# Advancing Assistive Mouse and Keyboard Control Scheme Using Two-Handed Gesture Recognition for Human-Computer Interaction

*Shawn Zhao*
*Brian A. Barsky*

Electrical Engineering and Computer Sciences
University of California, Berkeley

Technical Report No. UCB/EECS-2024-179
http://www2.eecs.berkeley.edu/Pubs/TechRpts/2024/EECS-2024-179.html

August 12, 2024

Acknowledgement

I would like to acknowledge and thank the earlier contributors to this project, including Michael Qi, Patricia Ouyang, Xinying Hu, Fang Hu, Yang Huang, Guanghan Pan, Juntao Peng, and Bill Tong for their setup and development of the codebase before I came to take over the project. I would like to thank my current research collaborators, Yash Pansari and Niall Mandall, for their continued help throughout the year. I would like to especially thank Prof. Brian Barsky for his continued guidance and support throughout the research process, and Prof. Paulos for being a second reader in reviewing this thesis. I would also like to thank my family and friends for their continued support.

**Advancing Assistive Mouse and Keyboard Control Scheme Using Two-Handed Gesture Recognition for Human-Computer Interaction**

by Shawn Zhao

**Research Project**

Submitted to the Department of Electrical Engineering and Computer Sciences, University of California at Berkeley, in partial satisfaction of the requirements for the degree of **Master of Science, Plan II**.

Approval for the Report and Comprehensive Examination:

**Committee:**

Brian A. Barsky

Professor Brian Barsky
Research Advisor

12 August 2024

(Date)

* * * * * * *

Professor Eric Paulos
Second Reader

9 Aug 2024

(Date)

# Advancing Assistive Mouse and Keyboard Control Scheme Using Two-Handed Gesture Recognition for Human-Computer Interaction

by

Shawn Zhao

A thesis submitted in partial satisfaction of the

requirements for the degree of

Master of Science in Engineering

in

Electrical Engineering and Computer Science

in the

Graduate Division

of the

University of California, Berkeley

Spring 2024

Abstract

Advancing Assistive Mouse and Keyboard Control Scheme Using Two-Handed Gesture Recognition for Human-Computer Interaction

by

Shawn Zhao

Master of Science in Engineering in Electrical Engineering and Computer Science

University of California, Berkeley

Professor Brian Barsky

Upon its introduction in 1968, the computer mouse arrived as a novel input device for controlling a computer pointer on a display. Even now, over 50 years later, the mouse and keyboard remain the ubiquitous combination of input devices for operating a computer. In those years, computers have undergone drastic changes to become increasingly more advanced and convenient for users, and while mice and keyboards themselves have undergone significant advancements in design, comfort, and sensing, the basic mechanical method for their usage as input devices remains the same. With the increasing prevalence of computers, there has likewise been increasing research into alternative input methods, including gestures. While gesture-based cursor control and keyboard input have been previously explored, we leverage a common hardware peripheral, the camera, and combine its capabilities with vision processing techniques to create a more natural handed input system.

We further develop an existing two-handed gesture control system to replace the functionality of a traditional mouse and keyboard for those who may find difficulty in operating those devices. The system uses a computer's webcam to detect hand movements and recognize gestures that correspond to cursor movements and other mouse and keyboard actions without the need for additional hardware. We add further features such as in-air writing and multi-frame gestures. Through evaluation of the accuracy of our gesture and writing recognition after the incorporation of trained datasets, we confirm that our system is a functional alternative control system to the traditional mouse-and-keyboard input methods, and have created a bundled executable for users to download and further test.

# Contents

# List of Figures

# List of Tables

## Acknowledgments

I would like to acknowledge and thank the earlier contributors to this project, including Michael Qi, Patricia Ouyang, Xinying Hu, Fang Hu, Yang Huang, Guanghan Pan, Juntao Peng, and Bill Tong for their setup and development of the codebase before I came to take over the project. I specifically want to thank Michael for his contribution to determining the palm's centroid, Xingying for implementing the affine transformation method for gesture recognition and browser control actions, and Patricia for setting up the initial two-handed gesture mapping scheme and for providing me with a well-established codebase and resources for me to finish the full implementation of cursor control. I would like to thank my current research collaborators, Yash Pansari and Niall Mandall, for their continued help throughout the year. Yash specifically implemented the dragging functionality, and Niall implemented the eye-tracking integration that could be used in future work. I would like to especially thank Prof. Brian Barsky for his continued guidance and support throughout the research process, and Prof. Paulos for being a second reader in reviewing this thesis. I would also like to thank my family and friends for their continued support.

# Chapter 1

# Introduction

## 1.1 Motivation

In 2020, an estimated 10 million people suffered from essential tremors and an additional 1 million suffered from Parkinson's Disease, with those numbers growing to 25 million people for essential tremors and 10 million for Parkinson's Disease worldwide [32, 36]. For people with such disabilities that limit their fine motor control, using a traditional mouse and keyboard as input methods to a computer can be challenging due to the precise micro-movements with their hands that using such hardware demands. Even for those without such disabilities, prolonged use of a mouse and keyboard for extended periods can lead to hand strain and the development of "technological diseases" such as carpal tunnel syndrome [34]. According to the Canadian Centre for Occupational Health and Safety, there are two main reasons that regular use of a mouse can be hazardous. The first reason is that the stationary positions and small and repetitive movements associated with using a computer mouse can lead to pain, discomfort, and Workplace Musculoskeletal Disorders (WMSDs) from straining the same small muscles over and over again for prolonged periods of time [24]. The second reason is that many office and personal workstations have limited space, so the mouse is placed outwards and forwards at a location that is uncomfortable to reach. To use the mouse, the person has to lean forward and reach outward and forward and remain unsupported in that posture while they use the mouse, which can cause strain on the upper back, shoulder, and neck muscles [24]. Thus, we have established that while the mouse and keyboard are the most popular methods of computer control, they are far from perfect and

not suitable for everyone.

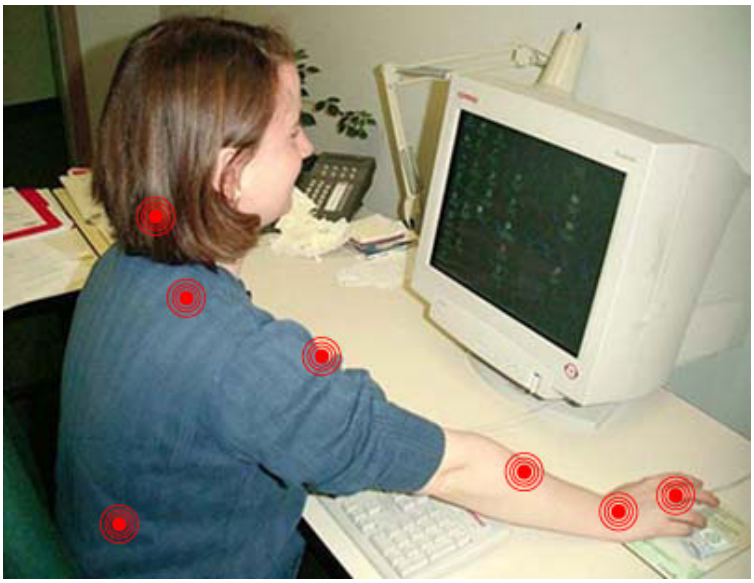


Figure 1.1: All the muscles and joints that could be strained or damaged by prolonged use of a mouse. Credit: [24]

Several ideas have been proposed from a hardware perspective, but they each pose their own problem. Alternative input devices exist, but they often require cumbersome and expensive machines. Ergonomic mice and keyboards are offered, and from a software perspective, there has been a commercial exploration into alternative input methods such as gesture recognition and eye tracking [23, 38, 11]. However, the specialized hardware and software can be expensive for the average user, often costing hundreds or thousands of dollars [1].

We develop a camera-based assistive mouse and keyboard using a two-handed gesture recognition that maps specific combinations of gestures to different mouse and keyboard inputs. Our gesture recognition system provides an alternative method of human-computer interaction for users who struggle to use traditional mice and keyboards and aims to address both of the aforementioned hazards associated with using a mouse that could lead to muscle strain or WMSDs. When users perform gestures rather than moving the mouse, they eliminate the stationary postures adopted by their arms and wrists and use more dynamic gestures that do not involve as much precision and micromovement, alleviating that hazard. Additionally, using the computer's camera, which is centered on the user, allows the user to

use the space close to them and around their natural sitting posture to make their gestures. This makes it so that users no longer have to reach to uncomfortable positions to use their mouse, reducing their risk of straining their muscles and developing WMSDs.

One major difference between our system and current approaches for gesture-based cursor control is the choice to use the palm centroid as the reference point for the cursor as opposed to the tip of an outstretched finger, which is what many similar systems use [29, 15]. The advantage of utilizing this point of reference is that the palm centroid is anchored by six different hand key points as opposed to only one using the tip of an outstretched finger, which leads to more stability in cursor control with a more natural hand gesture. We add further stabilization by implementing a 1€ filter, and we hope that both of these stabilization techniques improve performance and operability for those with tremors as described above. Another major difference between our system and current approaches is that current approaches use neural network classifiers as their gesture selection method, while we use affine alignment with templates as our gesture selection method. The usage of templates allows users to customize their range of gestures to be recognized when using the system, which can be particularly helpful to users who prefer specific gestures due to having difficulties in their fine motor control.

## 1.2 Overview

This project was a part of the larger Assistive Technology for Cursor Control project, whose overall goal is to enable users to replicate the ability to interact with a computer with a mouse and keyboard solely through hand gestures in view of the target computer's camera. We establish a two-handed control scheme, where different combinations of preset left and right-hand gestures map to different mouse and keyboard functionality. We separate the system into a mouse mode and keyboard mode that the user can switch between with a simple gesture. In mouse mode, users can perform general mouse actions such as moving the cursor and clicking, scrolling in the four cardinal directions, adjusting the volume, and zooming in and out in a browser window. In keyboard mode, users can draw letters to input them into a selected text box, or hold up gestures to input single digits.

## 1.3 Related Work

### Mouse and Keyboard Alternatives

Existing hardware alternatives to the traditional mouse include vertical mice, trackballs, trackpads, and joysticks. Trackballs in particular are used as assistive technology input devices for those with mobility impairments, as the control surface of trackballs is easier to manipulate and users only have to move their thumb or finger relative to their hand, instead of their whole hand [12]. Trackballs come in thumb-operated and finger-operated variants, where the thumb-operated variant places the ball towards the side of the device and the finger-operated variant places the ball in the middle of the device. Some people report difficulty holding a mouse still while clicking or double-clicking, whereas the trackball allows those people to keep the cursor still when pressing buttons by removing their finger from the ball [12]. One issue with trackballs as assistive technology is that while finger-operated trackballs are symmetric and thus usable by both hands, thumb-operated trackballs are generally not available in left-handed configurations due to lack of demand [12].


(a) Finger-operated trackball (b) Thumb-operated trackball

Figure 1.2: Variants of Trackball for Modern Use. Credit: [18]

Likewise, hardware alternatives for the traditional keyboard include one-handed keyboards, expanded keyboards, and split keyboards [16]. Each of these alternatives allows the user to position their hands differently than how they would have to be positioned on

a traditional keyboard, reducing the potential of damaging strain due to prolonged usage by enabling more comfortable hand and wrist arrangements [28]. Ergonomic keyboards are designed to improve the posture of the arms, wrists, and hands while operating a computer, allowing for postures that put less strain on the arms and reducing the risk of developing RSI and carpal tunnel syndrome [6]. For those with tremors, there are specially designed "big key" keyboards with extra large keys. The keys provide a larger target and are more spaced out to prevent accidental or incorrect keystrokes [6].

(a) Split Ergonomic Keyboard

(b) "Big Key" keyboard, with finger for scale

Figure 1.3: Various Alternative Keyboards. Credit: [6, 14]

## Alternative Control Schemes

From a software perspective, one example of an alternative control scheme for cursor control that bypasses the usage of hands altogether is eye tracking. Different eye tracking systems have been proposed to control the cursor using gaze detection based on where on the screen the user is looking [23, 38]. Likewise, head and face tracking is an adjacent alternative for cursor movement and functionality, where head position in a bounded reference frame would be used to determine cursor position, while cursor functions such as clicking would be triggered through the detection of blinking or other facial muscle movements [11].

However, there are a few drawbacks to these alternative control schemes. While eye-tracking software such as Precision Gaze Mouse and Gaze Pointer are readily available for users to download, they recommend expensive equipment to operate at a competent level

[26, 10]. Additionally, the head and eyes perform much more involuntary movements than the hands on average, so precise control using those methods may be difficult. The unique number and types of movements that can be made by the head and eyes are also much less than the movements that can be made by the hands, which limits the number of available actions in those cursor control schemes. The wide variety of possible hand gestures may remedy this limitation in our system.

## Existing Gesture Recognition Cursor Control Designs

At present, available cursor control schemes that use hand gesture detection and recognition can mainly be split into two different types of designs: machine learning approaches or hardware approaches. Machine learning approaches involve deep learning or training convolutional neural networks to classify hand gestures. However, the large number of available gestures becomes a detriment in this approach: the large number of potential gestures necessitates a classifier be trained across all of these gestures as classes, which may impact the accuracy of properly identifying any specific one gesture. However, a small number of available gestures would restrict the number of recognizable gestures, which would in turn restrict the functionality of the cursor control scheme. Regardless of balancing how many gestures are available in our classifier if we were to construct one to alleviate the aforementioned issues, it would be difficult to modify the classifier after training has been completed to add new gestures into the system.

These problems are also reflected in our decision not to build our own gesture classifier using convolutional neural networks to directly recognize gestures that appear in an image frame, as a good classifier would require a huge dataset of images of hand gestures covering every point on the screen and at different distances from the camera or zooms. Building this huge dataset and maintaining it to train a convolutional neural network classifier is out of the scope of this project and our group's manpower. Instead, we use existing machine learning frameworks such as Google's MediaPipe hand detection module [8] to identify relative key points, as constructing our own classifier would essentially be reconstructing this module from scratch. Once the key points have been identified, we apply image processing techniques, namely the affine alignment model, to associate the key point arrangements with specific

gesture templates. This method takes into account hand position on the screen, hand size, and distance from the hand to the camera, which are all factors that would have necessitated their own batches of images in the training dataset. This pipeline allows our system to avoid requiring large datasets that would be necessary to build a high-accuracy gesture classifier for effective implementation.

