7CCSMPRJ Final Year
Gesture Recognition in the Context of Touchless Interface in Surgery Rooms

Final Project Report

Author: Rakan Zabian

Supervisor: rakan Zabian

Student ID: 1741706

September 3, 2021

Abstract

The applications of gesture recognition in today's world are unknowable. There is a myriad of opportunity to further develop human to computer interaction. This project investigates the application of gesture recognition in the context of a surgical operating room where touchless interface is paramount to maintaining sterility. This project proposes a new method of building such systems, using Google's newly released MediaPipe framework to do so. A model has been built and tested with promising results.

Originality Avowal

I verify that I am the sole author of this report, except where explicitly stated to the contrary. I grant the right to King's College London to make paper and electronic copies of the submitted work for purposes of marking, plagiarism detection and archival, and to upload a copy of the work to Turnitin or another trusted plagiarism detection service. I confirm this report does not exceed 15,000 words.

Rakan Zabian

September 3, 2021

Acknowledgements

I would like to thank Dr. Timothy Neate for his consistent support throughout the development of this project.

Contents

1 Introduction **3**
- 1.1 Project Motivation . . . 3
- 1.2 Report Structure . . . 5

2 Background **7**
- 2.1 Hand Gesture Recognition . . . 7
- 2.2 Hand Gesture Recognition Application Areas . . . 9
- 2.3 Literature Review . . . 11

3 System Design and Specification **14**
- 3.1 System Design . . . 16
- 3.2 MediaPipe . . . 18

4 Methodology and Implementation **20**
- 4.1 Model . . . 21
- 4.2 Hand Tracking Module . . . 23
- 4.3 PyAutoGUI Module . . . 31
- 4.4 Cursor Control . . . 32
- 4.5 Slideshow . . . 35
- 4.6 Zoom . . . 37
- 4.7 Brightness . . . 38
- 4.8 Scroll . . . 39
- 4.9 Speech Recognition . . . 41

5 Results, Analysis and Evaluation **42**
- 5.1 Literature Available User Study . . . 42
- 5.2 Model Performance Analysis . . . 44
- 5.3 Individual Gesture Analyses . . . 46
- 5.4 Discussion and Evaluation . . . 48

6 Conclusion and Future Work **50**

7 Legal, Social, Ethical and Professional Issues **52**
- 7.1 Section Heading . . . 52

References **57**

A User Guide **58**
A.1 Instructions 58

B Source Code **60**
B.1 Instructions 60
B.2 Hand Tracking Module 60
B.3 PyAutoGUI Functions Module 63
B.4 Gesture Recognition Model 65
B.5 GUI 71

Chapter 1

Introduction

1.1 Project Motivation

Human language is a fundamental tool of our being that has enabled us to operate uniquely and substantially effectively in terms of cooperation in comparison to all other species on earth. Our ability to label things and ideas has accelerated our dominance of the natural world due to our enhanced ability to cooperate effectively in extremely large numbers. Let us consider animals for comparison. Chimpanzees build deep relationships based on trust, however, they cannot cooperate in groups consisting of over 200 chimpanzees. They operate flexibly, but not in large numbers. On the other hand, bee communities can consist of over 100,000 bees. They can cooperate in large numbers, but their cooperation is extremely rigid and inflexible, hence their limitation. Humans are the only species on earth which can cooperate deeply, effectively, flexibly, and in staggeringly large numbers. This is because humans connect over a myriad of attributes, much more than those which connect other species. If we consider a set consisting of the intersection of attributes that humans and other species connect through, we could consider the needs for land and food. Outside of that intersection is the bulk of what humans cooperate over, and the glue through which this occurs is language. Language has enabled humans to live in a dual reality: a reality consisting of the seen world, and a reality consisting of unseen, fictional entities. Examples of such entities are large corporations like Apple, countries like the United Kingdom, or concepts like money. These are all labels we have created, which are built on top of deeper layers of labels. The ability to label is in essence: language [11].

For the reasons mentioned above, human language has been extensively researched by philosophers, historians, and scientists. Modern research has been heavily focused on bridg-

ing this extensive analysis and study of language to the world of technology. In essence, if inanimate objects are developed enough to process information directly through human language, then humans will be empowered with previously unknowable scale and leverage. This is already occurring at present day with the progression of fields such as Natural Language Processing (NLP). A recent use case of NLP is Open AI Codex, which is an ambitious mission to develop NLP models which can translate human readable language like English into programming syntax. The paper [5] focused on investigating the potential of large language model training to effectively produce functional bodies of code from natural language docstrings. The models were found to perform promisingly well on a dataset of human written problems. Such literature illustrates the power lying at the intersection of language and technology.

Gestures are focused on transferring some forms of language through non-verbal communication, using body language, or in the context of hand gestures, hand positioning to communicate messages. Due to the recognition of hand gestures having the least limitations for users in human-computer interaction, they provide an excellent form of touchless interaction with computers. Their intuitive and natural communication mode also makes them favourable to humans to use when transferring simple information or commands [4]. Hand gestures can be classified as informative or communicative, depending on the information that is being communicated by the gesture. If the gesture is relaying information about the personality of the speaker, then it is considered to be informative. On the other hand, gestures that are intentionally relayed by a speaker to assist speech or individually communicate a message are considered to be communicative gestures [1, 9, 19]. The development of methods to inform computers through gestures could potentially substantially minimise the use of intermediaries such as mice and keyboards. The use cases of gesture recognition are numerous, and continue to increase with time. This was magnified during the COVID-19 pandemic, which was a harsh use case which demonstrated the power of technology to the world. Throughout the pandemic, people avoided touching machines at all costs. Gesture recognition systems would have been extremely helpful during such circumstances.

Similar to environments during the pandemic, an environment which consistently attempts to maintain sterility and is extremely weary of contamination is the Operating Room (OR) in which surgeons operate. Research by physicians has shown that post-operative contamination can cause sever complications including but not limited to discomfort, pain, hindered recovery and the need for further antibiotic treatments. Not only has it become mandatory by a myriad of regulations to make it standard procedure to be scrubbed and wear aseptic clothing prior to

entering the OR, but also, hospitals have developed sterile zones to maintain aseptic operating procedures. A product of this is the Picture Archive and Communication System (PACS), used by surgeons to navigate the process of operating on a patient. Surgeons use the PACS to obtain maps of obstructed areas, providing intricate details on how to operate on a patient's body with certain tools such as needles. However, the PACS system is controlled through the standard mouse and keyboard interface. This makes it substantially more complex to maintain a sterile environment. As a result of this, literature has been expanding on this topic, from Microsoft investigating the issue in [27] to further holistic analyses on the problem. [24] investigated this in depth to collect surgeon supported analysis on which gestures are best in this problem context.

This project aims to contribute to the literature of this problem by providing a solution using a new method, yet to be utilised in the literature. The majority of current research focuses on using sensors to acquire data, such as the LeapMotion [20] sensor. Machine learning methods have been primary investigated to classify the gestures [6]. However, as researchers we must be consistently alert to new innovations which could reshape the scope of the literature and redirect it. Google's recently developed MediaPipe framework [23] open sourced in 2019, has developed a real time hand detection and tracking system [8] which has been built on top of trained models to detect hands simply through a webcam. We aim to utilise this technology rather than opt for the heavily researched LeapMotion or other sensors as it could offer a far more effective strategy in terms of cost and overall simplicity and efficiency. Its simplicity can also make the methods easily reproducible for other use cases, which is also a target we have for this project. Also, due to the MediaPipe solutions being built on top of trained models which detect hands successfully, we have developed our gesture classifiers with a rule based approach rather than a machine learning model. This will offer a more simplistic approach, that is only possible due to the framework it has been built on top of.

1.2 Report Structure

This project aims to investigate and test a new approach which utilises Google MediaPipe for the development of a touchless user interface with clinical image viewers such as the PACS in the OR.

Chapter 2 will give a background on functional gesture recognition by exploring some computer vision approaches and use cases. The chapter will then flow to the use case which our project is handling. The chapter will end with a literature review on the research of gesture

recognition systems and the methodologies used in our problem context.

Chapter 3 will specify our solution space by describing the framework libraries used in our system, such as MediaPipe and OpenCV. Our system architecture and design will also be investigated.

Chapter 4 will provide an in depth investigation of the methods developed in this project to build the touchless interface system. This will be a rigorous review of the functions and rule classifiers developed for each component of the system and for every gesture.

Chapter 5 will provide an analysis and evaluation of the gesture recognition model and each of the gestures. The chapter will also investigate the literature to explore user studies on other systems, while looking into the demo software we developed to test our gesture recognition model and its usability.

Chapter 6 will summarise the project and discuss its limitations. The potential for future works on the researched topics will also be highlighted.

Chapter 2

Background

2.1 Hand Gesture Recognition

As previously highlighted, hand gestures offer a natural and intuitive way to communicate. As a result of this, research has grown in recent years with the accelerations in the fields and sub-fields of machine learning, deep learning and computer vision to develop methods of interacting with computers using only hand gestures and dismissing hardware input devices. There are primarily two forms of hand gesture recognition which have been extensively research. The first is static hand gesture recognition, which is focused on detecting the shape of a hand individually and statically, without paying heed to postures or sequences of static gestures. This approach is highly effective and rather simple when the gestures are easily recognisable. Use cases for this are touchless gesture control of smart home and car systems, media player systems and image viewing systems, such as the PACS. However, this approach is rather overly simple in more demanding problem contexts, hence, the development of research also in dynamic hand gesture recognition. This form of recognition is focused more on postures which affect the hand positions, alongside an investigation of static gesture sequences, or videos. This has a myriad of use cases from gaming to sign language translating. This project explores the use of static gestures, which consists of all forms of hand gestures produced literally single-handedly, without the assistance of posture. The palm, wrist and fingers are all that combine to convey meaning or information. Sequences of static gestures are explored in this project too, however, as some image viewing controls like zooming are slightly dynamic [28, 46].

A multitude of recognition methods exploring both types of gestures has been researched. From skeleton-based to motion-based recognition, the methods are numerous. This section will

focus on 3 of the most commonly researched methods: glove-based, depth-based, and vision-based recognition [40].

2.1.1 Glove-based Recognition

Glove-based recognition is effective with regards to the low latency, high speed, and less input data requirements to get 3D information on fingers or hands. A data glove is used to collect data on joints in the fingers and the hand [39], which is used to train a Deep Neural Network to classify numerous gestures in real time . The approach is also advantageous in its ability to recognise multiple hand gestures collectively [4]. The severe drawback of this approach, however, is its impracticality in many problem scopes. It is too dependant on the support of expensive and heavy hardware equipment, the gloves, which cannot be consistently worn in many problem scenarios [45] . Notwithstanding this, there has been leaps in the literature of this method and its usability has been demonstrated in some contexts. For example, [17] presents a largely experimented interface which can be operated using a wearable glove. Two DG5-VHand data gloves were used in [36] to translate Arabic Sign Language.

2.1.2 Depth-based Recognition

A methodology which has been highly effective due to its ability to strike a better balance between system and hardware complexity and reusability, and effectiveness in terms of performance, it depth-based recognition. Depth-based solutions are highly robust and have real time recognition and high precision attributes. Depth camera technology explores a new epoch for 3D geometric information acquisition [41] . Due to the construction of distance information the depth camera is capable of extracting, noise factors such as colour and illumination can almost be disregarded. Depth cameras can also use different distance information to separate components of an image, allowing users to potentially separate objects from one another, which is crucial in the case of object barriers and considered not doable by visible light [4]. Another technology which is capable of noise reduction is infrared sensors such as the LeapMotion [26]. However, both depth-based cameras and infrared sensors are expensive to use in comparison to vision-based technologies

2.1.3 Computer Vision-based Recognition

Camera vision based sensors, such as webcams, are commonly researched and utilised due to their capacity to completely eliminate intermediaries and provide contactless human-to-machine

communication. This enables a low cost environment in terms of set up and finance. The low setup cost also makes vision-based approaches highly reproducible in various problem cases. For these reasons, research has tilted towards computer vision-based gesture recognition (VGR). [25] developed a VGR sign language translator. This paper aimed to produce an application which can eliminate intermediaries between sign language communicating individuals and non-sign language communicating individuals. The device enables almost instantaneous gesture recognition and conversion to speech. However, the simplicity of the detection method comes at a cost. VGR systems are susceptible to noise due to its high lighting sensitivity, the effect of occlusions, and complex background issues [29]. Another common defect of VGR systems is that optical flow measurements require a cluster background.

2.2 Hand Gesture Recognition Application Areas

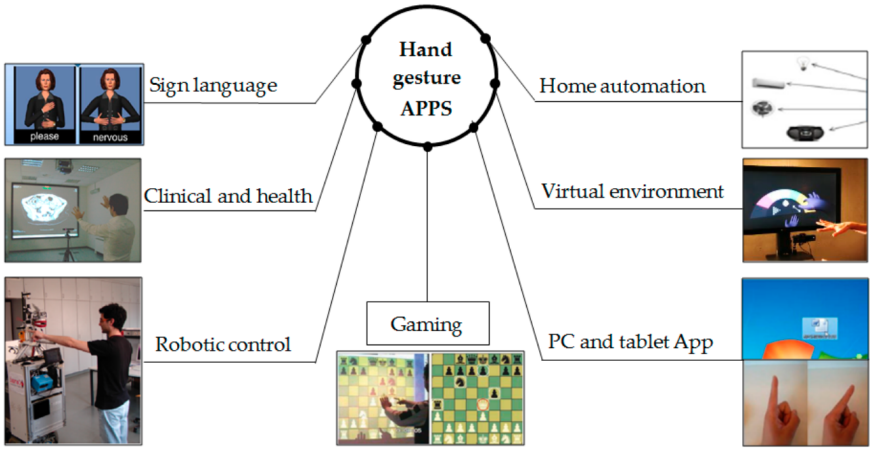

Figure 2.1: Common applications of hand gesture recognition [31]

2.2.1 Personal Computer

Hand gestures recognition can be used to control personal computers and act as alternative input devices to the systems. This enables a user to control the mouse or keyboard, control different functions such as zooming, scrolling, dropping and moving through file hierarchies in a desktop environment [18, 22, 35]. Through this, a plethora of use cases could emerge such as gesture controlled media players, TV screens, slideshow presentations [33], and machine learning applications such as sign language translators as previously discussed.

2.2.2 Robot Control

Robots can be used in a multitude of industry applications and assistive services in stores, entertainment, clinical applications, and sports [2]. This can be critical in the development of systems which the elderly can use for assistance, and the focus on intuitive gesture recognition since the elderly have a common problem of unfamiliarity with modern technology. Some research has developed in this problem case by focusing on the use of computer vision-based gesture recognition to enable a robot and a human to interact in a building [37]. The robotic systems integrate machine learning, artificial intelligence and 'robomechanics' to execute complex tasks which enable the robots to seamlessly interact with their environment [44].

2.2.3 Gaming

Microsoft Kinect Xbox is an excellent example of research on gesture interaction for gaming purposes and entertainment. The system enables a user to interact with a game using hand gestures and posture movements, tracked by the Kinect sensor. The system is also relatively simple to install as it is a screen mounted camera connected to the Xbox device through a cable port [16, 21].

2.2.4 Home Automation

Smart homes can benefit extensively from hand gesture recognition. Gestures can be used to control different housing commands such as media players (TV), air conditioners, lighting, curtains etc [7]. The standard for quality of life can be raised even further. This could also be extended to smart cars, which are already developing gesture controlled systems to control procedures like volume [32].

