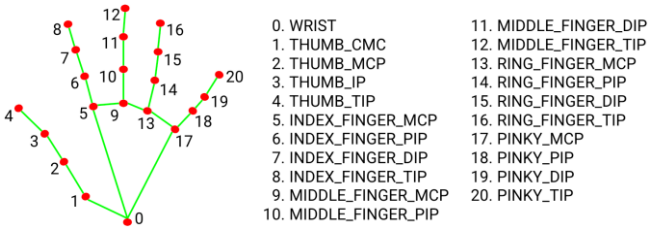

Fig. 5. 21 landmark coordinates from pose estimation

The calibration step was implemented as follows. The calibration happens on system start-up, giving the user a few seconds to move to the position they want to use the system from while the camera connects to the system. The code will then continue to accept hand coordinate inputs for about 10 seconds in which the user will move their hands through a comfortable range of motion. The cursor location transform function will record the maximum and minimum x and y coordinate values received within this time frame. These x and y values will be used to construct a rectangle that represents the user's range of motion. This rectangle will then be scaled to the device's screen size. Screen resolution is obtained using the ctypes library to directly obtain screen size using the GetSystemMetrics() method. After the calibration, all hand positional changes will be scaled based on this calibrated rectangle of motion.

Originally, we wanted to fit a bounding box that has dimensions of the interaction screen to the person to serve as a mapping between the hand location to on screen coordinates where we will put the mouse. Our initial functionality would have allowed the user to move in both vertical and lateral directions, so we would have to scale the bounding box initially calibrated to the user to follow their positional change. However, given that our use case shifted from giving presentations or demonstrations to a more general-purpose cursor control from a distance, the original bounding box idea was scrapped.

After calibration, the hand detection module continually sends the 21 landmark coordinates, seen in Fig. 5, to the OS interface to control cursor movement, and sends normalized coordinates to the gesture recognition module for hand gesture detection.

B. Gesture Recognition

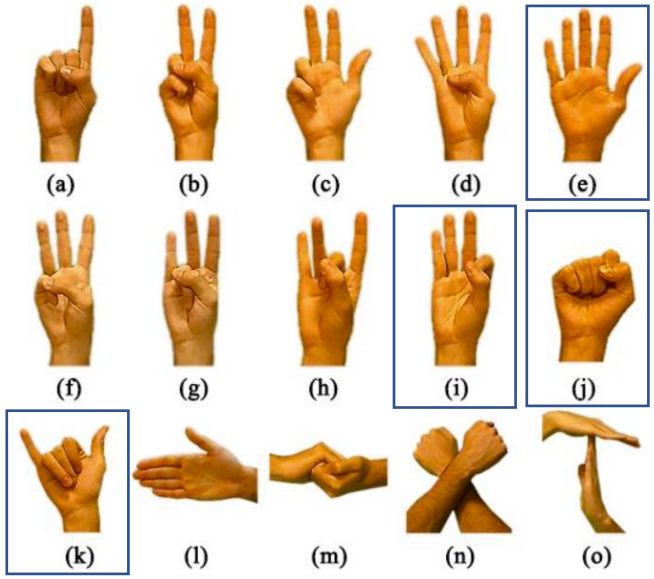

Fig. 6. Subset of gestures in the HANDS dataset with relevant gestures indicated

Gesture recognition was initially proposed to be implemented through transfer learning from pre-built gesture recognition models. This avenue did not end up being explored though, because of availability of time sequenced data containing relevant gestures on which to train. The gesture recognition model was intended to be trained on a data set of static images containing gestures, and no accessible pretrained models were found to meet this specification.

The gesture recognition model (GRM) for this project was instead custom designed and trained in python using the pytorch library. We used a dataset called HANDS [5][6], hosted by Mendeley Data, to perform this training. The GRM we created functions by taking in an input of 21 coordinate pairs corresponding to the location of the 21 hand landmarks detected by our pose estimation. These 42 features (2 for each of the 21 landmarks) are then normalized before being run through a fully connected neural network (FCNN) consisting of 13 hidden layers with two batch normalization layers with ReLU as an activation function. The output layer consists of the 13 one handed gestures represented within the dataset, and outputs are softmaxed to generate an estimation probability, with the highest probability gesture being selected as the model's output.

