Gesture-Based Smart Presentation Assistant

Introduction :

The aim of this project is to make a presentation assistant that works by using hand gestures instead of physical objects like remotes for example.
There will be many functionalities like hand-tracking to read the hand gestures , Registration to register people who are allowed to control the slides and an authentication function to find out if the user is registered or not.

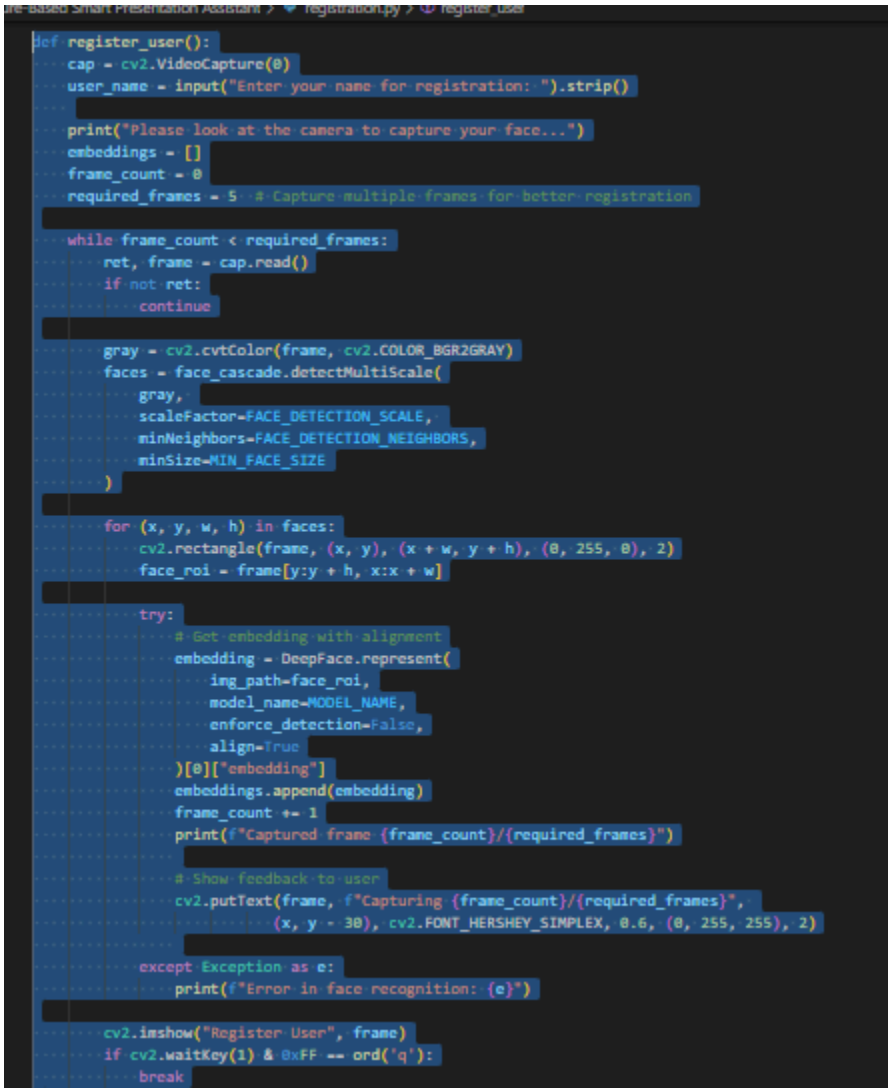
The Main Functionalities :

2- Hand Detector :

This part of the projects aims to recognise the presenter so that he can be the only one controlling the slides with his hand gestures, This is done by registering the users first by setting up some parameters like the embedding directory which is used to store the users face embeddings, (Face_Detections_Scale, Face_Detection_Neighpour, Min_Face_Size) which are the parameters used for the face detection in the opencv library , Cosine_similarity_threshold which is used to find the similarity between images, Then we define the model which is Facenet.

```
# Configuration
EMBEDDING_DIR = "user_embeddings/"
FACE_DETECTION_SCALE = 1.1
FACE_DETECTION_NEIGHBORS = 5
MIN_FACE_SIZE = (100, 100)
COSINE_SIMILARITY_THRESHOLD = 0.7  # Higher is more strict
MIN_CONSECUTIVE_MATCHES = 3  # Require multiple consecutive matches
MODEL_NAME = "Facenet"
```