2.2.5 Clinical and Health

Hand gestures have been extensively researched in the field of health due to the importance of the industry and the potential for its improvement through this research. Some use cases include the use of hand gestures to assist individuals, for example, in the case of wheelchairs [42]. This report focuses on the touchless interaction of the PACS medical image viewer. As discussed prior, a surgeon may need details about a specific part of a patients body, or an entire mapping of the body's structure. For these reasons, surgeons must access MRI, CT and X-ray scans mid-surgery, primarily orthopaedic surgeons [38, 42]. To do this without causing adverse

implications to the OR's mission to maintain a sterile environment is challenging. Hence, the introduction of touchless interaction with these systems. As previously concluded, gesture recognition is effective facilitating the touchless interaction and enabling surgeons to zoom, rotate, crop and go to the next or previous scan or image without any intermediary device such as a mouse.

2.3 Literature Review

Vision-based Gesture Recognition

Papers [4, 31] were paramount to developing our overview of historical and modern literature on hand gesture recognition. [31] was effective in evaluating the benefits and drawbacks of different gesture recognition techniques, while providing in depth descriptions of some vision-based gesture recognition use cases. The paper also evaluated image processing techniques such as the OpenCV library to provide real time interaction in a time efficient manner due to the real time processing. It did highlight some of the drawbacks of this, however, highlighting OpenCV's limitation with regards to illumination variation, background issues, distance limit and multi-gesture problems. Deep learning was investigated as a technique to match gestures to their labels in real-time, however, it was highlighted that they may miss some gestures due to the classification algorithms' accuracy contrast. [4] highlighted 3 difficulties in the context of hand gesture recognition: the difficulty encountered when detecting the gestures, and the difficulty encountered when recognising them. The research proceeds to mention how a hand is composed of over 20 degrees of freedom, making it difficult even to recognise the same gestures between two individuals. They considered multiple approaches to solving these problems, and concluded the depth-based recognition was currently solving problems that vision-based and gloves-based systems couldn't solve. The argument proposed for why gloves are ineffective on the large scale was fairly reasonable. As mentioned previously, gloves are impractical since they cannot be worn in all cases. An example of this is the surgeon context, which cannot really benefit from the method since the surgeon would still have to directly interact with hardware. However, the argument for why vision-based systems struggle to work well in bad conditions can be strongly cross-examined with today's research. The same goes for the argument presented in [31] which highlights that image processing techniques are highly limited when dealing with multiple hands or gestures. This was difficult to cross examine when computer vision was dealing with challenges when trying to implement robust real-time hand perception. This is

because hands lack high contrast patterns and tend to occlude themselves (handshakes) . However, Google's MediaPipe's framework is a strong case for why these challenges are being dealt with and why simple image processing techniques are starting to deal with noise and other issues that [4, 31] mentioned were consistent.

MediaPipe

MediaPipe has offered solutions which provide high-fidelity finger and hand tracking due to the employment of machine learning. Their 'MediaPipe hands' solution is capable of inferring 21 hand landmarks, from a single frame, and tracking them through multiple frames and even through obstruction, resolving some of the issues that the literature we discussed highlighted. The current state of the art relies heavily on powerful desktop environments to detect landmarks, while the MediaPipe solution achieves real-time performance on mobile phones, even scaling to multiple hands [23]. Through the emergence of MediaPipe, developers and researchers have now been given the opportunity to create new applications and research new venues. Hence, our proposition to use MediaPipe to build the contactless PACS system.

Contactless Interaction with Medical Image Viewers in the OR

The initial motivation for this project was [20]. The aim was to outperform the Deep Neural Network and to obtain a higher overall classification accuracy then their 86.46% using a CapsNet, around 10% higher than the accuracy they obtained with a CNN. However, as mentioned previously, out investigation of MediaPipe changed our approach as we found a previously unused method which can also be extended to applications outside of medical image viewing with only few software alterations due to MediaPipe's reproducibility. [30] obtained a mean accuracy of 94.86% using a CNN. Early research [10, 34] demonstrated some systems aimed at enabling gesture control in the OR. Some examples of current research focused on maintaining sterility include 'Gestix'. 'Gestix' is a medical device with a large software display used to browse primarily MRI images. The device is used by multiple medics, but orthopaedics are some of the surgeons which benefit greatly from the software. The device is interesting and relevant as it receives its input commands through hand gestures [38]. [12] proposes a system focused on controlling lights in the OR. The system is feasible and avoids non-sterile contact with light switches. The research also highlighted how gesture commands were faster and more accurate than voice commands. Another interesting example of an OR gesture recognition system is in [14], where the gesture control is investigated in the context of a robot called

'Gestonurse', responsible for the delivery of surgical instruments to the surgeons during their operations mid-surgery. The authors build and deploy a system which recognises hand gestures and commands the robot. In this project, we aim to demonstrate how our model built on top of the Mediapipe framework could produce similar results to far more complex models which use additional hardware and Deep Neural Networks.

Gestures in the PACS Context

The most commonly used gestures in all papers we investigated were cursor hover and click, zoom, previous image, next image. These were the primary gestures used in [20]. After learning more about the PACS, we decided to also add a brightness gesture since brightness is crucial to surgeons navigating the PACS as brightening helps focus on bones while dimming helps focus on organs. In [24], 34 commands were found to be a sufficient number of actions for manipulating the PACS. We have included around half of the commands in our model. Further research could add more, however, we chose to focus on those most used in almost every piece of research we investigated. Those we have focused on can also be extended to control media players and some other use cases, however, this will be revisited in the conclusion.

Chapter 3

System Design and Specification

This chapter aims to review the solution space for the problem we have at hand, reduce the space to our solution, and specify the design of our system. A redefinition of the problem would be as follows: how can we optimise functional gesture recognition methods to enable touchless interaction between surgeons and computers in the OR for effective and contamination-free navigation of an image viewer?

This problem must be approached by developing an application which is capable of perceiving the world around it. To successfully develop such a challenging task, a developer must firstly select an appropriate machine learning algorithm and model which is capable of classifying the information which is fed through the camera through which the application receives its input from. Secondly, the developer must demonstrate their model effectiveness through a series of demos and prototypes. Thirdly, the developer must optimise the trade-off between solution quality and resource consumption to obtain an effective and alluring balance. Finally, problems must be identified and mitigated [23].

As discussed in the Literature Review section, most of the existing volume of bibliography focused on this problem context attempt to tackle the presented challenges using additional sensors or camera systems, such as the LeapMotion sensor, to detect hands, and machine learning approaches to recognise and classify gestures. These methods are effective considering the results achieved in the literature, however, their efficiency can be improved as the additional hardware and software is associated with added complexity and cost with regards to the overall system. This added complexity and cost can be mitigated, improving the system's overall efficiency, by considering more optimal approaches which can attain the effective balance between solution quality and resource consumption. To improve on previous research and solve the pre-

sented developer challenges, this project has combined multiple state-of-the-art technologies, which are yet to be explored in detail in the literature of this problem context, in the quest to propose a more efficient system. Our system uses no additional hardware to detect the gestures, which eliminates all additional costs associated with previous sensor-based systems and allows for a simpler OR setup. This is due to our selection of the MediaPipe framework, open sourced by Google just 2 years prior to this research, which employs Deep Learning methodologies to infer 21 3D landmarks of a hand from a single frame [8]. The MediaPipe framework has been designed to address all the aforementioned developer challenges, from selecting an appropriate ML algorithm to identifying and mitigating problems. MediaPipe allows developers to build reproducible prototypes using its open-source framework . The library has enabled developers to collect sensory data while they focus on their model development. The MediaPipe environment facilitates the improvement of a developers application so that their results are reproducible across a multitude of use cases, platforms and devices [23].

Using OpenCV, the traditionally used library to feed input through a webcam, one is able to feed MediaPipe calculators with raw sensory data, and output 21 hand, 468 face and 33 pose 3D landmarks with just a traditional webcam [3, 15, 43]. Our recognition methods do not require machine learning model development such as training a Deep Neural Network, which is heavily depended on in the literature, due to our manipulation of MediaPipe and OpenCV's functionalities to develop a rule-based classification method instead. This is significantly simpler than using Deep Neural Networks, and in our context, more effective in terms of performance.

Our initial system was entirely gesture based until we investigated Microsoft's critical evaluation and discussion in [27]. This research was integral in our decision making with regards to the system, as we decided to also integrate speech recognition into our model to fine-tune the user experience in the context of surgeons when navigating the PACS or any image viewing system. The research was aimed at deepening the understanding of modalities (voice and gesture) in the problem context at hand, and concluded with the following statement: "In terms of design its not just a question of saying voice is better for this type of functionality and gesture is better for that type of functionality. Rather, there benefits are circumstantial." Based on this, we investigated their analysis to determine which circumstance entails a specific modality. To navigate to a set of images, such as a patient's folder, we will present the option for the user to use voice commands to specify a direct link into the PACS system, alongside the option to just hover the cursor using gesture, and to click on the desired folder. We propose the voice command option as the research also points out that gestures can be cumbersome

when specifying names of image folders (direct links into the PACS). To manipulate images once they have been selected and displayed, our system allows the user to communicate using static gestures, each of which is unique and aims to be intuitive to the user.

The following sections will describe our system holistically by outlining it's design and architecture, and then specify each of it's components by detailing the primary frameworks (Python libraries) that the components depend on.

3.1 System Design

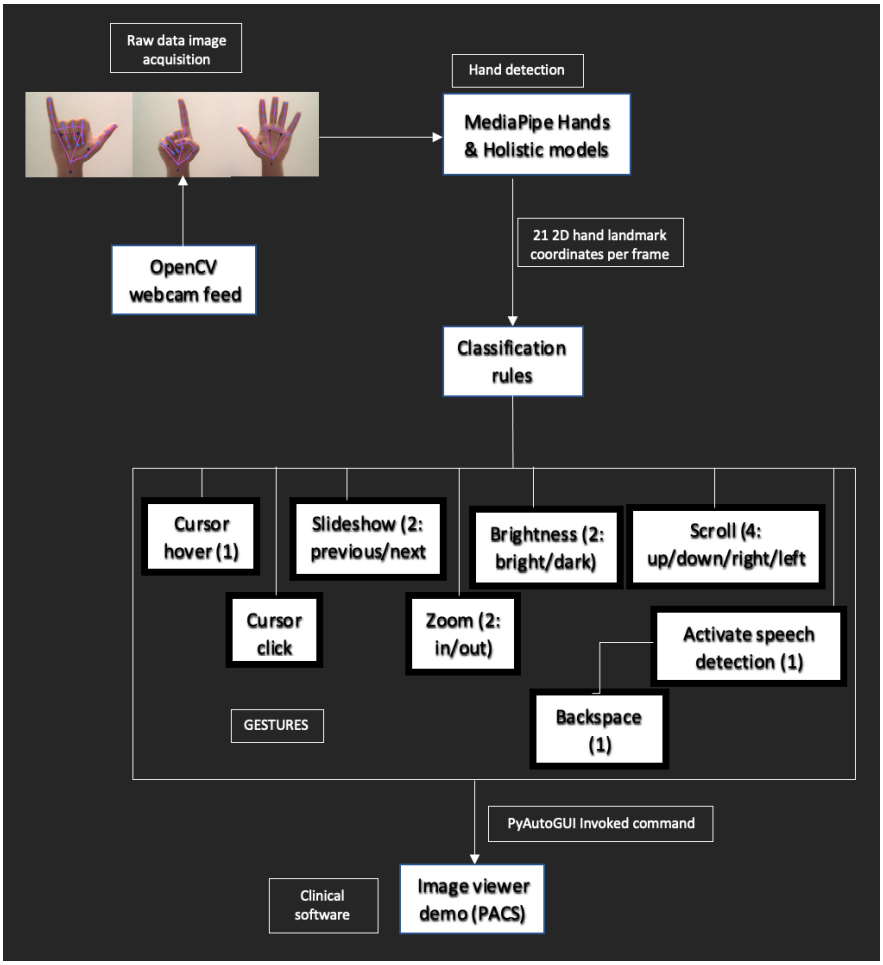

Figure 3.1: Overview of the touchless gesture interface system

The above system has been built to demonstrate the effectiveness of a model we have developed which enables a user to control a system using hand gestures. The model receives raw image data inputs through a webcam using OpenCV. This input is fed into MediaPipe detection models which output hand landmarks. Rules are then used to classify the initial frame

into the appropriate gesture label using the MediaPipe landmarks. Once the rule classifies the frame into a gesture, a PyAutoGUI command invokes an action in a PySimpleGUI Graphical User Interface which is a demo software representing the clinical software applications used for image viewing such as the PACS or Microsoft PowerPointTM. Below are all 14 gestures for reference, however, they will be investigated in the following chapter alongside the classification rules.

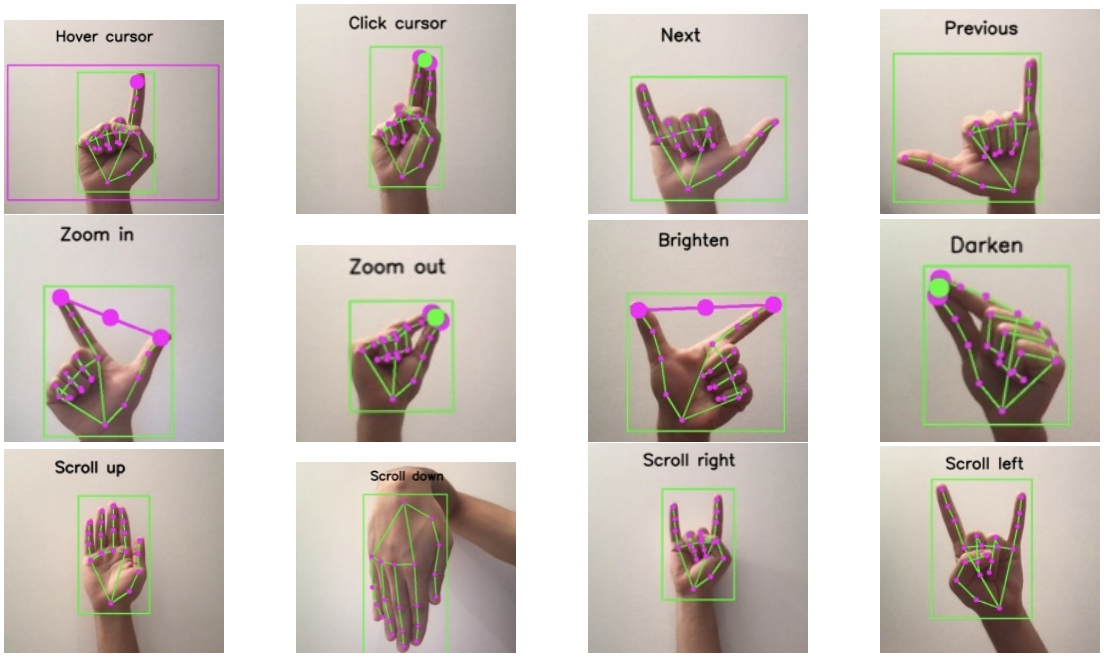

Figure 3.2: Gestures to control mouse commands

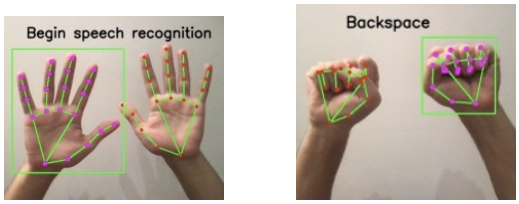

Figure 3.3: Gestures to control keyboard commands

Integrating this model with an actual PACS or other image viewer rather than this system's demo is simple and straightforward. All that is required is the screen coordinates of the buttons which invoke the desired commands. For example, this could be a button in the software which invokes a zoom action. In that case, we would require the screen coordinates of the zoom button in the software. With these coordinates, the model is able to invoke the action by automating a cursor click on that coordinate, which will click the button and invoke the command, without having to manually click the button. Finding the coordinates is straightforward using PyAutoGUI. If speech detection is activated using the first gesture in Figure 3.3, then there are three possibilities: a) the user does not speak within 3 seconds (duration can be altered to the users preference) and the model just passes on, deactivating speech recognition for the current frame. b) the user speaks and the system types the converted speech. c) the user makes a mistake and wants to backspace on the keyboard, which the user can do using the second gesture in Figure 3.3.