Hardware approaches require additional sets of cameras or wearable hardware in order to collect additional data from the user. One such proposed system is [4] which requires additional video cameras that enable the scheme to measure depth information needed for its gesture recognition system. Since the goal of our system is that it requires no additional hardware besides the computer's camera, we do not take this approach. This enables our system to be more lightweight and accessible to a wider variety of users.

## 1.4 Previous Work

While the previous section discussed available industry alternative cursor control schemes, this section details previous academic designs to cursor control systems and how our developed system differs from the previous work. Two systems particularly close to ours in design have been developed by [17] and [30].

In [17], the authors develop a system that utilizes a video capture and image processing approach to identify and track a designated color marker's motion and convert it to cursor control on the screen. The system uses OpenCV for video capture and reads the input frame-by-frame. It then smooths the image to remove noise and creates a region of interest (ROI) on the display window where the marker will be tracked inside of. Outside this region, any motion will not be tracked, and anything that is not the designated color will also not be tracked. The color marker is isolated using Canny edge detection and contour area maximization. The tracking of the color marker's motion within the ROI is finally converted to the cursor's position on the screen.

Our system follows a similar pipeline but uses further advancements in many steps to improve the system. Our system also uses OpenCV for video capture and reads in the input on a frame-by-frame basis, but is not limited to using a designated color to detect and detects

the user's hand gestures instead. In this regard, our system is much more flexible and enables many more functionalities. Furthermore, our system uses MediaPipe to detect the hands and uses a 1€ filter to smooth any sudden movements instead of smoothing the images from the frames together, which decreases the lag in reflecting cursor position. Our system also contains many more features that [17] does not, including the ability to type. Further details about design choices mentioned briefly here will be expanded on in subsequent sections.

In [30], the authors develop a cursor control system that detects hand position and gesture recognition and converts it to cursor position and action. It utilizes a pipeline split into an OpenCV camera module, a hand detection module using Haar cascade edge detection and background removal, and an interface module that takes the recognized gesture and position and performs cursor movements and actions depending on the detected hand location and gesture. Furthermore, the system contains an on-screen keyboard shown in the display window that the user can point to and gesture in order to type one letter at a time.

This system is closer to ours than [17], as our system is also based on hand position and gesture recognition, and comes with a typing functionality. However, some elements of our pipelines use different methods: for our hand detection, we use Google's MediaPipe module, and for gesture recognition, we use an affine alignment algorithm. [30] enables the entire display window to be used as an identifiable hand location for cursor position, but our system uses a bounding box that allows the user to keep their hand in relatively comfortable positions and still reach all points on the screen while increasing the accuracy of the gesture recognition by keeping the hand centered and thus a more complete gesture captured. Additionally, [30] enables the users to type by clicking on an on-screen keyboard, but our system's typing mode has users write symbols in the air to type letters and numbers. Further details about terms and design choices mentioned briefly here will be expanded on in subsequent sections.

# Chapter 2

# System Design

## 2.1 Control Pipeline

Our two-handed gesture recognition system receives video data from a computer's default camera using OpenCV's video capture class. The video stream is read in frame-by-frame, and if the frame is read successfully (not an empty frame reading which would indicate that an error occurred with the video stream), the actual frame data is converted to a NumPy array image representing the pixel values. Detection of both hands is attempted on each frame image, and if detected, each hand is overlayed with a skeleton connecting the hand's key points. Hand detection and hand key point determination are both performed using Google's MediaPipe framework [8].

The arrangement of key points is input into the gesture recognition system to determine the current hand gestures performed by the user. The gesture recognition system performs template matching by generating an affine alignment matrix and recognition score between the detected key points and a set of templates, one for each gesture, and on both hands if both are detected. The classified hand gestures are finally used as input to the control scheme in order to perform actions on the computer. The left hand's gesture controls the general action (click, adjust volume, scroll, etc.) while the right hand's gesture performs finer control options for the action specified by the left hand (left or right click, increase or decrease volume, scroll in a direction, etc.). If the user changes either hand's gesture, the current action will stop and the newly specified action will be executed, with the exception

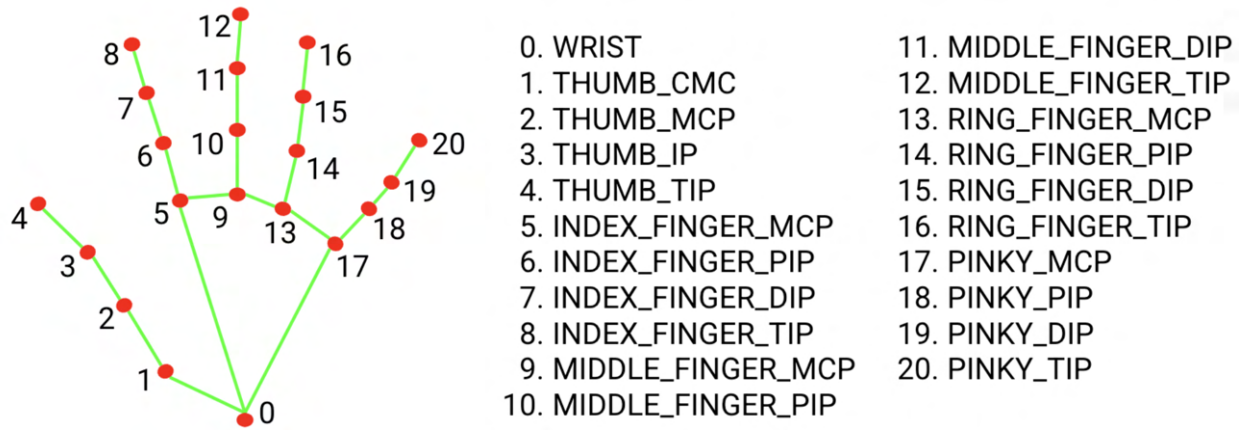



Figure 2.1: MediaPipe hand key points. Credit: [8]

of dragging. As such, the control pipeline of the system follows the Figure 2.2 for every frame:

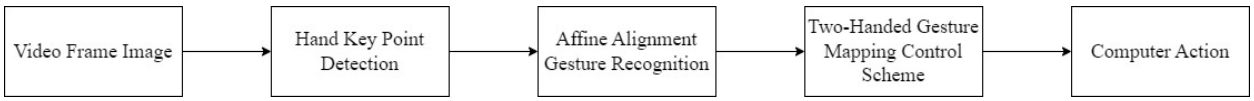


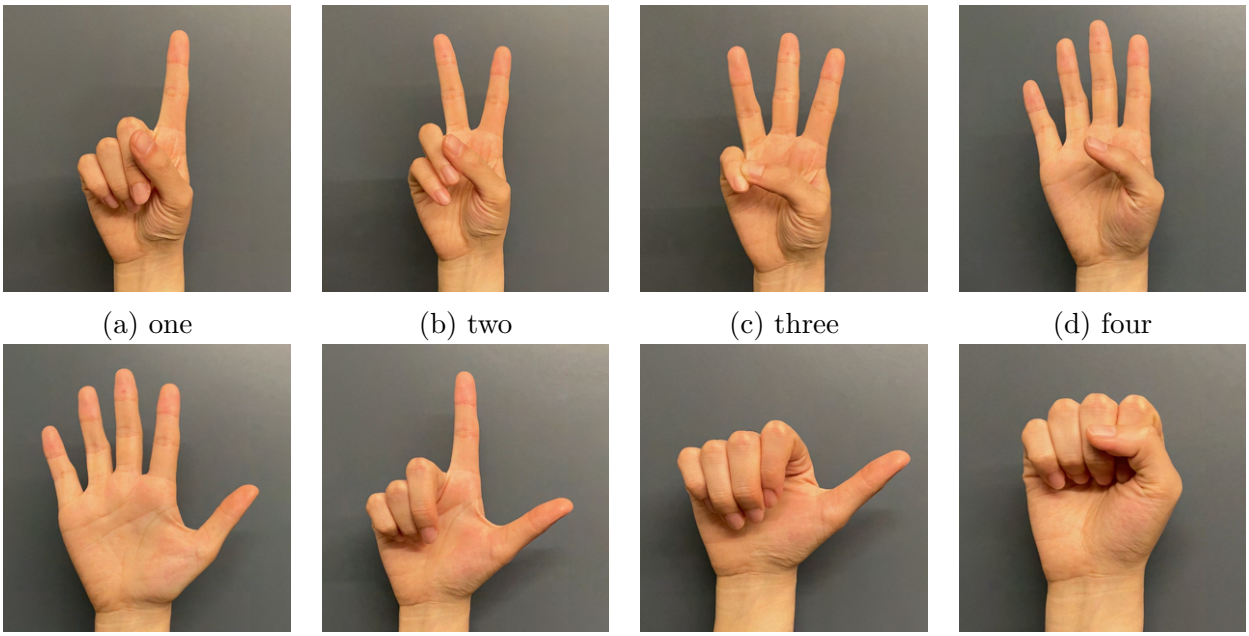

Figure 2.2: The control sequence for the system at every frame

## 2.2 Gesture Recognition

### Template Matching

An affine transformation is a function that can perform translations and linear transformations such as rotation, reflection, and scaling. Given a minimum of three noncolinear points in 2-dimensional space and a set of correspondingly transformed points, a unique affine transformation matrix can be determined that performs that transformation. Likewise, for two sets of at least three noncolinear points in 2-dimensional space, there exists a unique optimal affine transformation matrix such that, when applied to one set of points with the goal of transforming them into the other set of points, maximizes the norm of the difference between target set of points and the transformed set of points.

There are eight currently supported gestures: "one", "two", "three", "four", "five", "arrow", "thumb", and "fist", and these are differentiated into their left-hand and right-hand versions. These gestures were chosen to be simple yet clearly distinguishable. That way, all the gestures would be easily recognizable and there would be little chance of performing an unintended gesture when aiming to perform a different one. For each gesture, we save an image of the gesture that serves as a template to compare users' gestures to, and likewise save an arrangement of where each hand's key points are for each of those images. These images are saved with an aspect ratio of 16:9, which is a common aspect ratio for many webcams, computer monitors, laptop screens, and is a middle ground between 4:3 and 16:10, the other two common aspect ratios.

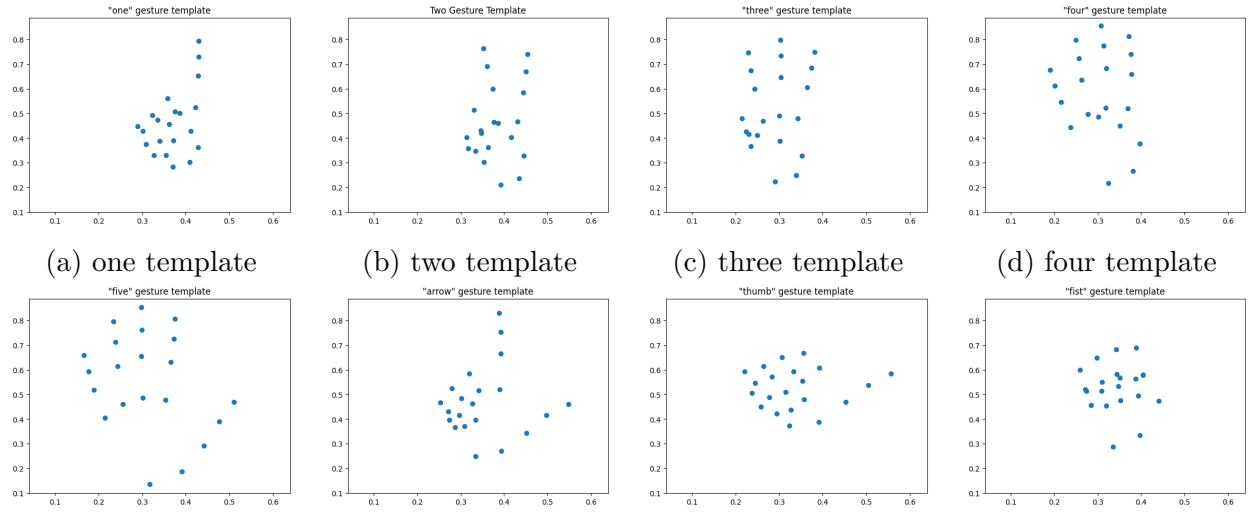


(a) one (b) two (c) three (d) four

(e) five (f) arrow (g) thumb (h) fist

Figure 2.3: The eight gestures supported by our system. The images were cropped down from their original aspect ratio.

The affine alignment method is originally proposed in [19] and further adapted to our specific system by [13]. To classify a user's intended gesture, we calculate an affine alignment matrix $A$ between the arrangement of the user's hand's key points $U$ and a template of key points $T$ for each available template. The optimal affine alignment matrix $A*$ for a designated




(a) one template (b) two template (c) three template (d) four template

(e) five template (f) arrow template (g) thumb template (h) fist template

Figure 2.4: The key points of the eight gestures. They were plotted using the same dimensions to show their relative sizes.

key point template is the matrix that transforms the user's gestures closest to that target key point templates, or in other words, results in the minimal difference between $A \cdot U\ T$. This process reduces to solving the following problem:

$$A^* = \arg\max_A ||A \cdot U - T||$$

Each affine transformation matrix is then reapplied to the original arrangement of user gesture key points. The transformed points will be closer to the template key points for the correct gesture, and relatively farther away for incorrect gestures. Therefore, each arrangement of transformed points is then compared to its corresponding template's key points and scored based on the error between them using the following formula:

$$S = \exp\left(-||A^* \cdot U - T||\right)$$

The gesture with the highest score $S$ across all $T$ is chosen as the prediction. However, if no gesture results in a score above 0.5, then a recognized gesture of "None" is returned. While this feature may seem like a limitation and result in instances where gestures fail to be recognized, we decided to implement this feature to ensure that the system only acts on

gestures that the user performs with intent, and does not just act on arbitrary actions the user may perform without even meaning to.