After detecting the face using the model the embeddings are created then the Cosin_similarity is used to find the similarity between the embeddings created by the model and the embeddings in the database in the authentication process.



```python
def register_user():
    cap = cv2.VideoCapture(0)
    user_name = input("Enter your name for registration: ").strip()

    print("Please look at the camera to capture your face...")
    embeddings = []
    frame_count = 0
    required_frames = 5  # Capture multiple frames for better registration

    while frame_count < required_frames:
        ret, frame = cap.read()
        if not ret:
            continue

        gray = cv2.cvtColor(frame, cv2.COLOR_BGR2GRAY)
        faces = face_cascade.detectMultiScale(
            gray,
            scaleFactor=FACE_DETECTION_SCALE,
            minNeighbors=FACE_DETECTION_NEIGHBORS,
            minSize=MIN_FACE_SIZE
        )

        for (x, y, w, h) in faces:
            cv2.rectangle(frame, (x, y), (x + w, y + h), (0, 255, 0), 2)
            face_roi = frame[y:y + h, x:x + w]

            try:
                # Get embedding with alignment
                embedding = DeepFace.represent(
                    img_path=face_roi,
                    model_name=MODEL_NAME,
                    enforce_detection=False,
                    align=True
                )[0]["embedding"]
                embeddings.append(embedding)
                frame_count += 1
                print(f"Captured frame {frame_count}/{required_frames}")

                # Show feedback to user
                cv2.putText(frame, f"Capturing {frame_count}/{required_frames}",
                            (x, y - 30), cv2.FONT_HERSHEY_SIMPLEX, 0.6, (0, 255, 255), 2)

            except Exception as e:
                print(f"Error in face recognition: {e}")

        cv2.imshow("Register User", frame)
        if cv2.waitKey(1) & 0xFF == ord('q'):
            break
```

register_user() Function: Capturing Your Facial Signature

This function's main goal is to capture several images of your face and convert each into a unique numerical "embedding" (like a digital fingerprint) using DeepFace.

How it Works Briefly:

Starts Webcam andAsks Name: Gets ready to see you and know who you are.

Repeatedly Captures & Processes:

Take a picture (frame) from the webcam.

Find your face in the picture.

If found, extracts just the face and uses DeepFace to create its numerical embedding.

Saves this embedding and shows progress.
Collects Multiple Embeddings: Do this a few times to get a good set of your facial data.
Cleans Up: Turns off the webcam and closes the window.
These collected embeddings are then typically averaged and saved to represent your registered face.

```python
def load_embeddings():
    embeddings = {}
    for file in os.listdir(EMBEDDING_DIR):
        if file.endswith(".npz"):
            user_name = file.split(".")[0]
            embeddings[user_name] = np.load(os.path.join(EMBEDDING_DIR, file))["embedding"]
    return embeddings
```

This part is used to load the embeddings in our database to compare it with the embeddings we got got from the register-user-function using the similarity function

```python
def authenticate_user(face_img, saved_embeddings):
    try:
        live_embedding = DeepFace.represent(
            img_path=face_img,
            model_name=MODEL_NAME,
            enforce_detection=False,
            align=True
        )[0]["embedding"]

        max_similarity = -1
        recognized_user = "Unknown"

        for user_name, saved_embedding in saved_embeddings.items():
            # Calculate cosine similarity (higher is better)
            similarity = np.dot(live_embedding, saved_embedding) / (
                np.linalg.norm(live_embedding) * np.linalg.norm(saved_embedding))

            if similarity > max_similarity:
                max_similarity = similarity
                recognized_user = user_name

        if max_similarity > COSINE_SIMILARITY_THRESHOLD:
            return recognized_user, max_similarity
        return "Unknown", max_similarity

    except Exception as e:
        print(f"Authentication Error: {e}")
        return "Unknown", 0
```

This is the function that compares the embeddings using the cosine-similarity threshold if the similarity was bigger than the threshold then the user is authenticated.