In conclusion, the system's components are: the OpenCV webcam feed image input, the MediaPipe detection models, the classification rules used to classify the current image frame data into a gesture label, the SpeechRecogntion model if the speech activation gesture is detected, and the PyAutoGUI invocation of the desired command in an image viewing GUI developed using PySimpleGUI. The relevant libraries are: OpenCV and MediaPipe for hand detection, SpeechRecongition for speech recognition, PyAutoGUI and AutoPy for command automation, and PySimpleGUI for GUI design.

3.2 MediaPipe

The MediaPipe framework was developed to enable the building of perception pipelines aimed at performing inference over arbitrary sensory data. The pipelines can be as a graph of a plethora modular components including model inference, media processing algorithms and data transformations etc. Each of these components acts as a node which receives an input and produces an output. The holistic graph receives sensory data as input, outputting a multitude of MediaPipe solutions which are perceived descriptions such as object localisation, hand landmark and face landmark streams. The solutions used in this project are the MediaPipe Hands and the MediaPipe holistic models. The graph showing the output of the hand landmark models when the hand detection model is triggered is displayed below [23].

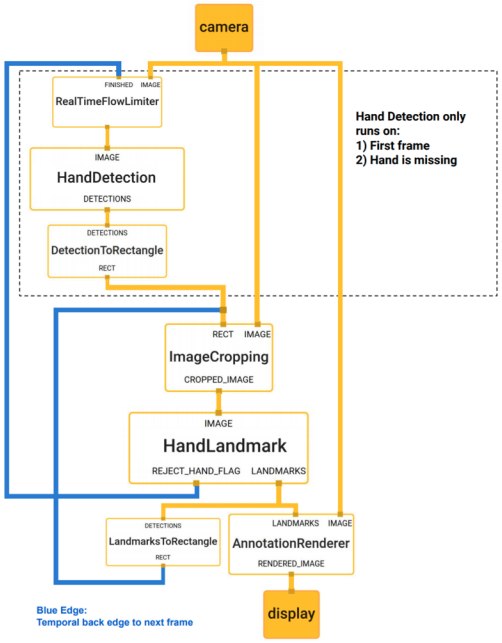

Figure 3.4: MediaPipe's powerful synchronisation components catalyzes high performance and optimal throughput of the Machine Learning pipeline [23].

Chapter 4

Methodology and Implementation

In the previous chapter, we presented an overview of the system's components from OpenCV's image data collection and MediaPipe's models feeding the classification rules with hand landmarks to the demo GUI development with PYSimpleGUI. This chapter will delve deeper into our classification methods, which use Object Oriented Programming in Python to develop rules which classify 14 different gestures using MediaPipe's landmark outputs.

Machine Learning models are effective when the patterns in the data are complex and difficult to represent manually. We train models to develop a mathematical function which is a knowledge representation as this function's parameters contain previously unrecognisable patterns in the data. However, if we can easily observe the patterns in an incoming dataset, which is our case, then structural patterns such as classification rules can be very effective as they may produce 100% correct predictions if our observation (the dataset) represents the totality of evidence. However, rules developed with an incomplete dataset could make inaccurate predictions since information is missing.

The rules used in our methods are in the form of an IF THEN statement. The IF part is the precondition or antecedent and consists of a series of tests. The THEN part is the conclusion or consequent and assigns values to one or more attributes. To make a rule based on ≥ 2 attributes, the statements use conjunctions (and) meaning that all tests must be true to fire the rule.

To detail our implementation, we will firstly provide an overview of our model, and then detail each of its components together with each of the gestures.

4.1 Model

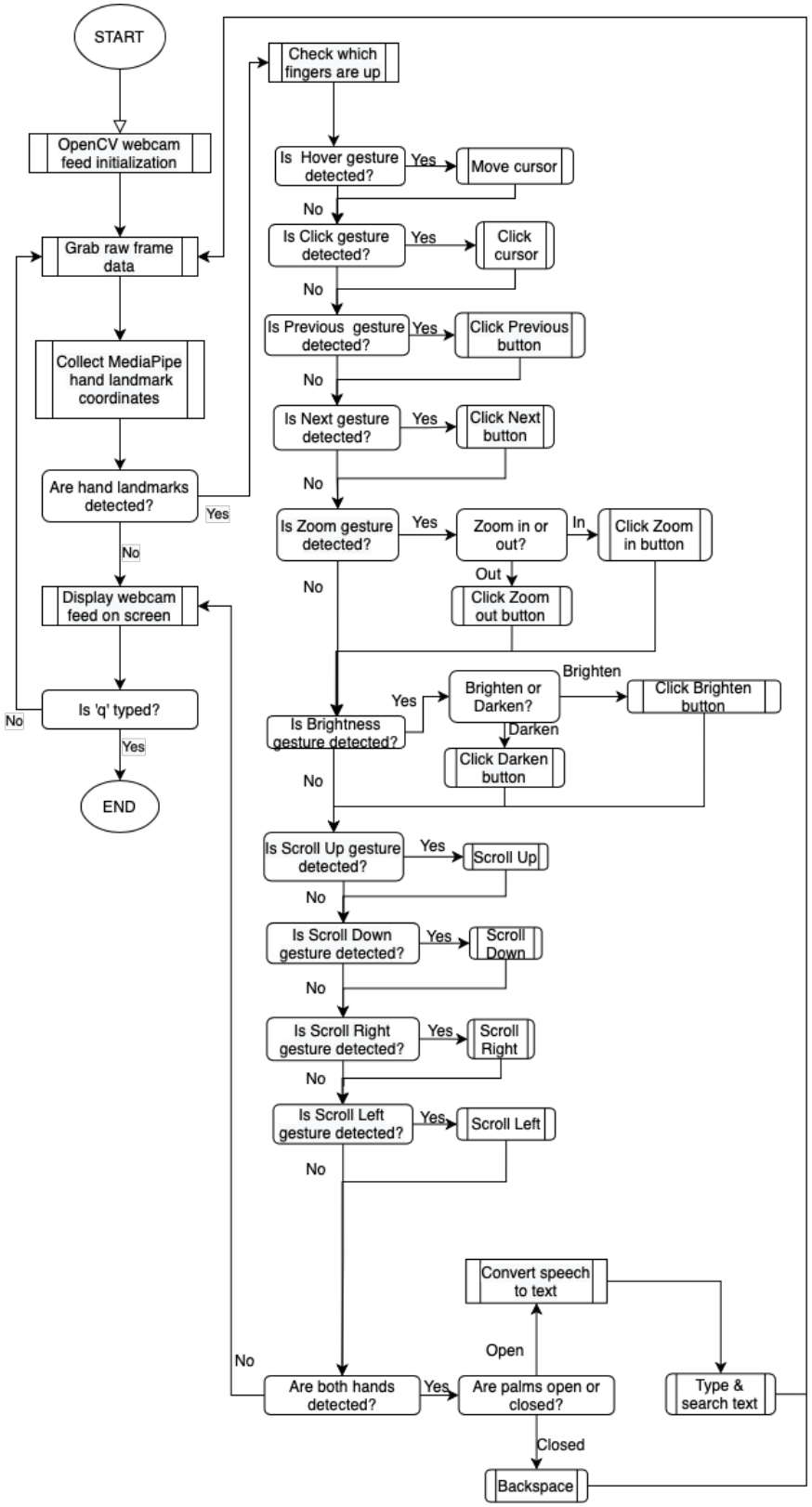

Figure 4.1: Flowchart diagram of our gesture recognition model

The flowchart in Figure 3.4 uses slightly different notation to fit it on one page. Decisions, or branching points, are represented by curved rectangles, processes are represented by sharp outlined rectangles, and the gestures (rule outputs) are represented by curved outlined rectangles. The flowchart illustrates the cycle of the system implementation, providing an overview of our gesture recognition model, from initialisation to each of the 14 gestures. The START node in the flowchart is representative of where all dependencies are imported and an object of a class 'handDetector' belonging to a module 'HandTrackingModule' we have developed and imported is constructed. This class attributes the MediaPipe models to the object and also contains all functions which are used to develop the gesture classification rules which direct the cycle illustrated in the flowchart. This makes it significantly simpler to reuse the MediaPipe models and detection functions for other use cases and recognition models, making our model reproducible and versatile.

After importing the dependencies, setting the parameters and creating the object, the program initialises an OpenCV webcam feed which captures about 10-15 frames per second depending on the system's load. A function of the class is then called, which feeds the frame to the attributed MediaPipe models, outputting the hand landmarks. For each frame, the gesture recognition cycle begins only if the MediaPipe models detect hand landmarks, as illustrated in the flowchart. If hand landmarks are detected, the program calls another function of the class which outputs a 5-item list containing binary values which represent whether a finger is up or down. This list is the basis of the classification rules. The loop then iterates through all the IF THEN statements represented by the questions in the flowchart.

To understand all of this in detail, we will firstly investigate the Hand Tracking module which contains the 'handDetector' class we developed together with another module which contains the PyAutoGUI functions which are called in the gesture recognition model to invoke the commands in the interface. That will underpin our understanding of the gesture classification rules, the logic of which will then be explored for each individual rule together with its gesture.

4.2 Hand Tracking Module

The Hand Tracking module we developed is primarily based on OpenCV and MediaPipe. The handDetector class contained in the module is composed of the initialisation alongside 5 functions: findHands, findPosition, fingersUp, fingersDown, and findDistance. Each of these will be individually investigated in this section.

4.2.1 Initialisation

Algorithm 1 handDetector Initialisation

Input: self | mode=False | maxHands=2 | detectionCon=0.5 | trackCon=0.5

```
// Set handDetector object attributes
```

1 $self.mode \leftarrow mode$

2 $self.maxHands \leftarrow maxHands$

3 $self.detectionCon \leftarrow detectionCon$

4 $self.trackCon \leftarrow trackCon$

5 $self.holistic \leftarrow mp.solutions.holistic.Holistic(min\_detection\_confidence = self.detectionCon, min\_tracking\_confidence = self.trackCon)$

6 $self.mpHands \leftarrow mp.solutions.hands$

7 $self.hands \leftarrow self.mpHands.Hands(self.mode, self.maxHands, self.detectionCon, self.trackCon)$

8 $self.mpDraw \leftarrow mp.solutions.drawing\_utils$

9 $self.tipIds \leftarrow [4, 8, 12, 16, 20]$

The above algorithm is the exact initialisation we composed for the handDetector class. We have provided the initialisation in such detail to provide the reader with an in depth understanding of each of the MediaPipe models, since the initialisation is straightforward and simple to understand considering that it only consists of attribute assignments. This will enable the reader to have a core understanding of the following pseudocodes which will use more intuitive language to call these attributes. The handDetector object is created with 4 parameters: a boolean value determining whether STATIC_IMAGE_MODE is on or not, an integer determining the maximum number of hands that could be detected, and two float arguments determining the detection and tracking confidences.

We initialise maxHands to 2 to enable the detection of two hands simultaneously. The detection and tracking confidence have been initialised to 0.5, which means that the hands model will detect a hand with at least 50% confidence. As for tracking confidence, this means

that the hands model will be tracked based on the initial detection with at least 50% confidence. If the confidence is below this, the hands model will invoke hand detection on the next input image (frame). Higher confidences could be opted for to build a more robust model, however, this will substantially increase latency. Setting mode to true means that MediaPipe will invoke a detection on every image. Hence, mode has been set to False to allow successful hand tracking from image to image so that a hand can be tracked throughout a video, which is our practical context. We opted for a 0.7 detection confidence when creating our handDetector object, while maintaining the default 0.5 tracking confidence. This was determined through trial and error. We initially set it at 0.7 for the zoom and brightness gestures only to avoid unintentional invocations of their commands, however, after testing it for other gestures too we experienced less difficulty navigating the user interface.

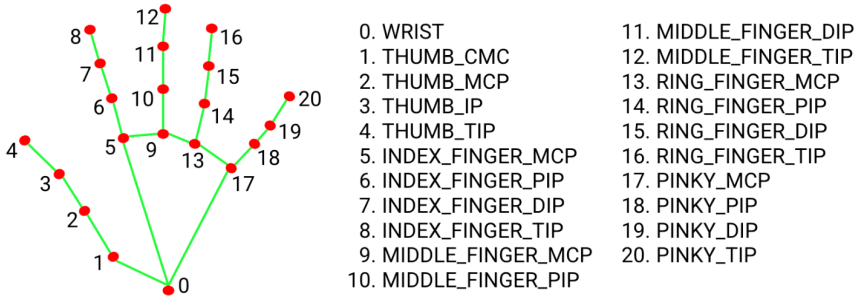

Figure 4.2: MediaPipe Hands solution's 21 hand landmarks

We will refer to the self.hands attribute as the 'hands model' and the self.holistic attribute as the 'holistic model', which are the MediaPipe Hands solution and MediaPipe Holistic solution respectively, investigated in the previous chapter. The holistic model is used to determine which hands are detected, by calling two instances of the model: left_hand_landmarks and right_hand_landmarks. Each is an instance containing all 21 hand landmark coordinates. If these instances are used to define our rules (IF statements), they will be set to a boolean value of True if there exists landmarks and False otherwise. Hence, our enablement to use these instances through the Holistic model to determine which hands are detected. The hands model is a model through which we can output the hand landmarks, illustrated in Figure 4.2, as an instance multi_hand_landmarks which contains the coordinates of whatever hand landmarks are detected.

The hands model has been attributed to the handDetector object by calling the MediaPipe library (mp), the solutions module, the hands module, the Hands class, which enables us to call the 'process' function. This function takes the an RGB (Red-Green-Blue) image its single

argument. The holistic model has been attributed to the handDetector object by calling the MediaPipe library (mp), the solutions module, the holistic module, the holistic class, which enables us to call the 'process' function, which also takes an RGB image input. The mpDraw attribute calls a class from the solutions module which enables us to call multiple functions. The one used in our project is the 'draw_landmarks' function, used to draw the hand landmarks and the connections between them. This function takes 3 arguments: the initial BGR image, the hand landmarks in the current frame, and the hand connections. Both these MediaPipe functions will be referred to when investigating the next handDetector function. We end the initialisation by attributing a list to the handDetector object, which contains the identification numbers for every fingertip (see Figure 4.2).

The initialisation of this class has enabled us to attribute MediaPipe models to our handDetector object and to build functions which will be used to construct our classification rules.

4.2.2 Function 1: findHands

Algorithm 2 findHands

Input: self | image | draw=True

Output: image, self.landmarks

```
imgRGB ← convert image to RGB
self.results ← CALL process(imgRGB) using hands model
self.landmarks ← CALL process(imgRGB) using holistic model
if hands are detected then
    if draw is True then
        for every hand's landmarks do
            CALL draw_landmarks(image, landmarks, HAND_CONNECTIONS)
        end
    end
end
```

The above pseudocode illustrates the findHands function in the naming domain of the problem rather than our actual implementation. As mentioned previously this function alongside the remaining 4 will be described in simpler language to transfer the logic of the problem as swiftly as possible. This function takes the handDetector object, the current frame and a boolean value as arguments. The function creates two new attributes and outputs an annotated image if the boolean value is True and an attribute through which we can obtain the left_hand_landmarks and right_hand_landmarks instances as previously mentioned.