Figure 2.5 shows the key points of a sample "five" gesture made. Figure 2.6 shows those input key points aligned with each of the eight templates, with their scores shown below. From a qualitative test, we observe that the the points almost perfectly line up with that of the "five" template (as intended), and are close to the "four" template and a bit farther away to the "three" template as well, but do not match the other templates well at all. This observation is reflected in the alignment score: since a "five" gesture was made, it almost perfectly matches the "five" template, resulting in a score of near 1. The "four" alignment score is high as well, and the "three" alignment score is lower but still noticeably higher than the score of all the other alignments. The alignment score behaviors as expected by our qualitative observations, as the gestures for "four" and "three" are closest to the gesture for "five", due to them all involving sticking the majority of the fingers straight up.

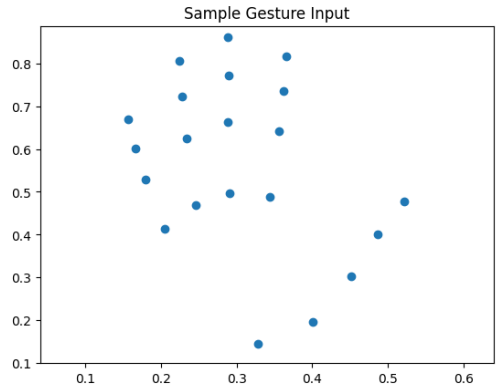
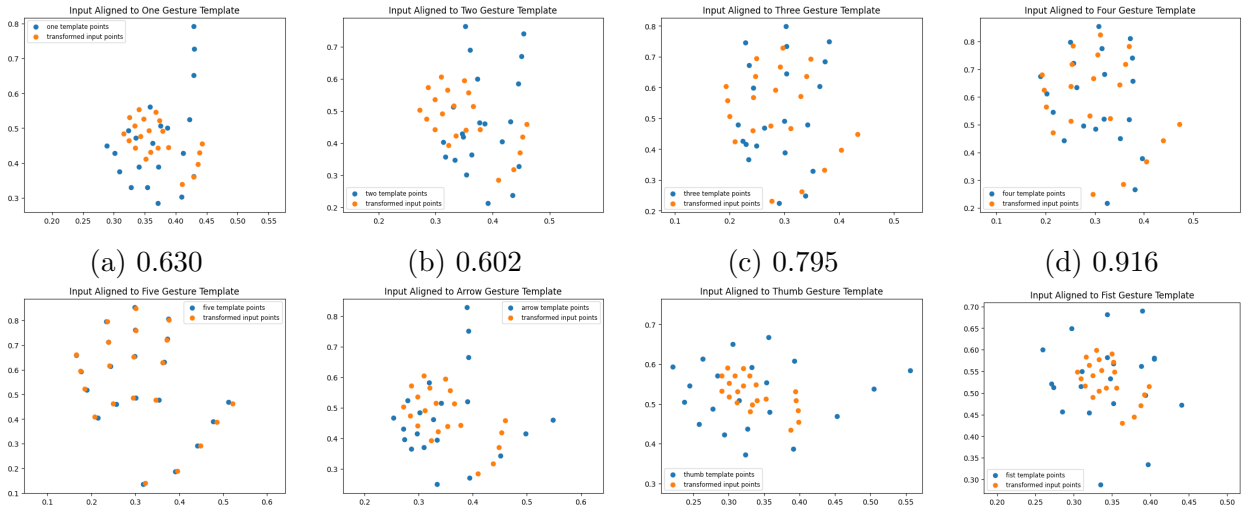



Figure 2.5: A sample "five" gesture to as an example to align

## Handedness

To determine whether the left hand or right hand is performing a specific gesture, we use MediaPipe's ability to determine whether or not a set of hand key points represents a left hand or a right hand. In a previous implementation, we only saved the templates for the right hand version of the gesture, and flipped the hand key points across the x-axis for the hand that MediaPipe determined was the left hand when performing affine alignment




(a) 0.630 (b) 0.602 (c) 0.795 (d) 0.916

(e) 0.998 (f) 0.602 (g) 0.617 (h) 0.670

Figure 2.6: The key points of the sample affinely aligned to the key points of each of the eight gestures. The caption of each image represents its score.

gesture recognition. We found that a faster implementation was to save a flipped version of the template key points as a left-handed version of the key points, and those key points specifically would be used when MediaPipe determined the hand was a left hand when performing affine alignment gesture recognition. In this way, the set of gestures is supported by both hands with the same accuracy and functionality without being limited by the user's handedness.

## Occlusion

The entire system is based on MediaPipe's identification of hand key points, so this method may face potential problems if hand key points are not identified properly. One potential problem that could hinder the proper identification of hand key points is the occlusion of the hands. According to MediaPipe's official documentation, the module should be able to interpolate occluded key points if enough of the hand is visible [8]. However, during our experience, we found that the interpolation could lead to deformed hand structures based on the direction of the occlusion. Furthermore, upon overlapping both hands, only the hand in front will show. The following images show how occlusion can lead to deformed hand key

points or the lack of identified key points altogether.

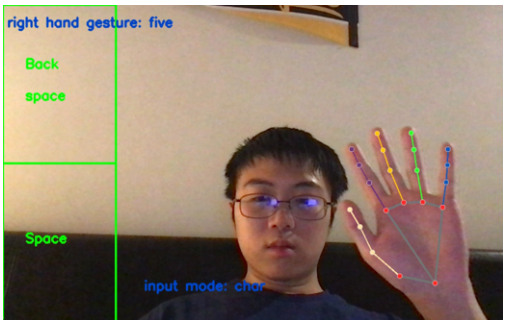
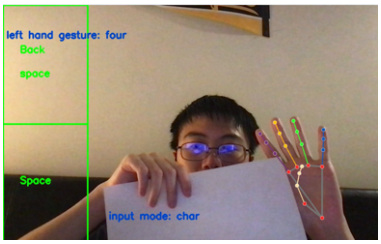
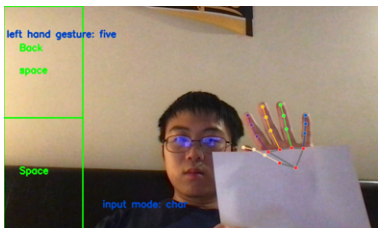



Figure 2.7: Not occluded, correctly recognized as Right Hand Gesture: Five




(a) Occluded, incorrectly recognized as Right Hand Gesture: Four




(b) Occluded, incorrectly recognized as Left Hand Gesture: Five


(c) Occluded, incorrectly recognized as Right Hand Gesture: Fist

Figure 2.8: The effects of occlusion on the identification of Right Hand Gesture: Five. The identified gesture is in the top left of each image

From observing figure 2.8, we can see that depending on the portions of the hands that are occluded, the key points are deformed to be "squished" into the region of the hand that is not occluded. This leads to incorrect hand key points being identified and subsequently leads to incorrect gestures being recognized since the key points do not match up with the expected templates. The correct hand gesture should be "Right Hand Gesture: Five", and while the gesture being performed does not change, the gesture being detected does become incorrect based on which portion of the hand was occluded. This finding also becomes the basis for our avoidance of any hand gestures toward the edges of the webcam's display, as hand parts could be off the screen and lead to similar "squishing" behavior.


(a) Not occluded, correctly recognized as Right Hand Gesture: Five and Left Hand Gesture: Five


(b) Self-occluded, only Right Hand Gesture: Five recognized


(c) Self-occluded, no hand key points identified

Figure 2.9: The effects of self-occlusion with one's own hands on the identification of Right Hand Gesture: Five and Left Hand Gesture: Five. The identified gestures are in the top left of each image

Similarly, from observing figure 2.9, we can see that self-occlusion of the user's hand by the user's other hand leads to the system failing to identify both hands properly, and in some cases fails to identify hands at all. This finding forces us to consider self-occlusion when designing our system and selecting gestures, as it is a natural motion and may happen unintentionally. This finding also prevents the current system from using any two-handed combination gesture (such as clasping both hands together), as it would not be able to recognize the key points of both hands.

## Lighting and Skin Tone

Another potential problem that could influence the identification of hand key points is the lighting of the environment and the skin tone of the user's hands. If the user's environment was too dark or was under a tinted light, the hand key points might not be identified properly. We tested the key point identification capabilities of the module in a low-light environment, where the room was enclosed without lighting except for the light coming from the laptop's screen at minimum brightness. The module was still able to accurately identify the hand key points in this low-light setting, so we claim with confidence that this gesture recognition method is robust to low-light environments. In this low-light environment, the hands were dark enough to be of the same color as the background, so we also claim that this system is robust to the skin tone of the hand matching the background to a visible degree.

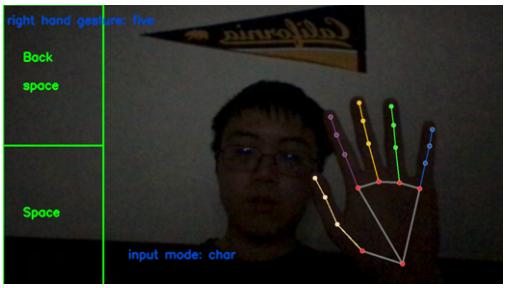



Figure 2.10: Successful gesture recognition under low-light conditions

(a) A hand color for which MediaPipe properly identifies hands.

(b) A hand color for which MediaPipe does not properly identify hands.

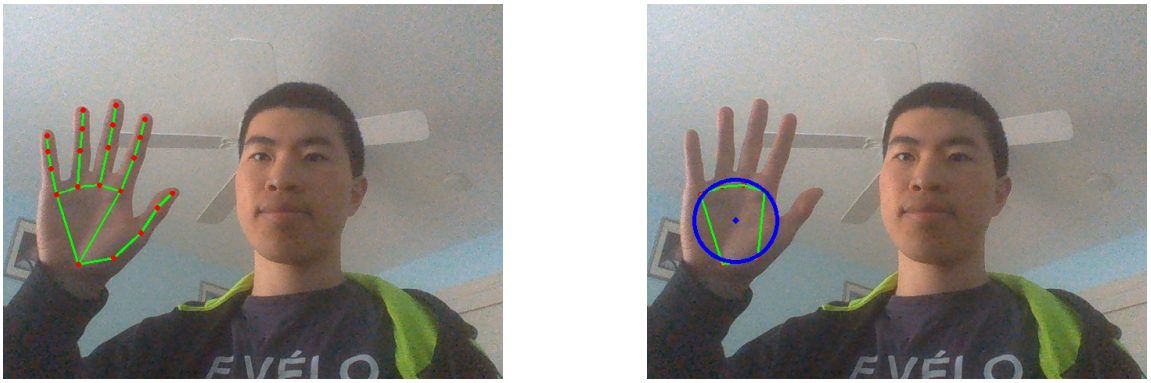
(c) A hand color for which MediaPipe does not properly identify hands.

Figure 2.11: Modifying the hues of a gloves template image to both find the range of MediaPipe's skin tone recognition and find colors where it no longer works. Shown are three examples, two of colors that do not fall within the range and one that does.

However, we found that taking the hue of a user's hands too far away from the human range of skin colors led to the hands not being recognized at all. We made this finding by taking a template image of solid-color gloves that had their properly detected, and offsetting their hue at regular intervals. Figure 2.11 shows examples of colored hands where MediaPipe no longer detects them gloves as hands. Thus, while we claim that human skin tones fall within the range of colors where key point identification behaves properly, we have found another potential limitation of the system: users wearing specific colored gloves, users working in tinted environments, or users tinted cameras may be unable to properly use this system due to the MediaPipe module not identifying hand key points at all.

## Dragging

One multi-frame gesture that was implemented was the ability to hold click and drag. Whether or not the user has begun dragging is determined by whether the user's palm has moved a certain minimum distance since the last frame while maintaining the same gesture. The location of the palm is calculated by constructing a polygon from key points surrounding the palm and determining its centroid. Figure 2.12 shows the process by which the palm centroid is calculated.



(a) Hand Keypoints (b) Palm Centroid

Figure 2.12: Calculating palm centroid from Mediapipe hand keypoints. Credit: [27]

The gesture recognized and the location of the palm centroid at the previous frame are stored, and the distance between the palm centroid at the current frame and the previous frame determines the distance traveled. If the distance traveled is above a certain threshold and the gestures at both the previous frame and the current frame correspond to "left click", then the system will enter the dragging state, and will remain in the dragging state until the "left click" gesture is no longer held. In this state, the cursor will follow the palm centroid's location, and the system creates a drag buffer and records the path between the start point of the drag until the point where the dragging state is exited. There are two ways to exit the dragging state: cancellation and confirmation. To cancel the drag, the user simply stops performing the "left click" gesture for a defined threshold of time. The drag will not be executed, and the drag path buffer will be cleared. To confirm the drag, the user performs the "double click" gesture. This will execute the drag, and the system will perform the

action of starting the cursor at the start point, holding down left click, moving the cursor to the endpoint, and releasing left click, as if it were the "click and drag" action performed by a traditional mouse.

This approach was chosen to balance intention with ease of use. We chose to implement the drag as a stored buffer that would then execute after the action was confirmed, rather than executing the drag directly with the movement of the cursor, because we did not want users to accidentally execute a drag without meaning to. Especially for those with tremors, without the implementation of the buffer system, a "click" action could very easily be recognized as a short drag action, and if executed without a confirmation failsafe, could make using the system difficult.

## 2.3 In-air Writing

### Previous Implementations and Changes

The initial implementations of the keyboard control scheme mapped each letter to a combination of left and right-hand gestures from one to five, such that each of the 25 letters from $a$ to $y$ maps to a different combination of left-hand gestures 1-5 and right-hand gestures 1-5 in order For instance, a left-hand gesture of "one" and a right-hand gesture of "one" maps to $a$, a left-hand gesture of "one" and a right-hand gesture of "two" maps to $b$, etc. As there were only 25 combinations of hand gestures, $z$ was mapped to a left-hand gesture of "five" and a right-hand gesture of "None". The premise behind this implementation's operation is similar to that of the chorded keyboard, which involves users enter characters or commands into a terminal by pressing combinations of several keys together at once, where the combination of keys equates to a combination of gestures in this case.