While training this model several data processing and augmentation tasks were performed. Initially, the training dataset consisted of RGB-A images of subjects performing gestures with each of their hands, along with a text file containing annotations, including labels, and bounding boxes within the image. To adapt this data to suit our model's needs,

python scripts were written to read through the annotations file, find each labeled gesture within each RGB-A image using its bounding box. Following this, our MediaPipe pose estimation software was applied to the labeled gesture to produce the coordinates of the hand's 21 landmarks, which were then standardized by designating the wrist landmark as the origin and recording each landmark's location as relative to this point. These relative landmarks were recorded into a CSV file along with the label, and this data is what was used to train the model.

Along with this data preprocessing, data augmentation was also utilized to improve the performance of the GRM on gestures made at various angles and with various orientations. This data augmentation involved representing each gesture as a three-dimensional matrix along the x-y plane, and then applying matrix rotations around the x, y, and z axes at random angles between -45 to 45, -45 to 45, and -90 to 90 degree angles for each axis respectively. Three randomly rotated gestures were created for each data point in our training set, resulting in a quadrupling of our model and vastly improving the accuracy of the model when in use.

The training of the GRM was performed over 50 epochs with a 1:3 validation to training data split, resulting in a final validation accuracy of about 88%.

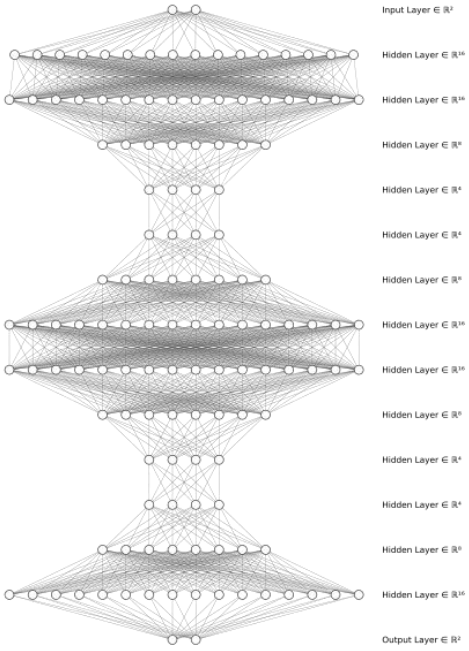

Fig. 7. Visual reprsentation of gesture recognition model

C. OS Interface

As mentioned before, the OS interface, which includes the cursor location transform, takes inputs from the hand detection and gesture recognition modules, and outputs simulated cursor actions for the OS to execute. The OS interface is implemented in Python and mainly makes use of the Python mouse library.

Firstly, the cursor location transform function is implemented by taking in hand coordinate data from the hand detection calibration block. While the system is running, the cursor location transform will continue to receive hand position coordinates and convert them to coordinates on the screen. The module will keep track of the currently received position as well as the three most recent previous positions. The x and y coordinates of the current position and the average of the previous three positions will be subtracted, and the result put into the mouse library move function. The averaging of the previous hand coordinates is done to smooth our cursor movement and decrease the amount of jitter. The function mouse.move takes in x, y, and absolute as input. The absolute input tells the function whether the x and y values represent set coordinates on the screen or if they represent the change in x and y that the cursor must move. This is an important distinction because our cursor movement is all relative since we are not scaling the range of motion to cover the entire screen. If we were to try to implement absolute cursor movement, the user would likely struggle to reach the corners of the screen as they would have to extend their arms as far as possible. The cursor location transform handles all the cursor movement.