The newly assigned handDetector results attribute will enable us to obtain the multi_hand_landmarks instance which will contain all hand landmarks. The newly assigned handDetector landmarks attribute will enable us to obtain the left and right hand landmarks as mentioned above. Following our attribute assignments, the findHands function returns an annotated image on the OpenCV webcam feed if hands are detected and draw is maintained at its default value, and returns the landmarks attribute.

This function will enable our gesture recognition model to obtain hand landmarks through the hands model, determine which hand is detected through the holistic model, and annotate

the webcam feed. The obtained hand landmarks are paramount to building our next function.

4.2.3 Function 2: findPosition

Algorithm 3 findPosition

Input: self | image | handNo=0 | draw=True

Output: self.lmList, box corner coordinates

```
1  self.lmList ← initialise to [ ]
2  if hands are detected then
3      Get hand landmarks using multi_hand_landmarks instance
4      for every landmark do
5          Get landmark number [ID] and (x, y) coordinates relative to screen shape
6          append (landmark[ID], x, y) to lmList
7      end
8      if draw is True then
9          Draw box around hand
10     end
11 end
```

This function takes the handDetector object, the current frame, an integer and a boolean value as arguments. The function creates a new attribute and outputs an annotated image if the boolean value is True.

If a hand is detected, the lmList attribute will be assigned to a list containing every individual hand landmark's coordinates together with their identification, in the range [0,20] (see Figure 4.2), and their 2D coordinates relative to the image shape. From the perspective of a user observing a screen, the webcam feed coordinate system is illustrated in Figure 4.3. After appending each of the coordinates to the list, the function draws a box enclosing the detected hand for the user's viewing convenience. The findPosition function then returns the hand landmarks list.

This function will enable our gesture recognition model to smoothly navigate individual hand landmarks and annotate the webcam feed. The list containing individual landmark identification and coordinates is paramount to building the following function.

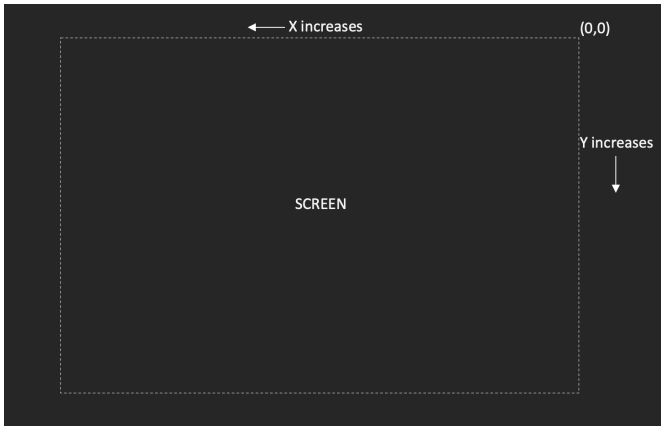

Figure 4.3: MediaPipe Hands solution's 21 hand landmarks

4.2.4 Function 3: fingersUp

Algorithm 4 fingersUp

Input: self

Output: self.fingers

```
1  self.fingers ← initialise to [ ]
2  if landmark[4].X > landmark[3].X then
3  |   append 1 to fingers
4  else
5  |   append 0 to fingers
6  end
7  for every landmark ID in [8,12,16,20] do
8  |   if landmark[ID].Y < landmark[ID-2].Y then
9  |   |   append 1 to fingers
10 |   else
11 |   |   append 0 to fingers
12 |   end
13 end
```

This function takes the handDetector object as its single argument. The function creates a new attribute and outputs this attribute, a list containing five boolean values, each representing whether a finger is up or down.

The thumb's openness is the trickiest to determine due to it being significantly short. The landmark's X coordinates are all close to one another, and the Y coordinates can't really be

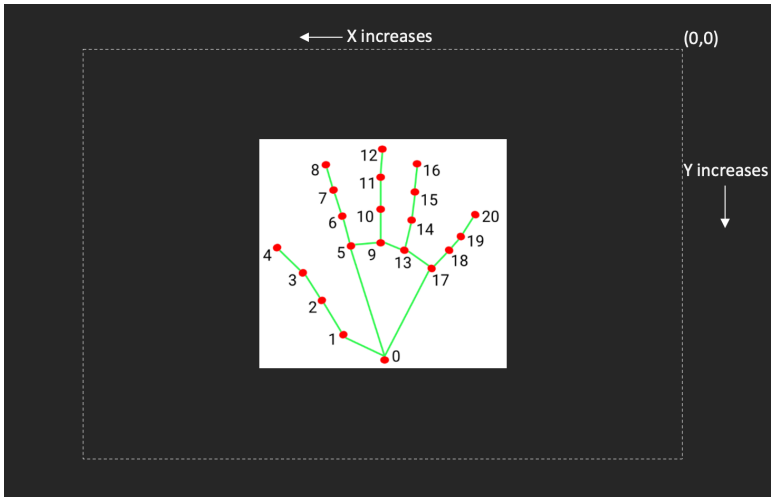

Figure 4.4: Hand landmark coordinates on screen

used to determine whether the thumb is closed or open. Considering Figure 4.4, for a right hand's thumb to be be open relative to the hand's palm, the thumb's tip - landmark 4, should be more to the left than its upper joint (the joint right below the thumb's fingertip) - landmark 3. We chose landmarks 4 and 3 rather than 4 and 2 or 1 to determine whether the thumb is open or not, notwithstanding, the fact that landmark 4 will also be to the left of landmarks 2 and 1 when the thumb is open. This is because when the thumb is closed, landmark 4 can be very close (in terms of X) to landmarks 1 and 2. We concluded that landmark 3 is the best determinant since a closed thumb (inside the palm) will always face the fifth finger (pinky), making landmark 4 consistently to the right of landmark 3, even if it is very close to landmarks 2 and 1. Consequently, if the 4th hand landmark's X coordinate is greater than that of the 3rd hand landmark, then we insert the first boolean value onto the string as 1. As a result of this, 1 would mean that the thumb is open if a right hand is detected, or closed if a left hand is detected, since the illustration is reversed. So, for a palm-open right hand that is detected, fingersUp will return [1,1,1,1,1].

For the remaining 4 boolean values of the list, the function simply iterates through each of the remaining 4 fingertips and appends a 1 for fingertips which are above (lower y) the first joint above the knuckle, and 0 otherwise. We have done this because in some cases the user may raise their index in a 'lazy' manner, which can be visualised as a hook using the index finger. In that case, the fingertip will potentially be lower than the joint right below it. For that reason, we decided to label a finger as down (0) only if its tip is lower than the joint above its knuckle.

This function is the foundation of the rule classifiers in the gesture recognition model, and

will enable the model to determine which fingers are up and whether the hand is open or not. This function will be directly called in 13 of our 14 rules (1 per gesture).

4.2.5 Function 4: fingersDown

Algorithm 5 fingersDown

Input: self

Output: down

```
down ← initialise to 0
for every landmark ID in [1,2,3,4] do
    if landmark[ID].Y > landmark[0].Y then
        down ← down + 1
    end
end
```

This function takes the handDetector object as its single argument. The function then outputs an integer value. If the integer value is 4, then we know that all 4 fingertips are below the wrist line (landmark 0 in Figure 4.4), meaning that the hand is pointing downwards. This function is straightforward as there is only one way for all 4 fingertips to be below the wrist line, which is for the hand to point downwards. This function is the foundation of the rule classifier in the gesture recognition model which classifies our 'scroll down' gesture. As mentioned previously, all 13 other gesture classifiers will use the fingersUp function.

4.2.6 Function 5: findDistance

Algorithm 6 findDistance

Input: self | landmark[ID_1] | landmark[ID_2] | image | draw=True | thickness=15 | radius=3

Output: length | image | [x1,y1,x2,y2,cx,cy]

```
x1, y1 ← get landmark[ID1] coordinates from lmList
x2, y2 ← get landmark[ID2] coordinates from lmList
cx, cy ← calculate midpoint between both landmarks
if draw is True then
    annotate landmarks, connection & midpoint on webcam feed
end
length ← euclidean distance between both landmarks
```

This function takes the handDetector object, two individual hand landmark identifications, the image, a boolean value, and two integers as arguments. The function then outputs the Euclidean distance between the landmarks, an annotated image with a line segment joining the landmarks if the boolean value is set to True, and a list containing the coordinates of the line segment together with its midpoint's coordinates.

This function is called multiple times in our gesture recognition model, to calculate attribute values for the rules which classify some of the gestures including click, zoom and brightness.

4.3 PyAutoGUI Module

Considering the flowchart in Figure 3.4, in the start node we import our system dependencies, set parameters, and create the handDetector object using the imported hand tracking module. The previous section investigated the module in depth, providing a detailed overview of its functions and their usability. As mentioned previously and illustrated in the flowchart, the gesture recognition model then initialises the OpenCV webcam. Following this, it grabs the current frame data and passes it through the handDetector findHands function which outputs the hand landmarks. If hands are detected, the fingersUp function is called to determine which fingers are currently up in the frame. The model then proceeds to determine which gestures are recognised through a series of decision rules we have developed. At the end of each rule's 'yes' branch, we see all 14 commands: from 'move cursor' to 'type & search text'. These commands are triggered in the image viewing system through the PyAutoGUI module, which is described below.

The module imports the PyAutoGUI library and composes 10 functions each of which is called in the gesture recognition model to automate the recognised gesture's allocated command as seen in the flowchart. The 10 gestures for which the functions were built for are: previous, next, zoom in, zoom out, brighten, darken, scroll up, scroll down, scroll right, scroll left (see Figure 3.2). The first 6 gestures exist to command a button click on the image viewing application, the remaining 4 exist to command a scroll in one of the 4 directions possible. The functions for the first 6 gestures automate a cursor click on the screen coordinates which are manually input by the user to create variables in the module. These variables contain the coordinates of the buttons on the screen that are to be clicked using the gestures in the image viewing application. They are displayed below.

1. previousXY = (x,y)

2. nextXY = (x,y)

3. inXY = (x,y)

4. outXY = (x,y)

5. brightenXY = (x,y)

6. darkenXY = (x,y)

The module simply passes each variable listed above to its function which automates the click on the button located at those coordinates. As for the the functions which are called when the last 4 gestures we listed are recognised (scroll gestures), each function simply scrolls 20 clicks in the direction based on the gesture that is recognised using PyAutoGUI's scroll() function. This can also be set according to the users preference. The scroll function can also take 2 more optional arguments, which are (x,y) coordinates of a point on the screen which the cursor will move to before scrolling. This was particularly helpful in our case since it was easy to forget to move the cursor before attempting to scroll when zoomed into an image.

4.4 Cursor Control

In the previous sections, we investigated the flowchart (see Figure 4.1) processes from the START node to checking which fingers are up. We have also investigated the system's dependencies. The following sections will investigate each rule individually. Cursor control uses AutoPy to automate cursor commands rather than PyAutoGUI simply because it was preferred when testing with the demo, however, both can be used and have the same functionality.

4.4.1 Hover

As mentioned prior, in the START node, an initialisation of parameters occurs. An explanation of the rule's inputs is below followed by its pseudocode.

1. fingers, the output of the fingersUp function, is the boolean values list determining which fingers are up. The values label the fingers in this order: [thumb,index,middle,ring,pinky]

2. lmList,the output of the findPosition function, is the list of every hand landmarks (ID, X, Y)

3. (x1,y1) are the index fingertip coordinates found using lmList

4. frameR, set to 100, is a frame reduction factor which will help map the finger movement onto the cursor
5. (wCam ,hCam), set to (640,480), is the OpenCV webcam feed shape
6. (wScr,hScr) are the screen dimensions which are found using AutoPy
7. smoothening, set to 5, is a factor used to slow the cursor automation down
8. (plocX,plocY), initialised to (0,0), is the initial cursor location
9. (clocX,clocY), initialised to (0,0), is the current cursor location
10. image is the image output from the findHands function

Algorithm 7 Hover

Input: fingers | lmList | (x1,y1) | frameR | wCam | hCam | wScr | hScr | smoothening | plocX | plocY | clocX | clocY | image

1 **if** *fingers excluding thumb = [1,0,0,0]* **then**
2 draw rectangle using wCam,hCam & frameR
3 find index fingertip coordinates relative to rectangle
4 interpolate new coordinates with respect to current coordinates, rectangle & screen shapes
5 update cloc using ploc and interpolated values and smoothen using smoothening
6 move cursor to current location (cloc) using autopy
7 annotate image
8 *ploc* ← *cloc*
9 **end**

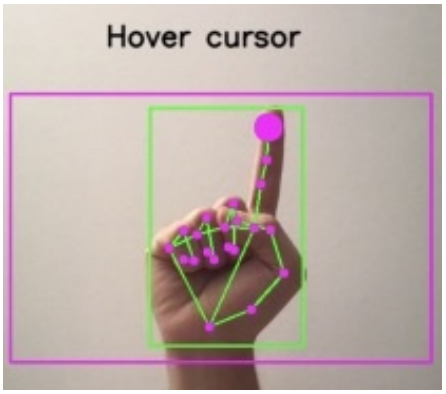

Figure 4.5: Hover gesture

The Hover function is run IF the index finger is the only finger raised excluding the thumb. The gesture for this command is shown in Figure 4.5. We have excluded the thumb because the only gesture which is similar the the hover is the zoom. The zoom command is only invoked if the thumb is down, so a false zoom cannot occur. The only possibility of falsity is a false hover when zooming. However, this is avoided when using the actual system because the overall system is busy with the zoom command so a false hover output has no effect on it. The cursor can also be controlled using the right or left hand, because, the mirroring effect has no bearing on the rule since the thumb is excluded.

If the condition is met, the algorithm proceeds to build a rectangle which will act like a screen. This makes it significantly easier for the user to control the cursor. The algorithm then interpolates new coordinates which represent the finger's coordinates relative to the screen with respect to the finger coordinates relative to the rectangle. This allows it to predict where the cursor is required to be. Finally, the algorithm invokes autopy to move the cursor to the location.

4.4.2 Click

An explanation of the rule's inputs which have not been previously explained is below followed by its pseudocode.

1. detector is the object we create at the initialisation which belongs to the handDetector class, through which we can call all the functions

2. click_threshold is a threshold value we set to 40 which we use to determine the recognition based on how close the index and middle fingertips are to each other

Algorithm 8 Click

Input: fingers | detector | image | click_threshold

```
1 if fingers excluding thumb = [1,1,0,0] then
2     length, image, lineInfo ←CALL detector findDistance function
3     if length < click_threshold then
4         annotate image automate cursor click using autopy
5     end
6 end
```

The Click function is run IF the index and middle fingers are the only fingers that are raised

excluding the thumb. The gesture for this command is shown in Figure 4.6. We have excluded the thumb because there is no risk of falsity. This also enables the user to control the cursor using any any hand, because, the mirroring effect has no bearing on the rule since the thumb is excluded.

If the condition is met, the algorithm proceeds to call the findDistance function which outputs the length between the index and middle fingertips, an annotated image with a line between the two fingertips, and a list containing the coordinates of the fingertips and the midpoint between them. If the length is below the threshold, the algorithm proceeds to invoke autopy to automate the click and modify the colour of the annotation.

