**The Project Report on**

**Hand Gesture Controller** **For Web Apps**

*Submitted in partial fulfilment of the requirements for the award of the degree of*

**BACHELOR OF TECHNOLOGY**

In

**CSE (DATA SCIENCE)**

By

|  |  |
| --- | --- |
| **P SANDHYA** | (23AG1A67H6) |
| **M SANIYA** | (23AG1A67G9) |
| **M BHANU PRASAD** | (23AG1A67G7) |
| **G GOPALAKRISHNA** | (23AG1A67E6) |

Under the guidance of

**Mrs. B SARITHA**

Assistant Professor



# DEPARTMENT OF CSE (DATA SCIENCE)

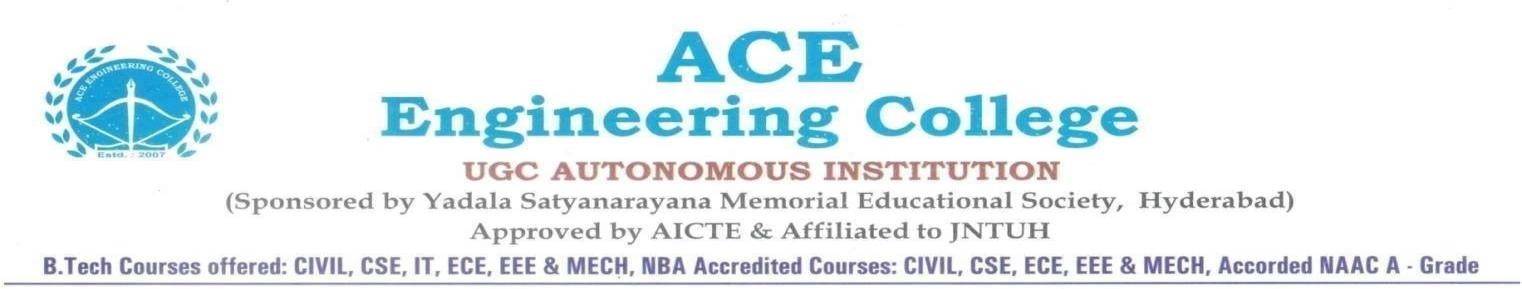
**ACE Engineering College**

Ankushapur(V), Ghatkesar(M), Medchal Dist. - 501 301

***(An Autonomous Institution, Affiliated to JNTUH, Hyderabad)***

[www.aceec.ac.in](http://www.aceec.ac.in)

**A.Y: 2024-2025**



## DEPARTMENT OF CSE (DATA SCIENCE)



### CERTIFICATE

This is to certify that the Societal Related project report entitled **“*HAND GESTURE CONTROLLER FOR WEB APPS* ”** is a Bonafide work done by ***P Sandhya (23AG1A67H6), M Saniya (23AG1A67G9), M Bhanu Prasad (23AG1A67G7), G GopalaKrishna (23AG1A67E6)*** in partial fulfilment for the award of Degree of BACHELOR OF TECHNOLOGY in CSE (Data Science)from JNTUH University, Hyderabad during the academic year 2024- 2025. This record of Bonafide work carried out by them under our guidance and supervision. The results embodied in this report have not been submitted by the student to any other University or Institution for the award of any degree or diploma.

|  |  |  |
| --- | --- | --- |
| **Mrs B Saritha**  Assistant Professor | **Dr. P Chiranjeevi**  Associate Professor | **External** |
|  | HOD**,** CSE-DS |  |

**ACKNOWLEDGEMENT**

We would like to express our gratitude to all the people behind the screen who have helped us transform an idea into a real-time application. We would like to express our heart-felt gratitude to our parents without whom we would not have been privileged to achieve and fulfill our dreams.

A special thanks to our General Secretary, **Prof. Y V Gopala Krishna Murthy,** for having founded such an esteemed institution. Sincere thanks to our Joint Secretary **Mrs. M Padmavathi**, for support in doing project work.

We profoundly thank **Dr. P Chiranjeevi**, Associate Professor and Head of the Department of Computer Science and Engineering (Data Science), who has been an excellent guide and also a great source of inspiration to our work.

We extremely thank **Mrs P Niharika,** Assistant Professor and Project coordinator, who helped us in all the way in fulfilling all aspects in completion of our Major-Project.

We are very thankful to our internal guide **Mrs B Saritha** who has been excellent and also given continuous support for the Completion of our project work. The satisfaction and euphoria that accompany the successful completion of the task would be great, but incomplete without the mention of the people who made it possible, whose constant guidance and encouragement crown all the efforts with success. In this context, we would like to thank all the other staff members, both teaching and non-teaching, who have extended their timely help and eased our task.

P Sandhya (23AG1A67H6)

M Saniya (23AG1A67G9)

M Banu Prasad (23AG1A67G7)

G GopalaKrishna (23AG1A67E6)

**DECLARATION**

We here by declare that the result embodied in this project report Entitled “**Hand** **Gesture Controller for Web Apps AI”** is carried out by us during the year 2024-2025 for the partial fulfilment of the award of **Bachelor of Technology in Computer Science and Engineering (Data Science)**, from **ACE ENGINEERING COLLEGE.** We have not submitted this project report to any other Universities/Institute for the award of any degree.

P Sandhya (23AG1A67H6)   
 M Saniya (23AG1A67G9)   
 M Bhanu Prasad (23AG1A67G7)   
 G GopalaKrishna (23AG1A67E6)

**Hand Gesture Controller**

**For Web Apps**

**ABSTRACT**

Hand gesture recognition is an innovative approach that enhances human-computer interaction by eliminating the need for traditional input devices like keyboards and mice. This project presents a gesture-controlled system for web applications, allowing users to navigate and interact using natural hand movements instead of physical peripherals. By leveraging computer vision and machine learning technologies, particularly MediaPipe for hand tracking and PyAutoGUI for cursor control, this system provides a seamless, touch-free interface. The project captures and processes hand movements in real-time, translating them into cursor actions and click events, thereby improving accessibility, hygiene, and intuitiveness.

The integration of multithreading ensures that gesture recognition occurs smoothly without lag, delivering a fluid, user-friendly experience. The need for intuitive and touchless interfaces has surged, especially for users with physical disabilities and in environments where hygiene is a priority. Traditional input devices can pose limitations, requiring direct contact and repetitive movements, which may not always be convenient or accessible. By utilizing real-time computer vision-based gesture recognition, this system provides a dynamic and adaptable alternative to conventional input methods.

The hand tracking pipeline accurately detects hand gestures, analyzing finger positions and movements to execute corresponding actions on the computer screen. The underlying deep learning models help refine gesture detection accuracy, ensuring users experience precise, responsive, and context-aware interactions.This system has potential plications across multiple domains, including assistive technology, gaming, smart environments, and virtual navigation. Future enhancements may focus on expanding gesture support, enabling custom gesture mapping, and integrating voice commands for multimodal control.

Additionally, improving gesture detection in low-light environments and extending compatibility to mobile and embedded systems will make the system even more versatile. As AI-driven interfaces continue to evolve, gesture-based controls can significantly redefine how users interact with digital content, offering a natural, immersive, and intelligent computing experience.

**CONTENTS**

|  |  |  |
| --- | --- | --- |
| **S. No** | **Chapter** | **Page No.** |

|  |  |  |
| --- | --- | --- |
| **1** | **INTRODUCTION** | **1 - 3** |

1.2 Background and Context of the Project   
1.2 Problem Statement and Objectives   
1.3 Significance and Motivation

|  |  |  |
| --- | --- | --- |
| **2** | **LITERATURE SURVEY** | **4 - 9** |

2.1 Existing System   
2.2 Existing System and its Limitations   
2.3 Proposed System

|  |  |  |
| --- | --- | --- |
| **3** | **REQUIREMENT ANALYSIS** | **10-18** |

3.1 Software Requirements   
3.2 Hardware Requirements   
3.3 Functional Requirements   
3.4 Non - Functional Requirements

|  |  |  |
| --- | --- | --- |
| **4** | **SYSTEM ANALYSIS** | **19-22 19-22** |

4.1 Methodology   
4.2 System Modules

|  |  |  |
| --- | --- | --- |
| **5** | **SYSTEM DESIGN** | **23-28 23-28** |

5.1 System Architecture   
5.2 Use Case Diagram   
5.3 Class Diagram   
5.4 Sequence Diagram   
5.5 State Chart Diagram   
5.6 Activity Diagram

|  |  |  |
| --- | --- | --- |
| **6** | **IMPLEMENTATION** | **29-39** |

|  |  |  |
| --- | --- | --- |
| **7** | **RESULT** | **40-44** |

|  |  |  |
| --- | --- | --- |
| **8** | **CONCLUSION AND FUTURE SCOPE** | **45-46** |

**REFERENCES**

**CHAPTER-1**

**INTRODUCTION**

**1.1 Background and Context of the Project:**

The modern era of computing is defined by the pursuit of seamless, natural, and human-centric interaction mechanisms. While traditional human-computer interaction (HCI) methods like the keyboard, mouse, and touchscreens have revolutionized digital engagement for decades, their limitations are increasingly evident as we strive for more immersive and accessible digital environments. These conventional tools often restrict interaction to two-dimensional inputs and are not always intuitive for all user demographics, particularly individuals with physical impairments or those working in environments that demand contact-free operations.

Gesture recognition, especially through hand movements, offers a new paradigm in interaction design. Hand gestures are a form of non-verbal communication that humans naturally use in day-to-day life. By translating these gestures into digital commands, users can control systems in a way that mimics natural behavior. This form of interaction eliminates the need for intermediary hardware and allows for direct, instinctive engagement with devices. It also opens up new possibilities in augmented reality (AR), virtual reality (VR), and artificial intelligence (AI)-based environments.

The development of computer vision and deep learning technologies has significantly improved the feasibility and accuracy of gesture recognition systems. With advancements in algorithms and processing power, it is now possible to perform real-time gesture tracking using standard webcams and open-source software libraries. This progress makes it viable to integrate gesture recognition into widely used platforms like web applications, expanding its accessibility and practical use.

This project, titled **"Hand Gesture Control for Web Apps"**, aims to harness these technological advancements to design a robust, real-time gesture recognition system tailored specifically for controlling web applications. By integrating computer vision with intelligent gesture mapping, users will be able to navigate, click, scroll, and interact with web interfaces using only their hands. The end goal is to redefine digital interaction in a way that is more natural, inclusive, and adaptable to various domains, including healthcare, education, gaming, and smart homes.

**1.2 Problem Statement and Objectives:**

Despite rapid advancements in digital technology, interaction with computing systems remains dependent on physical input devices, which are not always optimal or accessible. People with physical disabilities often face challenges using traditional devices, reducing their ability to access digital content independently. In many work environments—such as hospitals, research labs, and public kiosks—physical contact with devices is discouraged or even unsafe due to concerns about hygiene and contamination. Additionally, in hands-busy situations like cooking, manual labor, or industrial monitoring, touch-based inputs are impractical or disruptive.

Moreover, traditional interfaces lack the fluidity and naturalness of human interaction. While they function well for specific tasks, they do not accommodate diverse user needs or the increasing demand for seamless, intuitive user experiences. The absence of a universally accessible and contactless interface continues to pose a challenge in bridging the gap between human capability and digital technology. Therefore, a more natural and flexible input method is essential.

To address this challenge, this project proposes a gesture-controlled system that recognizes hand movements using a camera feed and translates them into actionable web commands. The system will operate in real time and provide users with an interactive experience that does not rely on physical contact or external devices. This will not only increase accessibility but also make interaction more hygienic, natural, and user-friendly.

The main objectives of this project include:

• Developing a robust and real-time hand gesture recognition system using computer vision techniques.

• Mapping recognized gestures to actions typically performed on web applications (e.g., mouse movement, click, scroll, navigation).

• Ensuring the system is lightweight, accurate, and responsive.

• Providing a user-friendly and accessible interface for users with physical limitations.

• Enabling customization and flexibility to add new gesture mappings in the future.

**1.3 Significance and Motivation:**

The significance of this project lies in its ability to enhance human-computer interaction by making it more inclusive, accessible, and hygienic. In particular, for people with mobility impairments, gesture recognition opens up new opportunities to interact with digital systems without needing specialized hardware or assistance. This promotes digital independence and aligns with global goals for accessibility and inclusion in technology. By integrating gesture control into web applications—arguably the most ubiquitous platform today—this project ensures broad utility and impact.

In environments such as hospitals, laboratories, and food processing units, where hygiene is a top priority, avoiding physical contact with shared devices is crucial. A gesture-controlled interface minimizes contamination risk by enabling contactless control, thus supporting infection control protocols. This has become especially relevant in light of global health events like the COVID-19 pandemic, which highlighted the urgent need for touch-free technologies.

Furthermore, gesture-based interfaces can dramatically enhance user experience by introducing a more intuitive and immersive way to interact with digital content. For example, in smart home environments or VR-based learning platforms, users can navigate systems more naturally using hand gestures. This form of interaction feels more human and reduces the learning curve associated with traditional device-based interfaces.

**The motivation** behind this project also stems from a fascination with how machine learning and computer vision can be combined to create intelligent systems that understand and adapt to human behavior. With open-source tools, readily available hardware, and rapidly evolving AI models, now is the perfect time to explore the development of a gesture recognition system for web platforms. The team’s academic background in data science and machine learning, along with a shared interest in solving real-world problems, further strengthens the drive to pursue and complete this innovative project

**CHAPTER-2**

**LITERATURE SURVEY**

Hand gesture recognition has been a growing area of research in human-computer interaction (HCI) for over two decades. Several systems have been developed based on different sensing technologies and recognition algorithms. Early systems used wearable devices like data gloves equipped with sensors to capture finger bends and hand orientation. Though accurate, these systems were expensive, uncomfortable, and limited in mobility. They also required users to wear cumbersome equipment, reducing the practicality for widespread use, especially in consumer applications.

Later developments utilized depth-sensing devices like the Microsoft Kinect sensor, which captured 3D spatial information of hand and body movements. These systems marked a significant improvement by allowing non-contact gesture recognition. However, such hardware was not always available or affordable for personal use and required specific environmental conditions (lighting, distance, and background) to function optimally. Additionally, these systems were largely restricted to gaming or niche industrial applications.