However, this implementation had several key limitations. First of all, it is both unintuitive and almost impossible to memorize which hand gestures correspond to which letters without extensive use, so the correspondence table of gestures must always be displayed in case users are not familiar with the system or do not remember. The need to constantly reference the table while performing unintuitive gestures may slow down typing significantly. Frequent users would likely be able to recall the specific combination for any desired letter

or system, but if we were to analogize this system's learning curve to that of a chorded keyboard, it would take them hundreds of hours [20]. Second of all, the number of mappable letters or symbols under this scheme is extremely limited, making it very difficult to extend to capital letters or special characters.

We implement a new keyboard control scheme where the user enters drawing mode and inputs into a selected text area by enabling the user to write the letter with their finger in the air in view of the camera. In drawing mode, there are two sub-modes: one for writing letters and one for writing digits. This separation was chosen due to the similarities between certain letters and numbers, particularly $O$ and 0 and 1 and $I$, which could have led to the system having difficulty differentiating those specific characters when their respective drawings are inputted. In our first approach, we neglected this potential problem and trained the model on the entire dataset; this led to poor performance that was vastly improved by splitting the models.

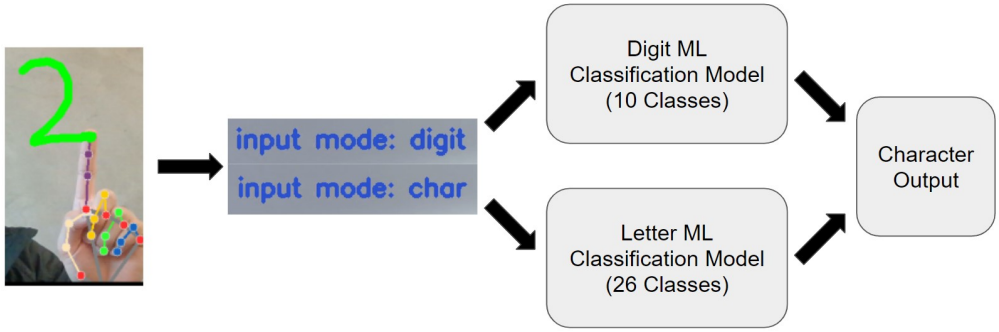



Figure 2.13: The pipeline for our two model approach that splits the classification of letters and numbers

## Training Data

To recognize the written in-air alphanumeric characters, two neural network models (one for digits, one for letters) were trained on the EMNIST dataset, which stands for the Extended MNIST. The original MNIST dataset is a popular benchmark dataset in the field of machine learning and computer vision and contains 70000 handwritten digits, normalized to $28 \times 28$ pixels and grayscaled [7]. The EMNIST dataset extends the original MNIST dataset by

not only digits, but also handwritten lowercase and uppercase letters, for a total of 814255 alphanumeric images, which are also normalized to $28 \times 28$ pixels and grayscaled [5]. Each image in the dataset is labeled with what the handwritten symbol should be among uppercase letters $A - Z$, lowercase letters $a - z$, and numbers $0 - 9$, for a total of 62 classes.

However, one issue with using the base dataset as a whole is that it is unbalanced. A balanced dataset is a dataset where each class is represented equally. There are many advantages to using a balanced dataset when training machine learning models: they can reduce biases caused by the model favoring classes with more samples, they can help the model generalize better to new data as models often fail to classify new data that falls within minority classes, and they reduce overfitting by preventing the model from memorizing the majority class rather than meaningful true patterns. Additionally, because our implementation uses different neural network models for digits and letters, we cannot use the entire dataset as it is, since it contains a mixture of both types of data.

To resolve these issues, we use two dataset splits provided by the EMNIST dataset: EMNIST MNIST and EMNIST Letters. The EMNIST MNIST dataset contains the original MNIST dataset of 70000 images of digits split across 10 balanced classes, while the EMNIST Letters dataset merges a balanced set of uppercase and lowercase letters into a single 26-class dataset. The limitation that comes with using these specific splits is that our trained model will not classify uppercase and lowercase characters of the same letter differently. However, that is a limitation we chose to accept to increase the confidence that training our models on these datasets will be successful.

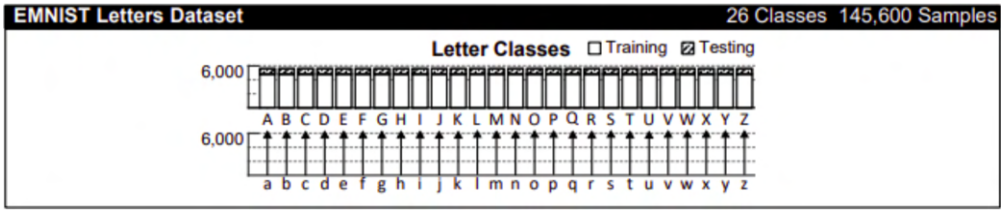



Figure 2.14: A visual representation of the EMNIST Letters dataset showing the balanced classes, the mix of uppercase and lowercase letters, and the train/test split. Credit: [5]

## Models

For the digit classifier, we implemented a shallow neural network model consisting of 3 linear layers with ReLU activation functions. The input dimension of the first layer is $28 \times 28$, representing the dimensions of the images that are inputted, and has an output size of 100. The second layer has input and output sizes of 100. The final layer has an input size of 100 and an output size of 10, representing the 10 classes of digits the drawings can be classified into. Figure 2.15 depicts the architecture used for this classifier.

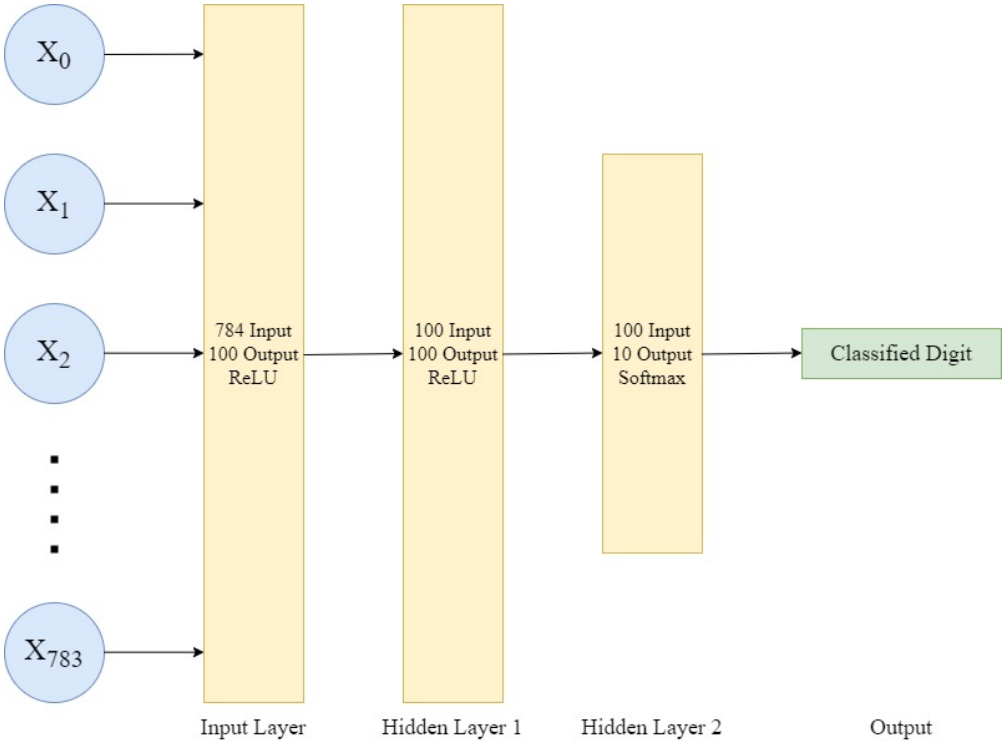



Figure 2.15: The neural network model for the digit classifier.

For the character classifier, we implemented a convolutional neural network model inspired by [35]. It consists of 4 convolutional layers of input sizes 1, 32, 64, and 128, followed by a linear layer that maps it back down to the 26 different classes, representing the 26 different letters. In between each convolutional layer is a batch norm, dropout, and ReLU layer. At the end of the network, we apply a final dropout layer and a softmax, allowing us to pick the class with the highest probability as our selected letter. Figure 2.16 depicts the

architecture used for this classifier.

We trained both models on the respective datasets (MNIST for the digit classifier, EMNIST Letters for the character classifier) for 50 epochs, with a learning rate of 0.005, and training and validation batch sizes of 2000, using the training and validation splits given by the datasets. For training, we used a Tesla T4 GPU from Google Colab. We uploaded these datasets to Google Drive to link them with our Google Colab notebook, and likewise downloaded the states of each of the models at the end of the training to be accessed locally in the system. To maximize robustness, we shuffled the datasets and performed data augmentation on both the training and validation sets such as rotating the image, reflecting the image across an axis, scaling the image, and brightening or darkening the image.

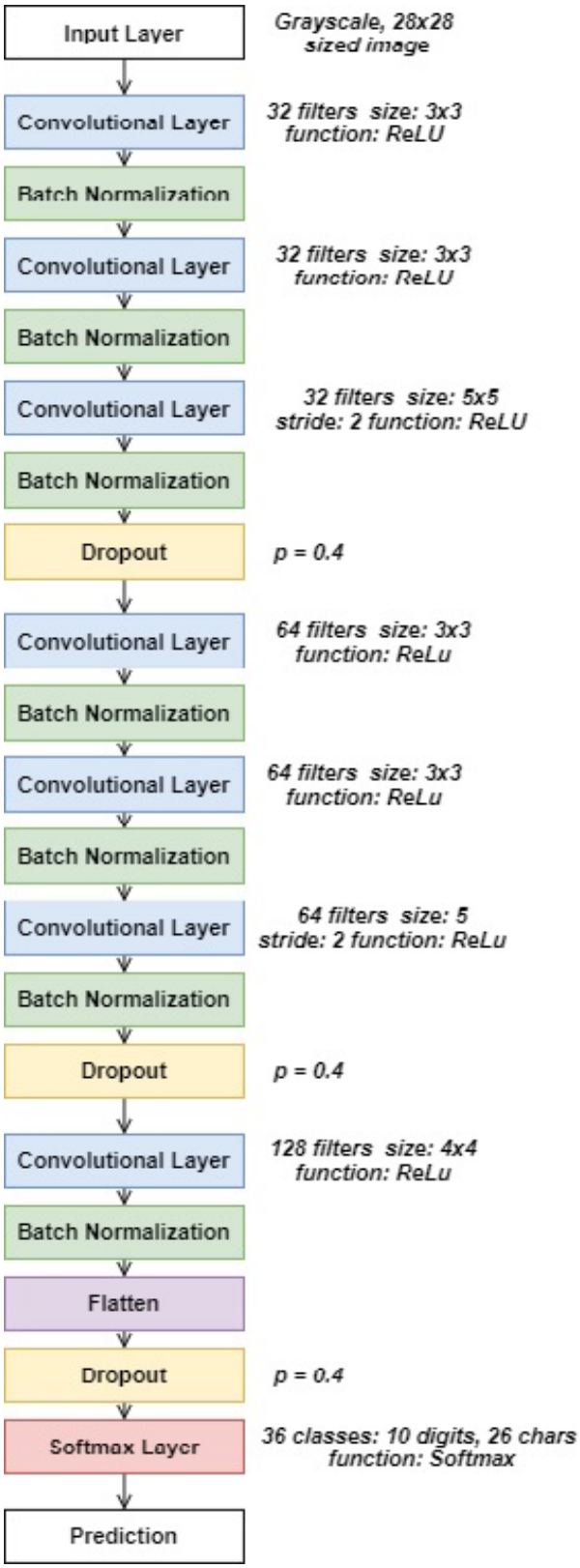



Figure 2.16: The neural network model for the letter classifier. Credit: [35].

# Chapter 3

# Control Scheme

The two-handed control system maps the left-hand gesture to different modes and the right-hand gesture to specific actions given the mode determined by the left hand. There are two main modes in this system: Mouse Mode and Keyboard Mode, which can be switched between with a designated gesture. By default, this designated gesture is set as a fist with both hands, as we felt that it was a gesture that would only be performed intentionally and thus would not result in any accidental mode switching behavior. However, we acknowledge that this gesture may be hard for some to perform and may cause fatigue if done repeatedly, so the gesture is able to be set to a different desired one using the affine alignment templates. Mode mappings can be switched and additional mappings can be added to accommodate more control options and extend system functionality. In this default set of controls, we assigned gestures based on the order of implementation (e.g. cursor control was implemented first and was assigned to the left-hand gesture "one", scrolling was implemented second and was assigned to the left-hand gesture "two", etc.), but we acknowledge that this default gesture assignment may not be the most convenient or intuitive for people, and thus they can be switched around.

## 3.1 Cursor Control

Cursor Control actions can be performed by holding up the left-hand gesture "one", while the right hand gesture controls actions such as moving the cursor, clicking, right clicking,

and double clicking.

## Previous Approaches

We previously implemented multiple methods of cursor control, such as relative cursor control, joystick cursor control, and absolute cursor control. Our current system uses absolute cursor control as it was found to be the best, but potential future development could enable the user to switch between these modes of cursor control to suit their preference.

### Joystick Cursor Control

Joystick Cursor Control is modeled after joystick controls on arcade machines by setting a point as the center, with cursor movement being based on the hand's relative position to the designated center point. For example, if the user's hand is to the left of the center point, the cursor will move continuously to the left on the screen, and the speed at which the cursor moves is proportional to the hand's distance away from the center point. This implies that the cursor will only remain still when the cursor is exactly at the designated center point, which is a near impossible task. To solve this, a radius around the center point is set such that any hand positioned within the circle determined by the radius and center point will not result in any cursor movement. This radius, as well as the center position itself, can be adjusted based on the user's preference and handedness.

However, in feedback and testing, we found that users found it unintuitive that the cursor would keep moving without any actual hand movement. Additionally, without actual hardware that the user can touch such as a real joystick, the center point and relative distance can be hard to keep track of as mere points in the system's display window.

