Mouse control using eyegaze

Aryaman Oberoi
IIT Kanpur
aryamano24@iitk.ac.in

Chandraveer Sisodia
IIT Kanpur
cssisodia24@iitk.ac.in

Shivam Kumar
IIT Kanpur
shivamkk24@iitk.ac.in

Anikait Dixit
IIT Kanpur
anikaitd24@iitk.ac.in

***Abstract*—This study develops a robust method for hands-free mouse cursor control, addressing the unreliability of direct eye-gaze tracking on common webcams. We first implement and compare two distinct frameworks, Dlib and MediaPipe, to track facial landmarks and calculate head pose using a Perspective-n-Point (PnP) algorithm. Our comparative analysis demonstrates that MediaPipe is objectively superior, offering more than double the Frames Per Second (FPS) and overcoming Dlib's critical failure to track non-frontal faces.**

Based on these conclusive findings, we then constructed a final, optimized controller built entirely on the MediaPipe framework. This novel system combines stable MediaPipe Face Mesh (for head-pose navigation) with MediaPipe Hands to detect both fist gestures (for reliable clicks) and pinch gestures (for scrolling). This unified, high-performance approach results in a stable, low-latency, and practical hands-free interface, making it a superior solution for accessible human-computer interaction and assisting individuals with severe Motor Disabilities, Spinal Cord Injuries, or Neurodegenerative diseases.

I. Introduction

In recent years, advancements in human-computer interaction (HCI) have led to the development of alternative input methods beyond traditional keyboard and mouse interfaces. Among these, intuitive, hands-free solutions are emerging with immense potential, particularly for individuals with motor impairments, offering them a more accessible way to navigate and interact with digital systems.

Our initial exploration focused on direct eye gaze tracking (which follows the pupil), as it is a powerful concept. However, we immediately encountered significant real-world failures. Our attempts to implement direct iris tracking using a standard, integrated laptop webcam were met with extreme instability. The poor quality and low resolution of the video feed made it impossible for the algorithms to maintain a stable lock on the pupil's small, noise-sensitive movements. This resulted in an unusable, "jittery" cursor that would jump erratically across the screen.

Similarly, our efforts to use blink detection for clicks were unsuccessful for the same reason. The low-fidelity video made it difficult to reliably distinguish an intentional blink from an open eye, leading to a frustrating experience with constant false positives (accidental clicks) and missed inputs.

Faced with these practical limitations, this project directly addresses this challenge by pivoting to a more robust alternative: head pose estimation. Instead of tracking the unreliable movements of the iris, our system analyzes the head's overall orientation (specifically, its yaw and pitch) as a stable proxy for the user's gaze direction. This approach is far more resilient and functions effectively even with the common, low-resolution webcams that made eye tracking impossible.

A primary challenge in developing such a real-time system is achieving an optimal balance between precision (is the cursor accurate enough to be useful?) and computational efficiency (does it run smoothly without lagging?). To find this balance, our research implements and compares two distinct computer vision frameworks for the core head-pose task: the classic, regression-tree-based Dlib library and the modern, deep-learning-based MediaPipe framework.

Furthermore, to create a complete and functional interface, we augment this head-pose navigation with robust gesture controls. We replace the failed traditional blink detection by integrating MediaPipe Hands into our system. This module is used to detect two specific gestures: a fist gesture for reliable clicks and a pinch gesture for scrolling. Our goal is to analyze this combined-model approach to develop an accessible, efficient, and user-friendly alternative to conventional input devices.

II. Landmark Detection

In computer science, landmark detection is the process of finding significant landmarks in an image. This originally referred to finding landmarks for navigational purposes – for instance, in robot vision or creating maps from satellite images. Methods used in navigation have been extended to other fields, notably in facial recognition where it is used to identify key points on a face. It also has important applications in medicine, identifying anatomical landmarks in medical images.

A. Facial Landmark

Detecting facial landmarks is a subset of the shape prediction problem. Given an input image (and normally an ROI that specifies the object of interest), a shape predictor attempts to localize key points of interest along the shape.

In the context of facial landmarks, our goal is detect important facial structures on the face using shape prediction methods. Some of the most popular frameworks used are Dlib, Active Shape Models (ASM), and the more recent deep learning-based Mediapipe FaceMesh. Detecting facial landmarks is therefore a two step process:

- Step #1: Localize the face in the image.
- Step #2: Detect the key facial structures on the face ROI.

B. Dlib

The pre-trained facial landmark detector inside the dlib library is used to estimate the location of 68 (x, y)-coordinates that map to facial structures on the face.

Fig. 1. Dlib 68 facial landmarks visualization.

For head pose estimation, a subset of these 68 landmarks is selected, corresponding to distinct anatomical features that provide a stable reference for estimating orientation. The selected landmarks and their Dlib indices are:

- **Nose Tip (index 30):** Used as the central reference point of the face.
- **Left Eye Outer Corner (index 36) & Right Eye Outer Corner (index 45):** These points help measure the horizontal direction of the face (left–right alignment).
- **Left Mouth Corner (index 48) & Right Mouth Corner (index 54):** Useful for estimating yaw — whether the face is turned left or right.
- **Chin (index 8):** Helps with vertical orientation and pitch — whether the face is looking up or down.

1) 2D Coordinates of These Landmarks: The 2D position of each selected landmark in the image is written as:

$$p_i = (x_i, y_i), \quad \text{for } i \in \{30, 36, 45, 48, 54, 8\} \tag{1}$$

Which simply means:

- Each landmark has an (x, y) coordinate,
- And we are using these six landmark indices: 30, 36, 45, 48, 54, 8.

These landmarks provide a stable geometric foundation for head pose estimation. This allows for the calculation of rotation angles (yaw, pitch, and roll) by contrasting their relative positions with a predefined 3D head model.

III. Architecture of MediaPipe FaceMesh

A. Overview of MediaPipe

MediaPipe is an open-source framework that enables developers to construct high-performance machine learning (ML) perception pipelines. The framework includes a library of adaptable modules for specific tasks, such as hand tracking, face detection, and pose estimation. These components are optimized for real-time operation on diverse platforms, from web browsers to mobile devices.