In recent years, computer vision techniques based on RGB cameras and deep learning have gained popularity for gesture recognition. Open-source frameworks like MediaPipe (by Google), TensorFlow, and OpenCV have enabled researchers and developers to create vision-based models that can detect hand landmarks from standard webcam inputs. These methods eliminate the need for specialized hardware and rely on machine learning models trained to identify and classify hand gestures with high accuracy.

Moreover, applications in sign language interpretation, robotics control, virtual reality, and smart TV navigation have been explored using gesture-based control systems. Most of these applications are platform-specific or designed for desktop environments rather than being optimized for web-based interfaces. While some commercial solutions provide gesture support, they often do not allow flexibility in gesture customization, nor are they open for integration with modern web apps through standard APIs or browser interfaces.

Despite significant progress, existing gesture recognition systems face multiple limitations that restrict their application in broader use cases. One of the major challenges is **reliability in real-time scenarios**. Many systems perform well in controlled laboratory environments but struggle when exposed to dynamic lighting conditions, background clutter, or varied hand orientations. The accuracy drops considerably in such settings, making them less effective in practical, real-world deployments.

Another limitation is **hardware dependency**. Many high-performance systems require depth cameras or infrared sensors, which are not commonly embedded in personal computing devices.

This limits accessibility, especially for users in educational or low-resource settings who only have access to basic webcams. Moreover, such specialized hardware solutions are not scalable for widespread use on web platforms.

The **lack of integration with web applications** is another major issue. While desktop or native apps may support gesture-based control, very few systems offer seamless gesture support within web browsers or progressive web apps. This restricts their usefulness in modern cloud-based workflows where most productivity tools, educational resources, and interactive applications reside on the web.

Additionally, **user customization and adaptability** are often absent in existing systems. Most gesture recognition platforms support a fixed set of predefined gestures. Users cannot define or train custom gestures without deep technical knowledge. This makes the system less adaptable to different cultures, languages, and specific user needs, particularly in assistive technology applications where personalization is critical.

**2.1 Existing Systems**

**Komura T (2024).**The authors explore gesture recognition for human-robot interaction, utilizing monocular and RGB-D cameras to improve real-time responsiveness. While the study demonstrates efficient tracking, it highlights the need for improved system reliability to address variations in gesture movement and environmental conditions.

**M. S. Amin (2024).**This study provides a systematic review of vision-based, sensor-based, and hybrid gesture recognition approaches. It discusses challenges related to dataset size and continuous gesture recognition, emphasizing improvements in scalable AI models to enhance real-time tracking accuracy.

**Abdulloh (2021).**The authors introduce MediaPipe-based gesture recognition, analyzing its application in user-guided interactions. While MediaPipe enables efficient hand tracking, the study notes its basic implementation lacks advanced functionality like adaptive learning or gesture refinement.

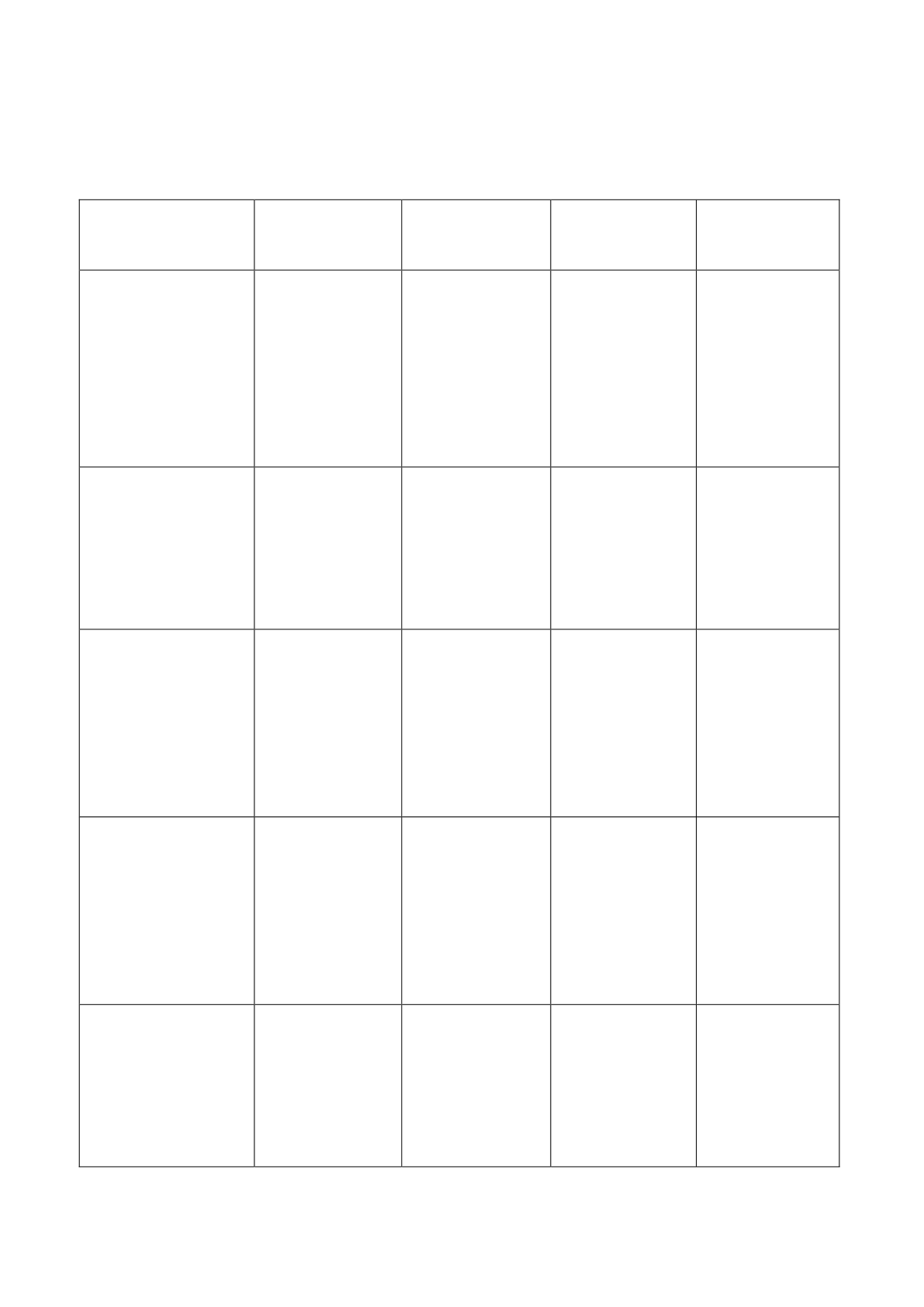
**S. S. Rautaray (2021).**This research explores dynamic gesture recognition using 3D pose estimation and deep learning models such as ConvLSTM networks. The paper discusses challenges related to complex environments and suggests refinements in CNN-based architectures for gesture classification improvements.

**Zhigang (2020).**A general overview of computer vision and sensor-based gesture recognition techniques, analyzing performance benchmarks across various methodologies. The study notes variations in recognition accuracy, recommending standardization for optimized gesture detection.

**Ali A. Alani (2019).**This research provides an in-depth survey on multi-modal gesture recognition, integrating skeleton tracking, depth sensing, and RGB data. It identifies fusion complexity as a challenge, suggesting optimized multi-modal integration techniques.

**Aksaç A (2019).**The study investigates low-cost human-computer interaction (HCI) using skin/motion detection and convex hull methods for gesture interaction. Environmental interference is identified as a limitation, leading to recommendations for robust feature extraction.

**D.L.Quam (2019).**A computer vision-based virtual mouse system utilizing color segmentation for gesture interaction. The paper discusses device-free interaction limitations, with suggestions to improve gesture tracking precision.



**2.2 Existing System and its Limitations:**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Title** | **Technology** | **Limitation** | **Authors** | **Year** |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Systematic Review | Vision, Sensor, | Dataset size, | M. S. Amin, S. | 2024 |
| Continuous |
| on Hand Gesture | T. H. Rizvi, M. |
| Hybrid Methods | gesture |
| Recognition | M. Hossain |
| recognition |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Gesture Recognition | Monocular & | Need for | Komura T, Lam | 2024 |
| for Human-Robot | improved system |
| RGB-D Cameras | W-C |
| Interaction | reliability |

Basic

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| MediaPipe-Based | MediaPipe ML | implementation, | Abdulloh, | 2021 |
| Gesture System | Framework | lacks advanced | Sudrajat Harris |

functionality

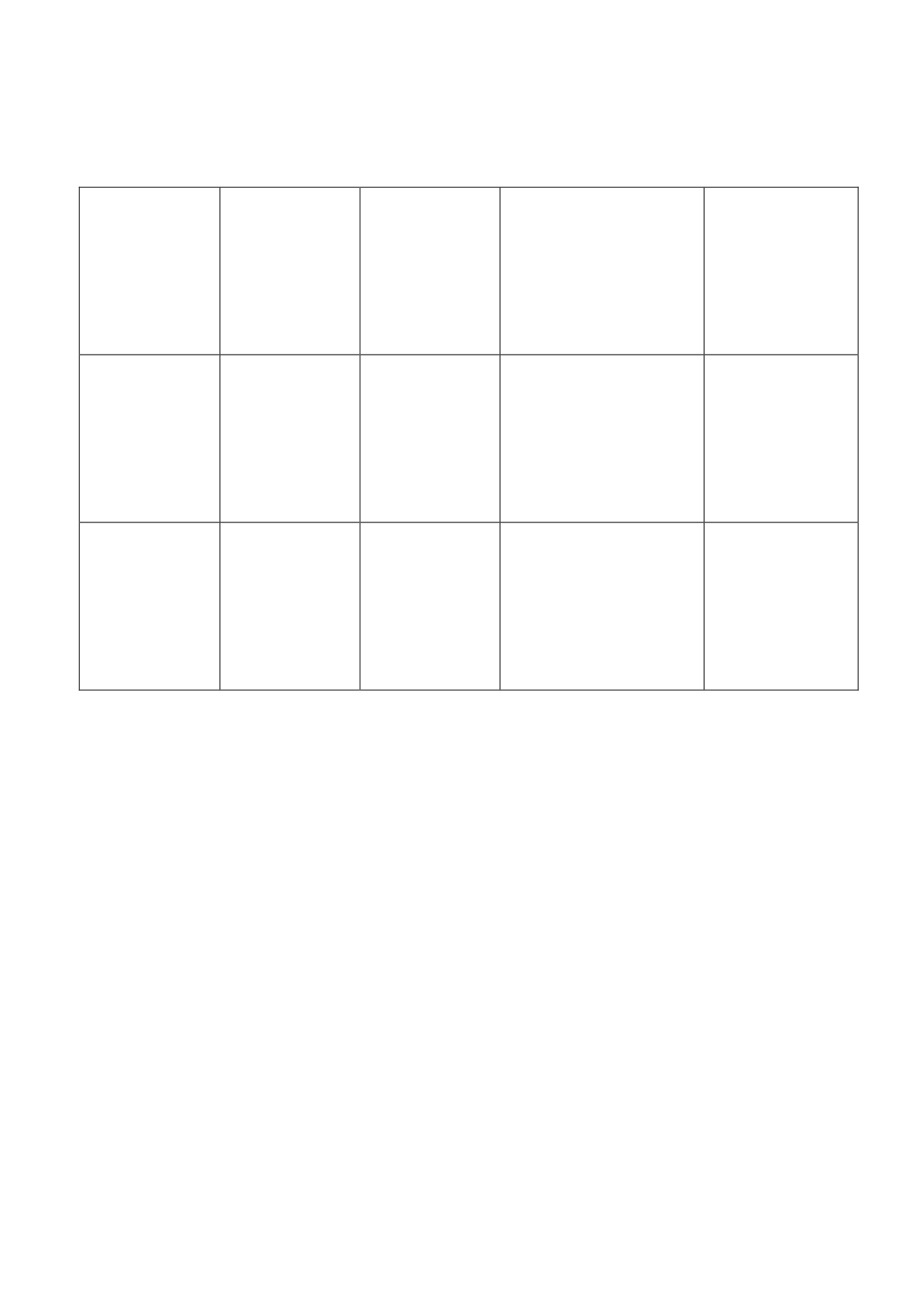
Complex

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Dynamic Gesture | 3D Pose | environment, | S. S. Rautaray, | 2021 |
| Recognition with 3D | Estimation, Deep | high processing |
| A. Agrawal |
| CNN + ConvLSTM | Learning | demands |

|  |  |  |
| --- | --- | --- |
| General Overview of | Computer | Performance |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Hand Gesture | Vision, Sensor | benchmarks vary | Zhigang | 2020 |

|  |  |  |
| --- | --- | --- |
| Techniques | Tech | per method |



|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Survey on Hand | Multi-modal | Fusion | Ali A. Alani, Georgina | 2019 |
| Gesture |
| (Skeleton, | Cosma, Aboozar, T. M. |
| Recognition | complexity |
| Depth, RGB) | McGinnity |
| (HGR) |

Vision

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Low-Cost HCI | (Skin/Motion | Environmental | Aksaç A, Öztürk O, | 2019 |
| with Vision | Detection, | interference | Özyer T |

Convex Hull)

|  |  |  |
| --- | --- | --- |
| Virtual Mouse | Computer | Device-free but |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| via Vision and | Vision (Color | limited | D.L. Quam | 2019 |

|  |  |  |
| --- | --- | --- |
| Gestures | Segmentation) | precision |

**2.3 Proposed System**

The proposed system, titled **Hand Gesture Control for Web Apps**, is a real-time hand gesture recognition application designed to provide a seamless, touchless interaction mechanism for web interfaces. The system leverages computer vision and machine learning technologies to recognize specific hand gestures and convert them into browser commands such as mouse movement, clicks, scrolling, and drag-and-drop operations. This allows users to interact with web applications using only their hand gestures, eliminating the need for traditional input devices.

The backbone of the system is built using the MediaPipe framework from Google, which is used for high-fidelity hand tracking and landmark detection. It detects 21 key points on the hand in real time using a webcam feed. These landmarks are processed using a gesture recognition algorithm implemented in Python, which determines the state of each finger and classifies the current gesture. For instance, a raised index finger is interpreted as a cursor move action, a peace sign triggers a left-click, an open palm executes a right-click, and thumbs up/down gestures are mapped to scrolling functions.