When we first tested our system, we noticed that if we based our mouse movement off a specific hand point like the wrist, we would get a lot of jagged cursor motion due to a combination of non-complete stability of the hand and noise in image and frame affecting landmark mapping. We wanted our cursor update location to mimic mouse functionality, and that would entail being able to move physical location on screen and moving the cursor in the same way, like how if you pick up a mouse in physical space and set it down, on screen cursor location does not change. As such, we needed to stick with the vectorization approach with the difference between the first and last cursor location. Ultimately, we added a couple different smoothing methods to get our final smooth cursor motion. First, we set the hand position at each frame to be the average of the 0, 1, 5, 9, 13, 17 landmarks which correspond to the hand parts bordering the palm. The reasoning for this is that we use the fingers to make gestures, so if we used an average of finger landmarks as well to estimate pose, cursor location would change when we're making a gesture which would heavily impede the accuracy of our system. We then added a 3-frame sampling window keeping track of the last 3 frames of hand location and set the current location equal to the average of those three frames. Since we were using a 60-fps camera, this corresponds to a 0.05 s window where we obtain this average hand location which serves as a low pass filter. Our final filtering step involves checking to see if the distance between the current frame position and the previous hand frame position is greater than 0.05-pixel distance after sensitivity calculations gathered from the initial calibration phase and only update the cursor in the direction of the movement vector if it's greater: another low pass filter. We manually experimented with sensitivity scaling as well as the sampling window frame to get our final smoothness of the cursor movement and settled on values that we found to be adequate for our user testing.

The other mouse operations including left and right clicking and holding will also be implemented using the mouse library and will use the gesture recognition output. The integer representation of hand gesture will be cased on, and different

mouse functions will be called corresponding to what was detected. The system remembers the three most recent gestures detected and takes the mode of this to determine what command to execute. This is to ensure that errors in the gesture recognition model output or gestures outputted in the middle of transition between two different gestures does not get inputted as a mouse command. The commands mouse.press, mouse.release, and mouse.click were used. Press, release, and click function exactly as they sound like, where press keeps the mouse button pressed until release is called and click issues a click to the OS. Originally, we had planned to only use press and release with clicks implemented as a press with release shortly after, but we later decided to use two different gestures to differentiate between clicking and holding instead. All the above functions take in mouse.LEFT or mouse.RIGHT as an argument to distinguish between left mouse button and right mouse button commands. We had planned on implementing a separate scrolling gesture but found that this was unreliable as the possible choices from our gesture output did not have high confidence in classification.

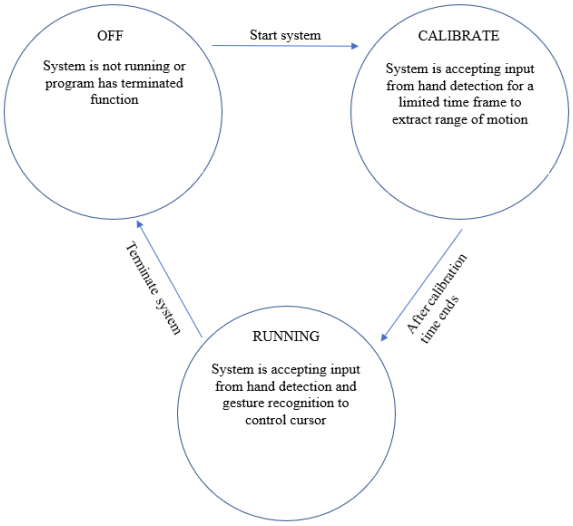

Fig. 8. Overall state machine view of our system

VI. Test and Validation

A. Latency

To test latency, we used Python's time.time() command to set benchmarks at different points in our code to measure how much time passed during the execution of each module and the system as a whole. We measured latency for the hand detection and gesture recognition modules separately, and then had an overall system latency. The hand detection module latency was measured as the time it took for an image to be inputted into the system until the landmark coordinates were assigned to the image. The gesture recognition module latency was measured as the time between when the landmark coordinates were received by the model and when the model outputted its classification. The overall system latency was the time between when a new image was inputted from the camera to after a mouse command had been executed in the OS. These latency measurements are collected whenever we run the system. There is a function that collects measurements and then outputs the minimum, maximum, and average latencies measured. The results are shown in Table 1.