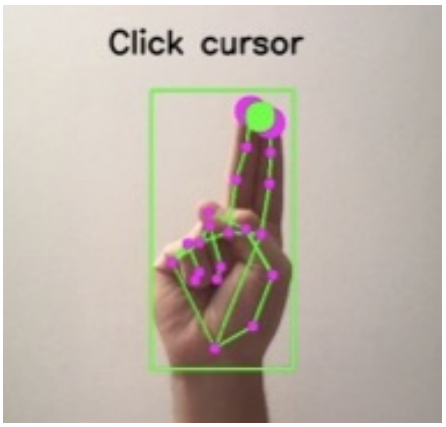

Figure 4.6: Click gesture

4.5 Slideshow

An explanation of the rule's inputs which have not been previously explained is below followed by its pseudocode.

1. landmarks is a boolean value determining whether a specific hand is detected or not

Algorithm 9 Slideshow

```
Input: landmarks | fingers
  if left hand is detected AND fingers = [0,0,0,0,1] then
    CALL pyautogui previous function
    sleep for 1 second
  else if right hand is detected AND fingers = [1,0,0,0,1] then
    CALL pyautogui next function
    sleep for 1 second
  end if
```

The Previous function is run IF the left hand is detected, the thumb is open and the pinky

finger is raised while all other fingers are down. The thumb is set to 0 since the left hand outputs a 0 rather than a 1 for an open thumb. The gesture for this command is shown in Figure 4.7. If the condition is met, the algorithm proceeds to call the previous function from the PyAutoGUI module. As discussed previously, the function takes the previous button's screen coordinates in the GUI as its single argument, and automates a click on that location. This successfully allows the user to click the button using the gesture. The system sleeps after automating the command to avoid a rapid continuous invocation of the command. This allows the user some time to make new gestures are move through the images at a steady pace.

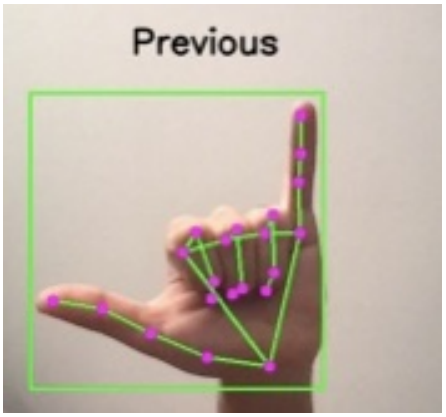

Figure 4.7: Previous gesture

The Next function is run IF the left hand is not detected AND the right hand is detected AND the thumb is open and the pinky finger is raised while all other fingers are down. The gesture for this command is shown in Figure 4.8. If the condition is met, the algorithm proceeds to call the next function from the PyAutoGUI module, automating the next command.

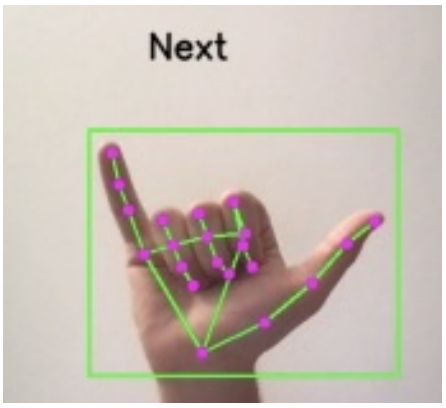

Figure 4.8: Next gesture

4.6 Zoom

An explanation of the rule's inputs which have not been previously explained is below followed by its pseudocode.

1. zoom_threshold is a threshold value we set to 50 which we use to determine the recognition based on how close the thumb and index fingertips are to each other

Algorithm 10 Zoom

Input: landmarks | fingers | detector | zoom_threshold | image

```
if right hand is detected AND fingers = [1,1,0,0,0] then
    length, image, lineInfo ←CALL detector findDistance function
    if length > zoom_threshold then
        CALL pyautogui zoomOut function
        sleep for 1 second
        annotate image
    else
        CALL pyautogui zoomOut function
        sleep for 1 second
    end
end
```

The zoom function is run if the right hand is detected AND the index finger is raised while the thumb is open. If the condition is met, the algorithm proceeds to call the findDistance function which outputs the length between the thumb and index fingertips, an annotated image with a line between the two fingertips, and a list containing the coordinates of the fingertips and the midpoint between them. If the length is above the threshold, the algorithm proceeds to call the zoom in function from the PyAutoGUI module, automating the zoom in command. If the length is below the threshold, the algorithm proceeds to call the zoom out function from the PyAutoGUI module, automating the zoom out command.

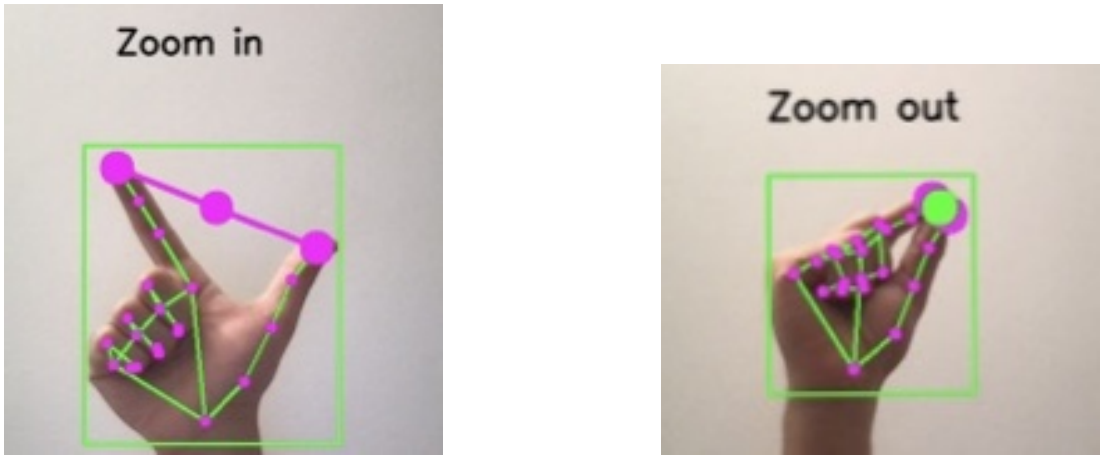

Figure 4.9: Zoom gestures

4.7 Brightness

Algorithm 11 Brightness

Input: landmarks | fingers | detector | zoom_threshold | image

```
1  if left hand is detected AND fingers = [0,1,0,0,0] then
2      length, image, lineInfo ←CALL detector findDistance function
3      if length > zoom_threshold then
4          CALL pyautogui Brighten function
5          sleep for 1 second
6          annotate image
7      else
8          CALL pyautogui Darken function
9          sleep for 1 second
10     end
11 end
```

The brightness function is run if the left hand is detected AND the index finger is raised while the thumb is open. If the condition is met, the algorithm proceeds to call the findDistance function which outputs the length between the thumb and index fingertips, an annotated image with a line between the two fingertips, and a list containing the coordinates of the fingertips and the midpoint between them. If the length is above the threshold, the algorithm proceeds to call the brighten function from the PyAutoGUI module, automating the brighten command. If the length is below the threshold, the algorithm proceeds to call the darken function from

the PyAutoGUI module, automating the darken command.

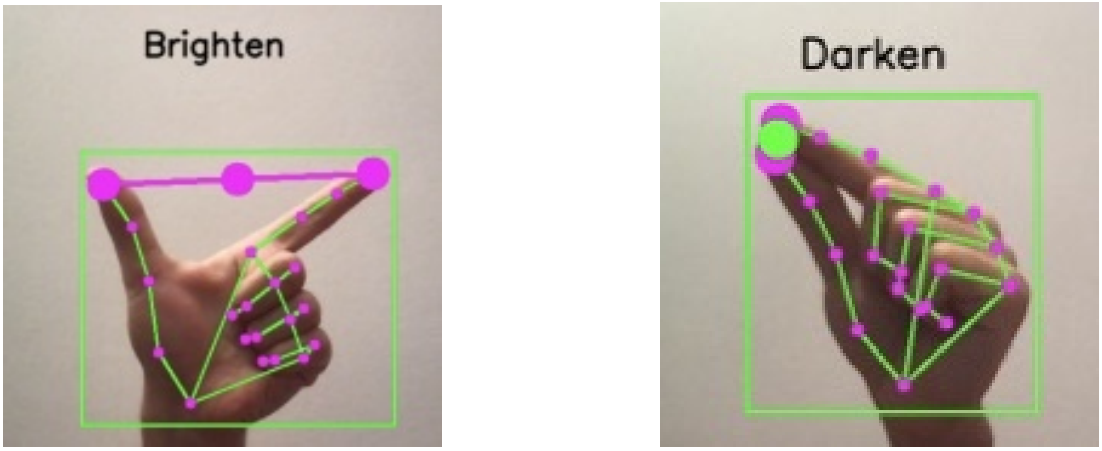

Figure 4.10: Brightness gestures

4.8 Scroll

Algorithm 12 Scrolling

Input: fingers | detector
  down ←**CALL** detector fingersDown function
  if fingers excluding thumb = [1,1,1,1] **then**
    CALL pyautogui scrollUp function
    sleep for 1 second
  else if down = 4 **then**
    CALL pyautogui scrollDown function
    sleep for 1 second
  else if right hand is detected AND fingers = [0,1,0,0,1] **then**
    CALL pyautogui scrollRight function
    sleep for 1 second
  else if left hand is detected AND fingers = [1,1,0,0,1] **then**
    CALL pyautogui scrollLeft function
    sleep for 1 second
  end if

Firstly, the fingersDown function is called. As mentioned prior, this function outputs an integer stating how many fingers are facing down. If all 4 fingers excluding the thumb are raised, in other words, if the palm is open, the algorithm proceeds to call the scrollUp function from the PyAutoGUI module, automating the upwards scrolling command. If all 4 fingers excluding the thumb are facing down, the algorithm proceeds to call the scrollDown function from the PyAutoGUI module, automating the downwards scrolling command. If the right hand is detected and the index and pinky only are raised, the algorithm proceeds to call the scrollRight function from the PyAutoGUI module, automating the horizontal scrolling

command to the right. If the left hand is detected and the index and pinky only are raised, the algorithm proceeds to call the scrollLeft function from the PyAutoGUI module, automating the horizontal scrolling command to the left.

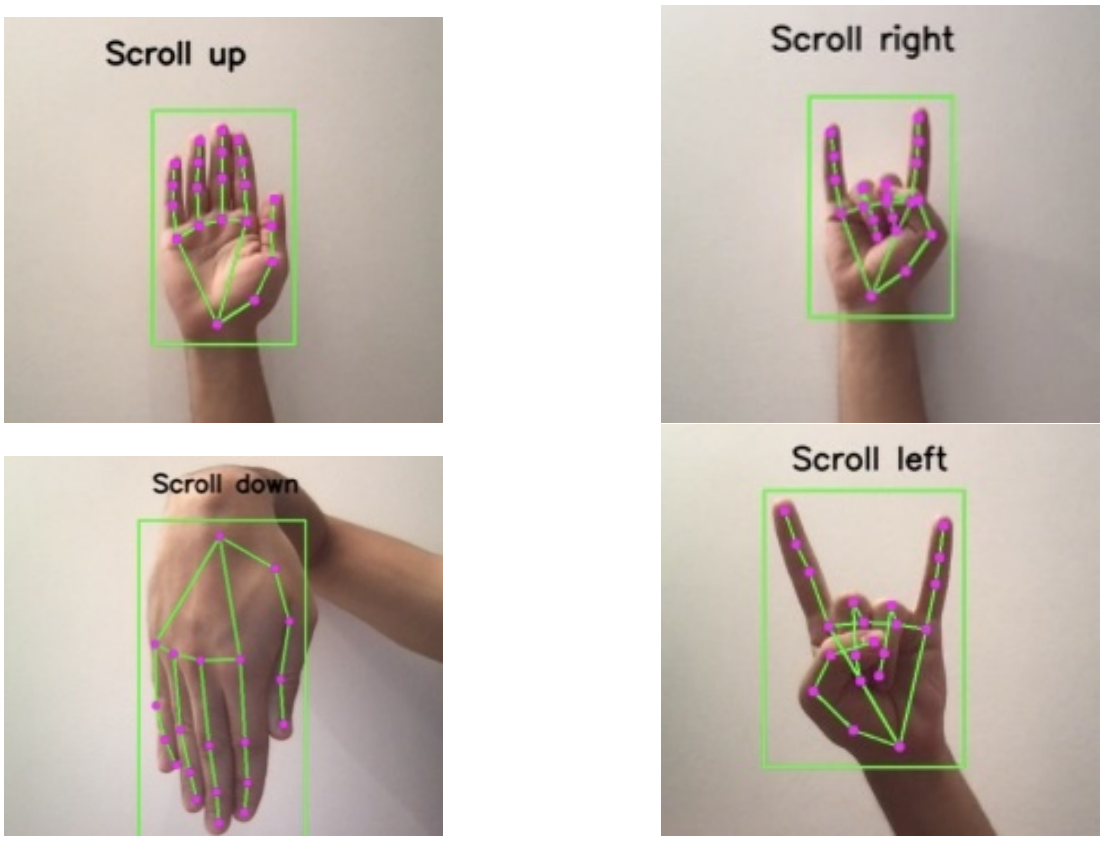

Figure 4.11: Scrolling gestures

4.9 Speech Recognition

An explanation of the rule's inputs which have not been previously explained is below followed by its pseudocode.

1. speech_duration is an integer representing a duration in seconds set to 3

Algorithm 13 Scrolling

```
Input: landmarks | fingers | speech_duration
  if both hands are detected then
    if fingers excluding the thumb = [0,0,0,0] then
      CALL pyautogui press('backspace') function
    else if fingers excluding the thumb = [1,1,1,1] then
      print out 'typing'
      r ← initialise speech recognition object
      source ← set microphone as source
      audio_data ← record speech with source
      if speech is detected then
        text ← convert audio_data to text with recognize_google function of r
        CALL pyautogui typewrite(text) function and press(enter) function
      end if
    end if
  end if
```

The speech recognition algorithm is run if both hands are detected. If a closed fist is detected, the algorithm calls a PyAutoGUI function which automates a keyboard click on the button that is passed as an argument. So, the algorithm presses the backspace key and deletes whatever was typed. If an open fist is detected, the algorithm initialises an object from the Recognizer class of the speech recognition, which allows speech detection using the microphone. The speech detection is set for 3 seconds, allowing the user some time to speak. The user can alter this to their preference. Following this, the recognize_google function from the Recognizer class is called to convert the speech into text. Finally, the algorithm calls a PyAutoGUI function which automates keyboard typing of whatever string is passed as input to the function.

Figure 4.12: Speech recognition activation gestures

Chapter 5

Results, Analysis and Evaluation

This chapter will begin by highlighting some points from a user study conducted in the available literature. Following this, the chapter will give an overview analysis on how the model runs in terms of complexity. The gestures will then be individually discussed and their usability will be reviewed. Finally, the chapter will conclude with an overview evaluation of the system holistically and how the demo is the primary method used to test our model.

5.1 Literature Available User Study

[13] developed a system which integrates with the PACS and can be controlled using hand gestures detected using a LeapMotion sensor. Considering the drastic contrast in approach, we can only consider user study results which are relevant to our system. Also, we must be weary of ethical concerns. Hence, no results or intricate details from that user study will be visually displayed in this report, we will only discuss the impact of the results.

The user study consisted of participants with diverse backgrounds in the medical field consisting of radiologists, surgeons, and other technicians and assistants that support the doctors inside the sterile environment [13]. Approximately 69% of participants logged that they always use imagings during a procedure. Approximately 88% logged that they consider patient's imagings during a procedure to be between important and very important. A crucial and relevant result is that all participants concluded that if they could manipulate imagings through a sterile fashion, they would be more inclined to view the imagings. This is critical as it conveys, based on this study, how beneficial touchless systems could be if fully integrated into medical institutions. This provides an adequate motive to further improve this project's research so

that it could potentially integrate with an actual institutions imagings software.