In addition to gesture recognition, the system incorporates PyAutoGUI, a Python library that allows programmatic control of the mouse and keyboard. Recognized gestures are translated into screen coordinates and actions using this library, enabling direct interaction with the web browser. To enhance user experience, the system includes smoothing algorithms to avoid jitter in cursor movement, gesture hold-time thresholds to prevent accidental inputs, and cooldown intervals to avoid repetitive or rapid clicks.

A Graphical User Interface (GUI) built using tkinter allows users to start and stop the camera, initiate gesture tracking, view current system status, and read gesture instructions. The GUI also supports a debug mode that displays finger status in real time, which is especially helpful during development or customization. The interface logs all major system actions, making it easy to monitor gesture recognition, camera status, and any errors encountered.

**CHAPTER-3**

**3.1 Software Requirements**

For the Hand Gesture Control for Web Apps system to function efficiently, it requires a well-

balanced combination of hardware and software.

The software stack used in this project is entirely open - source, ensuring accessibility,

customization, and platform independence. Each component was chosen carefully to fulfill the real-

time requirements of gesture recognition, system integration, and web-based interaction.

**1. Python (Version 3.7 or Above)**

**Purpose:** Python is the core programming language used in the development of the application.

**Justification:** Python is known for its simplicity, readability, and vast ecosystem of libraries in

machine learning, computer vision, and GUI development. Its ability to integrate various packages

with minimal overhead makes it ideal for real-time systems such as gesture control. Python' s support

for multi-threading, event-driven programming, and cross-platform development ensures the system

can run smoothly on Windows, Linux, or macOS.

**2. OpenCV (cv2)**

**Purpose:** OpenCV (Open Source Computer Vision Library) is used for accessing the webcam,

capturing video frames, and performing image preprocessing tasks.

**Key Functions in the Project:**

• Captures real-time video from the webcam.

• Flips and resizes video frames.

• Converts images from BGR to RGB format for compatibility with MediaPipe.

• Draws landmarks and gesture annotations on the frame for debugging and display.

**Justification:** OpenCV is a widely used computer vision library optimized for real-time performance.

Its efficient image – processing pipelines and broad compatibility make it the go-to solution for tasks

such as camera feed handling and frame analysis.

**3. MediaPipe (by Google)**

**Purpose:** MediaPipe is used for hand detection and landmark tracking.

**Key Functions in the Project:**

• Detects 21 hand keypoints using a pre-trained machine learning model.

• Identifies hand pose and finger positions in real time.

• Tracks hand movements continuously without reinitialization.

**Justification:** MediaPipe offers high-performance, lightweight hand-tracking capabilities optimized

for real - time applications. It provides precise landmark detection even in challenging lighting

conditions and with only partial hand visibility, which is essential for accurate gesture recognition.

**4. PyAutoGUI**

**Purpose:** PyAutoGUI is used to control the mouse pointer and simulate click or scroll actions based on the recognized gestures.

**Key Functions in the Project:**

• Moves the cursor across the screen.

• Performs left-click, right-click, drag, and scroll operations.

• Supports screen resolution detection and movement calibration.

**Justification:**  PyAutoGUI allows the gesture recognition system to directly interface with the

operating system’s input layer. This enables web app interaction through simulated mouse events,

effectively translating gestures into system - level inputs that work across browsers and platforms.

**5. Tkinter**

**Purpose:** Tkinter is used for creating the GUI (Graphical User Interface) for the application.

**Key Functions in the Project:**

• Provides buttons to start/stop the camera and tracking.

• Displays real-time status updates (e.g., camera status, gesture detected).

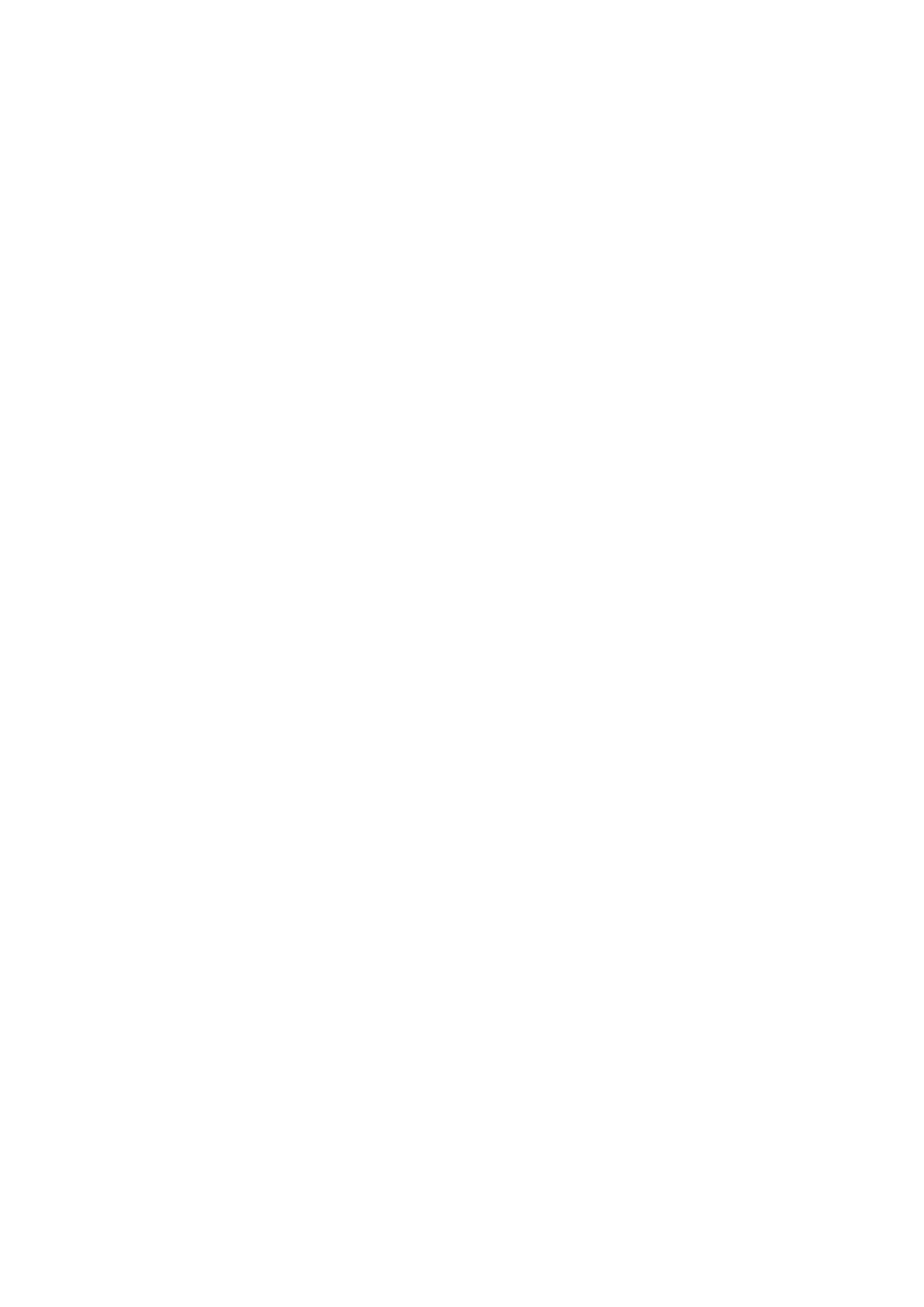
• Shows gesture instructions and an action log in the main window.

• Offers an optional debug window for finger position tracking.

**Justification:** Tkinter is Python’s standard GUI library, lightweight, easy to deploy, and does not

require external dependencies. It is ideal for providing a clean and functional interface for end users

who may not have technical expertise.



**6. NumPy**

**Purpose:** NumPy is used for mathematical operations and efficient handling of numerical data.

**Key Functions in the Project:**

• Assists in calculating distances between hand landmarks.

• Helps with gesture smoothing algorithms.

• Supports numerical transformations needed for cursor control.

**Justification:** Real - time gesture processing requires fast numerical computations, such as distance

calculations between keypoints, vector smoothing, and signal normalization. NumPy accelerates these

operations while maintaining code readability and efficiency.

**7. Additional Built-In Python Modules**

**Module**   **Purpose**

**math** Used for Euclidean distance calculation (e.g., for pinch detection).

**threading** Runs the video capture and gesture recognition asynchronously.

**time** Used for gesture hold detection and cooldowns for click events.

**datetime** Generates timestamps for the action log.

**messagebox** Displays error messages in a GUI pop-up format.

**Installation Instructions**

All required libraries can be installed using Python's package manager pip. The following command

installs all necessary dependencies:

**pip install opencv-python mediapipe pyautogui numpy**

Tkinter is included by default in standard Python installations. However, if using a minimal

environment, it can be installed via:

• **Windows/Linux:** Pre-installed with Python

• **Ubuntu/Debian:** sudo apt-get install python3-tk

**Development and Runtime Environment**

• **IDE/Editor**: Visual Studio Code / PyCharm / Jupyter Notebook (optional)

• **Operating System**: Windows 10/11, Ubuntu 20.04+, or macOS 12+

• **Browser Compatibility**: All modern browsers (Google Chrome, Mozilla Firefox, Edge)

**Python Environment**: Preferably virtualenv or conda for isolated package management.

**3.2 Hardware Requirements**

The hardware requirements for the **Hand Gesture Control for Web Apps** project have been defined to ensure real-time performance, high gesture recognition accuracy, and user-friendly experience without the need for specialized or expensive hardware. Since the system relies on visual input and real-time processing, a balance between processing power and affordability is maintained to ensure accessibility for both developers and end users.

**1. Processing Unit (CPU)**

• **Minimum Requirement**: Intel Core i5 (8th Gen or above) or AMD Ryzen 5 equivalent

• **Recommended**: Intel Core i7 / AMD Ryzen 7 or higher

**Purpose and Justification**: Gesture recognition and computer vision tasks involve continuous frame capturing, processing, landmark detection, and classification, all in real time. A multi-core processor ensures efficient parallel execution of GUI threads, camera input handling, and machine learning computations. Although GPU acceleration is not mandatory, a capable CPU significantly reduces latency and improves response time during gesture interaction.

**2. Random Access Memory (RAM)**

• **Minimum Requirement**: 8 GB

• **Recommended**: 12–16 GB (for smooth multitasking and higher frame processing rates)

**Purpose and Justification**: RAM is required to handle image buffers, application runtime, and background processes such as gesture tracking, GUI updates, and input simulation. Higher memory ensures smooth multitasking and prevents frame dropping, especially when the system is used alongside web browsers or development environments.

**3. Camera/Webcam**

• **Minimum Requirement**: 720p HD webcam (internal or external), 30 FPS capture rate

• **Recommended**: 1080p Full HD webcam, 60 FPS for high-accuracy tracking

**Purpose and Justification**:

The webcam acts as the primary sensor for capturing real - time video input for gesture recognition.

Higher resolution and frame rates improve the accuracy of hand landmark detection and allow for

smoother cursor movement. Although a basic 720p webcam is sufficient for prototyping, 1080p

cameras provide clearer input data and particularly in low - light environments or with fast hand

movements.

**4. Display Unit**

• **Minimum Requirement**: 13-inch monitor (720p resolution)

• **Recommended**: 15.6-inch or larger (1080p resolution)

**Purpose and Justification**:

A clear and sufficiently large display is essential for accurate cursor feedback, interface navigation, and

interaction with web applications. Higher - resolution screens enable finer cursor control, especially

when performing precision tasks such as drag -and - drop or scroll gestures. A larger screen also

improves visibility of GUI elements in the application dashboard.

**5. Storage Device**

• **Minimum Requirement**: 100 MB of available storage

• **Recommended**: 256 GB SSD (system) with at least 1 GB free for development logs and

updates

**Purpose and Justification**:

The application and its dependencies are lightweight and do not require large storage space. However,

using an SSD significantly improves system boot time, library loading, and overall responsiveness of

the development environment, especially when Python virtual environments or large browser-based

applications are used concurrently.

**6. Input Devices (For Setup and Debugging)**

• **Keyboard & Mouse**: Required during setup, testing, and debugging phases

• **Touchpad (Optional)**: Used for traditional fallback in case gesture recognition is inactive

**Justification**:

While the system is designed to function hands-free, traditional input devices are necessary for initial

configuration, system debugging, and launching the application and Once operational, the gesture

recognition system can replace most mouse-related tasks in the browser.

**7. Power Supply and Environment**

• **Power Supply**: Standard 65W laptop adapter or desktop PSU

• **Operating Environment**: Indoor environment with moderate, consistent lighting

**Justification**:

Gesture recognition systems require a well-lit environment for accurate landmark detection. Harsh

shadows, strong backlighting, or extremely low light can reduce model accuracy. While MediaPipe is

robust to lighting variations, maintaining consistent ambient lighting ensures better system reliability.

**Summary Table of Hardware Requirements**

|  |  |  |
| --- | --- | --- |
| Component | Minimum Specification | Recommended Specification |
| Processor | Intel i5 (8th Gen) / Ryzen 5 | Intel i7 / Ryzen 7 |
| RAM | 8 GB | 12–16 GB |
| Camera | 720p HD webcam @30fps | 1080p HD webcam @60fps |
| Display | 13" screen, 720p resolution | 15.6"+ screen, 1080p resolution |
| Graphics | Integrated GPU | NVIDIA GTX 1050+  (for future upgrades) |
| Lighting | Indoor with natural/LED lighting | Uniform lighting without  backlight glare |

**3.3 Functional Requirements**

Non-functional requirements describe how the system performs under specific conditions. These relate

to quality attributes such as performance, scalability, usability, and security.

• Real-Time Video Capture and Hand Detection

O Captures video from the webcam in real time using OpenCV.

O Uses MediaPipe to detect 21 hand landmarks per frame.

O Detects a single hand accurately with continuous tracking.

• Gesture Recognition and Classification

O Identifies predefined hand gestures:

▪ 👆 **Index Finger Only** – Cursor movement

▪ ✌️ **Peace Sign** – Left click

▪ 🖐️ **Open Palm** – Right click

▪ 👍 **Thumbs Up** – Scroll up

▪ 👎 **Thumbs Down** – Scroll down

▪ ✊ **Fist** – Drag mode

▪ 🤏 **Pinch** – Precision movement

O Uses finger states and relative landmark positions to classify gestures.