2- Hand Detector :

This function uses Google's MediaPipe library to perform real-time hand detection and tracking from a webcam feed. The class processes video frames to locate hands and identify 21 distinct landmark points on each detected hand. It offers helpful functions to draw these landmarks and bounding boxes directly onto the video, determine which fingers are extended, and calculate the distance between any two landmark points. The detector can track up to two hands at once and continuously displays the annotated video output, showing the live hand tracking and any

derived information like finger states. Essentially, it provides a toolkit for building applications that understand hand gestures and positions.

```python
def __init__(self, mode=False, maxHands=2, detectionCon=0.5, minTrackCon=0.5):
    """
    :param mode: In static mode, detection is done on each image: slower
    :param maxHands: Maximum number of hands to detect
    :param detectionCon: Minimum Detection Confidence Threshold
    :param minTrackCon: Minimum Tracking Confidence Threshold
    """
    self.mode = mode
    self.maxHands = maxHands
    self.detectionCon = detectionCon
    self.minTrackCon = minTrackCon

    self.mpHands = mp.solutions.hands
    self.hands = self.mpHands.Hands(static_image_mode=self.mode, max_num_hands=self.maxHands,
                                    min_detection_confidence=self.detectionCon,
                                    min_tracking_confidence=self.minTrackCon)
    self.mpDraw = mp.solutions.drawing_utils
    self.tipIds = [4, 8, 12, 16, 20]
    self.fingers = []
    self.lmList = []
```

This __init__ method sets up a new HandDetector object.
It takes your preferences (like how many hands to find and how confident it should be) and uses them to create and configure the main MediaPipe hand-finding tool. It also gets ready some internal helpers for drawing and storing hand information later.
Essentially, it initializes the hand detector with your desired settings so it's ready to start working.

```python
def findHands(self, img, draw_lm=True, draw_bbox=True, flipType=True):
    """
    Finds hands in a BGR image.
    :param img: Image to find the hands in.
    :param draw: Flag to draw the output on the image.
    :return: Image with or without drawings
    """
    imgRGB = cv2.cvtColor(img, cv2.COLOR_BGR2RGB)
    self.results = self.hands.process(imgRGB)
    allHands = []
    h, w, c = img.shape
    if self.results.multi_hand_landmarks:
        for handType, handLms in zip(self.results.multi_handedness, self.results.multi_hand_landmarks):
            myHand = {}
            ## lmList
            mylmList = []
            xList = []
            yList = []
            for id, lm in enumerate(handLms.landmark):
                px, py, pz = int(lm.x * w), int(lm.y * h), int(lm.z * w)
                mylmList.append([px, py, pz])
                xList.append(px)
                yList.append(py)

            ## bbox
```

```python
            ## bbox
            xmin, xmax = min(xList), max(xList)
            ymin, ymax = min(yList), max(yList)
            boxW, boxH = xmax - xmin, ymax - ymin
            bbox = xmin, ymin, boxW, boxH
            cx, cy = bbox[0] + (bbox[2] // 2), bbox[1] + (bbox[3] // 2)
            bx, by = xmin, ymin
            tx, ty = xmax, ymax


            myHand["lmList"] = mylmList
            myHand["bbox"] = bbox
            myHand["center"] = (cx, cy)
            myHand["bottom"] = (bx, by)
            myHand["top"] = (tx, ty)

            if flipType:
                if handType.classification[0].label == "Right":
                    myHand["type"] = "Left"
                else:
                    myHand["type"] = "Right"
            else:
                myHand["type"] = handType.classification[0].label
            allHands.append(myHand)

            ## draw
            if draw_lm:
                self.mpDraw.draw_landmarks(img, handLms,
                                           self.mpHands.HAND_CONNECTIONS)
            if draw_bbox:
                cv2.rectangle(img, (bbox[0] - 20, bbox[1] - 20),
                              (bbox[0] + bbox[2] + 20, bbox[1] + bbox[3] + 20),
                              (255, 0, 255), 2)
                # cv2.putText(img, myHand["type"], (bbox[0] - 30, bbox[1] - 30), cv2.FONT_HERSHEY_PLAIN,
                #             2, (255, 0, 255), 2)

    if draw_lm or draw_bbox:
        return allHands, img
    else:
        return allHands
```