The users in [13] were given a demonstration of the software and then enabled to test it, before being given a survey to rate their firsthand experience with it. It would be insensible to assume such results would apply to our system, unlike the previous results which were more generic. However, some of the detailed results provide an opportunity for insight into how our system is sufficient or could be improved. For example, the system in [13] consisted of 7 command areas: cursor navigation, two-finger image scrolling, two-finger image scrolling, Three-Finger Window Width (WW) and Window Level (WL), Four-Finger Image Pan, Window Width and Window Level Presets, and Next and Previous Image Scroll. Approximately 81% of the participants logged that the system consists of all necessary manipulation tools during a procedure. The system in this project consists of 6 command areas as well: cursor navigation, slideshow, zoom, brightness, scroll, and typing. Below is a venn diagram illustrating the crossover gestures.

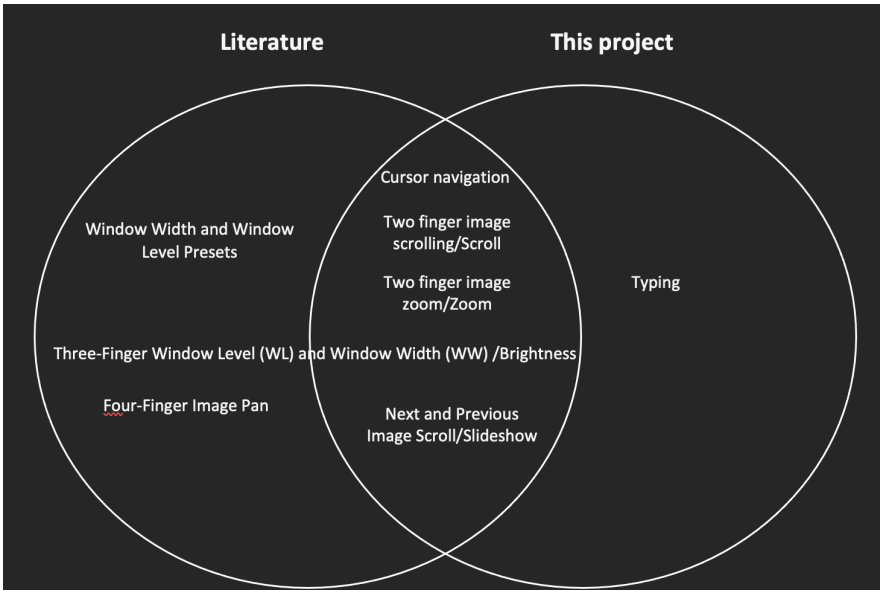

Figure 5.1: Project crossover with [13]

We see that the only command areas our project misses are window width and window level presets, which are gestures which enable a small change in brightness or contrast rather than a large one. So, in essence, it allows the user to change the brightness scale through gesture. That is an excellent command and would be useful in the case of a doctor which wants to review a region of interest in depth, and only modify brightness and contrast very slightly. Our system could theoretically do this if we just decrease the brightness factor in the GUI, but the brightness would then be very slow. So our system is missing such a function, and that is

most definitely an excellent point of focus for future improvements of this project. Alongside this, the system is missing the three window level, which represents contrast. We have utilised brightness, but not contrast. This would be an easy addition to the system with simply one more gesture for contrast. We considered this but struggled to demonstrate it in the demo GUI we built, so we opted to only include what we could demonstrate. This would also be a great focus point for future improvement of this project. As for the image pan feature, its essence is to allow the user to pan from one point of a zoomed in image to another (which the user cannot observe without panning). However, we have utilised the scroll feature to implement that same effect. Panning does speed it up though, so it would be a good point of focus for further improvement. As for speech recognition, [13] did not include such a functionality. Most of the current literature omits speech too, however, as discussed before and investigated in Microsoft's [27], voice commands could substantially improve the system, especially when dealing with a large database.

If we consider contrast and brightness to be two separate commands, then our system and the user experimented system share approximately 77% of the command areas in in the experimented system. Considering the 81% agreement that the user experimented system consists of all necessary manipulation tools during a procedure, , it is fair to say that our project can still be improved even further.

5.2 Model Performance Analysis

5.2.1 MediaPipe Models

Since MediaPipe's performance benchmarking is yet to be open sourced (see: MediaPipe Performance Benchmarking), we have no numerical analysis for the MediaPipe models yet. However, our model was tested rigorously using the GUI PACS demo. The demo was developed to test our model, hence, we continuously referred back to it for insight. We tried multiple MediaPipe solutions to output the hand landmarks, and found the Hands solution to be the most effective based on our user experience with the GUI and the overall cleanness of the code. However, for the model used to detect which hand is in frame, we found that MediaPipe Holistic model's left_hand_landmarks and right_hand_landmarks, each of which is an instance containing all hand landmark coordinates, far more effective compared to the MediaPipe Hands model's MULTI_HANDEDNESS instance, which determines which hand is detected through a string output. For the Holistic model approach, we would just use the instances as booleans

in the rules, since inputting into an IF statement will convert them to True if landmarks exist and False otherwise. Using the Hands model approach consisted of more bugging and overall latency to the system when tested. Hence, we opted for the Holistic model to determine which hands are detected. We also found that both the Hands model's other instances such as multi_hand_landmarks and the holistic model's landmark instances would output landmarks significantly before the multi$_h$*andednesswouldoutputthestring*.

5.2.2 Gesture Recognition Model

Algorithm 14 Gesture recognition model

do
    CALL findHands handDetector function
    CALL findPosition handDetector function
    if *hands are detected* **then**
        CALL fingersUp handDetector function
        CALL fingersDown handDetector function
    end
while *webcam is open*;

The above pseudocode is a skeleton of the gesture recognition model, including only loops. So, only the handDetector class functions that have been called in the model which contain loops, have been included. Considering that hands are detected, the findHands function runs a for loop which runs from 0-1 or 0-2 depending on how many hands are detected since the maximum number of lists is two since only two hands can be detected in this case. The findPosition function runs a for loop from 0-20 consistently since there are 21 landmarks per hand. The fingersUp and fingersDown functions each run a for loop from 1-4 consistently. The while loop engulfing these functions runs n times if n is the number of frames captured. Hence, the exact worse case running time of the model is $(n \times 2 \times 20 \times 4 \times 4)$. Considering the consistency of the upper bound of each of the for loops inside each handDetector function, the constants 2,20,4,4 will be disregarded as $n \to \infty$ in the worst case. Therefore, the underlying computational complexity of the Gesture recognition model , where n is the number of frames, is $O(n)$. The model runs in linear time, making the implementation quick.

5.3 Individual Gesture Analyses

5.3.1 Cursor Control

The cursor control gestures are both dynamic gestures since there is a consistent tracking of the finger positions over a sequence of frames. The hover gesture is intuitive and easy to use, however, some individuals may prefer different smoothening factors. This is up to the user to decide, in our implementation we found 5 to be a good factor based on our experience with the GUI. We did not experience any difficulty navigating using the hover gesture. With regards to the click gesture, it sometimes requires effort to position the middle without moving the index. However, in terms of effectiveness, both worked well. It may require a user a few takes to get familiar with it, but in our case it was a quick success.

5.3.2 Slideshow

The slideshow gestures are both static gestures since they are implemented simply with a detection of the finger positions. The right and left hand marking is intuitive since next is normally associated with the right and the same for previous. However, the finger positions may not be necessarily intuitive. They are simple though, and their implementation in the GUI was the easiest alongside some of the scrolling gestures, since they are static and easy to recognise in our model. They should familiarise quickly and work extremely well with every user.

5.3.3 Zoom and Brightness

The zoom and brightness gestures are both dynamic gestures since there is a consistent tracking of the finger positions over a sequence of frames. The right and left hand marking are not necessarily intuitive, since there is no association between the commands and specific directions. The zooming gesture is intuitive, since it mimics the gesture required on a cursor touchpad to zoom in/out. As for the brightness, it is not necessarily intuitive, however, it is easily adapted to considering its similarity to the zoom, in the sense that zooming in and brightening require the same finger positioning, and same for zooming out and darkening. Also, some individuals may prefer different zoom thresholds. This is up to the user to decide, in our implementation we found 50 to be a good threshold based on our finger proportionality. We sometimes experienced mild difficulty, in the sense that we would have to slightly re-position our hands in the frame to invoke the command. This is because, as previously mentioned, the thumb is very small and

is the most difficult in terms of determining whether it is closed or open. Moreover, there is some improvement that can be made to the gesture. We initially planned to scale the length between the fingertips and use it to determine the setting on the zoom and brightness scales. However, we struggled to find a way to demonstrate this using the GUI, so we opted for the more static approach which was easy to implement since it was just the automation of a button click. Future improvement on this could however link the length to the scale and allow the user to directly control the zoom function. This could be done very easily if a library is developed or found which can control zoom. Such libraries exist for volume control, hence, this gesture is very versatile. By importing the volume controlling libraries and scaling them based on the length between the fingertips, we could use our model to essentially control a media player such as a TV. Overall, in terms of effectiveness, the gesture worked well. It may require a user a few takes to get familiar with it. In our case it took a few takes to get the complete intuition.

5.3.4 Scroll

The scroll gestures are all static gestures since they are implemented simply with a detection of the finger positions. The right and left hand marking is intuitive since scrolling to the right is associated with the right and the same for scrolling to the left. However, the finger positions may not be necessarily intuitive. They are simple though, and their implementation in the GUI was the easiest alongside the slideshow gestures, since they are static and easy to recognise in our model. They should familiarise quickly and work extremely well with every user. They do take a bit of time to invoke scrolling though, but it is a second or two at most, we have set time.sleeps in the model to avoid commands running all the time, users can modify the periods to their preferences though. Also, we forgot to move the cursor inside the frame of the image when we wanted to scroll, since we would control the GUI without necessarily using the cursor (next, previous etc). As a result of this, nothing would happen when we make the scroll gestures. Hence, we decided to input the coordinates in the scroll functions in the PyAutoGUI module which automatically moves the mouse inside the frame when a scroll gesture is detected. Besides forgetting the put the cursor in frame, which we have resolved, and having to wait for a second before scrolling is invoked, which is modifiable by the user, the gestures are straightforward and easy to use.

5.3.5 Speech Recognition

The speech recognition gestures are both are both static gestures since they are implemented simply with a detection of both hands and specified finger positions. They are not necessarily intuitive, however, there is some relation between two open palms and typing. Either way, they are easy to familiarise with since they are the only two gestures requiring both hands and are simply open palms and closed fists. As for the speech recognition itself, it is probably the least smooth action in the overall model. The 3 second duration, modifiable by the user, is hard to gauge when controlling a software like the GUI. It takes a few tries to get the word recognised since a user will tend to speak before or after the duration. Also, backspacing is easy to do since once the gesture is recognised, it will continuously backspace, fairly quickly, until the gesture is not detected anymore. Backspacing a specific letter may be tricky, however, as navigating the cursor between letter requires some effort, and the backspace may delete multiple letters before the user can change the gesture. As for the speech detection itself, it works fine, similarly to mobile application AI assistants like siri, since the model is developed by google.

5.3.6 Resetting

If the wrong commands are being invoked, which does occur rarely, the user is advised to either close their fist (as long as only one hand is in frame), or remove all hands from the frame. This gives the system time to reset and the user can once again invoke commands.

5.4 Discussion and Evaluation

The gestures were concluded to be generally easy to use based on our experience testing the model with the GUI. Some are less intuitive than others, however, they are quick to grasp. The zoom/brightness gesture in particular can be utilised for any command which involves a scale such as zoom, brightness, contrast, and volume. This opens the door to controlling media players such as smart android TVs, Apple TVs, or gaming consoles. The model performs well since it runs in linear time, enabling real time gesture detection. Our system has shown to compare well to a system which yielded promising results in a user study consisting of medical professionals. We have highlighted further improvements that can be made to the model such as adding a contrast control gesture. This could simply be to raise the pinky finger while performing the brightness gesture. Our system has demonstrated through testing with the GUI that building lower cost systems can still yield good results and working prototypes, which yields

a better balance in the resource consumption and system effectiveness tradeoff. Moreover, our methods were void of any models which are trained. This was due to the framework we used which is already a composition of rigorous ML models. One drawback of this was that we could not numerically measure performance until MediaPipe release their performance benchmarking solutions. However, as discussed in the previous chapter's introduction page, rules produce 100% accuracy if the dataset represents the totality of evidence (future data). To transfer this logic to our context, the rules will predict with 100% accuracy if the user makes the gestures with 100% concordance with regards to their hand, finger and fingertip locations and the duration they maintain a gesture for, and if MediaPipe detects the hands. Moreover, we were weary of overpromising with the model since we continuously tested using the GUI we developed. This method of testing was effective because we were always able to quickly identify and assess how well the overall model and its components were working.

Chapter 6

Conclusion and Future Work

This project proposed the use of the MediaPipe framework to develop a gesture recognition system which could facilitate a touchless user interface in a surgical setting to maintain a sterile environment. The model developed was tested and demonstrated using a GUI which was a demo of the PACS image viewing system. In this project, we provided a detailed review of the literature which summarised results of papers which affected our approach, and critically analysed papers which discussed the drawbacks of computer vision. This was done due to the advent of the MediaPipe framework which tackled most of these drawbacks. We then specified our system and its architecture, before diving into the system's algorithms. The projected demonstrated the effectiveness of rule based approaches, which are only effective due to the classifying attributes provided by the MediaPipe models. A primary goal of this project was to contribute to the literature with a new method, that has only been recently developed. This was successful as a working prototype was built with MediaPipe and tested with PyAutoGUI.

Not only did this project aim to contribute to the existing literature concerning the surgical problem context, but also, this project developed a reusable model which is versatile and can be further researched and enhanced to control other systems in different problem domains using gestures. Some of these systems include media players, smart home interfaces and smart car interfaces. Moreover, MediaPipe enables a user to create large datasets in a few minutes. Due to this, we can develop more use cases which require Deep Neural Networks. Such use cases include the translation of sign language, or the prediction of emotional state. This would be effective at security checks such as airports.

In conclusion, we developed a system which contributes to the literature in two primary domains: the models used in MediaPipe, and the integration of speech recognition to further

assist surgeons in the OR.

Chapter 7

Legal, Social, Ethical and Professional Issues

Your report should include a chapter with a reasoned discussion about legal, social ethical and professional issues within the context of your project problem. You should also demonstrate that you are aware of the regulations governing your project area and the Code of Conduct & Code of Good Practice issued by the British Computer Society, and that you have applied their principles, where appropriate, as you carried out your project.

7.1 Section Heading

References

[1] Natasha Abner, Kensy Cooperrider, and Susan Goldin-Meadow. Gesture for linguists: A handy primer. *Language and linguistics compass*, 9(11):437–451, 2015.

[2] Monica Barragan, Nikolai Flowers, and Aaron M Johnson. Minirhex: A small, open-source, fully programmable walking hexapod. In *Proceedings of the Robotics: Science and Systems Workshop on "Design and Control of Small Legged Robots", Pittsburgh, PA, USA*, volume 30, 2018.

[3] Valentin Bazarevsky, Ivan Grishchenko, Karthik Raveendran, Tyler Zhu, Fan Zhang, and Matthias Grundmann. Blazepose: On-device real-time body pose tracking. *arXiv preprint arXiv:2006.10204*, 2020.

[4] Lingchen Chen, Feng Wang, Hui Deng, and Kaifan Ji. A survey on hand gesture recognition. In *2013 International conference on computer sciences and applications*, pages 313–316. IEEE, 2013.