• Action Execution and Mapping

O Maps each gesture to a mouse or browser action using PyAutoGUI:

▪ Move cursor, left/right click, scroll, and drag.

O Actions are triggered only when gestures are held for a short duration (e.g., 300ms) to

prevent accidental inputs.

• Cursor Movement with Smoothing

O Applies a smoothing algorithm to make cursor motion fluid and responsive.

O Reduces jitter and prevents abrupt changes in cursor position.

• Graphical User Interface (GUI)

O Allows users to:

▪ Start/Stop camera

▪ Enable/disable gesture tracking

▪ View gesture status

▪ Read gesture instructions

▪ See real-time logs and feedback

O Built using tkinter with color-coded elements for easy interaction.

• Gesture Debug Mode

O Shows current finger states (e.g., [1, 1, 0, 0, 0]) for troubleshooting.

O Can be toggled during development or testing.

• Action Logging

O Displays time-stamped logs of every recognized gesture and corresponding action.

O Logs are shown in a scrollable text window within the GUI.

• Error Handling and Safe Exit

O Gracefully handles errors (e.g., camera not available).

Allows users to close the application safely using the GUI or a keyboard interrupt

**3.4 Non-Functional Requirements :**

• High Performance and Responsiveness

O Gesture recognition and action execution should occur within 200 ms of user input.

O Ensures smooth, real-time interaction without noticeable lag.

• High Accuracy and Reliability

O Recognition accuracy should be above 90% under standard lighting.

O System should avoid false positives and only respond to clear gestures.

• Ease of Use and Accessibility

O GUI designed for users with minimal technical knowledge.

O Clearly labeled buttons, real-time feedback, and instructions included.

O Suitable for users with physical impairments who cannot use traditional input devices.

• Cross-Platform Compatibility

O Works on Windows, Linux, and macOS with Python 3 and a standard webcam.

O No platform-specific dependencies or device drivers required.

• Scalability and Extensibility

O Modular code structure allows for:

▪ Adding new gestures

▪ Supporting multiple hands

▪ Integrating voice commands or facial recognition in the future

O Gesture-action mappings can be easily modified or extended.

• Privacy and Security

O No image or video data is stored or transmitted.

O All processing is local on the user’s device.

O Includes a fail-safe (e.g., moving the mouse to a corner disables actions).

• Maintainability and Documentation

O Code is well-commented and structured for easy debugging and future development.

O README file with setup instructions provided.

O Includes usage guide and system logs to assist in troubleshooting.

• Robustness in Variable Conditions

O Designed to function under different lighting setups (within reasonable limits).

O Handles temporary hand loss or occlusion without crashing.

**CHAPTER-4**

**4.1 Methodology :**

The system follows a modular and event-driven approach, integrating multiple technologies such as computer vision, machine learning, and system-level input simulation to create a real-time hand gesture control mechanism. The methodology is divided into several interconnected stages, each handling a specific part of the gesture recognition and response pipeline.

1. Input Acquisition (Webcam Video Feed)

The system begins by accessing the user's webcam using OpenCV. A video stream is captured in real-time at a resolution of 640x480 pixels, which balances performance and detail. Each frame from the video stream is flipped horizontally to create a mirror effect, providing a natural user interface where right and left hand movements map intuitively to screen directions.

2. Hand Detection and Landmark Tracking (MediaPipe)

Each video frame is passed to MediaPipe’s hand tracking module, which detects the presence of a hand and maps 21 landmarks representing critical points on fingers and joints. MediaPipe uses a combination of machine learning models and hand pose estimation algorithms to detect and track these landmarks with high accuracy, even under slight variations in lighting and hand orientation.

The key advantages of using MediaPipe include:

• Real-time performance

• High precision in detecting landmark positions

• Ability to work on CPU (no GPU needed)

• Robustness to background noise

3. Gesture Recognition Logic

Once the landmarks are obtained, the system uses a custom logic layer to interpret which fingers are extended and which are folded. Based on the relative positions of specific landmarks (e.g., tips vs. joints), the system determines the hand’s shape and classifies it into a defined gesture.

For example:

• If only the index finger is extended → "Point" gesture

• If the index and middle fingers are extended → "Peace" gesture

• If all fingers are extended → "Open hand"

• If fingers are curled in → "Fist"

In addition, the distance between the thumb and index finger tips is used to detect "Pinch" gestures.

This logic is rule-based for transparency and customizability but can be enhanced later using machine

learning classifiers if needed.

4. Gesture-to-Action Mapping (pyautogui Integration)

Each recognized gesture is mapped to a system-level input action using the pyautogui library. These

include:

• Cursor movement: Mapping the x/y coordinates of the index finger to screen resolution

• Mouse clicks: Triggered on specific gestures like “peace” (left click) or “open hand” (right

click)

• Scrolling: Thumb-up and thumb-down gestures scroll the screen content

• Dragging: A “fist” gesture initiates mouse down (drag) and releases when the gesture ends

To prevent accidental actions, the system enforces gesture hold timing, requiring the gesture to remain

steady for at least 300 milliseconds before an action is performed and A click cooldown is also

implemented to avoid repeated clicks.

5. GUI and User Feedback (Tkinter)

A custom GUI is developed using tkinter to allow users to control and monitor the system. Features

include:

• Camera start/stop

• Tracking toggle

• Real-time status display (camera and hand detection)

• Gesture recognition display

• Action log panel with timestamped entries

• Debug panel showing finger states (for developers)

This GUI enhances usability for both end users and developers, and makes the system intuitive to

operate without command-line knowledge.

6. Threading and System Responsiveness

To maintain real-time performance and a responsive GUI, the system uses Python’s threading module

to separate video processing from the main application thread. This ensures:

• No freezing or lag in the user interface

• Simultaneous video feed handling and gesture recognition

• Smooth execution of actions and display updates

**4.2 System Modules :** The architecture is composed of six major functional modules, each responsible for a specific subsystem:

1. Camera & Input Module

• Uses OpenCV to capture and process frames from the webcam.

• Performs image transformations (flipping, resizing, color conversion).

• Manages camera start/stop and error handling.

2. Hand Tracking Module

• Powered by MediaPipe Hands.

• Detects and tracks 21 hand landmarks.

• Identifies hand presence and position in each frame.

3. Gesture Recognition Module

• Analyzes landmark data to determine which fingers are extended.

• Uses predefined rules and geometric analysis to classify gestures.

• Recognizes up to 7 distinct hand gestures.

4. Action Mapping & Control Module

• Maps recognized gestures to specific actions via pyautogui:

O Move mouse cursor

O Perform left/right clicks

O Scroll browser content

O Execute drag-and-drop operations

• Includes gesture cooldown timers and safety checks.

5. Graphical User Interface (GUI) Module

• Built with tkinter.

• Displays system status, instructions, and current gesture.

• Provides control buttons (Start Camera, Start Tracking).

• Shows action logs and optionally, debug data.

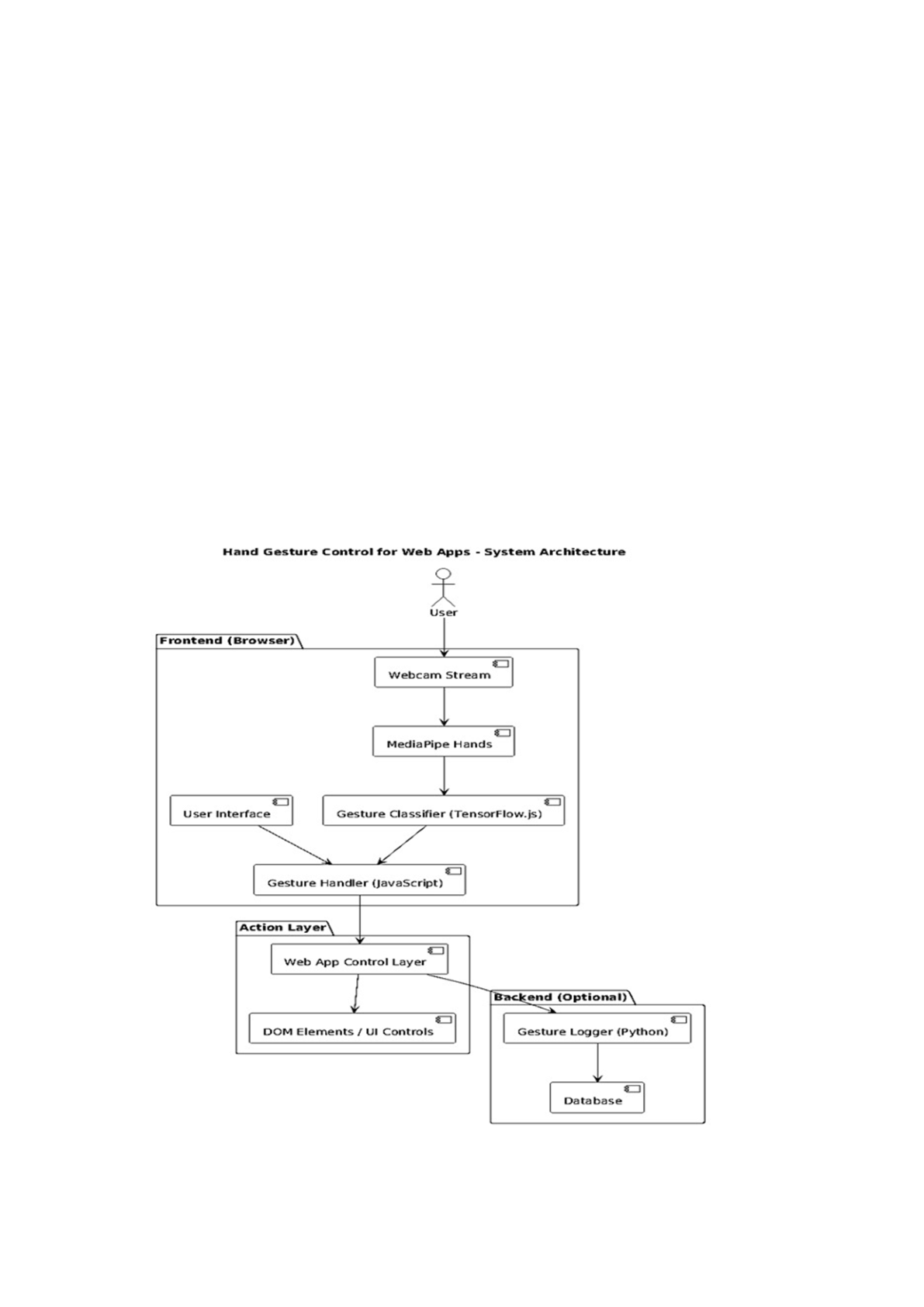
6. Logging & Debugging Module

• Logs actions in a scrollable text window.

• Displays system messages with timestamps.

• Provides internal feedback for gesture recognition and application state.

• Helps in debugging gesture mismatches or tracking failures.



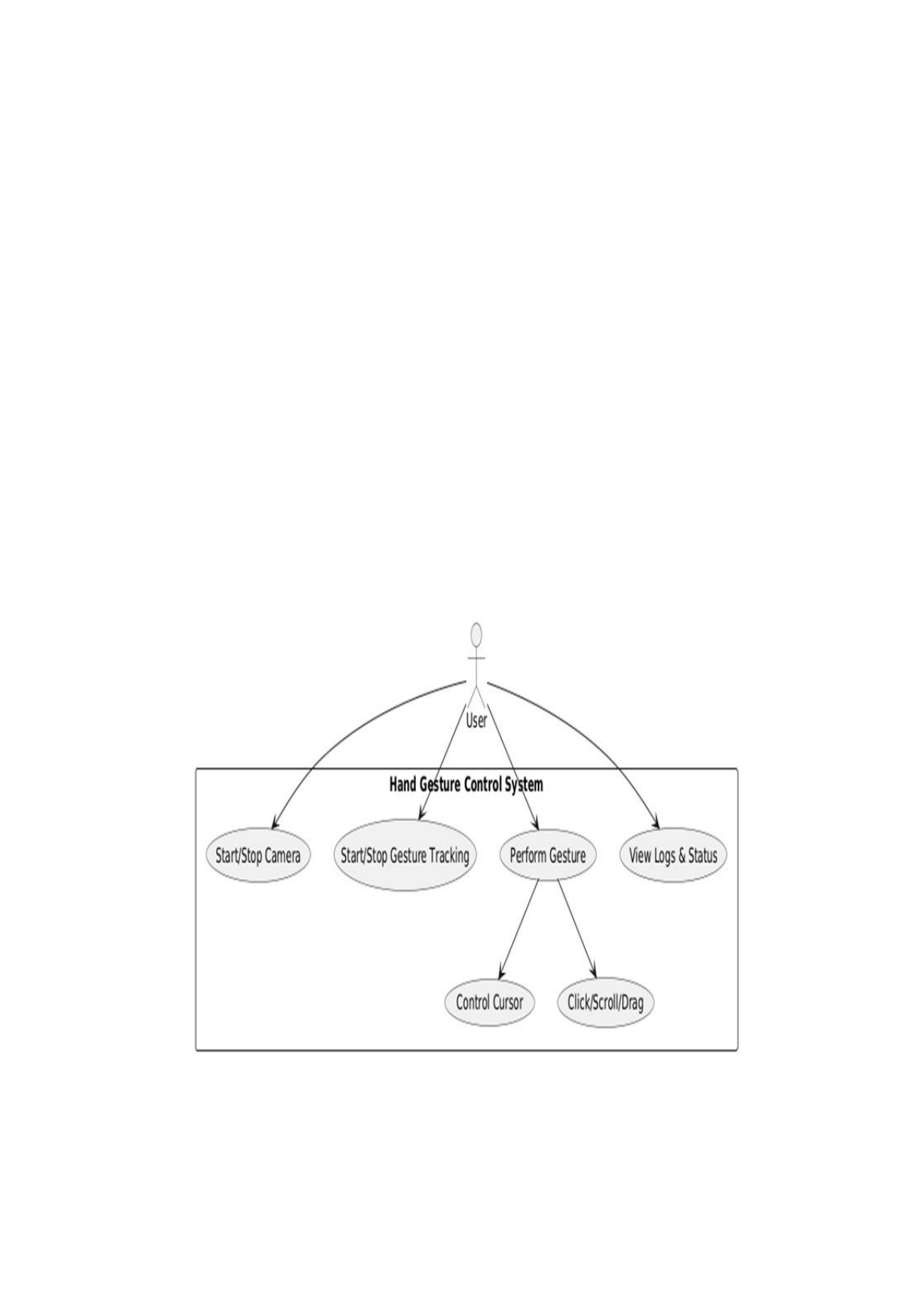
**CHAPTER-5**

**SYSTEM DEISGN:**

**5.1 SYSTEM ARCHITECHTURE:**

System architecture is a comprehensive blueprint that defines the structure, behavior, and interactions of various components within a system—whether it's a software application, a computer system, or a complex network of systems. It provides a high-level view of how the system is organized and how different parts such as hardware, software, data storage, processing units, communication protocols, and user interfaces interact to perform specific functions. In software systems, architecture describes how modules or services are divided, how they communicate (e.g., via APIs or message queues), and how data flows through the system. In hardware systems, it includes the design of processors, memory units, input/output devices, and how they are connected. System architecture also includes considerations for scalability (handling growth in users or data), security (protecting data and operations), maintainability (ease of updates and debugging), and performance (speed and efficiency).