B. FaceMesh Architecture

MediaPipe's FaceMesh module utilizes a lean convolutional neural network (CNN) to identify and map 468 distinct facial landmarks, all in real-time. This architecture is built on several key components:

- **Landmark Detection Network:** This network analyzes an input image to find facial features. It notably uses depthwise separable convolutions, a technique that lowers the computational cost without sacrificing accuracy. The model is also robust enough to work reliably under various conditions, such as different head orientations or facial expressions.
- **Facial Landmark Representation:** The 468 points provide a detailed map of the face, with each point corresponding to a specific feature like the eyes, nose, or mouth. Such a dense and comprehensive map enables in-depth analysis, like assessing facial symmetry.
- **Modular Pipeline Structure:** The framework is built modularly, which allows for the easy combination of different processing stages (e.g., face detection followed by landmark extraction). This pipeline design is engineered for highly efficient data handling, resulting in fast inference speeds and low latency—a crucial requirement for any real-time application.

C. Visualization Capabilities

The FaceMesh package also comes with integrated visualization tools. These utilities can be used to overlay the detected landmarks on an image, drawing features like the contours of the face, a full facial mesh (tessellation), or the iris, which provides clear and intuitive visual feedback.

D. Performance Optimizations

FaceMesh is specifically engineered for high performance on mobile and edge computing devices. This is achieved through techniques like model quantization and options for hardware acceleration. These strategies enable the system to run at high frame rates while using minimal computational resources, making it an ideal choice for demanding real-time tasks like augmented reality or live video conferencing.

The selected landmarks and their respective roles are:

- **Nose Tip (i = 1):** Acts as a central reference point for pose estimation.
- **Left Eye Outer Corner (i = 133) and Right Eye Outer Corner (i = 362):** Help determine horizontal alignment and yaw angle.
- **Left Ear (i = 234) and Right Ear (i = 454):** Provide additional spatial constraints to estimate head rotation.
- **Chin (i = 152):** Helps in estimating vertical orientation and pitch angle.

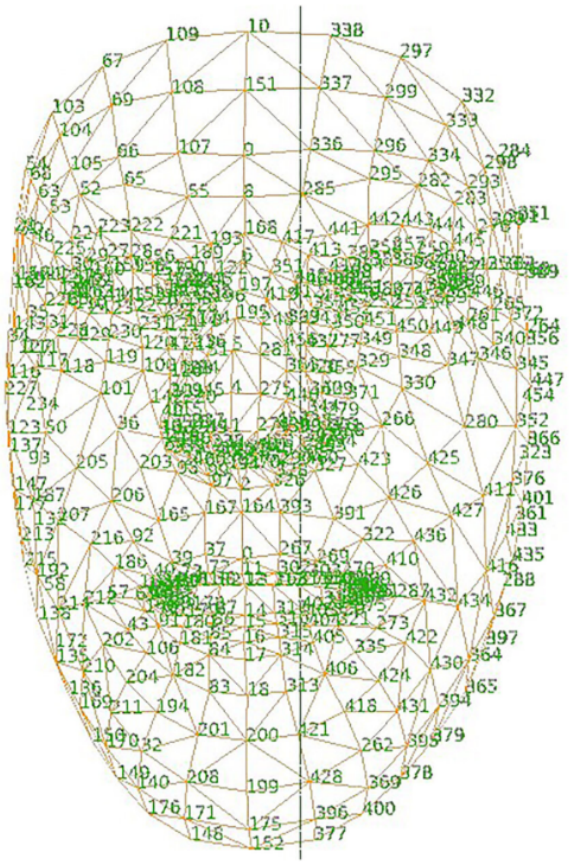

Fig. 2. MediaPipe FaceMesh 468 facial landmarks visualization.

The 2D image coordinates of these selected landmarks are given by:

$$p_i = (x_i, y_i), \quad i \in \{1, 133, 362, 234, 454, 152\} \tag{2}$$

These landmarks provide a stable geometric foundation for head pose estimation. This allows for the calculation of rotation angles (yaw, pitch, and roll) by contrasting their relative positions with a predefined 3D head model.

IV. Methodology

A. What is Pose Estimation?

In computer vision the pose of an object refers to its relative orientation and position with respect to a camera. You can change the pose by either moving the object with respect to the camera, or the camera with respect to the object.

The pose estimation problem described in this tutorial is often referred to as Perspective-n-Point problem or PNP in computer vision jargon. As we shall see in the following sections in more detail, in this problem the goal is to find the pose of an object when we have a calibrated camera, and we know the locations of n 3D points on the object and the corresponding 2D projections in the image.

B. Camera Calibration and Intrinsic Parameters

Accurate head pose estimation requires defining the camera's intrinsic matrix, K. This matrix mathematically models how 3D real-world points are projected onto the 2D image plane, assuming a standard pinhole camera model.

The intrinsic matrix K is defined as:

$$K = \begin{bmatrix} f_x & 0 & c_x \\ 0 & f_y & c_y \\ 0 & 0 & 1 \end{bmatrix} \tag{3}$$

where:

- f_x and f_y are the focal lengths in pixels along the x and y axes
- c_x and c_y are the coordinates of the principal point (optical center)
- The 3D-to-2D scaling of the scene is governed by the pixel-based focal lengths, f_x and f_y. In our practical application, we employ a common simplification, setting both f_x and f_y to a value equivalent to the image's total width.
- Concurrently, the camera's optical axis, or principal point (c_x, c_y), is assumed to be at the precise geometric center of the video frame.

1) Lens Distortion Assumption: Physical camera lenses invariably introduce image warping, which manifests as radial (k_1, k_2, k_3) and tangential (p_1, p_2) distortions that visibly curve straight lines.

The distortion coefficient vector is:

$$D = [k_1, k_2, p_1, p_2, k_3] \tag{4}$$

Correcting this warp is computationally intensive. To prioritize real-time performance and simplicity, our project bypasses this by adopting an ideal pinhole camera model, which is theoretically free of all distortion. This assumption is implemented by setting the distortion coefficient vector D to a zero vector:

$$D = [0, 0, 0, 0, 0] \tag{5}$$

The direct benefit of this simplification is that the projection of 3D points onto the 2D image plane becomes a purely linear transformation dictated by the K matrix. This significantly streamlines the calculations required for pose estimation.

2) Effect on Head Pose Estimation: These intrinsic parameters are the bedrock of the solvePnP algorithm. The algorithm is critically dependent on the K matrix; it uses K as the "key" to translate the 3D model's points from their fixed world coordinates to the 2D pixel locations detected in the camera's feed.

While using the image width as a substitute for the true focal length is a pragmatic shortcut, it provides a "reasonable approximation" of the camera's perspective projection. For our goal of stable, real-time control, this approximation is both sufficient and necessary to allow the algorithm to derive an accurate rotation estimation.