Specification	**Performance**
Hand Detection Latency	Min: < 0.1ms Max: 4.7ms Mean: 1.8ms
Gesture Recognition Latency	Min: < 1ms Max: 3ms Mean: 2ms
System Latency	Min: 16ms Max: 75 – 80ms Mean: 36ms

Table 1. Latency measurements

Our measured latency falls within our requirement of 50ms. In fact, our results are within our specification by a margin of about 47%. This means that our system functions fast enough so that our users perceive mouse movements and commands as if they were essentially instant, which should make the user experience extremely responsive and smooth.

Due to the choice of simplifying our model and using landmark coordinate data to classify gestures instead of image data, we were able to meet the requirements immediately as we started running tests, and our system managed to surpass our requirements as we continued to develop and make improvements to it.

B. Gesture Recognition Accuracy

The gesture recognition accuracy was measured and quantified in two ways. The first measure of accuracy is the pure validation accuracy that we have for our final gesture recognition model. This is the simulated accuracy of our machine learning model using verification data. We obtained a validation accuracy of about 88% for our final model.

While this value falls below the 90% gesture detection rate outlined in our requirements, our second test measures true gesture accuracy in practice while users are using the system. Our code includes lots of features that should make the gesture detection accuracy in practice much higher, and indeed it does.

The second test for gesture accuracy is done by performing different mouse commands at various distances and counting the number of missed commands. In each trial of this test, the user tried to perform 100 of a certain gesture, both in a row and as a mixture of different gestures and counted the number of successfully recognized and executed gestures. This was also done at variable distances to try and find the ideal distance for system use. For this metric, we only collected data from our team members, who are more experienced and used to using the device. In total, we performed around 30 total trials of 100 of each mouse command. We wanted to gather metrics for our system with optimal usage and decided to include new user testing as another section of testing. The results for our testing are shown in Table 2.

Distance	Left Click	Left Hold	Right Click
3 feet	99/100	99/100	92/100
6 feet	99/100	98/100	93/100
9 feet	97/100	98/100	91/100
12 feet	86/100	81/100	68/100

Table 2. Gesture accuracy measurements

According to our metrics, our system functions just as we required from the 3 feet to 9 feet range. However, at 12 feet the system does meet the requirements that we wanted. A big reason for this is that at 12 feet and further, the 21 landmark coordinates assigned to the user's hand that are used for gesture recognition start to become closely grouped together, and the distinction between different gestures is hard for our model to capture. Ultimately, this is a limitation of our camera resolution and the way MediaPipe works in assigning hand landmarks. Also, our data indicates that the gestures we selected for left click and left hold were detected with much higher confidence and accuracy than the gesture we chose for right click. In practice, the system is nearly flawless for just left clicking and left holding, which is what most of interacting with a web browser needs. Many trials ended in perfect 100/100 clicks and holds. There were less than 5 instances in which two mouse commands failed in a row, meaning that for any mouse command a user would only have to retry it once to get it to work properly in practice.

C. Cursor precision

To quantify cursor precision, we measured the amount of jitter the mouse cursor experienced while we kept our hands still at variable distances. In this test, we would keep our hand still for a 5 second interval and measure the standard deviation of the cursor during this time, as well as the maximum distance between two points the cursor moved during this time. This measurement of cursor precision allows us to quantify the uncertainty in cursor movement, since this jitter is what prevents the cursor from perfectly following user hand movements. We ran this jitter test over 5 second intervals around 20 times total between team members. Initial testing of our system yielded the following results for our cursor jitter.