**Fig 1: System Architecture**



**UML DIAGRAMS :**

UML is a method for describing the system architecture in detail using a blueprint. UML represents a collection of best engineering practices that have proven successful in the modeling of large and complex systems. UML is a very important part of developing object-oriented software and the software development process. UML uses mostly graphical notations to express the design of software projects. Using the UML helps project teams communicate, explore potential designs, and validate the architectural design of the software.

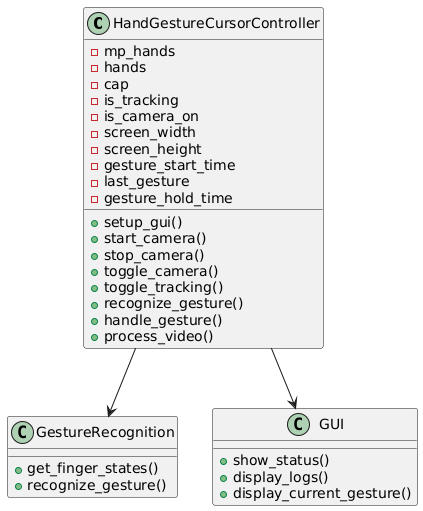
**5.2 USE CASE DIAGRAM :**

A Use Case Diagram is a type of Unified Modeling Language (UML) diagram that represents the interaction between actors (users or external systems) and a system under consideration to accomplish specific goals. It provides a high-level view of the system's functionality by illustrating the various ways users can interact with it.

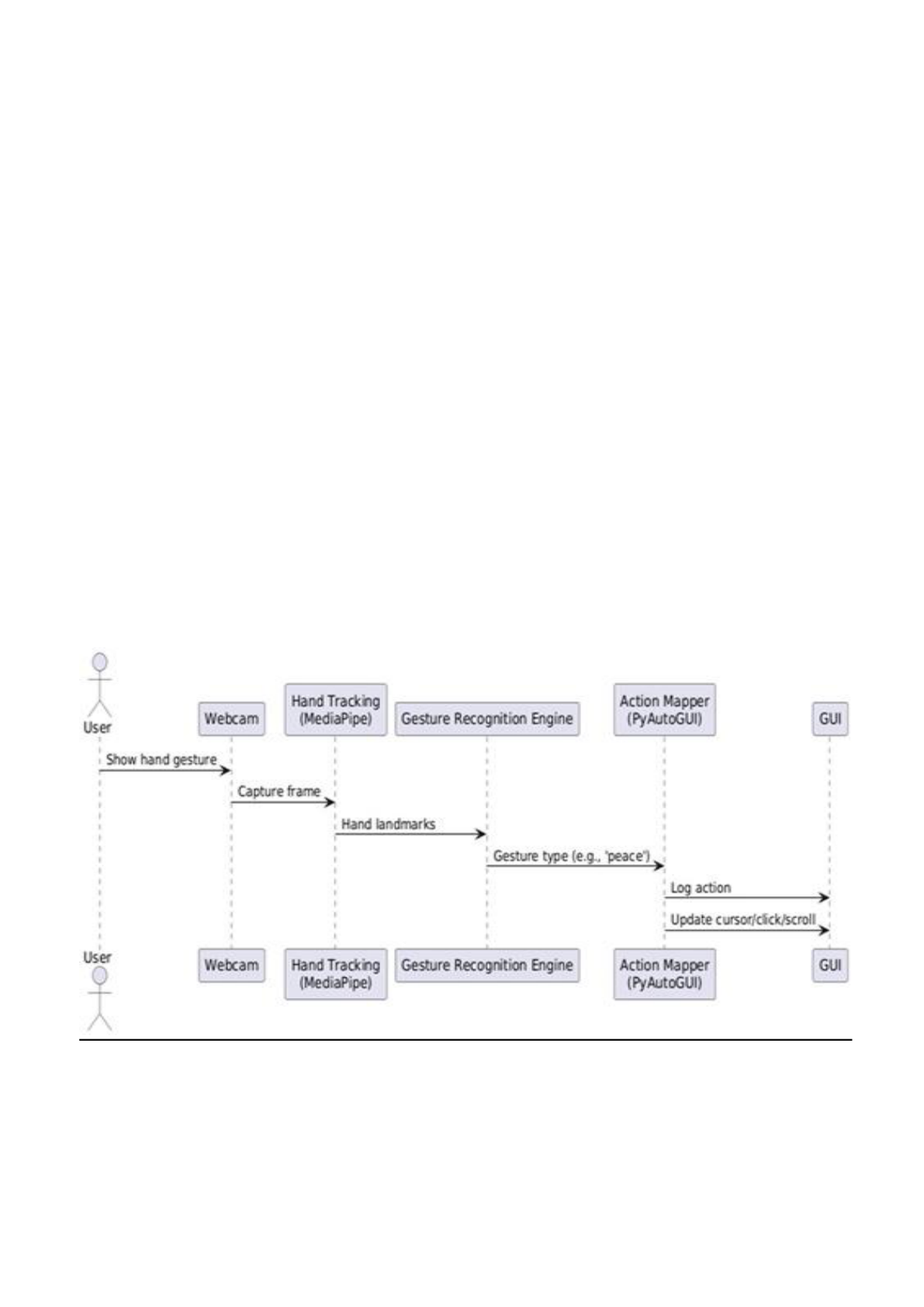
**Fig 2: Use Case Diagram**

**5.3 CLASS DIAGRAM :**

Class diagrams are widely used to describe the types of objects in a system and their relationships. Class diagrams model class structure and contents using design elements such as classes, packages, and objects. Class diagrams describe three different perspectives when designing a system: conceptual, specification, and implementation. These perspectives become evident as the diagram is created and help solidify the design. Class diagrams are arguably the most used UML diagram type. It is the main building block of any object-oriented solution. It shows the classes in a system, attributes and operations of each class, and the relationship between each class. In most modeling tools, a class has three parts: name at the top, attributes in the middle, and operations or methods at the bottom. In large systems with many classes, related classes are grouped together to create class diagrams. Different relationships between diagrams are shown by different types of Arrows. Beside is an image of a class diagram.



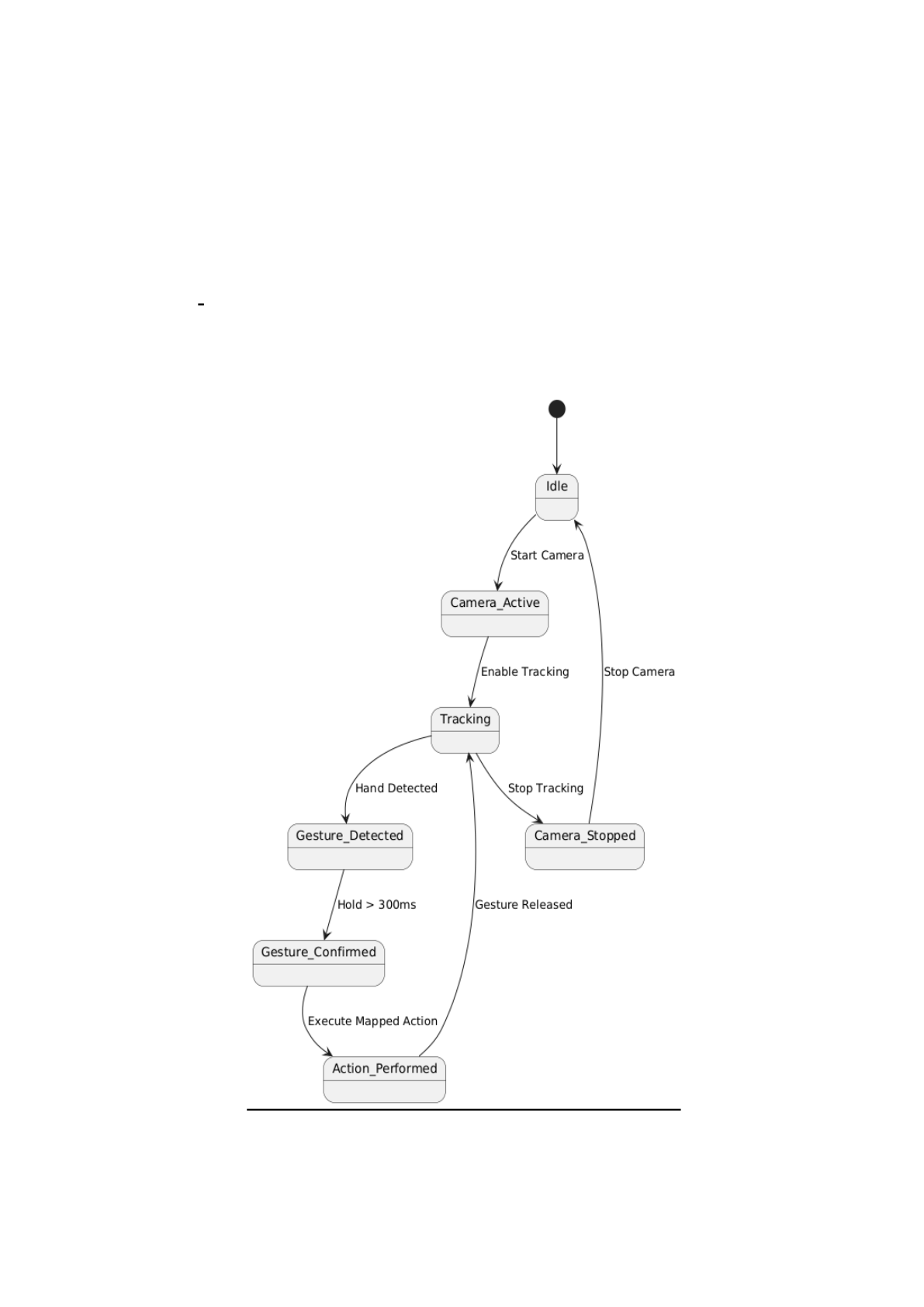
**Fig 3: Class Diagram**



**5.4 SEQUENCE DIAGRAM:**

A Sequence Diagram is a UML behavioral diagram that models the interaction between system components over time. It shows how objects or actors communicate with each other through a sequence of messages to accomplish a specific process or use case. The diagram reads top to bottom, where each participant (object, component, or actor) is represented by a lifeline, and horizontal arrows represent messages or method calls exchanged during the interaction. It's particularly useful for visualizing the order of operations and identifying timing, dependencies, or bottlenecks in a system's workflow. The sequence diagram shows the flow of interactions starting with the user requesting weather data through SkySentinal. SkySentinal calls the OpenWeatherMapAPI to retrieve the weather information. The API returns data, which is stored in the WeatherData class. SkySentinal processes the data and sends alerts if necessary. The diagram also depicts continuous updates and user interactions.

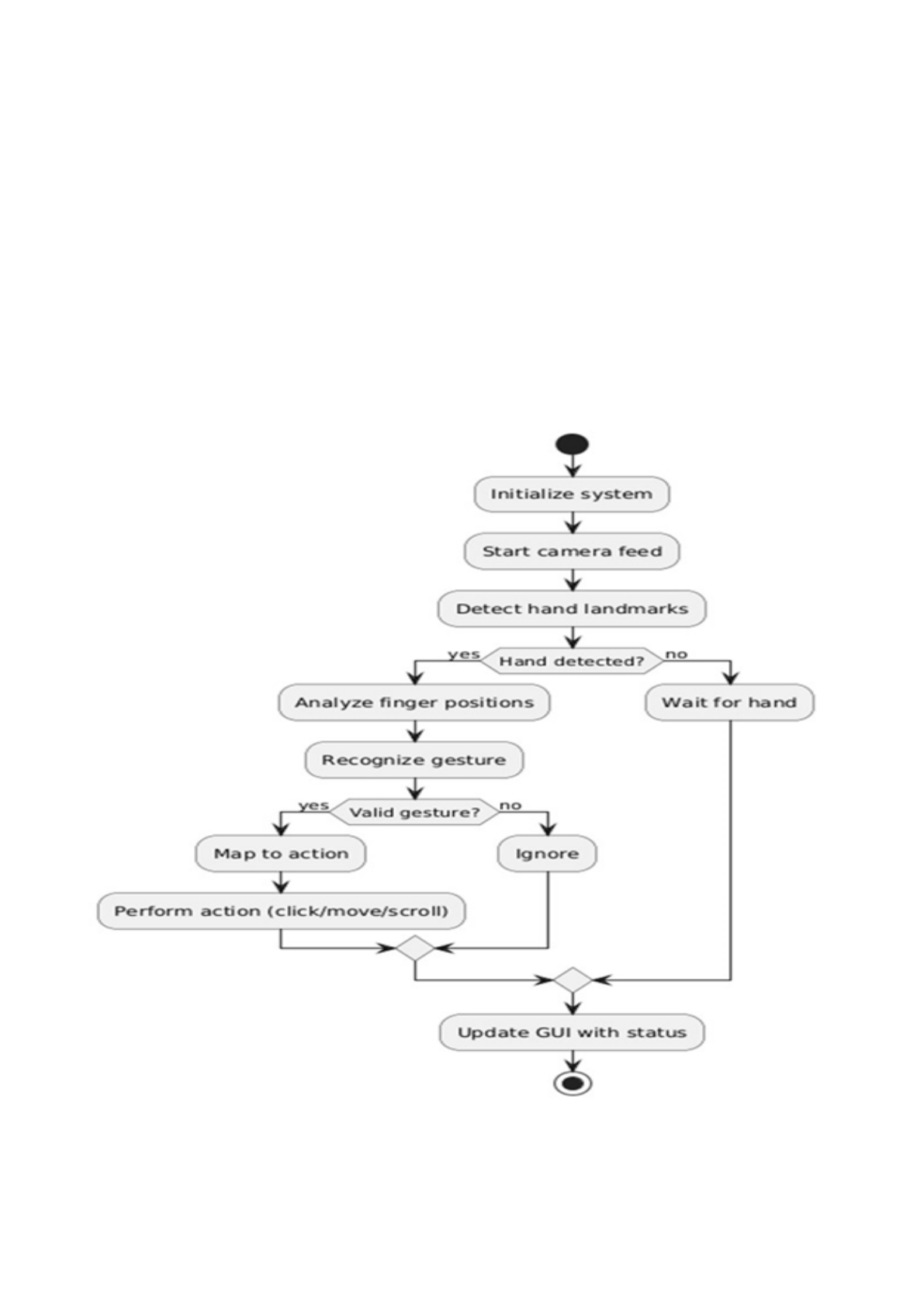
**Fig 4: Sequence Diagram**



**5.5 STATE CHART DIAGRAM:**

A State Chart Diagram, also known as a State Machine Diagram, is a UML behavioral diagram that models the dynamic states of a system or component over time. It shows the various states an object can be in and the transitions between those states based on events or conditions. The diagram consists of states represented by rectangles with rounded corners, and transitions are depicted as arrows between states, triggeredby events. It's particularly useful for visualizing how an object responds to different events, tracking state changes, and ensuring correct flow and behavior within a system.