C. Head Pose Estimation

To compute the head's orientation, we first require a generic 3D model of a face. This model is defined by a set of predefined 3D coordinates (P_i). These 3D points are strategically chosen based on their anatomical significance to provide a stable and accurate foundation for the pose estimation.

In our project, we implement two distinct 3D models (defined in our code as MODEL_POINTS) to align with the different landmark predictors:

- **For MediaPipe:** The 3D model is a 6-point array representing the Nose Tip, Left Eye, Right Eye, Left Ear, Right Ear, and Chin.
- **For Dlib:** The 3D model is a 6-point array representing the Nose Tip, Left Eye Outer Corner, Right Eye Outer Corner, Left Mouth Corner, Right Mouth Corner, and Chin.

These 3D model points (P_i) must correspond to the 2D pixel coordinates (p_i) of the facial landmarks detected in the live video feed. The specific landmark indices used to find these 2D points differ between our two frameworks:

- **MediaPipe 2D Indices:** $i \in \{1, 133, 362, 234, 454, 152\}$
- **Dlib 2D Indices:** $i \in \{30, 36, 45, 48, 54, 8\}$

With both sets of points—the 2D detected points (p_i) and their corresponding 3D model points (P_i)—we can calculate the head's orientation using the Perspective-n-Point (PnP) algorithm. The goal of the PnP algorithm is to find a solution for the main projection equation:

$$p_i = K \cdot (RP_i + t) \tag{6}$$

which can be expanded as:

$$\begin{bmatrix} u_i \\ v_i \\ 1 \end{bmatrix} \sim \begin{bmatrix} f_x & 0 & c_x \\ 0 & f_y & c_y \\ 0 & 0 & 1 \end{bmatrix} \left(\begin{bmatrix} r_{11} & r_{12} & r_{13} \\ r_{21} & r_{22} & r_{23} \\ r_{31} & r_{32} & r_{33} \end{bmatrix} \begin{bmatrix} X_i \\ Y_i \\ Z_i \end{bmatrix} + \begin{bmatrix} t_x \\ t_y \\ t_z \end{bmatrix} \right) \tag{7}$$

Where:

- K is the camera's intrinsic matrix, which was defined previously.
- R is the 3×3 rotation matrix. This is the primary output we need, as it mathematically describes the head's 3D orientation.
- t is the 3×1 translation vector, which accounts for the head's 3D position relative to the camera.
- To solve for R and t, we use the OpenCV solvePnP function, which employs methods such as the Levenberg-Marquardt algorithm or RANSAC-based iterative refinement.

D. Conversion to Euler Angles

The rotation matrix R, which is the output of the PnP algorithm, mathematically describes the head's 3D orientation. However, this 3×3 matrix is not directly interpretable for our control purposes. To make this orientation data human-readable, we must convert the matrix R into the more intuitive Euler angles.

The rotation matrix R has the form:

$$R = \begin{bmatrix} R_{11} & R_{12} & R_{13} \\ R_{21} & R_{22} & R_{23} \\ R_{31} & R_{32} & R_{33} \end{bmatrix} \tag{8}$$

We use the following formulas to extract the yaw, pitch, and roll:

$$\text{yaw} = \arctan 2(R_{21}, R_{11}) \tag{9}$$

$$\text{pitch} = \arctan 2(-R_{31}, \sqrt{R_{32}^2 + R_{33}^2}) \tag{10}$$

$$\text{roll} = \arctan 2(R_{32}, R_{33}) \tag{11}$$

These angles are defined as:

- **Yaw (θ_{yaw}):** Rotation around the vertical axis, corresponding to a left-right head movement.
- **Pitch (θ_{pitch}):** Rotation around the horizontal axis, corresponding to an up-down head movement.
- **Roll (θ_{roll}):** Rotation around the depth axis, corresponding to tilting the head sideways.

E. Smoothing of Values

The raw yaw and pitch angles computed from the PnP algorithm can be "jittery" due to small, noisy variations in the detected landmarks from frame to frame. To fix this and create a stable cursor, we apply an exponential moving average (EMA) to smooth the computed angles.

This is implemented in our code for both smoothed_yaw and smoothed_pitch. The formula is:

$$\theta^{(t)} = \alpha\theta_{\text{new}} + (1 - \alpha)\theta^{(t-1)} \tag{12}$$

where:

- $\theta^{(t)}$ is the new smoothed angle.
- $\theta^{(t-1)}$ is the previously smoothed angle.
- θ_{new} is the new, raw angle calculated from the PnP algorithm.
- α is the smoothing parameter.

  In our implementation, we set $\alpha = 0.7$. This α value represents the "weight" given to the new, raw data.

  - A higher alpha (like our 0.7) gives more weight to the new values, resulting in a more responsive but slightly "noisier" cursor.
  - A lower alpha would give more weight to the previous smoothed values, resulting in a much smoother but slower, "lagging" response.

  Our value of 0.7 was chosen as a good balance between responsiveness and stability for controlling the mouse.

F. Gaze Visualization

The estimated gaze direction is visualized as a vector $\mathbf{g}$ projected from the face center. Given the smoothed yaw and pitch angles, we approximate the gaze direction as:

$$\mathbf{g} = (g_x, g_y) = (-L \sin \theta_{\text{yaw}}, -L \sin \theta_{\text{pitch}}) \tag{13}$$

where:

- L is the predefined arrow length (typically a fixed scalar).
- θ_{yaw} and θ_{pitch} control the horizontal and vertical displacement of the arrow, respectively.

The endpoint of the gaze vector is computed as:

$$p_{\text{end}} = p_{\text{center}} + \mathbf{g} \tag{14}$$

where p_{center} is the face center, which is often approximated as the nose tip landmark (p_1). This allows real-time visualization of where the person is looking within the image frame.

1) Significance of Gaze Estimation: Gaze estimation is a foundational technology for a new class of interactive applications. The most critical applications for our project include:

- **Assistive Technologies:** This is the primary motivation. It provides a crucial "hands-free" interface, enabling individuals with significant motor impairments to access and control computers.
- **Human-Computer Interaction (HCI):** Our project is a core example of this, using "gaze-based controls" to create a new and intuitive way for users to interact with their systems.
- **Immersive Gaming and Simulation:** The same head-pose tracking can be used to control the in-game camera (e.g., in a flight simulator) or interact with virtual environments, offering a more engaging experience.