Distance	Average Jitter	Max Jitter
3 feet	5.76 pixels	38.6 pixels
6 feet	4.29 pixels	35.4 pixels
9 feet	3.73 pixels	25.0 pixels
12 feet	3.85 pixels	29.2 pixels

Table 3. Initial jitter measurements

Initially, our results indicated that maximum potential jitter at the 3 feet and 6 feet distances were beyond our 30-pixel cursor precision requirement. However, at 9 feet and 12 feet our system satisfied our requirements. These jitter values are caused by the pose estimation constantly estimating and re-estimating landmark coordinates, many times shifting the average position ever so slightly, which translates to bigger movements of the cursor. We wanted to try and lower this value and eliminate any jitter if we could, since jittering of the mouse on screen while the user's hand is stationary does not make the system seem very stable. After we made changes to our system, we ended up re-doing our cursor precision tests and ended up with virtually no jitter across 20 more trials of the test. In the end, our system has no observable cursor jitter, especially during use where the user corrects their own movement as they use the system.

D. New User Testing and Feedback

We originally planned on issuing a formal user survey for new users who tried out our system, including a somewhat quantitative rating between 1-10 of how easy to use and smooth our system was. However, we instead opted to go for a more qualitative approach in evaluating new user testing, since we noticed there was a lot of variability in our new user testing.

Starting from Thanksgiving break and through the final minutes of the Final Demo, our group had many friends, family members, students, faculty, and strangers test out our system. In total, we had around 25 new users try out our system either on our home setups or during the TechSpark and final demos. These new users covered a vast range of age groups and familiarity with engineering and computer vision technology.

Every single user seemed to be comfortable with the mouse movement immediately. Several users commented on how accurate and precise the mouse movement was. There was little to no trouble for new users to navigate their cursors to locations that they wanted to interact with. However, when it came to executing mouse commands with different gestures, there was a lot of mixed results.

Upon very first use of the system, around 20 of the 25 new users struggled to execute mouse commands immediately. We first gave them vague instructions such as "make an ok sign to click" or "make a fist to hold and drag." There were 5 users who were immediately able to use our system almost flawlessly and continued to do so during their time with our system. These users all said something along the lines of, "This thing is so intuitive and easy to use." There were around 12 users who were able to execute clicks and holds after a few tries. After they got the hang of it, a few of them also commented on how it was easy to use after they had some practice with it. Unfortunately for the remaining 8 users, but fortunately for us, they continued to struggle with executing mouse commands for a while, but we were able to gain insight into how to best use our system.

For these 8 users, we gave more specific details in how to use the system. An important aspect is keeping the hand in an open palm state that faces the camera when moving the mouse around. This way, the camera can pick up on the changes in gestures and execute the correct commands. Another thing is stay aware of where in the camera frame the user's hand is. Users that kept their hand near the center of the camera's view had greater success than those that ended up on the edge of the frame. Additionally, there were a lot of small intricacies to executing some of the commands that the team members had not realized, since we were so used to using the system. A big part of using the system is to exaggerate the hand gestures as much as possible, such as extending the thumb as far out as possible when trying to right click, or to really make the "o"

part of the "ok sign" visible when executing a left click. By the end, we were able to get every user who tried our system to successfully execute a few clicks and holds.

The qualitative data we collected was a bit disappointing, as we had hoped that all users could have had a smooth experience right off the bat. However, the new user testing was very informative in showing us what we might want to improve on if we had more time, and how we would show our system off in the future. The experiences we gained during the TechSpark demo allowed us to have smoother demonstrations and new user success for the final demo.

As an aside, nearly every user was impressed and commented that the project was very cool!

VII. Project Management

A. Schedule

Our schedule was designed based on team member responsibility and how our different modules connect. The hand detection module and the gesture recognition module can be developed in parallel before needing to be tested together before overall system integration. The OS interface can also largely be worked on in parallel and would just need to change the potential inputs based on modifications that are made to the hand detection and gesture recognition outputs. The schedule also leaves a lot of time near the end of the class to focus on integration and getting all our modules to work together. This time at the end also allows for lots of time to verify and test our system.