The findHands function finds hands in a picture and collects data about them.
Detects: It looks for hands in the input image using MediaPipe.
Retrieves Information: For every hand discovered, it computes:
the locations of the 21 landmarks (key points).
The hand is surrounded by a bounding box.The middle of the hand.

either the "Left" or "Right" hand (but if the camera is reversed, you can flip this).
Draws (Optional): The bounding box and/or landmarks are drawn onto the source image upon request.
Returns: If drawing was enabled, the updated image is returned together with a list of all identified hands (along with their data).
In essence, it locates hands, extracts their salient characteristics, and can visually draw attention to them.

```python
def fingersUp(self, myHand):
    """
    Finds how many fingers are open and returns in a list.
    Considers left and right hands separately
    :return: List of which fingers are up
    """
    myHandType = myHand["type"]
    myLmList = myHand["lmList"]
    if self.results.multi_hand_landmarks:
        fingers = []
        # Thumb
        if myHandType == "Right":
            if myLmList[self.tipIds[0]][0] > myLmList[self.tipIds[0] - 1][0]:
                fingers.append(1)
            else:
                fingers.append(0)
        else:
            if myLmList[self.tipIds[0]][0] < myLmList[self.tipIds[0] - 1][0]:
                fingers.append(1)
            else:
                fingers.append(0)

        # 4 Fingers
        for id in range(1, 5):
            if myLmList[self.tipIds[id]][1] < myLmList[self.tipIds[id] - 2][1]:
                fingers.append(1)
            else:
                fingers.append(0)
    return fingers
```

This method checks which of the fingers in a given hand are extended up.
It works by 1- identifying the hand type by first looking at whether the input myHand is a "Left" or "Right" hand, as this affects how the thumb is checked. It also gets the list of landmark coordinates (myLmList) for that hand 2- Checks Thumb: For a Right hand, it considers the thumb "up" if the thumb tip's x-coordinate is to the right of a point further down the thumb while the left is the opposite 3 - Checks Other Four Fingers : t loops through the index, middle, ring, and pinky fingers.For each of these fingers, it considers the finger "up" if its fingertip's y-coordinate is above (smaller y-value) a point two landmarks down that finger (closer to the palm). It appends 1 (up) or 0 (down) for each finger.

```python
def findDistance(self, p1, p2, img=None):
    x1, y1 = p1
    x2, y2 = p2
    cx, cy = (x1 + x2) // 2, (y1 + y2) // 2
    length = math.hypot(x2 - x1, y2 - y1)
    info = (x1, y1, x2, y2, cx, cy)
    if img is not None:
        cv2.circle(img, (x1, y1), 15, (255, 0, 255), cv2.FILLED)
        cv2.circle(img, (x2, y2), 15, (255, 0, 255), cv2.FILLED)
        cv2.line(img, (x1, y1), (x2, y2), (255, 0, 255), 3)
        cv2.circle(img, (cx, cy), 15, (255, 0, 255), cv2.FILLED)
        return length, info, img
    else:
        return length, info

def findPosition(self, img, handNo=0, draw=True):
    xList = []
    yList = []
    bbox = []
    self.lmList = []
    if self.results.multi_hand_landmarks:
        myHand = self.results.multi_hand_landmarks[handNo]
        for id, lm in enumerate(myHand.landmark):
            # print(id, lm)
            h, w, c = img.shape
            cx, cy = int(lm.x * w), int(lm.y * h)
            xList.append(cx)
            yList.append(cy)
            # print(id, cx, cy)
            self.lmList.append([id, cx, cy])
            if draw:
                cv2.circle(img, (cx, cy), 5, (255, 0, 255), cv2.FILLED)

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

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

    return self.lmList, bbox
```

Here's a more concise version:

`findDistance(self, p1, p2, img=None)` Calculates the Euclidean distance between `p1` and `p2` using `math.hypot`.