**Fig 5: State Chart Diagram**



**5.6 ACTIVITY DIAGRAM :**

Activity diagrams describe the workflow behavior of a system. Activity diagrams are similar to state diagrams because activities are the state of doing something. The diagrams describe the state of activities by showing the sequence of activities performed. Activity diagrams can show activities that are conditional or parallel.

**Fig 6: Activity Diagram**

**CHAPTER-6**

**IMPLEMENTATION**   
**6.1 SOURCE CODE :**

import cv2   
import mediapipe as mp   
import pyautogui   
import numpy as np   
import tkinter as tk   
from tkinter import ttk, messagebox   
import threading   
import time   
from datetime import datetime   
import math   
class HandGestureCursorController:   
 def \_\_init\_\_(self):   
 self.mp\_hands = mp.solutions.hands   
 self.hands = self.mp\_hands.Hands(   
 static\_image\_mode=False,   
 max\_num\_hands=1,   
 min\_detection\_confidence=0.7,   
 min\_tracking\_confidence=0.5)   
 self.mp\_draw = mp.solutions.drawing\_utils   
 self.cap = None   
 self.is\_tracking = False   
 self.is\_camera\_on = False   
 self.screen\_width, self.screen\_height = pyautogui.size()   
 self.camera\_width, self.camera\_height = 640, 480   
 self.last\_gesture = 'none'   
 self.gesture\_start\_time = 0   
 self.gesture\_hold\_time = 0.3 # Hold gesture for 300ms before action self.smoothing\_factor = 0.7

self.last\_x, self.last\_y = self.screen\_width // 2, self.screen\_height // 2   
 self.is\_dragging = False   
 self.drag\_start\_pos = None   
 self.last\_click\_time = 0   
 self.click\_cooldown = 0.5   
 self.debug\_mode = False   
 pyautogui.FAILSAFE = True # Move mouse to corner to stop   
 pyautogui.PAUSE = 0.01 # Small pause between actions   
 self.setup\_gui()   
 def setup\_gui(self):   
 self.root = tk.Tk()   
 self.root.title("Hand Gesture Cursor Controller")   
 self.root.geometry("500x700")   
 self.root.resizable(False, False)   
 self.root.configure(bg='#2b2b2b')   
 style = ttk.Style()   
 style.theme\_use('clam')   
 style.configure('Title.TLabel', font=('Arial', 16, 'bold'), background='#2b2b2b',   
foreground='#ff4080')   
 style.configure('Info.TLabel', font=('Arial', 10), background='#2b2b2b', foreground='white') style.configure('Status.TLabel', font=('Arial', 12), background='#2b2b2b', foreground='#00ff00')

title\_label = ttk.Label(self.root, text=" Hand Gesture Cursor Control", style='Title.TLabel') title\_label.pack(pady=20)   
 status\_frame = tk.Frame(self.root, bg='#2b2b2b')   
 status\_frame.pack(pady=10, padx=20, fill='x')

self.camera\_status\_label = ttk.Label(status\_frame, text=" Camera: OFF", style='Info.TLabel') self.camera\_status\_label.pack(side='left')

self.hand\_status\_label = ttk.Label(status\_frame, text=" Hands: 0", style='Info.TLabel') self.hand\_status\_label.pack(side='right')   
 gesture\_frame = tk.Frame(self.root, bg='#3b3b3b', relief='raised', bd=2)   
 gesture\_frame.pack(pady=10, padx=20, fill='x')   
 ttk.Label(gesture\_frame, text="Current Gesture:", font=('Arial', 12),

background='#3b3b3b', foreground='white').pack(pady=5)   
 self.current\_gesture\_label = ttk.Label(gesture\_frame, text="NONE",   
 font=('Arial', 14, 'bold'),   
 background='#3b3b3b', foreground='#ff4080')   
 self.current\_gesture\_label.pack(pady=5)   
 self.finger\_status\_frame = tk.Frame(self.root, bg='#3b3b3b', relief='raised', bd=2)   
 if self.debug\_mode:   
 self.finger\_status\_frame.pack(pady=10, padx=20, fill='x')   
 ttk.Label(self.finger\_status\_frame, text="Finger Status:", font=('Arial', 12),   
 background='#3b3b3b', foreground='white').pack(pady=5)   
 self.finger\_status\_label = ttk.Label(self.finger\_status\_frame, text="[0, 0, 0, 0, 0]",   
 font=('Arial', 12),   
 background='#3b3b3b', foreground='#4080ff')   
 self.finger\_status\_label.pack(pady=5)   
 control\_frame = tk.Frame(self.root, bg='#2b2b2b')   
 control\_frame.pack(pady=20)   
 self.camera\_button = tk.Button(control\_frame, text="Start Camera",   
 command=self.toggle\_camera,   
 bg='#ff4080', fg='white', font=('Arial', 12, 'bold'),   
 width=15, height=2)   
 self.camera\_button.pack(side='left', padx=10)   
 self.tracking\_button = tk.Button(control\_frame, text="Start Tracking",   
 command=self.toggle\_tracking,   
 bg='#4080ff', fg='white', font=('Arial', 12, 'bold'),   
 width=15, height=2, state='disabled')   
 self.tracking\_button.pack(side='left', padx=10)   
 instructions\_frame = tk.Frame(self.root, bg='#3b3b3b', relief='raised', bd=2)   
 instructions\_frame.pack(pady=20, padx=20, fill='both', expand=True)   
 ttk.Label(instructions\_frame, text="Gesture Controls:",   
 font=('Arial', 14, 'bold'), background='#3b3b3b', foreground='white').pack(pady=10) instructions = [ " Index Finger → Move Cursor",

" Two Fingers (Peace) → Left Click",

" Open Hand → Right Click",

" Thumbs Up → Scroll Up",

" Thumbs Down → Scroll Down",

" Closed Fist → Drag Mode",

" Pinch → Precision Mode" ]   
 for instruction in instructions:   
 ttk.Label(instructions\_frame, text=instruction, font=('Arial', 11),   
 background='#3b3b3b', foreground='#cccccc').pack(pady=3, anchor='w', padx=20) log\_frame = tk.Frame(self.root, bg='#3b3b3b', relief='raised', bd=2)   
 log\_frame.pack(pady=10, padx=20, fill='both', expand=True)   
 ttk.Label(log\_frame, text="Action Log:", font=('Arial', 12, 'bold'),   
 background='#3b3b3b', foreground='white').pack(pady=5)   
 log\_scroll\_frame = tk.Frame(log\_frame, bg='#3b3b3b')   
 log\_scroll\_frame.pack(fill='both', expand=True, padx=10, pady=5)   
 self.log\_text = tk.Text(log\_scroll\_frame, height=8, bg='black', fg='#00ff00',   
 font=('Courier', 9), wrap='word')   
 log\_scrollbar = tk.Scrollbar(log\_scroll\_frame)   
 self.log\_text.pack(side='left', fill='both', expand=True)   
 log\_scrollbar.pack(side='right', fill='y')   
 self.log\_text.config(yscrollcommand=log\_scrollbar.set)   
 log\_scrollbar.config(command=self.log\_text.yview)   
 self.log\_action("System initialized. Ready to start camera.")   
 self.root.protocol("WM\_DELETE\_WINDOW", self.on\_closing)   
 def log\_action(self, message):   
 timestamp = datetime.now().strftime("%H:%M:%S")   
 log\_entry = f"[{timestamp}] {message}\n"   
 self.log\_text.insert('end', log\_entry)   
 self.log\_text.see('end')   
 if self.log\_text.index('end-1c').split('.')[0] > '100':   
 self.log\_text.delete('1.0', '20.0')

def toggle\_camera(self):   
 if not self.is\_camera\_on:   
 self.start\_camera()   
 else:   
 self.stop\_camera()   
 def start\_camera(self):   
 try:   
 self.cap = cv2.VideoCapture(0)   
 self.cap.set(cv2.CAP\_PROP\_FRAME\_WIDTH, self.camera\_width) self.cap.set(cv2.CAP\_PROP\_FRAME\_HEIGHT, self.camera\_height) if not self.cap.isOpened():   
 raise Exception("Could not open camera")   
 self.is\_camera\_on = True   
 self.camera\_button.config(text="Stop Camera", bg='#ff4040')   
 self.tracking\_button.config(state='normal')

self.camera\_status\_label.config(text=" Camera: ON", foreground='#00ff00') self.video\_thread = threading.Thread(target=self.process\_video, daemon=True) self.video\_thread.start()   
 self.log\_action("Camera started successfully")   
 except Exception as e:   
 messagebox.showerror("Camera Error", f"Failed to start camera: {str(e)}") self.log\_action(f"Camera error: {str(e)}")   
 def stop\_camera(self):   
 self.is\_camera\_on = False   
 self.is\_tracking = False   
 if self.cap:   
 self.cap.release()   
 cv2.destroyAllWindows()   
 self.camera\_button.config(text="Start Camera", bg='#ff4080')   
 self.tracking\_button.config(text="Start Tracking", bg='#4080ff', state='disabled')

self.camera\_status\_label.config(text=" Camera: OFF", foreground='#ff4040')

self.hand\_status\_label.config(text=" Hands: 0")

self.current\_gesture\_label.config(text="NONE")   
 self.log\_action("Camera stopped")   
 def toggle\_tracking(self):   
 self.is\_tracking = not self.is\_tracking   
 if self.is\_tracking:   
 self.tracking\_button.config(text="Stop Tracking", bg='#ff4040') self.log\_action("Gesture tracking started")   
 else:   
 self.tracking\_button.config(text="Start Tracking", bg='#4080ff') self.current\_gesture\_label.config(text="NONE")   
 self.log\_action("Gesture tracking stopped")   
 def get\_finger\_states(self, landmarks):   
 tips = [4, 8, 12, 16, 20] # Thumb, Index, Middle, Ring, Pinky tips pips = [3, 6, 10, 14, 18] # PIP joints (middle knuckles)   
 mcps = [2, 5, 9, 13, 17] # MCP joints (base knuckles)   
 fingers\_up = [0, 0, 0, 0, 0]   
 if landmarks[tips[0]].x < landmarks[pips[0]].x:   
 fingers\_up[0] = 1   
 for i in range(1, 5):   
 if landmarks[tips[i]].y < landmarks[pips[i]].y:   
 fingers\_up[i] = 1   
 return fingers\_up   
 def recognize\_gesture(self, landmarks):   
 fingers\_up = self.get\_finger\_states(landmarks)   
 if self.debug\_mode:   
 self.finger\_status\_label.config(text=str(fingers\_up))   
 total\_fingers = sum(fingers\_up)   
 thumb\_tip = landmarks[4]   
 index\_tip = landmarks[8]   
 pinch\_distance = math.sqrt(