Our project's methodology, which focuses on PnP and smoothing, is a direct effort to achieve "stable and real-time gaze tracking," which is essential for making any of these applications robust and user-friendly.

V. Comparison Between the Two Models

We have compared the two frameworks side by side quantitatively using FPS for both the models.

A. FPS

Frames Per Second (FPS) is a performance metric that measures how many individual images, or "frames," a system can process and render in a single second.

1) Why FPS is a Good Metric for Comparison: FPS is an excellent metric for this project because it directly quantifies the computational efficiency and user-perceived responsiveness of each framework. As High FPS means low latency and fluid cursor motion.

B. Dlib Framework

Observations - Average FPS: 14-30 frames per second.

1) Dlib Pose-Tracking Failure: A major limitation was observed in the Dlib implementation: the cursor tracking completely stops when the head is turned even slightly. `dlib.get_frontal_face_detector()` is only trained to find "frontal" faces. The moment you turn your head, your face becomes a "profile" view, and the detector fails to find a face.

C. MediaPipe Framework

Observations - Average FPS: 100-135 frames per second.

The tracking failure for high values of Yaw and pitch were not observed, as the MediaPipe FaceMesh model is designed to be robust to varying head poses and does not suffer from this "frontal-only" limitation.

Fig. 3. Dlib framework performance showing low FPS.

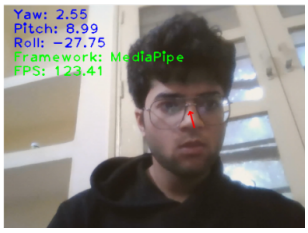

Fig. 4. MediaPipe framework performance showing high FPS.

TABLE I
Comparison of Dlib and MediaPipe

Feature	Dlib (Facial Landmark Detector)	MediaPipe (Face Mesh)
Core Technology	"Classic" Machine Learning: HOG + Regression Trees	Modern Deep Learning (Lightweight CNN)
Performance (FPS)	CPU-heavy, not optimized for real-time. Runs 15-30 FPS, which feels laggy	Highly optimized for real-time. Easily runs at 110+ FPS
Landmarks	Provides 68 2D landmarks (x, y coordinates)	Provides 478 3D landmarks (x, y, and z coordinates)
Robustness	Trained on frontal faces. Fails when head turned too far	Robust to head pose, lighting, and partial blockages
Model Files	External. Requires manual download of 80MB+ .dat file	Self-contained. Model bundled directly into library

VI. Click Trigger

Our system uses MediaPipe Hands, a high-fidelity, real-time hand-tracking model that provides 21 3D landmarks (or keypoints) for each hand, to serve as the "event detection output". We then use the `pyautogui` Python library for programmatic "control of the mouse";

upon detecting a valid gesture from the MediaPipe landmarks, `pyautogui.click()` is triggered to "emulate a mouse click at the current cursor position".

A. Controlling Scroll by a Simple Hand Gesture (via MediaPipe Hands)

To implement scrolling, we used a "pinch-to-scroll" gesture. This gesture is detected using the MediaPipe Hands model, which tracks all the landmarks of your hand in real-time. We continuously calculate the 2D distance between the tip of the thumb (Landmark 4) and the tip of the index finger (Landmark 8):

$$d_{\text{pinch}} = \sqrt{(x_4 - x_8)^2 + (y_4 - y_8)^2} \tag{15}$$

The scroll function "turns on" if this distance becomes smaller than a pinch threshold:

$$d_{\text{pinch}} < 0.05 \tag{16}$$

Once this pinch is active, we track the vertical (Y-axis) movement of the index finger; moving the pinched hand up triggers an upward scroll, and moving it down triggers a downward scroll, both of which are executed by the `pyautogui` library.

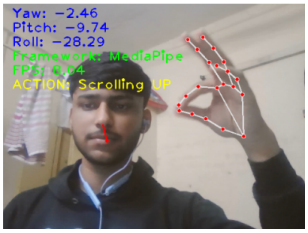

Fig. 5. Scrolling up gesture.

Fig. 6. Scrolling down gesture.

B. Fist Recognition (via MediaPipe Hands)

To address the limitations of eye blink detection (which we found unreliable with our low-resolution webcam), we implemented fist recognition as an alternative method for simulating mouse clicks. This approach proved to be "more reliable and easier to control in real-time".

We use the MediaPipe Hands framework, which employs the following logic:

Fist Detection (for Clicking): The code checks the y-coordinates of the landmarks. It confirms that all four fingertips (e.g., INDEX_FINGER_TIP) are positioned below their corresponding middle knuckles (e.g., INDEX_FINGER_PIP):

If this condition is true for all four fingers, the code registers a "fist."

This MediaPipe-based fist recognition offered "better stability and fewer false positives" than blink detection, all while being extremely computationally efficient.

By integrating our MediaPipe Hands detection with the `pyautogui` library, we achieved full, hands-free cursor control. Specifically, fist recognition served as a reliable trigger for clicks, while pinch gestures provided an intuitive signal for scrolling.

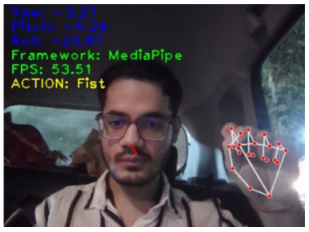

Fig. 7. Fist Gesture .

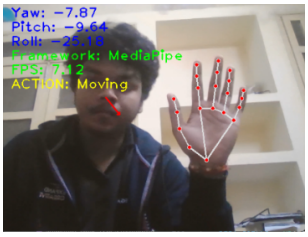

Fig. 8. Moving Gesture .

VII. Challenges Required

During the development of this project, we faced several significant challenges that required practical solutions.

1) **Initial Concept Failure (Eye-Gaze):** Our initial approach, based on the reference PDF, was to use direct eye-gaze and blink detection. This failed immediately due to the low resolution of our standard webcam. The video feed was not stable enough to track the small, rapid movements of the pupil, resulting in an unusable, "jittery" cursor. This forced us to pivot to the more robust method of head-pose estimation.
2) **Framework Performance (Dlib vs. MediaPipe):** Our first major challenge was selecting a viable framework. Our comparative test revealed that the Dlib implementation was unsuitable for two reasons:
   - **Performance:** Dlib was computationally heavy, achieving only ∼59 FPS, less than half of MediaPipe's ∼123 FPS.
   - **Robustness:** The Dlib `get_frontal_face_detector()` would fail the instant the head was turned, causing all tracking to stop. MediaPipe's model, by contrast, was "capable of functioning under varying conditions, including different head poses".
3) **Integration Performance Cost (The "FPS Drop"):** After selecting MediaPipe as the superior framework, we faced our next major challenge. While the head-pose-only script ran at a high ∼120 FPS, integrating the MediaPipe Hands model (in `gaze_controller.py`) caused a significant performance drop to as low as 5-15 FPS. This is because we are now running two computationally expensive deep-learning models (`face_mesh.process()` and `hands.process()`) on every single frame, which saturates the system's processing capabilities. This represents a critical trade-off between features (gestures) and real-time performance.