### Relative Cursor Control

Relative Cursor Control imitates the behavior of a trackpad by moving the mouse in line with the movement of your hand. For example, if you would like to move your cursor up and to the right, you would swipe up and to the right. The quicker the hand movement, the more distance the cursor travels. This allows the user to access all parts of the screen with minimal hand movement. Relative Cursor Control was implemented by determining

the change in hand position between the last few frames to calculate its speed and direction, then moving the cursor accordingly. The speed at which the cursor moves depending on hand speed could also be customized so that users with tremors do not keep moving the cursor unintentionally, making this method robust to noise.

However, in feedback and testing, we found that relative cursor control is not intuitive in practice since there is no physical hardware the user can touch to gauge how fast their hand should move. This method is also very sensitive to lag, as cursor movement is performed according to the previous few frames of hand movement, and these delays can make gauging the appropriate hand speed difficult.

### Absolute Cursor Control

Absolute Cursor Control imitates the behavior of a tablet or a touchpad by positioning the cursor at the exact position of your hand within the screen. As such, this method was found to be the most intuitive as it follows the user's hand in position, speed, and direction.

However, there were several limitations found with this method as well. By mapping the entirety of the screen to the entirety of the system's display window, users were forced to move their hands far away from their bodies to access the edges of the screen, leading to exaggerated gestures that were not comfortable. When their hands did reach the edge of the screen, it was also more difficult for the system to recognize gestures as some parts of the hand would be outside of the camera's view. To remedy these issues, we set a bounding box towards the middle of the system's display window, and map this bounding box to the entire screen instead. This allows the user's hand to move a lot less in order to reach all parts of the screen, and enables gesture recognition to still work as intended as the entire hand is still in view. The position and size of the bounding box can be adjusted based on the user's preference and handedness.

The tradeoff with decreasing the screen-mapped size through the implementation of the bounding box is that the smaller the bounding box, the more sensitive the control scheme is to noise. Small changes in hand position may cause the cursor to move a large distance across the screen. Additionally, users must make precise movements while also trying to remain within the bounding box in order to move their cursor to a select location, and this

problem is exacerbated with smaller bounding boxes. While the absolute cursor control with bounding box method has its limitations, it was overwhelmingly found to be the most intuitive and easiest to use.

## 1€ Filter

To prevent the previously discussed problem with absolute cursor control being extremely sensitive to small movements given a small bounding box, we implemented a 1€ filter to smooth out the cursor movement. A 1€ filter is a type of digital first-order low-pass filter that adapts its filtering strength based on the rate of change of the target signal. Since its strength is smoothing noisy signals while preserving sharp edges, it is particularly useful for filtering noisy motion tracking data from computer input devices, which fits our needs very well [3].

The discrete time realization of a first-order low-pass filter is given by equation 3.1, where where $X_i$ and $\hat{X}_i$ denote the raw and filtered data at time $i$ and $\alpha$ is a smoothing factor in $[0, 1]$.

$$\hat{X}_i = \alpha X_i + (1 - \alpha)\hat{X}_{i-1} \tag{3.1}$$

The first term of the equation is the contribution of the newly input data value, while the second term adds smoothing and inertia based on previous values. In our case, the first term represents the palm position at the current frame, the second term represents the filtered palm position at the previous frame, and the result represents the filtered palm position at the current frame. As $\alpha$ decreases, noisy jitter is decreased but lag is increased. It is this tradeoff that the 1€ filter balances.

The 1€ filter calculates $\alpha$ as a function of the sampling period $T_e$ and a time constant $\tau$, as shown in equation 3.2. $\tau$ itself is calculated given the cutoff frequency $f_c$, as shown in equation 3.3.

$$\alpha = \frac{1}{1 + \frac{\tau}{T_e}} \tag{3.2}$$

$$\tau = \frac{1}{2\pi f_c} \tag{3.3}$$

The above values can be substituted into the original equation 3.1 to result in equation 3.4, which results in the equation 3.4 for the calculated filtered value.

$$\hat{X}_i = \left( X_{i-1} + \frac{\tau}{T_e} \hat{X}_{i-1} \right) \frac{1}{1 + \frac{\tau}{T_e}} \tag{3.4}$$

Among the governing equations, the only configurable parameter is the cutoff frequency $f_c$. $f_c$ itself is adapted for each new sample according to an estimate of the signal's speed, which is represented by its derivative value, using equation 3.5.

$$f_c = f_{c_{min}} + \beta \left| \dot{\hat{X}}_i \right| \tag{3.5}$$

Thus, the only configurable parameter is the minimum cutoff frequency $f_{c_{min}}$. We use the governing equations to implement a 1€ filter for cursor movement. We adapt the parameters in these equations to fit our system, with number of frames as time constants and sampling periods, cursor movement distances as frequencies, and data points and filtered points as unsmoothed/smoothed cursor locations. We create one filter for the x-direction and one filter for the y-direction. When the palm center moves, the new location is input into the filter. The filter combines this new data point with previous filtered data to output a new location for the cursor, and the system moves the cursor to that location. Through qualitative analysis, the 1€ filter resulted in the cursor jittering significantly less than both averaging across the last 5 frames and the last 20 frames, with no noticeable increase in lag.

## Gaze Tracking

One alternative method to cursor control that was investigated was gaze tracking, mentioned during the introduction as very convenient but very prone to jittery and involuntary eye movements. However, we felt that it would be a significant improvement to user experience, as they could then perform gestures with both hands while still controlling the cursor. We also believed that our implemented 1€ filter (to be discussed later) would help smooth out any sudden involuntary eye movements and keep the cursor stable as long as the user was focused on a specific point on the screen.

Similar to how we found hand key points, we utilized Google's MediaPipe framework to get key points for the user's face and specifically their eyes. It contains the key points for

the outline of the eyes, the corners of the pupil to find a bounding box, the center of the pupil, and the center of the eye [9]. The key points are shown in Figure 3.1.

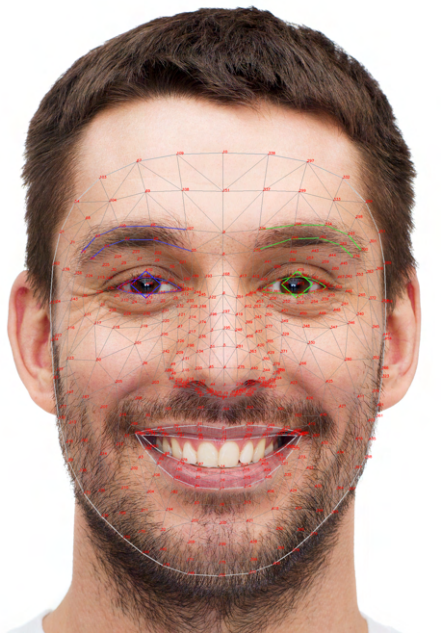


Figure 3.1: Mediapipe's facial key points, with the eye outlines highlighted. Credit: [9].

Inspired by [2], we construct a vector between the current center of the user's pupil and the center of the user's eye to construct a "'gaze vector" that represents the user's pupil movement. Figure 3.2 shows the keypoints and the gaze vector. The gaze vector can then be converted to gaze location. To do this, the user instantiates the limits of their screen by looking at each of the four corners of their screen. The four pupil locations when they do this establish a bounding box on the pupil, and the "gaze vector" within the bounding box represents where on the screen they are looking. That location would be calculated, and the cursor would be moved there.

One significant problem with this approach is that the calculated pupil bounding box being extremely small meant that the model was extremely sensitive to any eye movements, even with the 1€ filter, making doing any precise movements with the cursor impossible. Additionally, in some of our testing of the method, the location we were gazing at and the location that the cursor was put to were wildly far apart, implying that there was a

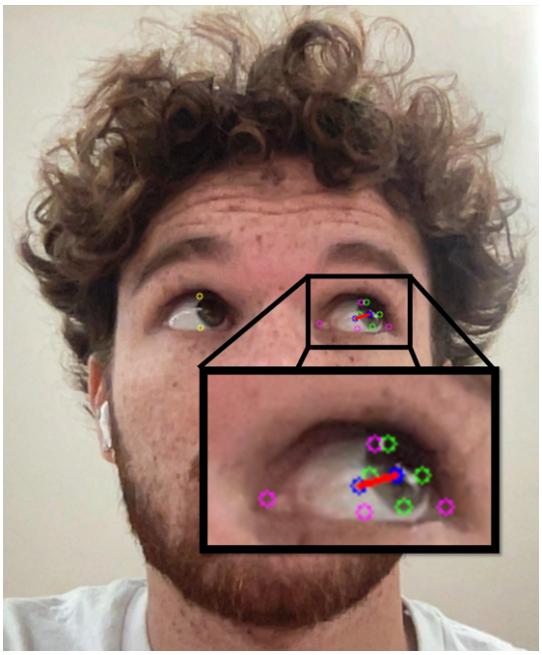


Figure 3.2: Eye keypoints with gaze vector. Credit: Nial Mandall.

major error in our algorithm. That could also be the result of only using four anchor points to create a bounding box, when that may not be enough. We also neglected to take into account user's head movements when they looked around: if they moved their head in that direction, then their eye would not need to move as much, which contributed to incorrect cursor positions. As such, the process of translating the pupil vector to on-screen locations for the cursor was ultimately a failure.

Future work in this regard may involve a more robust model, or training a machine learning model where the data are head-and-eye positions and the labels are on-screen locations. Some simpler intermediate steps to see if the approach has further potential would be to make the user set more anchor points, so the calculations for triangulating the gaze location will potentially be more accurate due to there being more points to reference and calculate from.

## 3.2 Scrolling

Scroll actions can be performed by holding up the left-hand gesture "two", while the right-hand gesture controls the direction of the scroll (up, down, left, or right). If the right hand is not detected to be performing one of the four gestures mapped to each direction, no scrolling occurs. The scrolling speed is a parameter that can be adjusted. However, one limitation is that even though the scroll speed is adjustable, it remains fixed for all directions and for an entire session at once. This means that users cannot scroll both slowly and quickly in the same session of the system, and they would have to stop, make the modification to the scroll speed, and relaunch the system in order for the changes to take effect. One way to control the scroll speed would be to create an interface similar to that for joystick cursor control. There would be an established center point, and the right hand's position relative to that center point would determine the direction and speed of the scrolling: the triangular quadrant the hand is located in relative to the center point would determine the direction of the scroll (up, down, left, or right, although this can be fine-tuned to include 45-degree angles as well), and the distance from the hand to the center point would determine the speed of the scroll.

## 3.3 Volume Control

Volume Control actions can be performed by holding the left-hand gesture "three", while the right-hand gesture controls increasing, decreasing, and muting/unmuting the computer's sound output. Despite there being four different actions, there are only three different gestures; This is due to the fact that on computers' sound system, there are not two different actions for mute and unmute. Instead, the mute and unmute actions act as a switch for whether the speaker output at all. We emulate this behavior by mapping the muting and unmuting actions to the same gesture, which acts as a toggle for the sound output. Since the gesture is recognized frame-by-frame, repeated holding of the gesture would cause the volume to increase or decrease very quickly, leading to undesirable behavior and possible sudden loud sounds. To prevent the volume from changing too quickly, we include a short pause between each volume increment so that users can stop holding the gesture at the

desired volume level and prevent overshooting and undershooting the target level.

## 3.4 Window Management and Browser Actions

Unlike the previously described control actions, window management and browser control actions consist solely of instances rather than continuous actions that require a steady stream of input for control and adjustment such as moving the cursor, adjusting volume, and scrolling. It is easy to accidentally to repeat an action unintentionally if the right hand holds the gesture for too long, and while this issue is relatively harmless for non-continuous mouse actions such as clicking, actions such as closing tabs and minimizing windows may cause great inconvenience and frustration if done repeatedly and unintentionally. To safeguard against this situation, we implement a few-frame window after one of these modes' actions where the system is not actively recognizing any gestures, so that the next action will not be performed too quickly after the initial action. This gives users enough time to switch gestures once they have performed the desired action.

### Window Management

Window Management actions can be performed by holding up the left-hand gesture "four", while the right-hand gesture controls performing common window management actions such as closing or minimizing the current window and switching the active window to the previously used application.

### Browser Control

Browser Control actions can be performed by holding up the left-hand gesture "five", while the right-hand gesture controls performing common browser and tab management actions such as opening a new tab, closing a tab, switching tabs, zooming in and out, or selecting the address bar as the text field to type in. These actions are implemented by mapping gestures to keyboard shortcuts, so any action that can be performed using a keyboard shortcut can be added to the control scheme if the user has different preferences of which actions they use most commonly.

One limitation of this implementation is that some browsers have keyboard shortcuts that are specific to their browser, but many of the commonly used functions have the same universal keyboard shortcut across browsers. The default actions we implement all have universal shortcuts, but they differ between operating systems (`ctrl+x` on Windows and Linux vs. `cmd+x` on MacOS). We check the operating system of the user and adjust the keyboard shortcuts accordingly, so our system is not limited by a specific operating system or browser.

## 3.5 Keyboard Control

The previously mentioned control modes all fall under the larger bracket of Mouse Mode. Unlike the actions in Mouse Mode. where the left hand continuously holds up a gesture in order to keep performing actions in those modes, Keyboard Control Mode is a toggle can be activated by both hands holding up a "fist" gesture. When the system is in Keyboard Mode, the gesture mapping scheme changes completely, and the mapping scheme in Mouse Mode no longer applies. Keyboard Mode is used for typing when a text field has been selected. Keyboard Mode can be toggled off by performing the same "fist" gesture with both hands, and the system will switch back to Mouse Mode and all of its previously discussed gesture mapping schemes.