[5] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde, Jared Kaplan, Harri Edwards, Yura Burda, Nicholas Joseph, Greg Brockman, et al. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*, 2021.

[6] Yongwon Cho, Areum Lee, Jongha Park, Bemseok Ko, and Namkug Kim. Enhancement of gesture recognition for contactless interface using a personalized classifier in the operating room. *Computer methods and programs in biomedicine*, 161:39–44, 2018.

[7] Smit Desai and Apurva Desai. Human computer interaction through hand gestures for home automation using microsoft kinect. In *Proceedings of International Conference on Communication and Networks*, pages 19–29. Springer, 2017.

[8] Z Fan, B Valentin, V Andrey, T Andrei, S George, C Chuo-Ling, and G Matthias.

Mediapipe hands: On-device real-time hand tracking. arxiv 2020. *arXiv preprint arXiv:2006.10214*.

[9] Susan Goldin-Meadow and Diane Brentari. Gesture, sign, and language: The coming of age of sign language and gesture studies. *Behavioral and Brain Sciences*, 40, 2017.

[10] Chauncey Graetzel, Terry Fong, Sebastien Grange, and Charles Baur. A non-contact mouse for surgeon-computer interaction. *Technology and Health Care*, 12(3):245–257, 2004.

[11] Yuval Noah Harari. *Sapiens: A brief history of humankind*. Random House, 2014.

[12] Florian Hartmann and Alexander Schlaefer. Feasibility of touch-less control of operating room lights. *International journal of computer assisted radiology and surgery*, 8(2):259–268, 2013.

[13] Derick Hsieh. *Touchless gesture recognition system for imaging controls in sterile environment*. PhD thesis, University of British Columbia, 2014.

[14] Mithun Jacob, Yu-Ting Li, George Akingba, and Juan P Wachs. Gestonurse: a robotic surgical nurse for handling surgical instruments in the operating room. *Journal of Robotic Surgery*, 6(1):53–63, 2012.

[15] Yury Kartynnik, Artsiom Ablavatski, Ivan Grishchenko, and Matthias Grundmann. Real-time facial surface geometry from monocular video on mobile gpus. *arXiv preprint arXiv:1907.06724*, 2019.

[16] Harpreet Kaur and Jyoti Rani. A review: Study of various techniques of hand gesture recognition. In *2016 IEEE 1st International Conference on Power Electronics, Intelligent Control and Energy Systems (ICPEICES)*, pages 1–5. IEEE, 2016.

[17] Holger Kenn, Friedrich Van Megen, and Robert Sugar. A glove-based gesture interface for wearable computing applications. In *4th International Forum on Applied Wearable Computing 2007*, pages 1–10. VDE, 2007.

[18] Min-Soo Kim and Choong Ho Lee. Hand gesture recognition for kinect v2 sensor in the near distance where depth data are not provided. *International Journal of Software Engineering and Its Applications*, 10(12):407–418, 2016.

[19] Robert M Krauss, Yihsiu Chen, and Purnima Chawla. Nonverbal behavior and nonverbal communication: What do conversational hand gestures tell us? *Advances in experimental social psychology*, 28:389–450, 1996.

[20] A-reum Lee, Yongwon Cho, Seongho Jin, and Namkug Kim. Enhancement of surgical hand gesture recognition using a capsule network for a contactless interface in the operating room. *Computer methods and programs in biomedicine*, 190:105385, 2020.

[21] Doe-Hyung Lee and Kwang-Seok Hong. Game interface using hand gesture recognition. In *5th International Conference on Computer Sciences and Convergence Information Technology*, pages 1092–1097. IEEE, 2010.

[22] Unseok Lee and Jiro Tanaka. Finger identification and hand gesture recognition techniques for natural user interface. In *Proceedings of the 11th Asia Pacific Conference on Computer Human Interaction*, pages 274–279, 2013.

[23] Camillo Lugaresi, Jiuqiang Tang, Hadon Nash, Chris McClanahan, Esha Uboweja, Michael Hays, Fan Zhang, Chuo-Ling Chang, Ming Guang Yong, Juhyun Lee, et al. Mediapipe: A framework for building perception pipelines. *arXiv preprint arXiv:1906.08172*, 2019.

[24] Naveen Madapana, Glebys Gonzalez, Richard Rodgers, Lingsong Zhang, and Juan P Wachs. Gestures for picture archiving and communication systems (pacs) operation in the operating room: Is there any standard? *PloS one*, 13(6):e0198092, 2018.

[25] Yellapu Madhuri, G Anitha, and Ml Anburajan. Vision-based sign language translation device. In *2013 International Conference on Information Communication and Embedded Systems (ICICES)*, pages 565–568. IEEE, 2013.

[26] Tomás Mantecón, Carlos R del Blanco, Fernando Jaureguizar, and Narciso García. Hand gesture recognition using infrared imagery provided by leap motion controller. In *International Conference on Advanced Concepts for Intelligent Vision Systems*, pages 47–57. Springer, 2016.

[27] Helena M Mentis, Kenton O'Hara, Gerardo Gonzalez, Abigail Sellen, Robert Corish, Antonio Criminisi, Rikin Trivedi, and Pierre Theodore. Voice or gesture in the operating room. In *Proceedings of the 33rd Annual ACM Conference Extended Abstracts on Human Factors in Computing Systems*, pages 773–780, 2015.

[28] Sushmita Mitra and Tinku Acharya. Gesture recognition: A survey. *IEEE Transactions on Systems, Man, and Cybernetics, Part C (Applications and Reviews)*, 37(3):311–324, 2007.

[29] GRS Murthy and RS Jadon. A review of vision based hand gestures recognition. *International Journal of Information Technology and Knowledge Management*, 2(2):405–410, 2009.

[30] Ebrahim Nasr-Esfahani, Nader Karimi, SM Soroushmehr, M Hossein Jafari, M Amin Khorsandi, Shadrokh Samavi, and Kayvan Najarian. Hand gesture recognition for contactless device control in operating rooms. *arXiv preprint arXiv:1611.04138*, 2016.

[31] Munir Oudah, Ali Al-Naji, and Javaan Chahl. Hand gesture recognition based on computer vision: a review of techniques. *journal of Imaging*, 6(8):73, 2020.

[32] Muneeba Raja, Viviane Ghaderi, and Stephan Sigg. Wibot! in-vehicle behaviour and gesture recognition using wireless network edge. In *2018 IEEE 38th International Conference on Distributed Computing Systems (ICDCS)*, pages 376–387. IEEE, 2018.

[33] Ram J Rajesh, D Nagarjunan, RM Arunachalam, and R Aarthi. Distance transform based hand gestures recognition for powerpoint presentation navigation. *Advanced Computing*, 3(3):41, 2012.

[34] Helman I Stern, Juan P Wachs, and Yael Edan. Optimal consensus intuitive hand gesture vocabulary design. In *2008 IEEE International Conference on Semantic Computing*, pages 96–103. IEEE, 2008.

[35] R Suriya and V Vijayachamundeeswari. A survey on hand gesture recognition for simple mouse control. In *International Conference on Information Communication and Embedded Systems (ICICES2014)*, pages 1–5. IEEE, 2014.

[36] Noor Tubaiz, Tamer Shanableh, and Khaled Assaleh. Glove-based continuous arabic sign language recognition in user-dependent mode. *IEEE Transactions on Human-Machine Systems*, 45(4):526–533, 2015.

[37] Michael Van den Bergh, Daniel Carton, Roderick De Nijs, Nikos Mitsou, Christian Landsiedel, Kolja Kuehnlenz, Dirk Wollherr, Luc Van Gool, and Martin Buss. Real-time 3d hand gesture interaction with a robot for understanding directions from humans. In *2011 Ro-Man*, pages 357–362. IEEE, 2011.

[38] Juan Pablo Wachs, Mathias Kölsch, Helman Stern, and Yael Edan. Vision-based hand-gesture applications. *Communications of the ACM*, 54(2):60–71, 2011.

[39] Han-liang WENG and Yin-wei ZHAN. Vision-based hand gesture recognition with multiple cues [j]. *Computer Engineering & Science*, 2, 2012.

[40] Daniel Wigdor and Dennis Wixon. *Brave NUI world: designing natural user interfaces for touch and gesture*. Elsevier, 2011.

[41] Xueqin Xiang, Zhigeng Pan, and Jing Tong. Depth camera in computer vision and computer graphics: an overview. *Jisuanji Kexue yu Tansuo*, 5(6):481–492, 2011.

[42] Jinhua Zeng, Yaoru Sun, and Fang Wang. A natural hand gesture system for intelligent human-computer interaction and medical assistance. In *2012 Third Global Congress on Intelligent Systems*, pages 382–385. IEEE, 2012.

[43] Fan Zhang, Valentin Bazarevsky, Andrey Vakunov, Andrei Tkachenka, George Sung, Chuo-Ling Chang, and Matthias Grundmann. Mediapipe hands: On-device real-time hand tracking. *arXiv preprint arXiv:2006.10214*, 2020.

[44] Xinshuang Zhao, Ahmed M Naguib, and Sukhan Lee. Kinect based calling gesture recognition for taking order service of elderly care robot. In *The 23rd IEEE international symposium on robot and human interactive communication*, pages 525–530. IEEE, 2014.

[45] WANG Zhao-qi. Research review of virtual human synthesis. *Journal of the Graduate School of the Chinese Academy of Sciences (in Chinese)*, 17(2):89–97, 2000.

[46] Fang Zhigang. Computer gesture input and its application in human computer interaction. *Mini-micro Systems*, 20(6):418–421, 1999.

Appendix A

User Guide

A.1 Instructions

1---- Before running ensure that the following libraries are pip installed:

cv2

mediapipe

numpy

time

autopy

math

pyautogui

speech_recognition

io

pathlib

PIL

PySimpleGUI

os

Once the libraries are installed, run the system by following the steps below.

2---- Using an IDE, run GUI_init.py and as you click on each button, its coordinates will be outpu

3---- Modify pyautoguiFunctions.py to include these coordinates, no need to pass the last 2 coordi

4---- Using terminal (bash):

cd ~/folderPath---->python GRmodel.py & python GUI.py &

5---- Use the gestures to control the system. Make sure that you, your arm and your wrist are faci

Appendix B

Source Code

B.1 Instructions

B.2 Hand Tracking Module

```
### This was developed using a tutorial from


https://www.computervision.zone/courses/ai-virtual-mouse/


import cv2

import mediapipe as mp

import time

import math


class handDetector():

    def __init__(self, mode=False, maxHands = 2, detectionCon=0.5,trackCon=0.5):

        self.mode = mode

        self.maxHands = maxHands

        self.detectionCon = detectionCon

        self.trackCon = trackCon


        self.holistic = mp.solutions.holistic.Holistic(min_detection_confidence=

        self.detectionCon, min_tracking_confidence=self.trackCon)
```

```
        self.mpHands = mp.solutions.hands
        self.hands = self.mpHands.Hands(self.mode, self.maxHands,self.detectionCon, self.trackCon)
        self.mpDraw = mp.solutions.drawing_utils
        self.tipIds = [4, 8, 12, 16, 20]


    def findHands(self, img, draw=True):
        imgRGB = cv2.cvtColor(img, cv2.COLOR_BGR2RGB)
        self.results = self.hands.process(imgRGB)
        self.landmarks = self.holistic.process(imgRGB)

        if self.results.multi_hand_landmarks:
            for handLms in self.results.multi_hand_landmarks:
                if draw:
                    self.mpDraw.draw_landmarks(img, handLms,
                                               self.mpHands.HAND_CONNECTIONS)
        return img, self.landmarks

    def findPosition(self, img, handNo=0, draw=True):
        xList = []
        yList = []
        bbox = []
        self.lmList = []
        if self.results.multi_hand_landmarks:
            myHand = self.results.multi_hand_landmarks[handNo]
            for id, lm in enumerate(myHand.landmark):
                # print(id, lm)
                h, w, c = img.shape
                cx, cy = int(lm.x * w), int(lm.y * h)
                xList.append(cx)
                yList.append(cy)
                self.lmList.append([id, cx, cy])
                if draw:
```

```
                cv2.circle(img, (cx, cy), 5, (255, 0, 255), cv2.FILLED)

        xmin, xmax = min(xList), max(xList)
        ymin, ymax = min(yList), max(yList)
        bbox = xmin, ymin, xmax, ymax

        if draw:
            cv2.rectangle(img, (xmin - 20, ymin - 20), (xmax + 20, ymax + 20),
                          (0, 255, 0), 2)

    return self.lmList, bbox

def fingersUp(self):
    self.fingers = []
    # Thumb
    if self.lmList[self.tipIds[0]][1] > self.lmList[self.tipIds[0] - 1][1]:
        self.fingers.append(1)
    else:
        self.fingers.append(0)

    # Fingers
    for id in range(1, 5):

        if self.lmList[self.tipIds[id]][2] < self.lmList[self.tipIds[id] - 2][2]:
            self.fingers.append(1)
        else:
            self.fingers.append(0)


    return self.fingers


def fingersDown(self):
```

```
        down = 0
        for id in range(1,5):
            if self.lmList[self.tipIds[id]][2] > self.lmList[0][2]:
                down+=1
        return down


    def findDistance(self, p1, p2, img, draw=True,r=15, t=3):
        x1, y1 = self.lmList[p1][1:]
        x2, y2 = self.lmList[p2][1:]
        cx, cy = (x1 + x2) // 2, (y1 + y2) // 2

        if draw:
            cv2.line(img, (x1, y1), (x2, y2), (255, 0, 255), t)
            cv2.circle(img, (x1, y1), r, (255, 0, 255), cv2.FILLED)
            cv2.circle(img, (x2, y2), r, (255, 0, 255), cv2.FILLED)
            cv2.circle(img, (cx, cy), r, (0, 0, 255), cv2.FILLED)
        length = math.hypot(x2 - x1, y2 - y1)

        return length, img, [x1, y1, x2, y2, cx, cy]
```

B.3 PyAutoGUI Functions Module

```
import pyautogui as pg


previousXY = (229,333)
nextXY = (91,363)
inXY = (99,332)
```

```
outXY = (162,331)
brightenXY = (141,361)
darkenXY = (201,360)


def previousImg():
    pg.click(previousXY)


def nextImg():
    pg.click(nextXY)


def zoomIn():
    pg.click(inXY)


def zoomOut():
    pg.click(outXY)


def Brighten():
    pg.click(brightenXY)


def Darken():
    pg.click(darkenXY)


def scrollUp():
    pg.scroll(20, x=571, y=429)
```

```
def scrollDown():
    pg.scroll(-20, x=571, y=429)


def scrollRight():
    pg.hscroll(-20, x=571, y=429)


def scrollLeft():
    pg.hscroll(20, x=571, y=429)
```