VIII. Conclusion

The comparative analysis conducted in this study demonstrates a clear and definitive result: MediaPipe is the superior framework for our real-time, head-pose-based HCI system.

Its modern, lightweight architecture places a significantly lower load on the hardware, as evidenced by its FPS being over 5 times that of Dlib's. This high efficiency, combined with its robust pose tracking, makes it the clear choice.

Therefore, for all future advancements—such as integrating our YOLOv8 fist detection for clicks and MediaPipe Hands for scrolling—we will be building exclusively on the MediaPipe foundation.

Despite the limitations, our final application uses stable head-pose estimation for navigation, combined with reliable fist-detection for clicks and pinch-gestures for scrolling.

This project's success was achieved by systematically identifying and solving several key challenges. We began by pivoting from unreliable eye-tracking to stable head-pose, and then proved that MediaPipe was a superior framework to Dlib in both performance and robustness. The final system is a stable, low-latency controller that demonstrates the feasibility of a truly accessible, vision-based interface.

Future work could build on this stable foundation to include dynamic screen mapping, which would allow the controller to learn and adapt to a user's specific range of head motion.

References

[1] Dlib Face Landmarks Detector. Shape Predictor 68 Face Landmarks. Available: https://www.kaggle.com/datasets/sajikim/shape-predictor-68-face-landmarks

[2] J. Hou, Face Yaw Roll Pitch from Pose Estimation using OpenCV. GitHub Repository. Available: https://github.com/jerryhouuu/Face-Yaw-Roll-Pitch-from-Pose-Estimation-using-OpenCV/blob/master/pose_estimation.py

[3] Landmark Detection. Wikipedia. Available: https://en.wikipedia.org/wiki/Landmark_detection

[4] Head Pose Estimation Using OpenCV and Dlib. Learn OpenCV. Available: https://learnopencv.com/head-pose-estimation-using-opencv-and-dlib/

[5] R. Neupane, Facial Landmark Detection. Medium, 2023. Available: https://medium.com/@RiwajNeupane/facial-landmark-detection-a6b3e29eac5b

[6] C. Lugaresi, J. Tang, H. Nash, M. McGuire, N. Kalantri, F. Fei, and J. Kosslyn, "MediaPipe: A Framework for Building Perception Pipelines," arXiv preprint arXiv:1906.08172, 2019. Available: https://arxiv.org/abs/1906.08172

Appendix

This appendix contains the complete Python implementation code for the head pose estimation system. Two main scripts are presented: the first enables comparison between Dlib and MediaPipe frameworks, while the second provides the final integrated system combining MediaPipe FaceMesh with MediaPipe Hands for gesture-based control.