Our schedule was updated extensively right after our design report. Firstly, we were able to highlight more specific and relevant tasks now that we knew what exactly our project was and what would go into completing it. We also scheduled much more integration earlier on instead of working purely in parallel until the final few weeks. Our project's submodules were much more connected than we initially thought, and our new modified schedule allowed us to work out issues together much sooner. Our new schedule is shown on a separate page at the end of the report.

B. Team Member Responsibilities

Alan's main responsibility is to develop the OS interface module. He will develop most of the code in Python to interface with the OS and cursor. Since Alan's part largely relies on receiving inputs from the other modules, he will be designated some lighter tasks in earlier weeks to help the other team members with getting their components up and running. Although not explicitly depicted on the schedule, Alan's secondary responsibility will be to aid both Andrew and Brian with the hand detection and gesture recognition modules respectively, especially in the first weeks. In the last few weeks, he will work with the whole team on integration and testing of the entire system.

Andrew's main responsibility will be to develop the hand detection module. He will be responsible for turning the image data received from the camera into hand position data for the OS interface as well as sending enhanced and cropped hand image data to the gesture detection algorithm. Since his responsibilities overlap with the gesture detection, his secondary responsibility will be to work with Brian and ensuring that images are properly sent from the hand detection module into the gesture detection module. In the last few weeks, he will work with the whole team on integration and testing of the entire system.

Brian's main responsibility will be to develop the gesture detection module. He will be responsible for training our neural network model to receive image input and produce gesture classification output. Since the gesture detection module is largely dependent on the hand detection module, Brian's secondary responsibility will be to work with Andrew and ensure that the proper images are sent into the gesture detection module. In the last few weeks, he will work with the whole team on integration and testing of the entire system.

The division of labor above was essentially what happened while we developed our system, however there was a lot more overlap where every team member was involved in making improvements to every submodule and the system together. The main developments of the submodules were an individual task but bringing the whole system together and improving performance based on collected metrics was a job that everybody contributed to.

Budget

As shown in table 1 below, our budget is only used for our cameras to capture image data and AWS credits that we planned to use to train and run the gesture recognition model but ended up not using.

Description	Model	Manufacturer	Quantity	Cost
Camera	C922x	Logitech	1	$99.99
Camera	BRIO	Logitech	1	$199.99
AWS Credits	N/A	Amazon	1	$50
Laptop	Varies	Varies	1	$0

Table 4. Bill of Materials

C. Risk Management

The biggest risk in our project is the integration of all our individual components and the final product meeting all our design requirements. We foresee that integration will likely be a tough challenge, and so we have allocated sufficient time in our schedule to focus on this aspect. We decided to do a lot of our individual development in parallel so we could all come together at the end to sort out problems that came up during integration and testing. The risk of not meeting design requirements is also largely present since we are developing our modules in parallel, so even if individual testing and verification passes, once the entire system comes together, we may run into additional issues with meeting our requirements. Our resources should not be a point of risk at all since we chose a project that requires simple resources. However, this means that a lot of the success of the project will fall onto our individual responsibilities of developing the proper software. By following the proper process that we were taught to in this

course, starting from the proposal to the design review, we can think about all these risks ahead of time and plan/schedule accordingly.

VIII. Ethical Issues

There are always potential ethical issues that arise when it comes to any new product. Our Virtual Whiteboard is no exception to this fact, but we have carefully considered these issues. One ethical issue is accessibility for who can use our system. There was some discussion at the demos about how our system could also help those with physical disability who cannot use a mouse, and that we could potentially customize gestures for those that can only make certain gestures. However, as our system is now, many of our gestures are still difficult to make for those with physical disabilities such as carpal tunnel. Those with these types of disabilities would still not be able to interact with their desktop environment using our system.