(thumb\_tip.x - index\_tip.x)\*\*2 +   
 (thumb\_tip.y - index\_tip.y)\*\*2

)   
 if fingers\_up == [0, 1, 0, 0, 0]: # Only index   
 return 'point'   
 elif fingers\_up == [0, 1, 1, 0, 0]: # Index and middle (peace)   
 return 'peace'   
 elif fingers\_up == [1, 1, 1, 1, 1]: # All fingers   
 return 'open\_hand'   
 elif fingers\_up == [0, 0, 0, 0, 0]: # Fist   
 return 'fist'   
 elif fingers\_up == [1, 0, 0, 0, 0]: # Only thumb   
 if landmarks[4].y < landmarks[9].y: # Compare thumb tip to middle finger MCP return 'thumbs\_up'   
 else:   
 return 'thumbs\_down'   
 elif pinch\_distance < 0.05: # Pinch gesture (thumb and index close together)   
 return 'pinch'   
 return 'unknown'   
 def handle\_gesture(self, gesture, landmarks):   
 current\_time = time.time()   
 gesture\_display = gesture.replace('\_', ' ').upper()   
 self.current\_gesture\_label.config(text=gesture\_display)   
 if gesture != self.last\_gesture:   
 self.last\_gesture = gesture   
 self.gesture\_start\_time = current\_time   
 return   
 if current\_time - self.gesture\_start\_time < self.gesture\_hold\_time:   
 return   
 index\_tip = landmarks[8]

screen\_x = int((1 - index\_tip.x) \* self.screen\_width)

screen\_y = int(index\_tip.y \* self.screen\_height)   
 smooth\_x = int(self.last\_x \* self.smoothing\_factor + screen\_x \* (1 - self.smoothing\_factor)) smooth\_y = int(self.last\_y \* self.smoothing\_factor + screen\_y \* (1 - self.smoothing\_factor))

self.last\_x, self.last\_y = smooth\_x, smooth\_y   
 try:   
 if gesture == 'point':   
 pyautogui.moveTo(smooth\_x, smooth\_y)   
 if self.is\_dragging:   
 pass   
 elif gesture == 'peace':   
 if current\_time - self.last\_click\_time > self.click\_cooldown:   
 self.log\_action(f"Left click attempt at ({smooth\_x}, {smooth\_y})") pyautogui.click(smooth\_x, smooth\_y)   
 self.last\_click\_time = current\_time   
 elif gesture == 'open\_hand':   
 if current\_time - self.last\_click\_time > self.click\_cooldown:   
 self.log\_action(f"Right click attempt at ({smooth\_x}, {smooth\_y})") pyautogui.rightClick(smooth\_x, smooth\_y)   
 self.last\_click\_time = current\_time   
 elif gesture == 'thumbs\_up':   
 pyautogui.scroll(3)   
 self.log\_action("Scrolled up")   
 self.gesture\_start\_time = current\_time # Reset to prevent rapid scrolling elif gesture == 'thumbs\_down':   
 pyautogui.scroll(-3)   
 self.log\_action("Scrolled down")   
 self.gesture\_start\_time = current\_time # Reset to prevent rapid scrolling elif gesture == 'fist':

if not self.is\_dragging:

pyautogui.mouseDown(smooth\_x, smooth\_y)   
 self.is\_dragging = True   
 self.drag\_start\_pos = (smooth\_x, smooth\_y)   
 self.log\_action(f"Started dragging from ({smooth\_x}, {smooth\_y})") else:   
 pyautogui.moveTo(smooth\_x, smooth\_y)

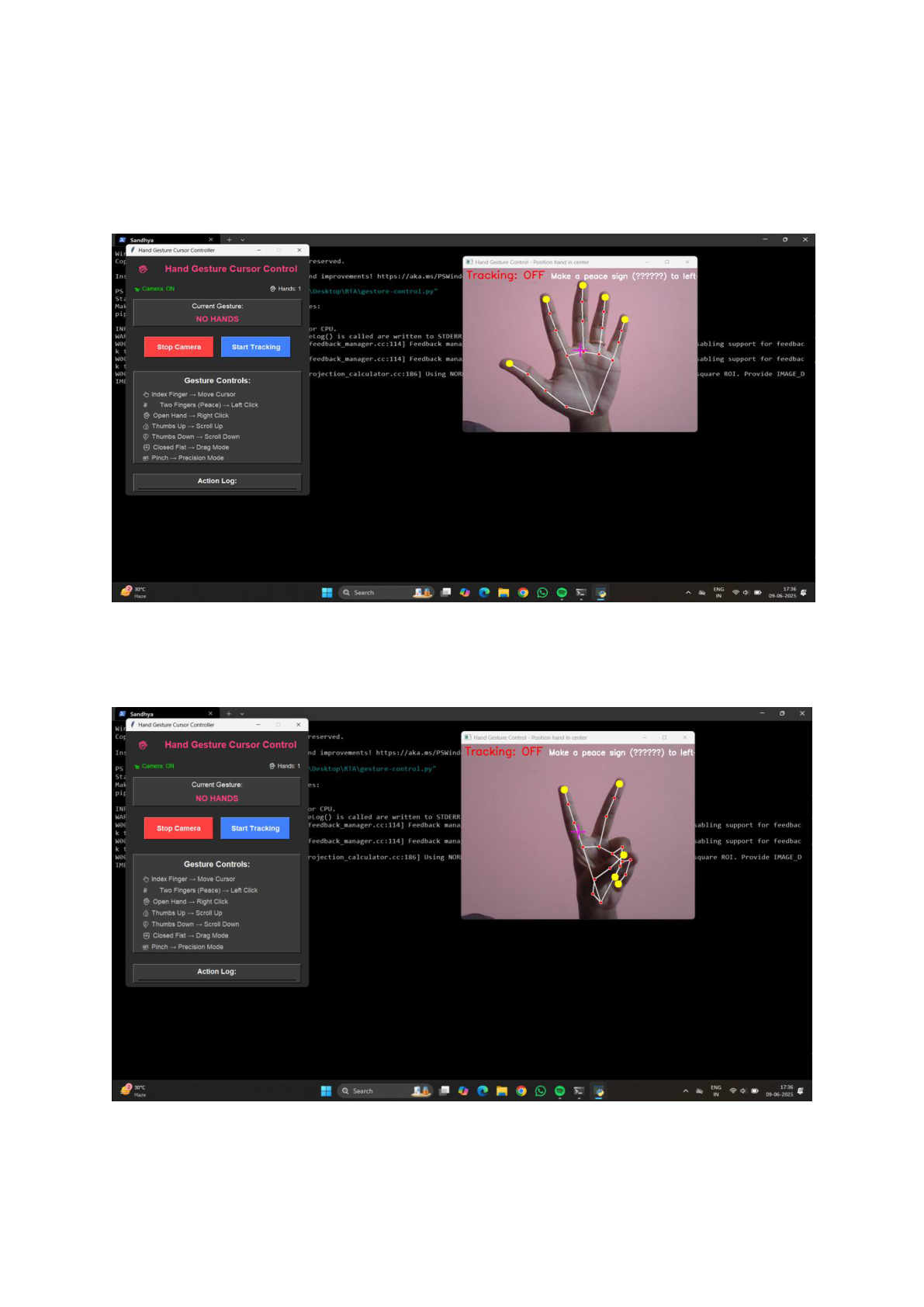
elif gesture == 'pinch':   
 precision\_x = int(self.last\_x \* 0.9 + screen\_x \* 0.1) precision\_y = int(self.last\_y \* 0.9 + screen\_y \* 0.1) pyautogui.moveTo(precision\_x, precision\_y)   
 self.last\_x, self.last\_y = precision\_x, precision\_y   
 else:   
 if self.is\_dragging:   
 pyautogui.mouseUp()   
 self.is\_dragging = False   
 self.log\_action("Stopped dragging")   
 except Exception as e:   
 self.log\_action(f"Error performing action: {str(e)}")   
 def process\_video(self):   
 while self.is\_camera\_on:   
 ret, frame = self.cap.read()   
 if not ret:   
 break   
 frame = cv2.flip(frame, 1)   
 rgb\_frame = cv2.cvtColor(frame, cv2.COLOR\_BGR2RGB) results = self.hands.process(rgb\_frame)   
 hand\_count = 0   
 if results.multi\_hand\_landmarks:   
 hand\_count = len(results.multi\_hand\_landmarks)

self.hand\_status\_label.config(text=f" Hands: {hand\_count}")   
 if results.multi\_hand\_landmarks:   
 for hand\_landmarks in results.multi\_hand\_landmarks:   
 self.mp\_draw.draw\_landmarks(   
 frame, hand\_landmarks, self.mp\_hands.HAND\_CONNECTIONS) tips = [4, 8, 12, 16, 20] # Thumb, Index, Middle, Ring, Pinky tips for tip in tips:   
 tip\_x = int(hand\_landmarks.landmark[tip].x \* frame.shape[1]) tip\_y = int(hand\_landmarks.landmark[tip].y \* frame.shape[0])

cv2.circle(frame, (tip\_x, tip\_y), 10, (0, 255, 255), -1)   
 if self.is\_tracking:   
 gesture = self.recognize\_gesture(hand\_landmarks.landmark)   
 self.handle\_gesture(gesture, hand\_landmarks.landmark)   
 cv2.putText(frame, f"Gesture: {gesture.upper()}",   
 (10, 70), cv2.FONT\_HERSHEY\_SIMPLEX, 0.7,   
 (255, 255, 0), 2)   
 else:   
 self.current\_gesture\_label.config(text="NO HANDS")   
 if self.is\_dragging:   
 pyautogui.mouseUp()   
 self.is\_dragging = False   
 self.log\_action("Stopped dragging (no hands)")   
 status\_text = f"Tracking: {'ON' if self.is\_tracking else 'OFF'}"   
 cv2.putText(frame, status\_text, (10, 30), cv2.FONT\_HERSHEY\_SIMPLEX, 1, (0, 255, 0) if self.is\_tracking else (0, 0, 255), 2)

cv2.putText(frame, "Make a peace sign () to left-click",   
 (frame.shape[1] - 400, 30), cv2.FONT\_HERSHEY\_SIMPLEX, 0.7, (255, 255, 255), 2)   
 h, w = frame.shape[:2]   
 cv2.line(frame, (w//2-20, h//2), (w//2+20, h//2), (255, 0, 255), 2) cv2.line(frame, (w//2, h//2-20), (w//2, h//2+20), (255, 0, 255), 2) cv2.imshow('Hand Gesture Control - Position hand in center', frame) if cv2.waitKey(1) & 0xFF == ord('q'):   
 break   
 def on\_closing(self):   
 self.stop\_camera()   
 self.root.destroy()   
 def run(self):   
 try:   
 self.log\_action("Application started. Click 'Start Camera' to begin.") self.root.mainloop()

except KeyboardInterrupt:   
 self.log\_action("Application interrupted by user") finally:   
 self.stop\_camera()   
if \_\_name\_\_ == "\_\_main\_\_":   
 print("Starting Hand Gesture Cursor Controller...")   
 print("Make sure you have installed the required packages:") print("pip install opencv-python mediapipe pyautogui") print()   
 try:   
 app = HandGestureCursorController()   
 app.run()   
 except Exception as e:   
 print(f"Error starting application: {e}")   
 input("Press Enter to exit...")