A. Head Pose Comparison Script: Dlib vs MediaPipe

The following script implements both Dlib and MediaPipe frameworks for head pose estimation.This code demonstrates the comparative analysis discussed in Section 5 of the paper.

```python
import cv2
import numpy as np
import dlib
import mediapipe as mp
import pyautogui
import time

# --- 0. CONFIGURATION ---
# Set this to True to use MediaPipe, False to use Dlib
USE_MEDIAPIPE = False

# --- 1. DLIB SETUP ---
print("Loading Dlib model...")
try:
    dlib_detector = dlib.get_frontal_face_detector()
    dlib_predictor = dlib.shape_predictor("shape_predictor_68_face_landmarks.dat")
except RuntimeError:
    print("ERROR: Could not find 'shape_predictor_68_face_landmarks.dat'")
    print("Download it from: http://dlib.net/files/shape_predictor_68_face_landmarks.dat.bz2")
    print("And unzip it in the same folder as this script.")
    exit()

# Dlib 68-point model indices
DLIB_LANDMARK_IDS = [
    30,  # Nose Tip
    36,  # Left Eye Outer Corner
    45,  # Right Eye Outer Corner
    48,  # Left Mouth Corner
    54,  # Right Mouth Corner
    8    # Chin
]

# 3D model points for Dlib
DLIB_MODEL_POINTS = np.array([
    ( 0.0,  0.0,    0.0),   # Nose Tip
    (-22.0, -17.0, -26.0),  # Left Eye Outer Corner
    ( 22.0, -17.0, -26.0),  # Right Eye Outer Corner
    (-15.0,  27.0, -20.0),  # Left Mouth Corner
    ( 15.0,  27.0, -20.0),  # Right Mouth Corner
    ( 0.0,  60.0, -35.0)    # Chin
], dtype=np.float32)

# --- 2. MEDIAPIPE SETUP ---
print("Loading MediaPipe model...")
mp_face_mesh = mp.solutions.face_mesh
face_mesh = mp_face_mesh.FaceMesh(
    static_image_mode=False,
                                  max_num_faces=1,
                                  refine_landmarks=True)

# MediaPipe 6-point model indices
MEDIAPIPE_LANDMARK_IDS = [
    1,    # Nose Tip
    133,  # Left Eye Outer Corner
    362,  # Right Eye Outer Corner
    234,  # Left Ear
    454,  # Right Ear
    152   # Chin
]

# 3D model points for MediaPipe
MEDIAPIPE_MODEL_POINTS = np.array([
    ( 0.0,   0.0,    0.0),    # Nose Tip
    (-30.0, -30.0, -30.0),    # Left Eye
    ( 30.0, -30.0, -30.0),    # Right Eye
```

```
    (-60.0,  60.0, -60.0),    # Left Ear
    ( 60.0,  60.0, -60.0),    # Right Ear
    ( 0.0,  100.0, -100.0)    # Chin
], dtype=np.float32)

# --- 3. SHARED PARAMETERS & FUNCTIONS ---
# Smoothing parameters
alpha = 0.7
smoothed_yaw = None
smoothed_pitch = None

# Camera Parameters (will be set in functions)
FOCAL_LENGTH = 1
CENTER = (0, 0)

def rotation_matrix_to_euler_angles(R):
    """ Converts a rotation matrix to Euler angles (yaw, pitch, roll) """
    yaw = np.arctan2(R[1, 0], R[0, 0]) * (180.0 / np.pi)
    pitch = np.arctan2(-R[2, 0], np.sqrt(R[2, 1]**2 + R[2, 2]**2)) * (180.0 / np.pi)
    roll = np.arctan2(R[2, 1], R[2, 2]) * (180.0 / np.pi)
    return yaw, pitch, roll

def draw_info(img, gaze, fps, framework_name):
    """ Draws the gaze arrow and info text on the frame """
    face = gaze["face"]
    arrow_length = img.shape[1] / 3
    dx = -arrow_length * np.sin(np.radians(gaze["yaw"]))
    dy = -arrow_length * np.sin(np.radians(gaze["pitch"]))

    cv2.arrowedLine(
        img,
        (int(face["x"]), int(face["y"])),
        (int(face["x"] + dx), int(face["y"] + dy)),
        (0, 0, 255), 2, cv2.LINE_AA, tipLength=0.2
    )

    cv2.putText(img, f"Yaw: {gaze['yaw']:.2f}", (20, 30),
                cv2.FONT_HERSHEY_PLAIN, 2, (255, 0, 0), 2)
    cv2.putText(img, f"Pitch: {gaze['pitch']:.2f}", (20, 60),
                cv2.FONT_HERSHEY_PLAIN, 2, (255, 0, 0), 2)
    cv2.putText(img, f"Roll: {gaze['roll']:.2f}", (20, 90),
                cv2.FONT_HERSHEY_PLAIN, 2, (255, 0, 0), 2)

    # Display the framework and FPS
    color = (0, 255, 0) if framework_name == "MediaPipe" else (0, 0, 255)
    cv2.putText(img, f"Framework: {framework_name}", (20, 120),
                cv2.FONT_HERSHEY_PLAIN, 2, color, 2)
    cv2.putText(img, f"FPS: {fps:.2f}", (20, 150),
                cv2.FONT_HERSHEY_PLAIN, 2, color, 2)

# --- 4. POSE ESTIMATION FUNCTIONS ---
def estimate_head_pose_dlib(image):
    """ Estimates head pose using Dlib """
    global FOCAL_LENGTH, CENTER
    gray = cv2.cvtColor(image, cv2.COLOR_BGR2GRAY)
    faces = dlib_detector(gray)

    if not faces:
        return None

    shape = dlib_predictor(gray, faces[0])
    ih, iw = image.shape[:2]

    image_points = np.array([
        [shape.part(i).x, shape.part(i).y]
        for i in DLIB_LANDMARK_IDS
    ], dtype=np.float32)

    # Camera Intrinsics
    FOCAL_LENGTH = iw
    CENTER = (iw/2, ih/2)
    CAMERA_MATRIX = np.array([
        [FOCAL_LENGTH, 0, CENTER[0]],
        [0, FOCAL_LENGTH, CENTER[1]],
        [0, 0, 1]
    ], dtype=np.float32)

    DIST_COEFFS = np.zeros((4, 1))  # No distortion

    # SolvePnP
    success, rotation_vector, translation_vector = cv2.solvePnP(
        DLIB_MODEL_POINTS, image_points, CAMERA_MATRIX, DIST_COEFFS
    )

    if not success:
        return None

    rotation_matrix, _ = cv2.Rodrigues(rotation_vector)
    yaw, pitch, roll = rotation_matrix_to_euler_angles(rotation_matrix)
    yaw = -yaw  # Adjust sign

    # Get face center
    x_coords = [p[0] for p in image_points]
    y_coords = [p[1] for p in image_points]
    face_x = np.mean(x_coords)
    face_y = np.mean(y_coords)

    return {"yaw": yaw, "pitch": pitch, "roll": roll,
            "face": {"x": face_x, "y": face_y}}

def estimate_head_pose_mediapipe(image):
    """ Estimates head pose using MediaPipe """
    global FOCAL_LENGTH, CENTER
    img_rgb = cv2.cvtColor(image, cv2.COLOR_BGR2RGB)
    results = face_mesh.process(img_rgb)

    if not results.multi_face_landmarks:
        return None

    face_landmarks = results.multi_face_landmarks[0]
    ih, iw = image.shape[:2]

    # Get 2D image points from MediaPipe
    image_points = np.array([
        [face_landmarks.landmark[i].x * iw, face_landmarks.landmark[i].y * ih]
        for i in MEDIAPIPE_LANDMARK_IDS
    ], dtype=np.float32)

    # Camera Intrinsics
    FOCAL_LENGTH = iw
    CENTER = (iw/2, ih/2)
    CAMERA_MATRIX = np.array([
        [FOCAL_LENGTH, 0, CENTER[0]],
        [0, FOCAL_LENGTH, CENTER[1]],
        [0, 0, 1]
    ], dtype=np.float32)

    DIST_COEFFS = np.zeros((4, 1))  # No distortion

    # SolvePnP
    success, rotation_vector, translation_vector = cv2.solvePnP(
        MEDIAPIPE_MODEL_POINTS, image_points, CAMERA_MATRIX, DIST_COEFFS
    )

    if not success:
```

```
        return None

    rotation_matrix, _ = cv2.Rodrigues(
        rotation_vector)
    yaw, pitch, roll =
        rotation_matrix_to_euler_angles(
        rotation_matrix)
    yaw = -yaw  # Adjust sign

    # Get face center (use nose tip)
    face_x = image_points[0][0]
    face_y = image_points[0][1]

    return {"yaw": yaw, "pitch": pitch, "roll":
        roll,
            "face": {"x": face_x, "y": face_y}}

# --- 5. MAIN WEBCAM LOOP ---
print("Running Head Pose Comparison...")
print(f"*** USING {'MEDIAPIPE' if USE_MEDIAPIPE
    else 'DLIB'} FOR POSE ESTIMATION ***")

cap = cv2.VideoCapture(0)
cv2.namedWindow("Head Pose Comparison")

while cap.isOpened():
    ret, frame = cap.read()
    if not ret:
        break

    # --- Gaze Estimation (SWITCHABLE) ---
    framework_name = ""
    gaze_data = None

    if USE_MEDIAPIPE:
        start_time = time.time()
        gaze_data = estimate_head_pose_mediapipe(
            frame)
        processing_time = time.time() - start_time
        framework_name = "MediaPipe"
    else:
        start_time = time.time()
        gaze_data = estimate_head_pose_dlib(frame)
        processing_time = time.time() - start_time
        framework_name = "Dlib"

    # Calculate FPS
    if processing_time > 0:
        fps = 1 / processing_time
    else:
        fps = float('inf')

    if gaze_data:
        # Smoothing
        if smoothed_yaw is None:
            smoothed_yaw = gaze_data["yaw"]
            smoothed_pitch = gaze_data["pitch"]
        else:
            smoothed_yaw = alpha * gaze_data["yaw"
                ] + (1 - alpha) * smoothed_yaw
            smoothed_pitch = alpha * gaze_data["
                pitch"] + (1 - alpha) *
                smoothed_pitch

        gaze_data["yaw"] = smoothed_yaw
        gaze_data["pitch"] = smoothed_pitch

        # Draw info
        draw_info(frame, gaze_data, fps,
            framework_name)

        # --- PyAutoGUI Mouse Control ---
        screen_width, screen_height = pyautogui.
            size()

        # This mapping can be tuned
        target_x = screen_width * (gaze_data["yaw"
            ] + 40) / 80
        target_y = screen_height * (gaze_data["
            pitch"] + 30) / 60

        target_x = np.clip(target_x, 0,
            screen_width - 1)
        target_y = np.clip(target_y, 0,
            screen_height - 1)

        pyautogui.moveTo(target_x, target_y)

    # Display the result
    cv2.imshow("Head Pose Comparison", frame)

    if cv2.waitKey(1) & 0xFF == ord("q"):
        break

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

Listing 1. Head Pose Comparison: Dlib vs MediaPipe Framework

B. Final Integrated System: MediaPipe with Gesture Control

This script represents the final, optimized implementation built entirely on the MediaPipe framework.