Another concern is that our project relies heavily on camera usage to feed image data into our system. Any product that uses a camera has some privacy concern attached to it, whether it be camera data collected or a possible hack that steals camera data. Our system only uses the camera to move the mouse, but a malicious party may take advantage of the camera part of our system to invade privacy of our users. Additionally, if our system were to be used extensively, many users may become used to having their camera constantly on since they would need it to control their computer. This may lead to user's being unaware that their camera is currently on, and users may not be aware that their image data is being captured and could potentially be used by malicious attackers.

In terms of accessibility, there are other potential hand gestures or even body parts that users can use to control their mouse. Our development of a system that uses hands is just a start, but the technology may be there for users to control their desktop using something like just their eyes. As for the camera aspect of our project, we can only warn the user about the possible dangers of keeping their camera on all the time and tell them that even though we are not invading their privacy through camera usage, there is always a risk of being hacked.

IX. Related Work

Basically any product in the field of Virtual Reality could potentially be extended to do the same thing that our Virtual Whiteboard does. VR systems such as the Oculus can be used to interact with the cursor. These systems that allow users to interact with a virtual world using their hands can probably also have their application extended to using a web browser with just their hands. Additionally, touchless touchscreen technology in products such as kiosks are like our system and could potentially be extended to work with users at further distances. The standard of touchless touchscreen is very short range but accurate detection, but it is only a matter of time before the range and accuracy continue to increase. There are also other ECE capstone projects from previous semesters that are like our project, such as the Gesture Glove from this semester, which uses similar machine learning techniques to classify hand gestures for sign language applications, and the AR FruitNinja project from a previous semester that utilizes hand tracking for more of a game application.

X. Summary

Overall, our system was able to meet our design specifications. Our system latency and cursor precision were easily within our requirements and allowed for very smooth operation of certain aspects of our system. As for gesture recognition, left clicking and left button holding work fantastically and are as reliable as a mouse for users who get used to the system, however there is still room for improvement in incorporating other gestures and mouse commands. Right now, the system works super well just for using the left mouse button, which is frankly enough for using a web browser and a lot of activities on the computer. However, there are improvements that could be made to the system. If we had more time, we would have experimented more with the options outlined in our design trade studies. If we could combine our computer vision approach with an IMU to possibly get more accurate detection of gestures in a 3D space, or if we were able to develop several different complete machine learning models for classifying gestures, our system accuracy and extendibility could be improved greatly.

While we obviously learned a lot about working with computer vision and machine learning models during our time working with our project, but we learned that it is worth looking into all your options and experimenting with a lot of ideas early on before you must commit to something. We decided on using computer vision and simple neural net earlier, but I would say that weighing all your options and doing more work earlier, even if you might have to abandon most of your options to focus on one, is worth it to get the best result as possible in the end. On the project management side, it is very important to pay attention to all the early semester assignments, especially the design review report, and to really listen to the feedback from professors and TA's.

XI. AWS

For this project we were fortunate enough to have access to AWS credits for cloud computing, which we used to develop and train our gesture recognition model. Our AWS credit usage only tallied up to about $15, using up only one $50 credit. Due to the relatively simple design of our model, a large amount of computational power was not required, and we were able to use the low priced "a" instances. The access to cloud computing was extremely valuable to the development of the machine learning portion of this project, and we would like to thank Amazon for providing us with these credits.

Glossary of Acronyms

CMU – Carnegie Mellon University
IMU – Inertial Measurement Unit
OS – Operating System
RPi – Raspberry Pi
VR – Virtual Reality

References

[1] MediaPipe Hands https://google.github.io/mediapipe/solutions/hands.html
[2] mouse https://pypi.org/project/mouse/
[3] pywin32 https://pypi.org/project/pywin32/
[4] PyAutoGUI https://pyautogui.readthedocs.io/en/latest/
[5] HANDS: an RGB-D dataset of static hand gestures for human-robot interaction https://doi.org/10.1016/j.rcim.2020.102085
[6] MEGURU: a gesture-based robot program builder for Meta-Collaborative workstations https://doi.org/10.1016/j.rcim.2020.102085