### Previous Mapping Implementation

The previous implementation of the keyboard mode gesture mapping scheme mapped a combination of left and right hand gestures from "one" to "five" to the letters of the alphabet. To keep the mapping scheme as intuitive as possible, we approached the order of the mapping from the same perspective as that of mouse mode: we mapped the letters in order with a left-hand-gesture-first, right-hand-gesture-second approach. As this method only yields 25 mappable gesture combinations for the 26 letters of the alphabet, we made it so that $Z$ maps to left-hand gesture "five", right hand gesture "None", which represents not inputting one of the recognized gestures. Table 3.1 shows the full keyboard mapping control scheme under this previous implementation.

| | | Right Hand Gesture | | | | | |
|---|---|---|---|---|---|---|---|
| | | "one" | "two" | "three" | "four" | "five" | "None" |
| Left Hand Gesture | "one" | *A* | *B* | *C* | *D* | *E* | N/A |
| | "two" | *F* | *G* | *H* | *I* | *J* | N/A |
| | "three" | *K* | *L* | *M* | *N* | *O* | N/A |
| | "four" | *P* | *Q* | *R* | *S* | *T* | N/A |
| | "five" | *U* | *V* | *W* | *X* | *Y* | *Z* |

Table 3.1: Original Keyboard Gesture Mapping Scheme

There are a few limitations to this implementation. Despite the best efforts to make the order intuitive, it may be difficult for users to remember the mapping, which necessitates displaying the mapping scheme on the system's display window. This could cause delays in typing as the user needs to constantly reference the mapping scheme table, and cause a lot of text to block parts of the screen. Additionally, not every character that could be typed is mapped, particularly differentiating uppercase and lowercase letters, as well as special characters. While one solution to this limitation could be to extended the scheme to include all of the recognized gestures that are currently unused such as arrow, thumb, and fist, this would cause even more confusion and bloat the mapping scheme even more, leading an even larger table display in the system's display window and even more blocking text on the screen. In reality, extending the scheme to incorporate all the characters that could be typed is infeasible under this implementation.

## New Drawing Implementation

The new implementation of the keyboard mode gesture mapping scheme involves recognizing numbers and letters based on drawings. As such, the actions available are draw, submit drawing, clear drawing, backspace, and switching between typing letters and digits. To draw, the user holds up a "one" gesture with their right hand and begins drawing in the air with their single finger. This gesture was chosen for its intuitiveness, as holding up one finger is the equivalent to air writing. No left hand gesture is needed for this action, as it is considered the default state when in this mode. The left-hand gestures include backspace and switching between typing letters and digits, while the right-hand gestures include drawing,

submitting the drawing, and clearing the drawing. This separation was chosen such that the left hand controls actions external to the drawing itself, allowing the right hand to control everything related to the drawing itself.

In summary, Figure 3.3 summarizes the complete two-handed gesture control scheme.

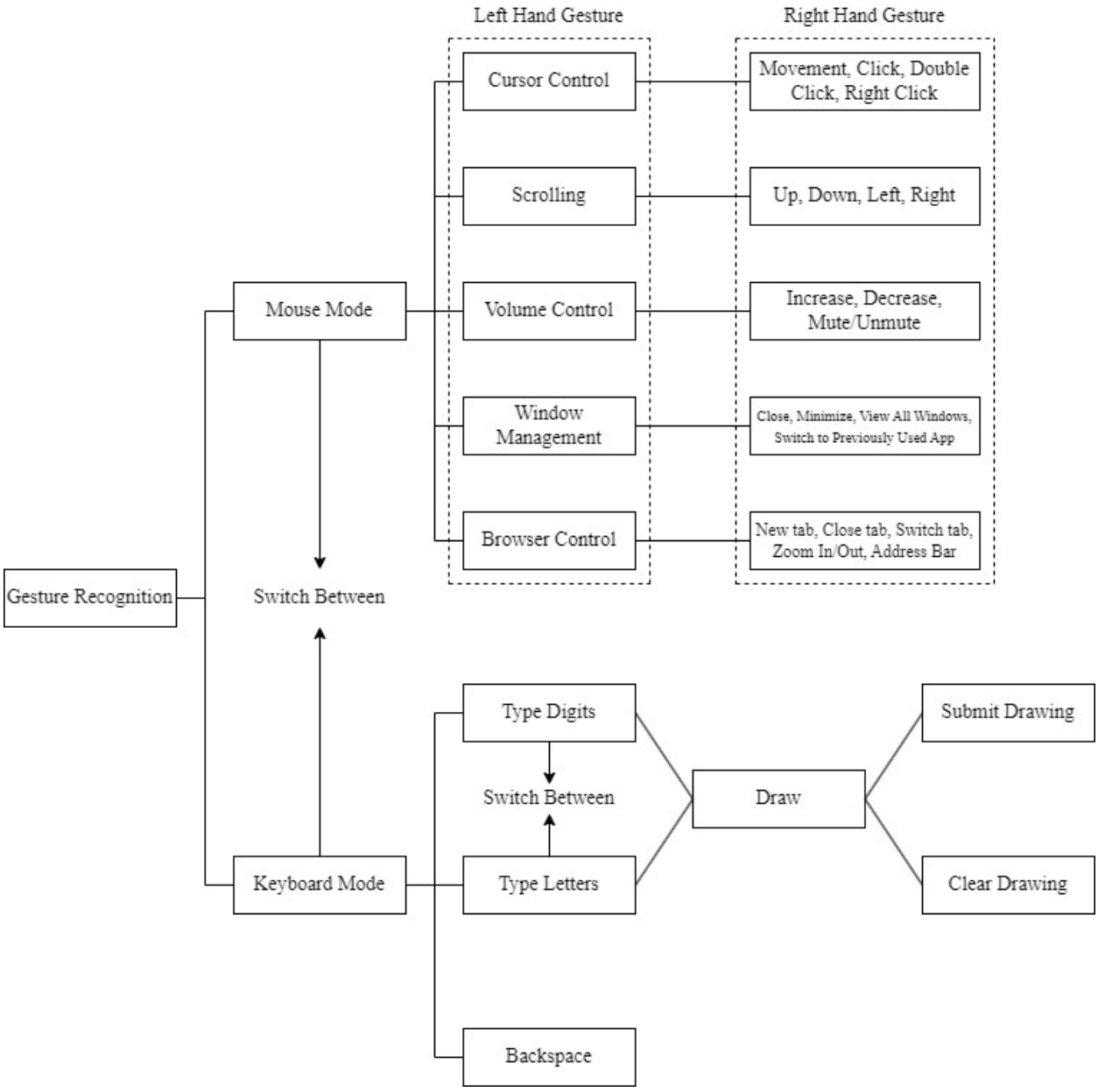



Figure 3.3: Full Two-Handed Gesture Control Scheme

# Chapter 4

# Results

## 4.1 Smoothing Methods

Prior to our implementation of the 1€ filter to smooth cursor movement, we also tried averaging the positions of the palm centroid among the last 5 and last 20 video capture frames. We compared the jitter and lag among each of the smoothing methods as we performed the same series of tasks of opening a web browser through a shortcut on the taskbar, opening a new tab, dragging the new tab out to create a new browser window, and closing both web browser windows. This series of actions was chosen because it forced the user to perform a variety of actions including both large, cross-screen movements and micromovements, clicking, and dragging. Because the series of actions was preset, a straight-line route between locations the cursor was meant to travel to could be drawn. Jitter was measured as the average number of pixels away from the straight-line route the cursor was at a given frame. Lag was measured as the average of the following times: the time between when a click gesture was made and a click was executed, the time between when a hand that was previously at rest begins to move and the time that the corresponding cursor that was at rest begins to move, and the time between when a hand had arrived at a desired location and stopped moving and when the corresponding cursor arrived at that location and stopped moving. All experimentation was done on a $3072 \times 1920$ pixel 2020 Macbook Pro monitor using the built-in FaceTime HD camera, which records at 720p. Figure 4.1 shows the average jitter and lag measured for these three smoothing methods.

| Smoothing Method | Jitter (Pixels) | Input Lag (seconds) |
|---|---|---|
| Average of Previous 5 | 94.74 | 0.085 |
| Average of Previous 20 | 78.26 | 0.353 |
| 1€ Filter | 55.38 | 0.108 |

Table 4.1: Jitter and Input Lag for different Smoothing Methods

The table shows the tradeoff between jitter and lag among the average of the previous 5 and previous 20 palm centroid positions. While the 1€ filter smoothing method is not as responsive as averaging the previous 5 positions (which had basically no lag), it also smooths out significantly more jitter than just averaging the previous 5 positions and even averaging the previous 20 positions. This could be due to the fact that it has much less lag than averaging the previous 20 positions, and because of its responsiveness, it is actually easier for the user to follow their cursor as it updates on the display and guide it to the desired location.

Due to our default control state being that a visible palm signifies cursor movement while a closed palm signifies a click, the act of clicking was equivalent to closing your hand into somewhat of a fist. This motion itself caused the cursor to shift downward during the act of clicking, and when we only averaged the previous 5 palm centroid positions, it became difficult to click our desired target. It was especially difficult to perform the act of closing tabs, since the button to close tabs is at the top of the screen, but the clicking motion would shift the cursor too far down to reach the button, and it would end up taking multiple attempts before we could successfully click the "close" button. This problem was not present when using the other two stronger smoothing methods, as the downward movement of the cursor when clicking was negligible. For this reason, despite the lower lag of the weaker smoothing method, the difference between lag times is negligible relative to human reaction time and 1€ filter actually felt more responsive due to not having to correct unsuccessful actions. For this combination of low jitter, low lag, and perceived ease of use, we claim that the 1€ filter is the superior smoothing method for cursor control in this system.

## 4.2 One Model vs. Two Model Approach

Our initial approach to letter and digit classification involved classifying both with one model, such that the model took in the drawing and output it among all 36 different classes (10 digits, 26 letters). However, we quickly found the accuracies to be poor and not improving. Having two separate models, where one classified the digits and one classified the letters, improved classification accuracy drastically and to a satisfactory level.

To compare the performance of the models and measure the improvement in our two-model design over our one-model design, we tested both designs on the test splits of three of EMNIST's datasets, corresponding to letters, numbers, and a mix of both. We felt this division was appropriate as we split our model design for the purpose of better-classifying letters and numbers as individual categories. The EMNIST Letters dataset test split ("Letters Only") contains 14800 handwritten uppercase and lowercase letters merged into a balanced dataset across all 26 letter classes. The EMNIST Digits dataset test split ("Numbers Only") contains 40000 handwritten numbers merged into a balanced dataset across all 10 number classes. The EMNIST Balanced dataset test split ("Numbers and Letters") contains 18800 balanced across the 10 number classes and the 26 letter classes. Figures 4.2 and 4.3 show the classification accuracies for these models on the EMNIST validation dataset splits of numbers only, letters only, and both numbers and letters.

**Single Model**

| **Dataset** | **Accuracy (%)** |
|---|---|
| Numbers Only | 34.7 |
| Letters Only | 69.1 |
| Numbers and Letters | 61.8 |

Table 4.2: Validation Accuracies on Several Datasets While Classifying Using a Single Model

Both tables clearly reflect that our new approach to classify the drawings using two split models was a significant improvement in our system.

**Two Models**

| Dataset | Accuracy (%) |
|---|---|
| Numbers Only | 89.0 |
| Letters Only | 92.7 |
| Numbers and Letters | 91.7 |

Table 4.3: Validation Accuracies on Several Datasets While Classifying Using Two Models

## 4.3 EMNIST Letter Accuracies

We also analyze the Type I and Type II errors present in our letter classifier specifically. We perform individual letter analyses on our letter classifier to determine if there are any letters that are classified incorrectly significantly more than the rest, or if a specific letter is incorrectly chosen as the classified letter more than others. If any incorrect outputs dominate the rest of the letters, we make specific note of those errors. Figures 4.4 and 4.5 show the type 1 errors and type 2 errors of the classifier per letter.

Between the overall accuracies and the letter-by-letter analyses, the tables show that the letter classifier performs with above 90% accuracy on the validation datasets, and above 95% error avoidance on both Type I and Type II errors for most letters, with some specific outliers. This meets our group-set threshold that the letter classifier is satisfactory enough to be functional. With the classifier itself reaching satisfactory level, future work would be to extend the model to other characters, including splitting by uppercase and lowercase letters and including special characters in the list of typable letters to be classified.

From the table and graph of the correctness accuracy, we note that a few letters have noticeably lower accuracies than the rest: G, I, L, Q, and U. Likewise, from the table and graph of the predicted letter accuracies, we note that G, I, L, Q, V. When looking at the combined graph of the correctness accuracy and the prediction accuracy for each letter, figure 4.3, we note that the two values were similar for all letters, with the only significant discrepancy of a difference greater than 5% coming from V, as V's predicted letter accuracy was 5.5% higher than its correctness accuracy. This means that except for V, no letter is being disproportionally classified as another letter, but rather that pairs of letters may be hard to distinguish. From the table, we observe that this discrepancy comes from many Us

| Correct Letter | Accuracy (%) | Significant Incorrect Predictions (letter: %) |
|---|---|---|
| A | 94.875 | |
| B | 97.125 | |
| C | 97.250 | E: 1.38 |
| D | 95.500 | O: 2.88 |
| E | 97.125 | C: 1.12 |
| F | 96.250 | T: 1.75 |
| G | 84.205 | Q: 11.12, A: 1.75 |
| H | 95.750 | N: 1.38 |
| I | 75.625 | L: 23.12 |
| J | 93.875 | I: 3.38 |
| K | 97.625 | |
| L | 76.375 | I: 21.12 |
| M | 98.625 | |
| N | 97.000 | |
| O | 97.75 | D: 1.25 |
| P | 98.75 | |
| Q | 85.625 | G: 9.75, A: 1.88 |
| R | 96.25 | V: 1.38 |
| S | 98.000 | |
| T | 98.125 | |
| U | 92.500 | V: 6.25 |
| V | 95.625 | U: 1.62 |
| W | 97.625 | |
| X | 97.625 | |
| Y | 94.750 | V: 1.62, X: 1.12 |
| Z | 99.375 | |

Table 4.4: Type II errors of the classifier per letter: the percentage of letters for which the classifier predicted incorrectly.

being incorrectly classified as Vs, and with this knowledge we can construct a dataset to further train our model to remove this inaccuracy.