```
import cv2
import numpy as np
import mediapipe as mp
import pyautogui
import time

# --- MEDIAPIPE SETUP ---
print("Loading MediaPipe models...")
mp_face_mesh = mp.solutions.face_mesh
mp_hands = mp.solutions.hands
mp_drawing = mp.solutions.drawing_utils

face_mesh = mp_face_mesh.FaceMesh(
    static_image_mode=False,
    max_num_faces=1,
    refine_landmarks=True,
    min_detection_confidence=0.5,
    min_tracking_confidence=0.5
)

hands = mp_hands.Hands(
    static_image_mode=False,
    max_num_hands=1,
    min_detection_confidence=0.7,
    min_tracking_confidence=0.5
)

# --- CONFIGURATION ---
# Smoothing parameters for head pose
alpha = 0.7
smoothed_yaw = None
smoothed_pitch = None
smoothed_roll = None

# Click detection parameters
click_cooldown = 0.5  # Seconds between clicks
last_click_time = 0

# Scroll detection parameters
scroll_active = False
last_scroll_y = None
scroll_threshold = 0.05  # Pinch distance
    threshold
scroll_sensitivity = 2

# MediaPipe 6-point model indices for head pose
LANDMARK_IDS = [1, 133, 362, 234, 454, 152]

# 3D model points for MediaPipe
MODEL_POINTS = np.array([
    (0.0, 0.0, 0.0),       # Nose Tip
    (-30.0, -30.0, -30.0),  # Left Eye
    (30.0, -30.0, -30.0),   # Right Eye
    (-60.0, 60.0, -60.0),   # Left Ear
    (60.0, 60.0, -60.0),    # Right Ear
    (0.0, 100.0, -100.0)    # Chin
], dtype=np.float32)

# Camera Parameters
```

```python
FOCAL_LENGTH = 1
CENTER = (0, 0)

def rotation_matrix_to_euler_angles(R):
    """Converts a rotation matrix to Euler angles (yaw, pitch, roll)"""
    yaw = np.arctan2(R[1, 0], R[0, 0]) * (180.0 / np.pi)
    pitch = np.arctan2(-R[2, 0], np.sqrt(R[2, 1]**2 + R[2, 2]**2)) * (180.0 / np.pi)
    roll = np.arctan2(R[2, 1], R[2, 2]) * (180.0 / np.pi)
    return yaw, pitch, roll

def estimate_head_pose(image):
    """Estimates head pose using MediaPipe FaceMesh"""
    global FOCAL_LENGTH, CENTER
    img_rgb = cv2.cvtColor(image, cv2.COLOR_BGR2RGB)
    results = face_mesh.process(img_rgb)

    if not results.multi_face_landmarks:
        return None

    face_landmarks = results.multi_face_landmarks[0]
    ih, iw = image.shape[:2]

    # Get 2D image points from MediaPipe
    image_points = np.array([
        [face_landmarks.landmark[i].x * iw, face_landmarks.landmark[i].y * ih]
        for i in LANDMARK_IDS
    ], dtype=np.float32)

    # Camera Intrinsics
    FOCAL_LENGTH = iw
    CENTER = (iw/2, ih/2)
    CAMERA_MATRIX = np.array([
        [FOCAL_LENGTH, 0, CENTER[0]],
        [0, FOCAL_LENGTH, CENTER[1]],
        [0, 0, 1]
    ], dtype=np.float32)

    DIST_COEFFS = np.zeros((4, 1))  # No distortion

    # SolvePnP
    success, rotation_vector, _ = cv2.solvePnP(
        MODEL_POINTS, image_points, CAMERA_MATRIX, DIST_COEFFS
    )

    if not success:
        return None

    rotation_matrix, _ = cv2.Rodrigues(rotation_vector)
    yaw, pitch, roll = rotation_matrix_to_euler_angles(rotation_matrix)
    yaw = -yaw  # Adjust sign

    # Get face center (use nose tip)
    face_x = image_points[0][0]
    face_y = image_points[0][1]

    return {
        "yaw": yaw,
        "pitch": pitch,
        "roll": roll,
        "face": {"x": face_x, "y": face_y}
    }

def detect_fist(hand_landmarks):
    """Detects if hand is making a fist gesture"""
    # Check if all fingertips are below their PIP joints
    fingers_down = []

    # Index finger
    fingers_down.append(
        hand_landmarks.landmark[mp_hands.HandLandmark.INDEX_FINGER_TIP].y >
        hand_landmarks.landmark[mp_hands.HandLandmark.INDEX_FINGER_PIP].y
    )