B.4 Gesture Recognition Model

```
import cv2
import mediapipe as mp
import numpy as np
import HandTrackingModule as htm
import time
import autopy
import math
import pyautoguiFunctions
import pyautogui
import speech_recognition as sr


#### Inputs

########################
wCam, hCam = 640, 480  # Set webcam screen size
frameR = 100
# Set frame reduction factor to build rectangle (reduced screen) which represents actual screen fo
wScr, hScr = autopy.screen.size()  # Get actual screen size
smoothening = 5  # Smoothening factor to determine cursor control speed
plocX, plocY = 0, 0  # Initialization of initial cursor location
```

```
clocX, clocY = 0, 0  # Initialization of current cursor location
detector = htm.handDetector(detectionCon = 0.7)
# Object of handDetector class from htm module, creates attributes containing MediaPipe model outp
speech_duration = 3 # Speech detection duration
click_threshold = 40
# Distance between index & middle fingertips to determine wether to click cursor or not
zoom_threshold = 50
# Distance between thumb and index fingertips to determine wether zoom in or out/brighten or darke
########################


#### Open webcam & set to initialized size

cap = cv2.VideoCapture(0)
cap.set(3, wCam)
cap.set(4, hCam)



while True: # While webcam is open

    #### Find hand landmarks using MediaPipe detection models

    success, img = cap.read() # Collect raw image data using OpenCV

    img, landmarks = detector.findHands(img)
    # img is the frame with annotated landmark connections and landmarks contains hand landmarks w

    lmList, bbox = detector.findPosition(img)
    # lmList [landmark ID, x, y] contains MediaPipe multi hand landmark coordinates and bbox conta
```

```
if len(lmList) != 0: # If landmarks are detected

    fingers = detector.fingersUp() # check which fingers are up: 1 if finger up, 0 if down. [b

    #### CURSOR CONTROL

    # Get the (x,y) of the tips of the index and middle fingers
    x1, y1 = lmList[8][1:]
    x2, y2 = lmList[12][1:]


    # HOVER
    if fingers[1:5] == [1,0,0,0]: # If hover gesture is detected

        # create rectangle to represent screen
        cv2.rectangle(img, (frameR, frameR), (wCam - frameR, hCam - frameR), (255, 0, 255), 2)

        # find coordinates of index fingertip
        x3 = np.interp(x1, (frameR, wCam - frameR), (0, wScr))
        y3 = np.interp(y1, (frameR, hCam - frameR), (0, hScr))

        # smoothen values to ease cursor control
        clocX = plocX + (x3 - plocX) / smoothening
        clocY = plocY + (y3 - plocY) / smoothening

        autopy.mouse.move(wScr - clocX, clocY) # move mouse relative to fingertip coordinates
        cv2.circle(img, (x1, y1), 15, (255, 0, 255), cv2.FILLED)
        plocX, plocY = clocX, clocY

    # CLICK
    if fingers[1:5] == [1,1,0,0]: # If click gesture is detected

        length, img, lineInfo = detector.findDistance(8, 12, img) # find distance between inde
```

```
    if length < click_threshold:

        cv2.circle(img, (lineInfo[4], lineInfo[5]), 15, (0, 255, 0), cv2.FILLED)
        autopy.mouse.click()  # click mouse if distance is below threshold



#### SLIDESHOW
if landmarks.left_hand_landmarks and fingers == [0,0,0,0,1]:
    pyautoguiFunctions.previousImg() # click previous button if previous gesture is detect
    time.sleep(1) # to slow down the command invocation
elif landmarks.right_hand_landmarks and fingers == [1,0,0,0,1]:
    pyautoguiFunctions.nextImg() # click next button if next gesture is detected
    time.sleep(1)



#### ZOOM
if landmarks.right_hand_landmarks and fingers == [1,1,0,0,0]: # If zoom gesture is detecte

    length, img, lineInfo = detector.findDistance(4, 8, img) # find distance between index

    if length > zoom_threshold:
        pyautoguiFunctions.zoomIn() # click zoom in button if distance is above threshold
        time.sleep(1)
        cv2.circle(img, (lineInfo[4], lineInfo[5]), 15, (0, 255, 0), cv2.FILLED)
    else:
        pyautoguiFunctions.zoomOut() # click zoom out button if distance is below threshol
        time.sleep(1)

#### BRIGHTNESS
if landmarks.left_hand_landmarks and fingers == [0,1,0,0,0]: # If brightness gesture is de
```

```
    length, img, lineInfo = detector.findDistance(4, 8, img) # find distance between index

    if length > zoom_threshold:
        pyautoguiFunctions.Brighten() # click brighten button if distance is above thresho
        time.sleep(1)
        cv2.circle(img, (lineInfo[4], lineInfo[5]), 15, (0, 255, 0), cv2.FILLED)
    else:
        pyautoguiFunctions.Darken() # click darken button if distance is above threshold
        time.sleep(1)

#### SCROLL
down = detector.fingersDown() # variable with number of fingers below wrist

if fingers[1:5]==[1,1,1,1]: # if scroll up gesture is detected
    pyautoguiFunctions.scrollUp() # automate mouse scrolling upwards
    time.sleep(1)
elif down == 4: # if scroll down gesture is detected
    pyautoguiFunctions.scrollDown() # automate mouse scrolling downwards
    time.sleep(1)
elif landmarks.right_hand_landmarks and fingers == [0,1,0,0,1]: # if scroll right gesture
    pyautoguiFunctions.scrollRight() # automate mouse scrolling right
    time.sleep(1)
elif landmarks.left_hand_landmarks and fingers == [1,1,0,0,1]: # if scroll left gesture is
    pyautoguiFunctions.scrollLeft() # automate mouse scrolling left
    time.sleep(1)


#### SPEECH

if landmarks.right_hand_landmarks and landmarks.left_hand_landmarks: # If both hands are d

    if fingers[1:5] == [0,0,0,0]: # and If backspace gesture is detected
```

```
                pyautogui.press('backspace') # Backspace on keyboard


            elif fingers[1:5] == [1,1,1,1]: # else If speech activation gesture is detected

                print('typing')
                r = sr.Recognizer() # Create object from speech_recognition
                with sr.Microphone() as source: # Input audio data from default microphone

                    audio_data = r.record(source, duration=speech_duration) # read the audio data


                    # Convert speech to text
                    try:
                        text = r.recognize_google(audio_data) # text string variable
                        pyautogui.typewrite(text) # automate keyboard commands to type text
                        pyautogui.press('enter') # enter to search
                    except:
                        pass # if no speech detected, pass onto CURSOR CONTROL


    #### Display webcam feed on screen
#     cv2.imshow("OpenCV feed", img)

    # Close webcam if 'q' is clicked
    if cv2.waitKey(10) & 0xFF == ord('q'):
        break

cap.release()
cv2.destroyAllWindows()
cv2.waitKey(1)
```

B.5 GUI

```
from io import BytesIO
from pathlib import Path
from PIL import Image, ImageEnhance
import numpy as np
import PySimpleGUI as sg
import os



sg.theme('Black')

# Class title
class GUI:

    # the init function
    def __init__(self):
        # initiation variables used in rest of class
        self.column_width = 960
        self.column_height = 720
        self.width = 0
        self.height = 0
        self.diff = 10
        self.white = (255, 255, 255, 0)
        self.font = ('Courier New', 16, 'bold')
        # This is a graph, where the image resides (used in layout[])
        col = [[self.graph()]]
        # This is the left half of the GUI
        file_list_column = [
                            # variable creation within the PySimpleGUI class
                           [sg.Text("Image Folder:"),
                            sg.In(size=(25,1), enable_events=True, key="-FOLDER-"),
                            sg.FolderBrowse()],
```

```
                    [
                    self.button('Zoom In'),
                    self.button('Zoom Out'),
                    self.button('Previous')],
                    [self.button('Next'),
                    self.button('Brighten'),
                    self.button('Darken')],
                    [sg.Text("Search Image:"),
                    sg.Input(do_not_clear=True, size=(20, 1), enable_events=True, key='-IN
                    [sg.Listbox(values=[], enable_events=True, size=(40, 20), key="-FILE L

                    ]
# This is the entire layout (right side inside)
layout = [
    [
        # Left side
        sg.Column(file_list_column),
        # Line separating two sides
        sg.VSeperator(),
        # Right side (image display)
        self.column(col, key='Column'),
    ]
]
# More instance variables
self.window = sg.Window('Image Viewer - PACS Demo', layout, finalize=True,
    use_default_focus=False)
self.draw   = self.window['Graph']
self.im     = None
self.key    = None
self.scale  = 1
self.folder = None
self.fnames = None
self.file_list = None
```

```
        self.filename = None
        self.listNum = None
        self.enhancer = None
        self.factor = None
        self.output = None
        self.results = None
        self.new_values = None

    # Button function used when creating buttons so all are uniform
    def button(self, text):
        return sg.Button(button_text=text, enable_events=True,
            key=text)

    # Column function used when creating columns so all are uniform
    def column(self, layout, key='Column'):
        return sg.Column(layout, background_color='grey', scrollable=True,
                         size=(self.column_width, self.column_height), key='Column')

    # File function used when creating files so all are uniform
    def file(self, save=False):
        return sg.popup_get_file('message', save_as=save, no_window=True,
            font=self.font, file_types=(
            ("ALL Files", "*.*"), ("PNG Files", "*.jpg")),
            default_extension='png')

    # Graph function used when creating graphs so all are uniform
    def graph(self):
        return sg.Graph(
            (self.column_width, self.column_height), (0, self.column_height),
            (self.column_width, 0), pad=(0, 0), background_color='grey',
            enable_events=True, key='Graph')

    # Image function used when creating images so all are uniform
```

```
def image(self):
    return sg.Image(key='Image', enable_events=True)

# function to draw the image on screen when it is updated
def draw_image(self):
    try:
        # if theres something there
        if self.key:
            # delete it
            self.draw.delete_figure(self.key)
        # set new key to the new image at location 0, 0
        self.key = self.draw.draw_image(data=self.data, location=(0, 0))
        # draw
        self.draw.Widget.configure(width=self.width*self.scale, height=self.height*self.scale)
        # set max width height of image
        max_width = max(self.width*self.scale, self.column_width)
        max_height = max(self.height*self.scale, self.column_height)
        # canvas
        canvas = g.window['Column'].Widget.canvas
        canvas.configure(scrollregion=(0, 0, max_width, max_height))
    except Exception as e:
        print(e)

@property
# Gets information about image
def data(self):
    try:
        # if scale is same, do nothing
        if self.scale == 1:
            im = self.im
        else:
            im = self.im.resize(
                (int(self.width*self.scale), int(self.height*self.scale)),
```

```
                resample=Image.NEAREST)
        with BytesIO() as output:
            im.save(output, format="PNG")
            data = output.getvalue()
        return data
    except Exception as e:
        print(e)


# grey function
def grey(self):
    try:
        im_grey = np.array(self.im.convert(mode="L"), dtype=np.uint8)
        im = np.array(self.im, dtype=np.uint8)
        return im, im_grey
    except Exception as e:
        print(e)


# open file function
def open_file(self):
    try:
        # if its the right file
        if self.filename and Path(self.filename).is_file():
            # open the image in PIL library
            self.im = Image.open(self.filename).convert(mode='RGBA')
            # get properties of it
            self.width, self.height = self.im.size
            # reset scale to 1
            #self.scale = 1
            # draw the image
            self.draw_image()
    except Exception as e:
        print(e)
```

```
# remove function
def remove(self):
    try:
        x, y = values['Graph']
        x, y = x//self.scale, y//self.scale
        image, image_grey = self.grey()
        pixel = image_grey[y, x]
        lst = [(x, y)]
        checked = set()
        while lst:
            tmp = set()
            for point in lst:
                x1, y1 = point
                if point not in checked:
                    checked.add((x1, y1))
                    if pixel-self.diff <= image_grey[y1, x1] <= pixel+self.diff:
                        image[y1, x1] = self.white
                        for dx, dy in [(-1, 0), (1, 0), (0, -1), (0, 1)]:
                            if 0<=x1+dx<self.width and 0<=y1+dy<self.height:
                                tmp.add((x1+dx, y1+dy))
            lst = tmp
        self.im = Image.fromarray(image)
        self.draw_image()
    except Exception as e:
        print(e)


# savesfile under filename
def save_file(self):
    try:
        if self.key:
            filename = self.file(save=True)
            if filename:
                g.im.save(filename)
```

```
        except Exception as e:
            print(e)

    # Zoom in function
    def zoom_in(self):
        try:
            # set scale up
            self.scale = min(self.scale+1, 10)
            # redraw image
            self.draw_image()
        except Exception as e:
            print(e)

    # Zoom out function
    def zoom_out(self):
        try:
            # set scale down 1
            self.scale = max(self.scale-1, 1)
            # draw image
            self.draw_image()
        except Exception as e:
            print(e)

    # called when folder selected, display Files function
    def displayFiles(self, values):
        # get the value of the folder
        self.folder = values["-FOLDER-"]
        try:
            # Get list of files in folder
            self.file_list = os.listdir(self.folder)
        # if theres an error (no files in there)
        except:
            # make the list empty
```

```
        self.file_list = []

    # take out all non .png/.gif
    self.fnames = [
        f
        for f in self.file_list
        if os.path.isfile(os.path.join(self.folder, f))
           and f.lower().endswith((".png", ".gif"))
    ]
    # update the file list
    self.window["-FILE LIST-"].update(self.fnames)

# display image function called when image is clicked
def displayImage(self, values):
    # try this
    try:
        # joins file path of folder and image for later use
        self.filename = os.path.join(
            values["-FOLDER-"], values["-FILE LIST-"][0]
        )
        # sets the list index for prev/next function
        self.listNum = self.fnames.index(values['-FILE LIST-'][0])
        # opens the image
        self.scale = 1
        self.open_file()
    # if theres an error, don't quit the program, just keep going
    except:
        pass

# Previous image function
def previousImage(self):
    # try this
    try:
```

```
            # if its the beginning of the list, set listnum to the end + 1
            if self.listNum == 0:
                self.listNum = len(self.fnames)
            # subtract one
            self.listNum -= 1
            # join file paths
            self.filename = os.path.join(
                values["-FOLDER-"], str(self.fnames[self.listNum])
            )
            # open new file
            self.open_file()

        except Exception as e:  # if theres an error, don't quit program, just print the error
            print(e)

    # go to next image
    def nextImage(self):
        # try this
        try:
            # if its the last element, set it to one below first
            if self.listNum == len(self.fnames) - 1:
                self.listNum = -1
            # add one to the listnum
            self.listNum+=1
            # join file paths
            self.filename = os.path.join(values["-FOLDER-"], str(self.fnames[self.listNum]))
            # open the file
            self.open_file()
        # if theres an error, don't quit, just print the error
        except Exception as e:
            print(e)

    def brighten(self):
```

```
        try:
            self.enhancer = ImageEnhance.Brightness(self.im)
            self.factor = 1.1
            self.output = self.enhancer.enhance(self.factor)
            self.output.save(self.filename)
            self.open_file()
        except Exception as e:
            print(e)

    def darken(self):
        try:
            self.enhancer = ImageEnhance.Brightness(self.im)
            self.factor = 0.9
            self.output = self.enhancer.enhance(self.factor)
            self.output.save(self.filename)
            self.open_file()
        except Exception as e:
            print(e)

    def search(self, values):
        try:
            self.results = values['-INPUT-']
            self.new_values = [x for x in self.fnames if self.results in x]
            self.window["-FILE LIST-"].update(self.new_values)
        except Exception as e:
            print(e)

# create an instance of the class
g = GUI()
# function list for each button/action
function = {'Open':g.open_file,  'Save':g.save_file, 'Graph':g.remove,
            'Zoom In':g.zoom_in, 'Zoom Out':g.zoom_out, '-FOLDER-':g.displayFiles,
            '-FILE LIST-':g.displayImage, "Previous":g.previousImage, "Next":g.nextImage,
```

```
            'Brighten':g.brighten, 'Darken':g.darken}
# main loop for program
while True:
    # get the events and values that happen every time
    event, values = g.window.read()
    # if that event is a close button, then close
    if event in (sg.WINDOW_CLOSED, 'Exit'):
        break
    elif values['-INPUT-'] != '':
        g.search(values)
    else:
        g.displayFiles(values)
    # if a new folder has opened, or file has been clicked, open the function but with parameter v
    if '-FOLDER-' in event or '-FILE LIST-' in event:
        function[event](values)
    # if its any other function just call the function
    elif event in function:
        function[event]()

# when loop ends, close the program
g.window.close()
```