These data from the tables and graphs and our earlier observations also lead to the conclusion that the model has trouble distinguishing between the specific pairs of letters (I, L) and (G, Q). To solve the problem of a few pairs of letters being difficult to distinguish in particular, we can construct extra layers of binary classifiers to help differentiate between

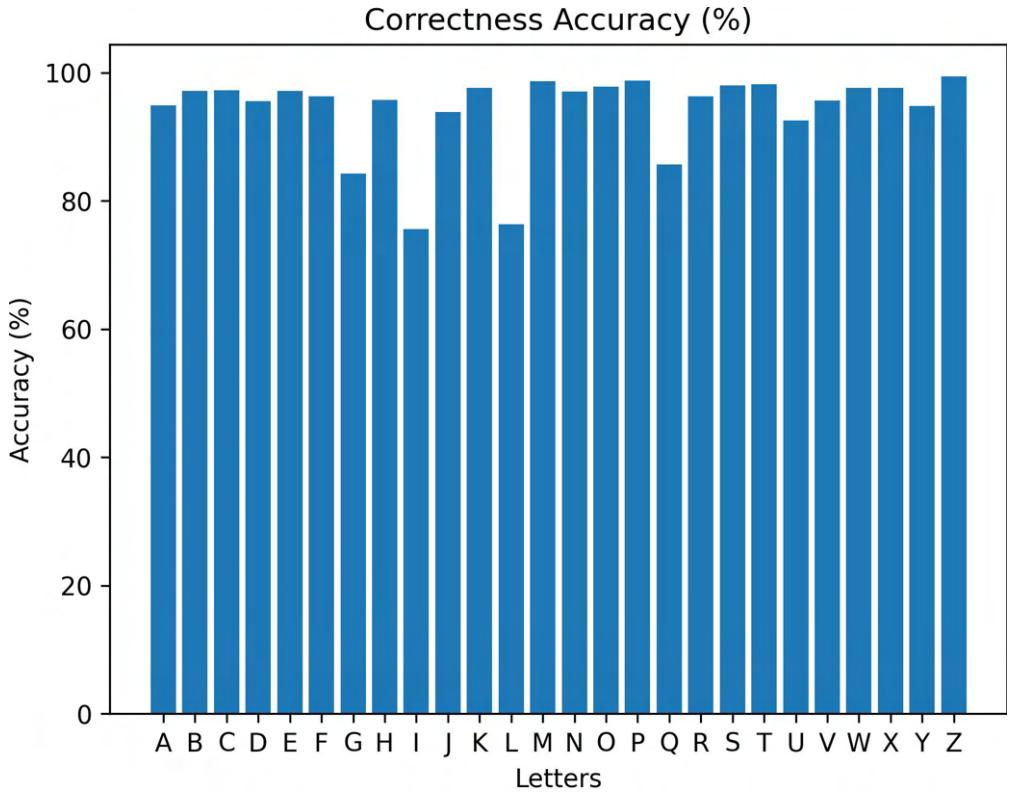



Figure 4.1: Correctness accuracy of each letter: the percentage of letters for which the classifier was meant to classify the data as a letter and correctly classified it as that letter. Conversely, the smaller the bar, the more Type II errors occurred for that letter.

those letters in particular. The model would classify an image, and the predicted result was one of these designated letters, it would feed it to a designated binary classifier for that letter pair to further refine its prediction. Because these binary classifiers would be specifically trained to differentiate a certain pair of letters, their accuracies should be much higher than the general character model's accuracy when it comes to specifically distinguishing amongst those letters. This additional potential future work should result in another significant increase in the model's accuracy. Another way to solve these outlier pairs would be to show an on-screen selector on the system's display window with the two commonly confused letters as options, from which the user would select their desired letter to be typed. However, while the user is guaranteed to type their desired character, this method introduces an extra delay and inconvenience for the user, when the objective of the system is to be accessible and fast.

| **Predicted Letter** | **Accuracy (%)** | **Significant Correct Letters (letter: %)** |
|---|---|---|
| A | 93.820 | Q: 1.85, G: 1.73 |
| B | 98.354 | |
| C | 96.406 | E: 1.12 |
| D | 96.100 | O: 1.26 |
| E | 97.004 | C: 1.37 |
| F | 98.845 | |
| G | 85.533 | Q: 9.90, A: 1.02 |
| H | 95.870 | |
| I | 75.062 | L: 20.97, J: 3.35 |
| J | 97.028 | |
| K | 98.000 | |
| L | 74.603 | I: 22.59 |
| M | 98.748 | |
| N | 95.449 | H: 1.35 |
| O | 95.600 | D: 2.81 |
| P | 97.651 | |
| Q | 87.150 | G: 11.32 |
| R | 96.25 | |
| S | 98.990 | |
| T | 95.966 | |
| U | 96.104 | V: 1.69 |
| V | 90.106 | U: 5.89, Y: 1.53, R: 1.3 |
| W | 99.490 | |
| X | 97.019 | |
| Y | 96.931 | |
| Z | 98.148 | |

Table 4.5: Type I errors of the classifier per letter: the percentage of the classifier's predictions of a specific letter where the true label was a different letter.

Thus, we conclude that our model is overall satisfactory for typing use, but starts to break down with regard to the letters G, I, L, Q, and U specifically. We hope that future versions of our model, after the implementation of the binary classifiers described here, can alleviate the majority of the problems surrounding the correct classification of these letters.

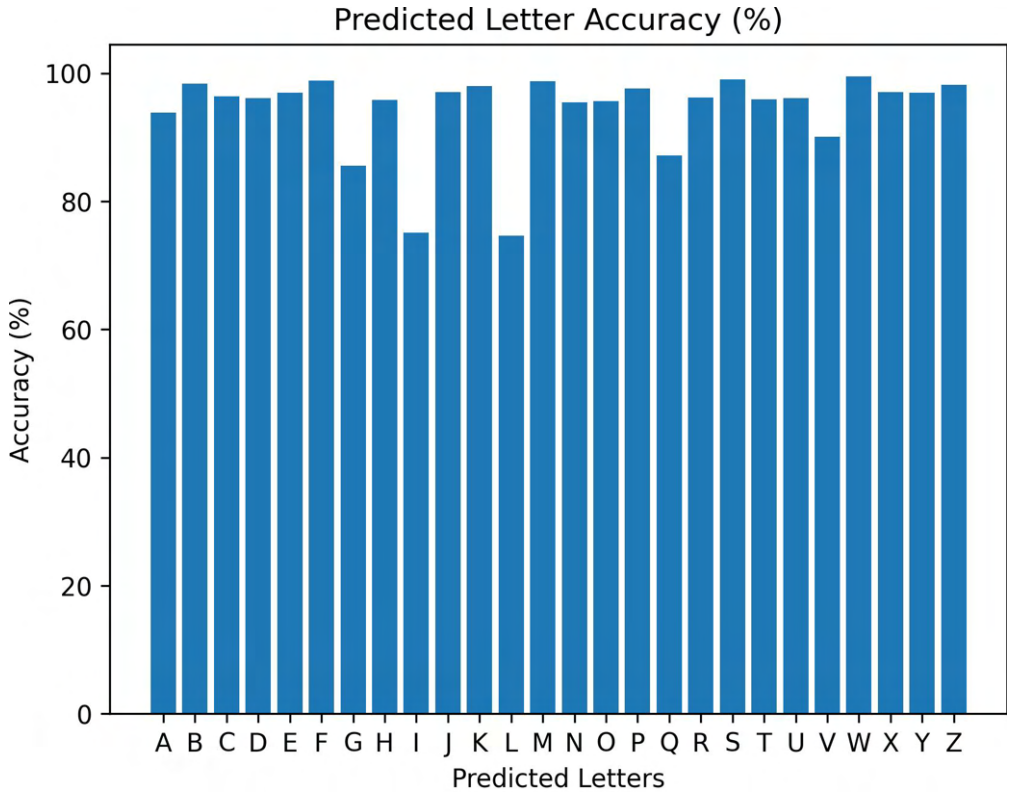



Figure 4.2: Predicted Letter accuracy of each letter: the percentage of letters for which the classifier classified the data as a letter and the true label was also that same letter. Conversely, the smaller the bar, the more Type I errors occurred for that letter.

## 4.4 Bundled Executable

The current method to run the system requires significant amounts of computer science knowledge such as command line inputs, Python environments, and package installation that we cannot expect the target audience of this application to have. Since the objective of this system is accessibility, we wanted to create a simple method to run this application that is easy and intuitive for users. Ideally, the user is able to start running the system just like any other application: by double-clicking an application shortcut.

Our system is built in Python and requires many Python package dependencies to perform certain tasks and execute functions. However, we do not want a user to need to have Python installed or know how to install these packages in order to use the application. Thus, we start at the main Python file, trace all of the package dependencies that the file needs (including

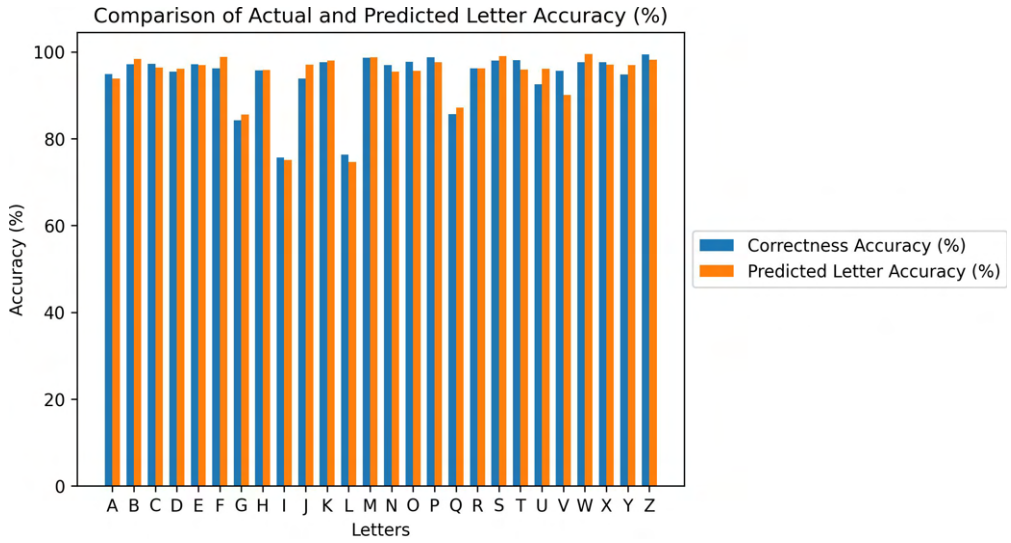



Figure 4.3: A side-by-side comparison between the correctness accuracy and predicted letter accuracy for each letter. This graph shows any trends if a letter is being consistently over-represented or underrepresented in the data. No huge discrepancies between the two columns for any given letter means that using a balanced dataset during training worked as intended and that errors come from hard-to-distinguish letters, not imbalance within the dataset.

other files and those files' packages) to build a dependency tree, and then link all of the files according to the dependency tree into a bundled executable. There are slight differences in Python installation and package installations between MacOS and Windows, as packages not only use some different dependency files but also the file paths that they should be retrieved from. To ensure that we are not limited by the type of operating system, we create a version of the executable for both MacOS on intel-based Macs (pre-M1) and Windows 10/11. There is currently no version of the executable for MacOS on ARM (post-M1) Macs due to the different CPU architecture conflicting with some of the package versions used in our codebase, but getting the executable for that type of operating system architecture is currently in progress.

Aside from the installed packages, the main Python file also uses relative file pathing to retrieve the saved neural network models for the digit and letter classifiers, as well as the hand keypoint data for the templates. This works for the current way that the project's directories are set up, but would not work once the project is exported as a standalone bundled executable. One thing that is different is unlike in a project directory structure, the

bundled executable's location when run is not the where the file is run from; rather, it is the root directory of the terminal, so following the same relative file pathing will not retrieve the correct files. This issue is solved by replacing all instances where the current directory is referenced with an absolute path to the location of the executable file, such that all further relative file paths remain properly referenced.

We include the template and model data files, as well as a README with instructions on how the system is used, with the executable, and wrap all of the files in a ZIP file to be distributed. The creation of this file allows the full system to be distributed to a target audience for testing without them needing to install any external software that they may be unfamiliar with. Ideally, we would not need to include any other files other than the executable so that there is zero confusion among users when they download the application, but that is not something we have managed yet.

We test this executable on MacOS on an Intel-based MacBook, a laptop running Windows 10, and a PC running Windows 11. On all of these devices, the application properly opens and runs, all functionalities are present, and the system runs at over 60 frames per second. Future work in this regard would be to host the bundle on a publicly available webpage, such as the lab's website, so users can download and test it.

# Chapter 5

# Limitations, Future Work and Conclusion

## 5.1 Limitations

Although our two-handed gesture control system provides users with satisfactory and functional means to control their computers for basic tasks, the system is still not able to completely replicate the full functionality of the traditional mouse or trackpad and keyboard combination in certain aspects.

### Fatigue

One major problem to account for during operation is fatigue and repeating the same gestures over and over. Despite the stationary nature of operating a computer mouse being previously described as a hazard, it does provide comfort in that you can rest your arms on a flat surface when operating it, which reduces fatigue. However, using a front-facing camera that is present on laptops involves users bringing their hands and arms upwards for prolonged periods of time, which can cause fatigue in the arms very quickly. One potential solution investigated was to use a top-down camera: that way, users can keep their hands and arms on their desk as they gesture to use the system. The major issue with this solution is that top-down-facing cameras are much less common than the front-facing cameras present on

almost every laptop, and one major goal of this project was to avoid having users purchase any additional hardware. Thus, the issue of fatigue is something that still must be taken into account when further designing gesture recognition systems.

## Speed and Incorrect Gesture Identification

Furthermore, the current system has limitations due to the level of technology we work with. It currently takes registering the same gesture for 5 frames for it to be recognized by the system, which imposes a limit on the speed and convenience with which this technology can be used by users. The gesture recognition system is also not perfect: there will be false positives and false negatives and moments where the gesture recognized is incorrect and an unintended action is performed, so we must utilize the further increasing research into this space to increase the correctness of our system to match the user's intent, while ensuring the safety and convenience of their computer if an unintended action were to occur, so as to minimize any potential negative impact. The current system also has certain core features that can added or improved, many of which have been previously discussed and future work has been proposed.