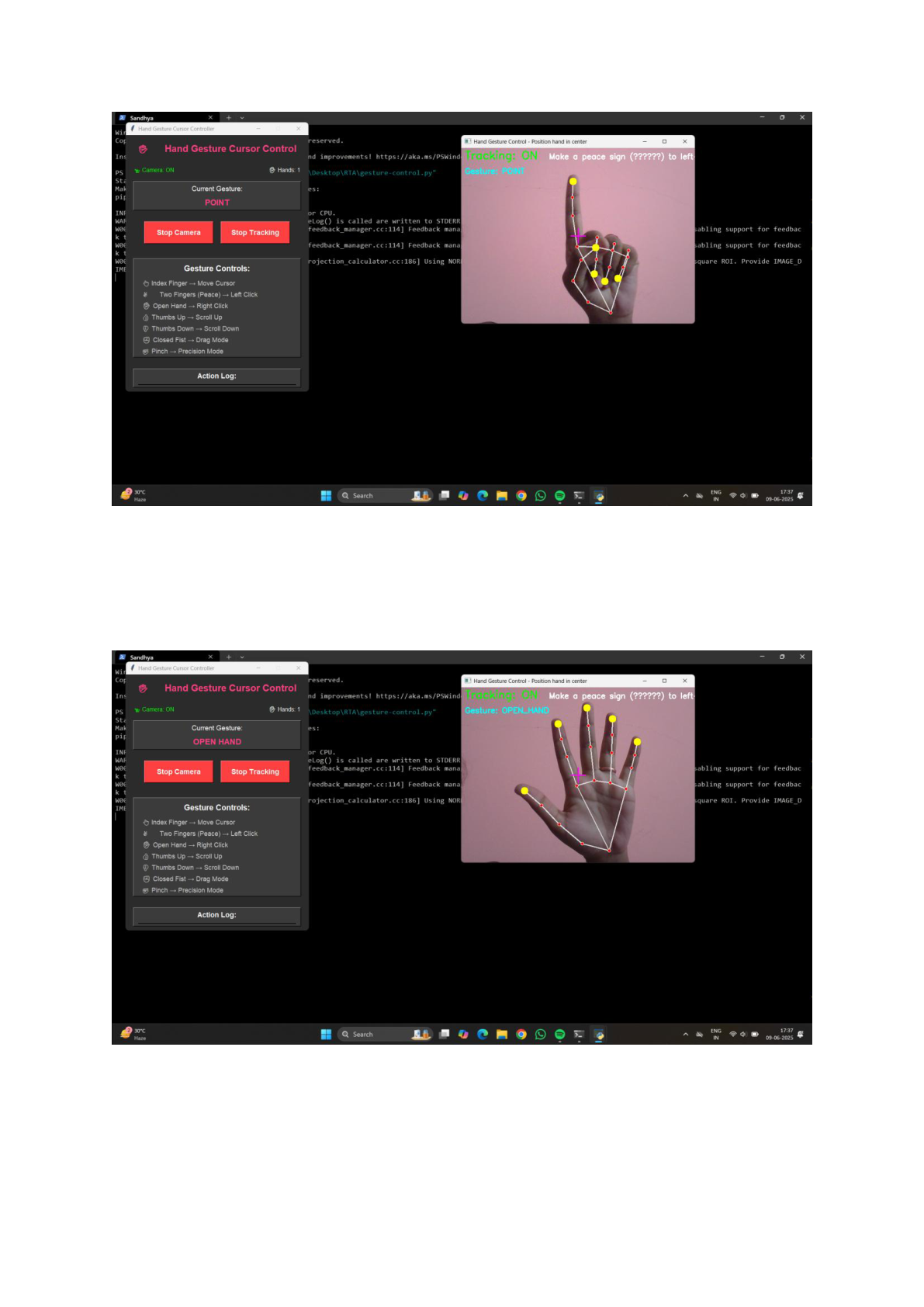


**CHAPTER-7**

**OUTPUT SCREENS :**

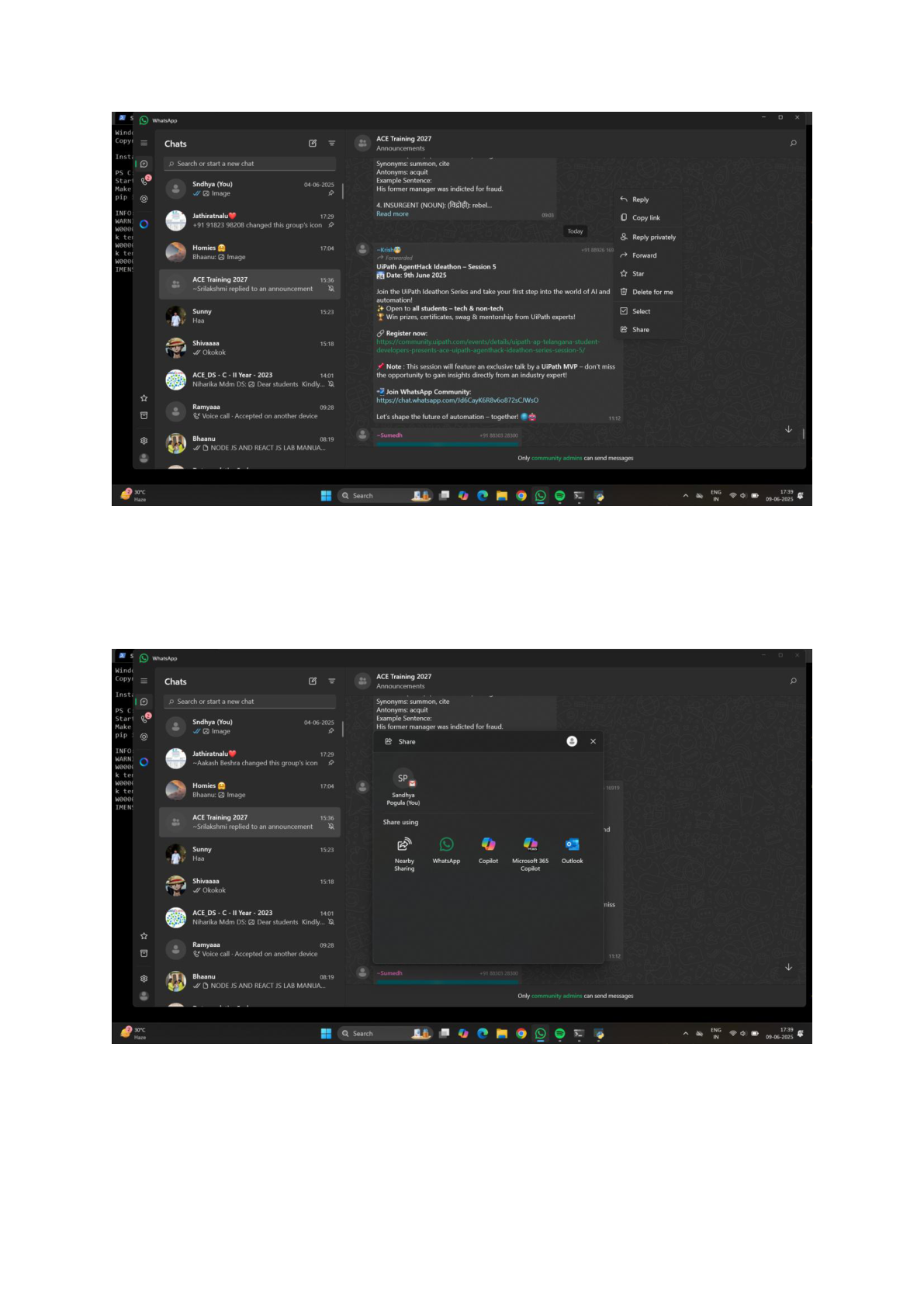
**Fig: 7 Starting the camera without Tracking**

**Fig: 8 Starting the camera without Tracking**



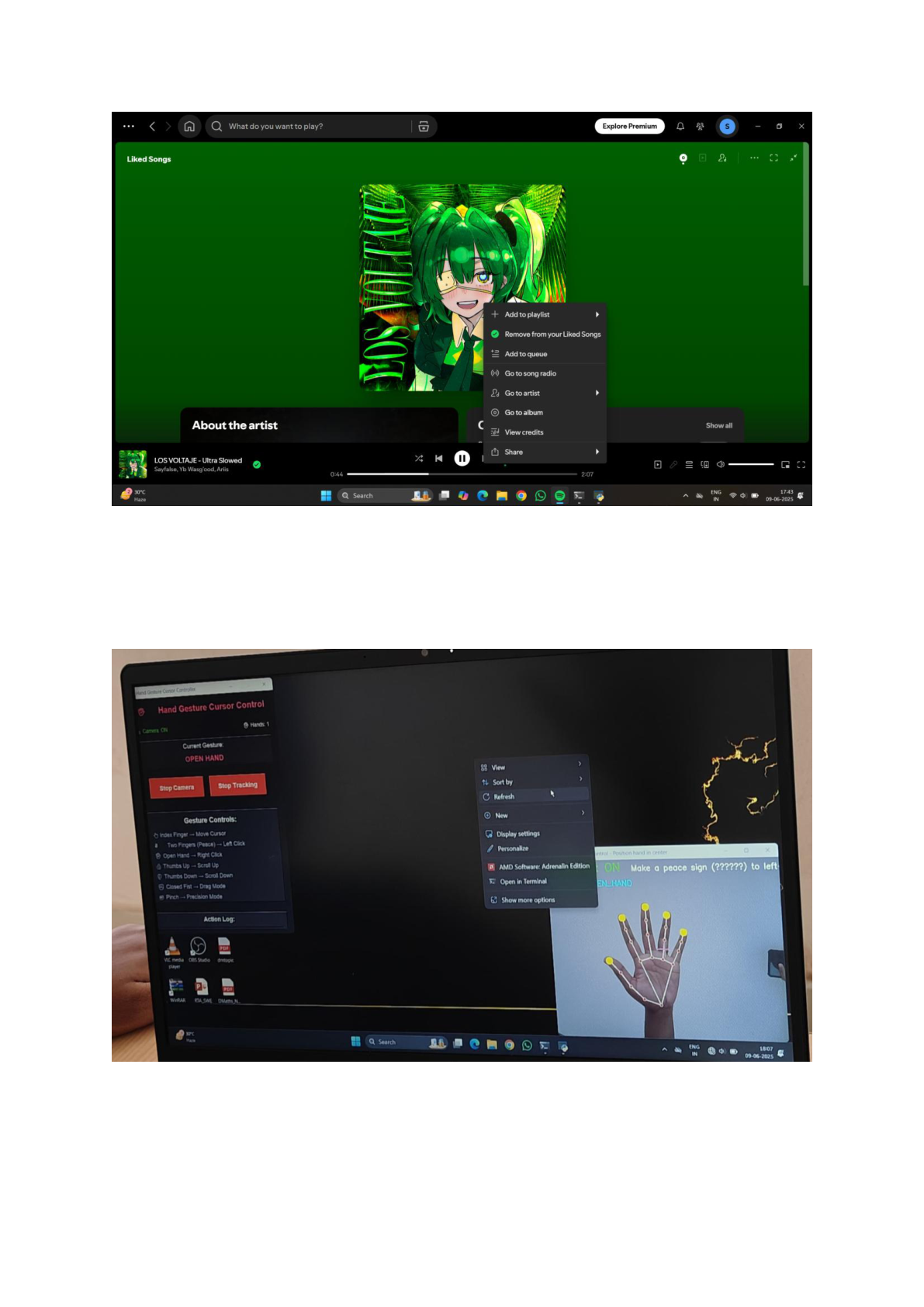
**Fig: 9 Controlling the cursor with the help of Index finger (point gesture)**

**Fig: 10 Recognizing the hand gestures**



**Fig: 11 Controlling the whatsapp [ selecting a message ] using pointer gesture and peace gesture**

**Fig: 12 Sharing the message the using peace gesture(left click)**

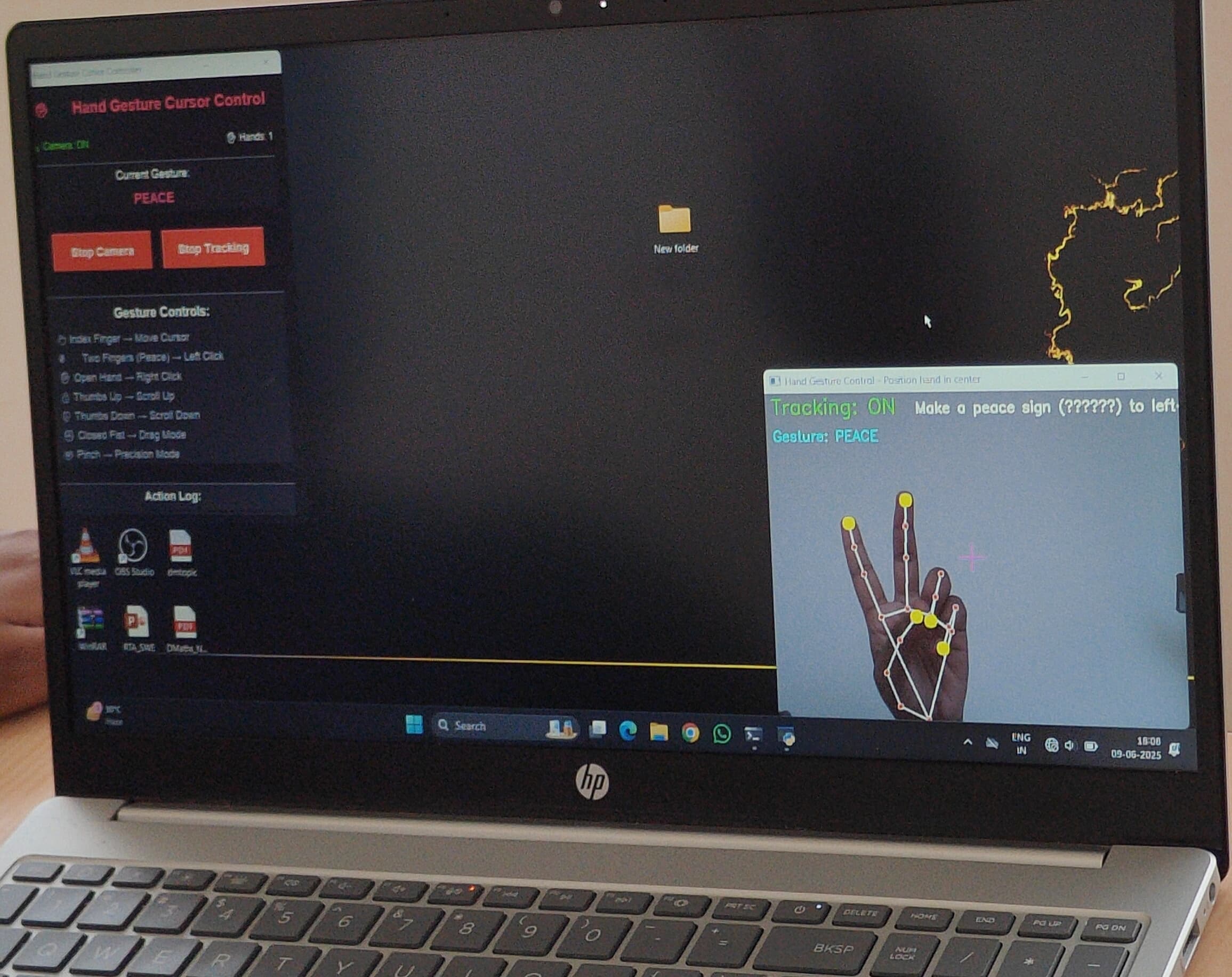


**Fig: 13 Playing the song on spotify and viewing the options by using open hand gesture(right click)**

**Fig: 14 Controlling the desktop - refreshing the desktop using open hand gesture(right click)**



**Fig: 15 Creating a folder using peace gesture.**



**Fig: 16 Created a folder**

**CHAPTER-8**

**CONCLUSION**

we’ve developed a real-time hand gesture control system that works with just a webcam and Python code. Our system is easy to use, touchless, and does not require any special hardware, making it ideal for use in daily life, especially for those with physical limitations or in hygiene-sensitive environments. It shows promising results in controlling apps like WhatsApp, managing files, and more. With planned enhancements, it could revolutionize how we interact with computers, apps, and smart devices. We thank our guide and the faculty at Ace Engineering College for their support, and we look forward to exploring this project further.

**FUTURE SCOPE**

To enhance usability, scalability, and application reach, the following future developments are recommended:

1. Multimodal Control: Integrate voice commands alongside gestures for better accessibility. Enable facial expressions for command input (e.g., eye blink for click).
2. Gesture Expansion: Allow custom gesture mapping by users for different actions. Add multi-hand gesture support for complex controls.
3. Environmental Adaptability: Improve accuracy in low-light and cluttered backgrounds. Add adaptive brightness compensation in the vision pipeline.
4. Cross-Platform Integration: Convert the system into a browser extension/plugin for real-time web app integration. Port the system to Android/iOS devices using front cameras.
5. AI-Based Enhancements: Use deep learning for gesture classification to reduce false positives. Implement gesture learning to adapt to user habits over time.
6. Use Case Expansion: Smart homes: Control appliances via gesture.

Education: Use in digital whiteboards and online teaching tools.

Healthcare: Use in sterile environments like operating rooms.

**REFERENCES**

• M. S. Amin, S. T. H. Rizvi, and M. M. Hossain, “A comparative review on applications of different sensors for sign language recognition,” *J. Imag.*, vol. 8, no. 4, p. 98, Apr. 2022.

• Ali A Alani, Georgina Cosma, Aboozar and T M McGinnity, “Hand Gesture Recognition Using an Adapted Convolutional Neural Network with Data Augmentation,” 4th *IEEE*  *International Conference on Information Management*.

• Zhigang, F. Computer gesture input and its application in human computer interaction. Mini Micro Syst., 6, 418–421.

• D.L. Quam, “Gesture recognition with a Data Glove,” 10.1109/NAECON.1990.112862, vol. 2, pp. 755 - 760.

• Komura T, Lam W-C (2006) Real-time locomotion control by sensing gloves. Comput Anim Virtual Worlds 17(5):513–525.

• L. Dipietro, A. M. Sabatini, and P. Dario, “A survey of glove-based systems and their applications,” *IEEE Trans. Syst. Man Cybern. Part C Appl. Rev.*, vol. 38, no. 4, pp. 461– 482

• Patil, S.; Bidari, I.; Sunag, B.; Gulahosour, S.; Shettar, P. Application of HMI Technology in Automotive Sector. In Proceedings of the International Conference on Electrical, Electronics, Communication, Computer and Optimization Techniques (ICEECCOT), Mysuru, India.

• S. S. Rautaray and A. Agrawal, “Vision based hand gesture recognition for human computer interaction: A survey,” *Artif. Intell. Rev.*, vol. 43, no. 1, pp. 1–54