    # Middle finger
    fingers_down.append(
        hand_landmarks.landmark[mp_hands.HandLandmark.MIDDLE_FINGER_TIP].y >
        hand_landmarks.landmark[mp_hands.HandLandmark.MIDDLE_FINGER_PIP].y
    )

    # Ring finger
    fingers_down.append(
        hand_landmarks.landmark[mp_hands.HandLandmark.RING_FINGER_TIP].y >
        hand_landmarks.landmark[mp_hands.HandLandmark.RING_FINGER_PIP].y
    )

    # Pinky finger
    fingers_down.append(
        hand_landmarks.landmark[mp_hands.HandLandmark.PINKY_TIP].y >
        hand_landmarks.landmark[mp_hands.HandLandmark.PINKY_PIP].y
    )

    return all(fingers_down)

def detect_pinch(hand_landmarks):
    """Detects pinch gesture and returns distance between thumb and index finger"""
    thumb_tip = hand_landmarks.landmark[mp_hands.HandLandmark.THUMB_TIP]
    index_tip = hand_landmarks.landmark[mp_hands.HandLandmark.INDEX_FINGER_TIP]

    # Calculate Euclidean distance
    distance = np.sqrt(
        (thumb_tip.x - index_tip.x)**2 +
        (thumb_tip.y - index_tip.y)**2
    )

    return distance

def draw_info(img, gaze, hand_gesture, fps):
    """Draws information overlay on frame"""
    # Draw gaze arrow
    if gaze:
        face = gaze["face"]
        arrow_length = img.shape[1] / 3
        dx = -arrow_length * np.sin(np.radians(gaze["yaw"]))
        dy = -arrow_length * np.sin(np.radians(gaze["pitch"]))

        cv2.arrowedLine(
            img,
            (int(face["x"]), int(face["y"])),
            (int(face["x"] + dx), int(face["y"] + dy)),
            (0, 0, 255), 2, cv2.LINE_AA, tipLength=0.2
        )

        # Display angles
        cv2.putText(img, f"Yaw: {gaze['yaw']:.1f}", (20, 30),
                    cv2.FONT_HERSHEY_SIMPLEX, 0.7, (255, 0, 0), 2)
        cv2.putText(img, f"Pitch: {gaze['pitch']:.1f}", (20, 60),
                    cv2.FONT_HERSHEY_SIMPLEX, 0.7, (255, 0, 0), 2)

    # Display gesture status
    if hand_gesture:
        cv2.putText(img, f"Gesture: {hand_gesture}", (20, 90),
                    cv2.FONT_HERSHEY_SIMPLEX, 0.7,
```

```
                          (0, 255, 0), 2)

    # Display FPS
    cv2.putText(img, f"FPS: {fps:.1f}", (20, 120),
                cv2.FONT_HERSHEY_SIMPLEX, 0.7, (0,
                     255, 255), 2)

# --- MAIN LOOP ---
print("Starting integrated gaze and gesture
     control system...")
cap = cv2.VideoCapture(0)
cv2.namedWindow("Gaze Controller with Gestures")

while cap.isOpened():
    start_time = time.time()
    ret, frame = cap.read()
    if not ret:
        break

    frame = cv2.flip(frame, 1)  # Mirror for
         intuitive control
    img_rgb = cv2.cvtColor(frame, cv2.
         COLOR_BGR2RGB)

    # --- HEAD POSE ESTIMATION ---
    gaze_data = estimate_head_pose(frame)

    if gaze_data:
        # Apply smoothing
        if smoothed_yaw is None:
            smoothed_yaw = gaze_data["yaw"]
            smoothed_pitch = gaze_data["pitch"]
        else:
            smoothed_yaw = alpha * gaze_data["yaw"
                 ] + (1 - alpha) * smoothed_yaw
            smoothed_pitch = alpha * gaze_data["
                 pitch"] + (1 - alpha) *
                 smoothed_pitch

        gaze_data["yaw"] = smoothed_yaw
        gaze_data["pitch"] = smoothed_pitch

        # --- MOUSE CURSOR CONTROL ---
        screen_width, screen_height = pyautogui.
             size()

        # Map head pose to screen coordinates
        target_x = screen_width * (gaze_data["yaw"
             ] + 40) / 80
        target_y = screen_height * (gaze_data["
             pitch"] + 30) / 60

        target_x = np.clip(target_x, 0,
             screen_width - 1)
        target_y = np.clip(target_y, 0,
             screen_height - 1)

        pyautogui.moveTo(target_x, target_y,
             duration=0)

    # --- HAND GESTURE DETECTION ---
    hand_results = hands.process(img_rgb)
    hand_gesture = None

    if hand_results.multi_hand_landmarks:
        for hand_landmarks in hand_results.
             multi_hand_landmarks:
            # Draw hand landmarks
            mp_drawing.draw_landmarks(
                frame,
                hand_landmarks,
                mp_hands.HAND_CONNECTIONS
            )

            # Detect fist for clicking
            if detect_fist(hand_landmarks):
                current_time = time.time()
                if current_time - last_click_time
                     > click_cooldown:
                    pyautogui.click()
                    last_click_time = current_time
                    hand_gesture = "CLICK"

            # Detect pinch for scrolling
            pinch_distance = detect_pinch(
                 hand_landmarks)

            if pinch_distance < scroll_threshold:
                hand_gesture = "SCROLL"
                index_tip = hand_landmarks.
                     landmark[mp_hands.
                     HandLandmark.INDEX_FINGER_TIP
                     ]

                if last_scroll_y is not None:
                    scroll_delta = (last_scroll_y
                         - index_tip.y) *
                         scroll_sensitivity * 100
                    if abs(scroll_delta) > 10:  #
                         Minimum movement
                         threshold
                        pyautogui.scroll(int(
                             scroll_delta))

                last_scroll_y = index_tip.y
            else:
                last_scroll_y = None

    # Calculate FPS
    processing_time = time.time() - start_time
    fps = 1 / processing_time if processing_time >
          0 else 0

    # Draw overlay
    draw_info(frame, gaze_data, hand_gesture, fps)

    # Display frame
    cv2.imshow("Gaze Controller with Gestures",
         frame)

    if cv2.waitKey(1) & 0xFF == ord("q"):
        break

cap.release()
cv2.destroyAllWindows()
face_mesh.close()
hands.close()
```

Listing 2. Final Integrated System Using MediaPipe FaceMesh and MediaPipe Hands

The Github repo for the project is https://github.com/BlackBetty-SaysHello/Mouse-Detection-using-Head-Pose-.git