- Find the midpoint `(cx, cy)`.
- If `img` is provided, it marks points and draws a connecting line.
- Returns the distance, midpoint info, and modified image (if applicable).

`findPosition(self, img, handNo=0, draw=True)`

Extracts landmarks and a bounding box for a detected hand.

- Converts 21 landmarks to pixel coordinates, storing them in `self.lmList`.
- Finds bounding box dimensions `(xmin, ymin, xmax, ymax)`.

- If `draw=True`, it marks landmarks and highlights the bounding box on `img`.
- Returns `self.lmList` and bounding box info.
- This keeps the essential details while making it more compact. Let me know if you need it even shorter!

`3- main.py` Documentation

This script runs a Gesture-Based Smart Presentation Assistant, allowing users to control PowerPoint slides using hand gestures after facial authentication.
Core Functionalities:

1. User Authentication: Registers new users or authenticates existing ones via facial recognition (`registration.py`, `authentication.py`).
2. PPTX to Image Conversion: Converts a specified PowerPoint file into a series of images using Microsoft PowerPoint automation (`comtypes.client`).
3. Gesture Control Loop:
    - Captures webcam feed and detects hands using `HandDetector` (MediaPipe).
    - Interprets finger gestures (e.g., thumb up for previous, pinky for next) within a defined screen zone to navigate slides.
    - Allows drawing annotations, showing a pointer, and erasing annotations on slides using specific finger configurations.
    - Overlays the webcam feed onto the displayed slide.

Workflow:

1. Login/Register: User chooses to register (captures face embeddings) or authenticate (verifies face against stored embeddings). Exits on failure.
2. Slide Conversion: If authentication is successful, the script converts the `pptx_path` presentation into images in `output_folder`. Exits on error.
3. Presentation Mode:
    - Loads converted slide images.
    - Initializes webcam and `HandDetector`.
    - Enters a loop to display the current slide and process hand gestures from the webcam.
    - Gestures control slide changes (`slide_num`), drawing (`annotations`), pointing, and erasing.
    - A small delay (`delay`) prevents accidental multiple gesture activations.
    - The webcam feed is shown in a corner of the slide.
    - Press 'q' to quit.

```python
# ========== USER LOGIN SYSTEM ==========
print("Welcome to the Hand Gesture Presentation System")
print("1. Register a new user")
print("2. Authenticate to start the presentation")

choice = input("Enter your choice (1 or 2): ").strip()

if choice == '1':
    register_user()
    print("Now please restart the program and choose authentication to proceed.")
    exit(0)

elif choice == '2':
    access_granted = authentication_system()
    if not access_granted:
        print("Authentication failed or canceled. Exiting...")
        exit(1)

else:
    print("Invalid input. Please restart the program and choose 1 or 2.")
    exit(1)

# ========== CONVERT PPTX TO IMAGES ==========
def convert_pptx_to_images(pptx_path, output_folder):
    # Check if PowerPoint file exists
    if not os.path.exists(pptx_path):
        raise FileNotFoundError(f"PowerPoint file not found at: {pptx_path}\nPlease make sure the file exists and the path is correct.")

    if not os.path.exists(output_folder):
        os.makedirs(output_folder)
```

Key Settings (Constants):

- `pptx_path`, `output_folder`: Paths for presentation and converted slides.
- `width`, `height`: Display dimensions.
- `ge_thresh_x`, `ge_thresh_y`: Define the gesture activation zone.