## Changing Gesture Mapping Scheme

One major limitation of the system not yet mentioned is the difficulty with which changing specific gesture mapping schemes can be. Some users may find specific hand gestures more intuitive or easier to carry out certain actions than others. being able to add more recognizable gestures or remap hand gestures to different actions would be a great improvement to the user experience. Our system is currently capable of adding new gestures to be recognized and remapping gestures to different tasks, but as it currently stands it requires significant knowledge of the codebase to go in and change, which should not be expected of the average target user. Future work in this aspect may be to include a better user interface for the user to walk through and add a gesture/customize a gesture mapping without much difficulty, and should be a priority given the emphasis on convenience and accessibility this system is designed to have.

### Handedness Asymmetry

Enabling the user to remap gestures directly from the system window without requiring significant knowledge of the codebase would also help establish hand symmetry in the system. In particular, our control scheme is mapped from the perspective of someone who is right-handed: the left hand often performs simpler gestures and often just needs to hold their gesture still to indicate the specific control mode, while the right hand switches gestures frequently. In contrast, left-handed users may find the system more intuitive and comfortable to use if the gestures on the hands were swapped in the control scheme. As such, the ability to remap the gestures, or even the ability to choose a handedness and have the gestures automatically swapped if the handedness is swapped, would greatly benefit those users.

## 5.2 Future Tasks and Directions

In discussing some of the limitations of the system in previous sections, some future directions for features that could be added to the system to address those limitations. Ideas mentioned previously but incomplete, unsuccessful, or not implemented include: gaze tracking for cursor control, joystick scroll control, splitting the letter classifier by uppercase and lowercase, incorporating special characters into the keyboard control scheme, and binary classifiers for outlier-pair letters. Below are some future directions extending on implemented features that have not been mentioned yet.

### Full Word Recognition Typing Scheme

Our current Keyboard Control Scheme has users draw letters one-by-one, and while the current system is very flexible and extendable to more characters and features, it is extremely slow. One way to improve this scheme would be to allow users to write whole words at once, have the model recognize it letter by letter, then type out the entire word at once. This method would not only be faster but more intuitive as well: in daily life, we rarely write individual letters deliberately one at a time to form an entire word, we write the whole word at once.

However, there are a few challenges in the implementation of this scheme. People often write individual letters differently from how they would write the letter if it were in a word: they are more deliberate with the individual letter, and in words sometimes letters connect to other letters. This potential lack of separation may be confusing to the model and decrease the accuracy significantly. Furthermore, the model has to get the entire word right, or else it is inconvenient for the user to have to backspace half of the word to correct a mistake by the model in the middle of a word. Even though the model has high accuracy for most letters, it has a chance of getting each individual letter wrong, which makes getting the entire word correct difficult as the inaccuracies of the model compound multiplicatively as the written word gets longer.

## 5.3 Conclusion

Our two-handed gesture control system provides an accessible and functional alternative to the traditional mouse and keyboard. By utilizing only a computer's default camera and the user's hand gestures, users are able to replicate common mouse, keyboard, application window, and browser actions. We continually develop our system to include more features while being more accessible and customizable. In particular, people who find using traditional mice and keyboards challenging or uncomfortable may benefit from our system. Looking outside of the mouse and keyboard, gesture control systems may become more prevalent in other spaces such as virtual or augmented reality, where users use their gestures to interact with the virtual world as if it were analogous to gesturing in reality, or in remote control robotics, where the potential to have robots mimic human gestures can lead to advancements in medicine, service, and other sectors. As technology becomes more widespread and its users more diverse, we believe that our two-handed gesture control system is a step towards making technology usage more inclusive for everybody.

# Bibliography

[1] Ability411. *What solutions are available to computer users who have tremors impacting their use of a keyboard or mouse?* University of Victoria, 2024. URL: https://www.ability411.ca/answer/computers-use-with-tremors-what-solutions-are-available-to-computer-users-who-have-tremors-impacting-their-use-of-a-keyboard-or-mouse.

[2] Saravanan Alagarsamy, S. Abhiram Reddy, V. Venkata Reddy, V. Bharath Reddy, and Y.V. Praneeth Reddy. "Control the Movement of Mouse Using Computer Vision technique". In: *2022 6th International Conference on Electronics, Communication and Aerospace Technology*. 2022, pp. 395–399. DOI: 10.1109/ICECA55336.2022.10009394.

[3] Géry Casiez, Nicolas Roussel, and Daniel Vogel. "1€ Filter: A Simple Speed-based Low-pass Filter for Noisy Input in Interactive Systems." In: *CHI'12, the 30th Conference on Human Factors in Computing Systems* (2012), pp. 2527–2530. DOI: 10.1145/2207676.2208639.

[4] Ilya Chugunov and Avideh Zakhor. "Duodepth: Static Gesture Recognition Via Dual Depth Sensors". In: *2019 IEEE International Conference on Image Processing (ICIP)*. 2019, pp. 3467–3471. DOI: 10.1109/ICIP.2019.8803665.

[5] Gregory Cohen, Saeed Afshar, Jonathan Tapson, and André van Schaik. "EMNIST: an extension of MNIST to handwritten letters". In: *CoRR* abs/1702.05373 (2017). arXiv: 1702.05373. URL: http://arxiv.org/abs/1702.05373.

[6] University of Colorado Denver College of Engineering Design, Computing Center for Inclusive Design, and Engineering. *Alternative Keyboards*. 2024. URL: https://www.ucdenver.edu/centers/center-for-inclusive-design-and-engineering/

community-engagement/colorado-assistive-technology-act-program/technology-and-transition-to-employment/alternative-keyboards.

[7] Li Deng. "The mnist database of handwritten digit images for machine learning research". In: *IEEE Signal Processing Magazine* 29.6 (2012), pp. 141–142.

[8] Google Developers. *Hand Landmarks Detection Guide*. Ed. by MediaPipe Solutions. Nov. 2023. URL: https://developers.google.com/mediapipe/solutions/vision/hand_landmarker.

[9] Google Developers. *Face Detection Guide*. Ed. by MediaPipe Solutions. Jan. 2024. URL: https://developers.google.com/mediapipe/solutions/vision/face_detector.

[10] GazeRecorder. *GazePointer*. URL: https://gazerecorder.com/gazepointer/.

[11] Diyar Gür, Niklas Schäfer, Mario Kupnik, and Philipp Beckerle. "A Human–Computer Interface Replacing Mouse and Keyboard for Individuals with Limited Upper Limb Mobility". In: *Multimodal Technologies and Interaction* 4.4 (2020). DOI: https://doi.org/10.3390/mti4040084.

[12] Dennis van der Heijden. *Alternative Pointing Systems for Mobility Impaired People*. Axistive, Mar. 2006. URL: http://www.axistive.com/alternative-pointing-systems-for-mobility-impaired-people.html.

[13] Fang Hu, Xingying Hu, Guanghan Pan, and Juntao Peng. *SimpleGest: Assistive Hand-Gesture Recognition Technology Based on Computer Vision*. M. Eng Capstone report. Berkeley, CA, USA: University of California, Berkeley, May 2022.

[14] Greystone Digital Inc. *BigKeys keyboards*. 2007. URL: bigkeys.com.

[15] Anushka Jain. *Real-Time AI Virtual Mouse System Using Computer Vision*. Geeks for Geeks, Nov. 2023. URL: https://www.geeksforgeeks.org/ai-virtual-mouse/.

[16] *Keyboard and mouse alternatives and adaptations*. URL: https://abilitynet.org.uk/factsheets/keyboard-and-mouse-alternatives-and-adaptations.

[17] Faiz Khan, Basit Halim, and Asifur Rahman. "Computer Vision Based Mouse Control Using Object Detection and Marker Motion Tracking". In: *International Journal of Computer Science and Mobile Computing* 9 (May 2020), pp. 35–45. URL: https://www.researchgate.net/publication/341255951_Computer_Vision_Based_Mouse_Control_Using_Object_Detection_and_Marker_Motion_Tracking.

[18] Thorin Klosowski. *The Best Trackballs*. The New York Times Wirecutter, Aug. 2024. URL: https://www.nytimes.com/wirecutter/reviews/best-trackballs/.

[19] Ruilong Li, Xin Dong, Zixi Cai, Dingcheng Yang, Haozhi Huang, Song-Hai Zhang, Paul L. Rosin, and Shi-Min Hu. "Pose2Seg: Human Instance Segmentation Without Detection". In: *CoRR* abs/1803.10683 (2018). arXiv: 1803.10683. URL: http://arxiv.org/abs/1803.10683.

[20] K. Lyons, D. Plaisted, and T. Starner. "Expert chording text entry on the Twiddler one-handed keyboard". In: *Eighth International Symposium on Wearable Computers*. Vol. 1. 2004, pp. 94–101. DOI: 10.1109/ISWC.2004.19.

[21] Rui Ma, Zhendong Zhang, and Enqing Chen. "Human Motion Gesture Recognition Based on Computer Vision". In: *Complexity* 2021 (2021), pp. 1–11. DOI: https://doi.org/10.1155/2021/6679746.

[22] Hasan Mahmud, Mashrur Mahmud Morshed, and Md. Kamrul Hasan. "A deep-learning-based multimodal depth-aware dynamic hand gesture recognition system". In: *CoRR* abs/2107.02543 (2021). arXiv: 2107.02543. URL: https://arxiv.org/abs/2107.02543.

[23] Mohamed Nasor, K K Mujeeb Rahman, Maryam Mohamed Zubair, Haya Ansari, and Farida Mohamed. "Eye-controlled mouse cursor for physically disabled individual". In: *2018 Advances in Science and Engineering Technology International Conferences (ASET)*. 2018, pp. 1–4. DOI: 10.1109/ICASET.2018.8376907.

[24] Canadian Centre for Occupational Health and Safety. *Office Ergonomics - Computer MOuse - Common Problems from Use*. June 2017. URL: https://www.ccohs.ca/oshanswers/ergonomics/office/mouse/mouse_problems.html.

[25] Patricia Ouyang. "Accessible Two-handed Gesture Control for Human Computer Interaction". Master's thesis. Berkeley, CA, USA: University of California, Berkeley, May 2022.

[26] *Precision Gaze Mouse*. 2018. URL: https://precisiongazemouse.org.

[27] Michael Qi. "Designing an Assistive Mouse for Human Computer Interaction Using Hand Gestures". MA thesis. EECS Department, University of California, Berkeley, May 2021. URL: http://www2.eecs.berkeley.edu/Pubs/TechRpts/2021/EECS-2021-119.html.

[28] David Rempel. "The Split Keyboard: An Ergonomics Success Story". In: *Human factors* 50 (July 2008), pp. 385–92. DOI: 10.1518/001872008X312215.

[29] Murtaza's Workshop - Robotics and AI. *AI Virtual Mouse — OpenCV Python — Computer Vision*. Youtube, 2021. URL: https://www.youtube.com/watch?v=8gPONnGIPgw.

[30] Devika Rani Roy, Shubham Jadhav, Tedav Joshi, and Farhan Momin. "Virtual Interactive Keyboard and Mouse Using OpenCV". In: (2020).

[31] Abir Sen, Tapas Kumar Mishra, and Ratnakar Dash. *A Novel Hand Gesture Detection and Recognition system based on ensemble-based Convolutional Neural Network*. 2022. arXiv: 2202.12519 [cs.CV].

[32] Peige Song, Yan Zhang, Mingming Zha, Qingwen Yang, Xinxin Ye, and Igor Rudan. "The global prevalence of essential tremor, with emphasis on age and sex: A meta-analysis". In: *Journal of Global Health* 11.04028 (2021). DOI: 0.7189/jogh.11.04028.

[33] Jing-Hao Sun, Ting-Ting Ji, Shu-Bin Zhang, Jia-Kui Yang, and Guang-Rong Ji. "Research on the Hand Gesture Recognition Based on Deep Learning". In: *2018 12th International Symposium on Antennas, Propagation and EM Theory (ISAPE)*. 2018, pp. 1–4. DOI: 10.1109/ISAPE.2018.8634348.

[34] Merita Tiric-Campara, Ferid Krupic, Mirza Biscevic, Emina Spahic, Kerima Maglajlija, Zlatan Masic, Lejla Zunic, and Izet Masic. "Occupational overuse syndrome (technological diseases): carpal tunnel syndrome, a mouse shoulder, cervical pain syndrome". In:

*Acta Informatica Medica* 22.5 (2014), pp. 333–340. DOI: 10.5455/aim.2014.22.333-340.

[35] Vishal Vaidya, T. Pravanth, and D Viji. "Air Writing Recognition Application for Dyslexic People". In: *2022 International Mobile and Embedded Technology Conference (MECON)*. 2022, pp. 553–558. DOI: 10.1109/MECON53876.2022.9752119.

[36] Ryen W. White and Eric Horvitz. "Population-scale hand tremor analysis via anonymized mouse cursor signals". In: *npj Digital Medicine* (Sept. 2019). DOI: https://doi.org/10.1038/s41746-019-0171-4. URL: https://www.nature.com/articles/s41746-019-0171-4.

[37] Fan Zhang, Valentin Bazarevsky, Andrey Vakunov, Andrei Tkachenka, George Sung, Chuo-Ling Chang, and Matthias Grundmann. "MediaPipe Hands: On-device Real-time Hand Tracking". In: *CoRR* abs/2006.10214 (2020). arXiv: 2006.10214. URL: https://arxiv.org/abs/2006.10214.

[38] Xuebai Zhang, Xiaolong Liu, Shyan-Ming Yuan, and Shu-Fan Lin. "Eye Tracking Based Control System for Natural Human-Computer Interaction". In: *Computational Intelligence and Neuroscience* 2017 (Dec. 2017), pp. 1–9. DOI: 10.1155/2017/5739301.

[39] Yifan Zhang, Congqi Cao, Jian Cheng, and Hanqing Lu. "EgoGesture: A New Dataset and Benchmark for Egocentric Hand Gesture Recognition". In: *IEEE Transactions on Multimedia* 20.5 (2018), pp. 1038–1050. DOI: 10.1109/TMM.2018.2808769.