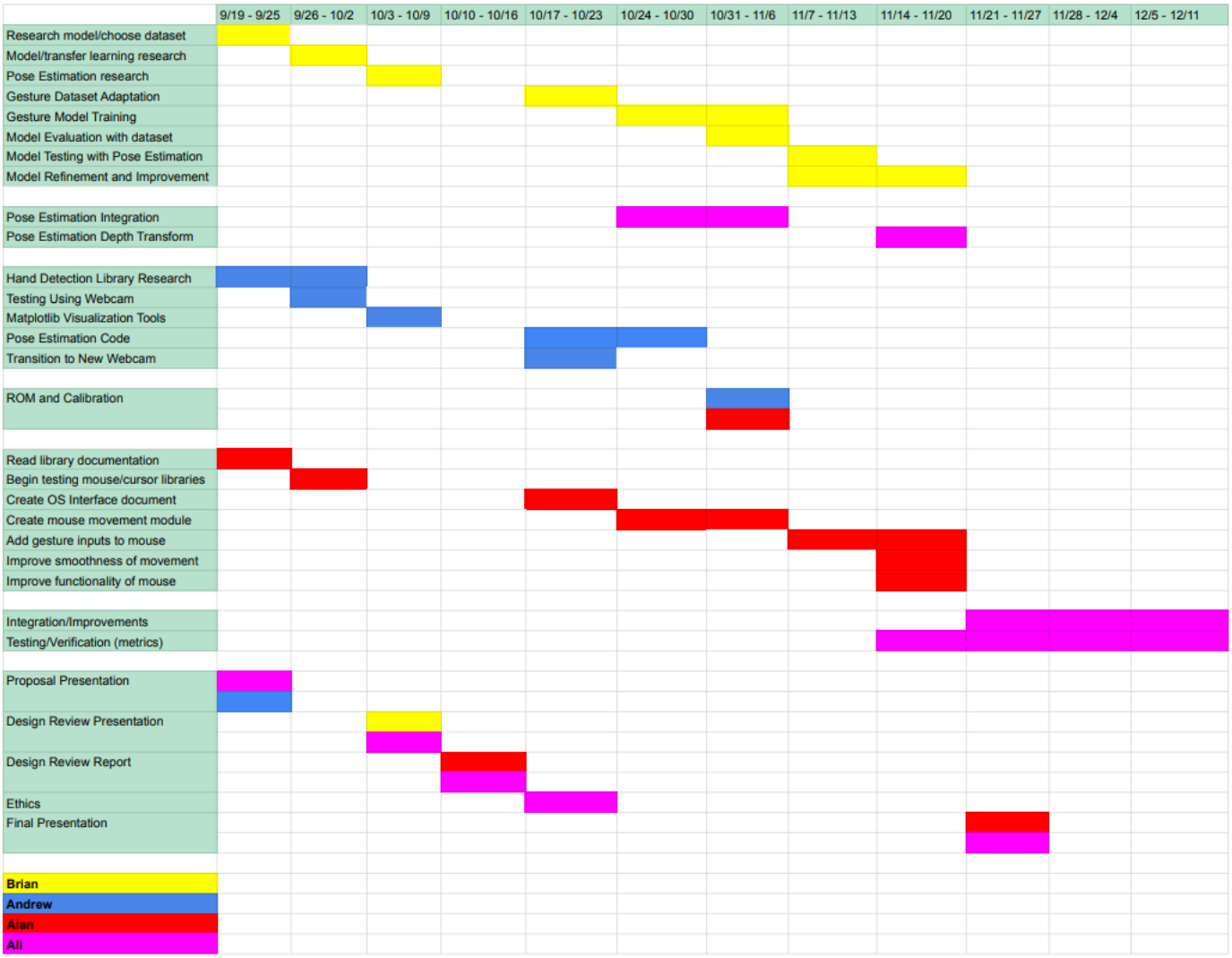

Fig. 9. Detailed schedule