Hand Tracking 30 FPS using CPU | OpenCV Python (2021) | Computer Vision

In this hand-tracking project, I am engaging with **MEDIAPIPE**, a robust framework from Google that provides pre-trained models for tasks like face detection, hand tracking, and object recognition. My current project leverages the hand-tracking model from MEDIAPIPE. The main goal of the project is to control the volume of a device through hand gestures.

The model functions on two core algorithms: Palm Detection and Hand Landmarks. Palm Detection initially processes the entire image to detect and isolate the hand region. Subsequently, the Hand Landmarks algorithm identifies 21 specific points within this isolated hand region. This model's precision is achieved through training on a dataset of 30,000 hand images, each carefully annotated to enhance the landmark detection accuracy.

In my "HandTrackingProject," I set up the development environment in PyCharm, where I navigated through File > Settings to the interpreter settings. There, I installed essential packages such as **opencv-python** for image processing and **mediapipe** for hand-tracking functionality.

A compatibility issue arose with Python 3.12.1, which was unsupportive of **MEDIAPIPE**, prompting me to downgrade to Python 3.9.13. This version was compatible and facilitated the installation of the required packages in PyCharm's virtual environment, thus isolating project dependencies and preventing potential conflicts with other projects.

In this project, I initiated a file named **HandTrackingMin**, aiming to encapsulate the fundamental code needed for hand tracking. To enhance the project's accessibility, I used my mobile phone as a webcam, connecting it to my PC through EpoCam, an application designed for such integrations.

---

import cv2
This imports **OpenCV**, a library used for image and video processing in Python.

import mediapipe as mp
This imports the **MEDIAPIPE** library, essential for hand-tracking functionalities.

import time
This imports the time module, which can be used for time-related functions such as checking the frame rate.

cap = cv2.VideoCapture(0)
Initializes the webcam (or mobile camera through EpoCam) to capture video. The parameter 0 generally refers to the default webcam.

while True:
Starts an infinite loop to continuously capture frames from the camera.

success, img = cap.read()

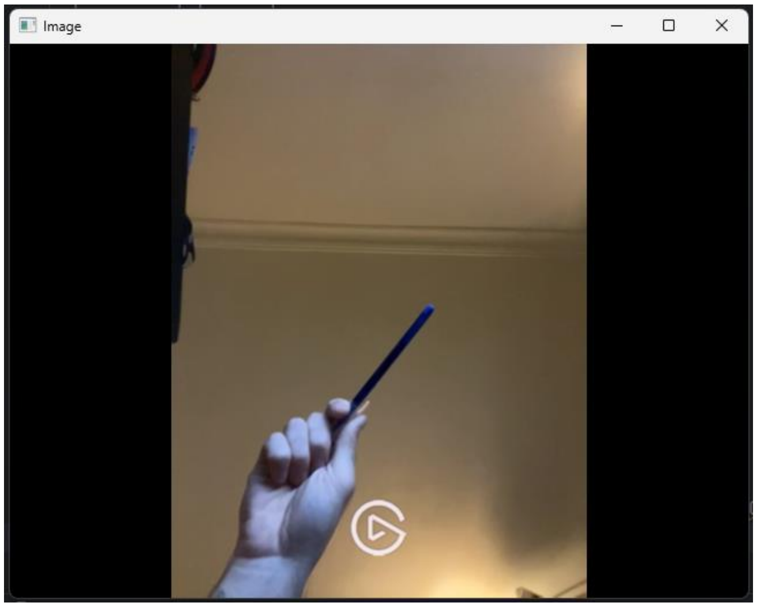
Captures each frame from the video capture object; **success** is a boolean indicating if the frame was successfully captured, and **img** is the frame itself.

cv2.imshow("Image", img)
Displays the captured frame in a window titled "**Image**".

cv2.waitKey(1)
Waits for 1 millisecond before capturing the next frame, effectively making the loop suitable for video processing by updating the displayed frame continuously.

```
import cv2


import mediapipe as mp
import time

cap = cv2.VideoCapture(0)

while True:
    success, img = cap.read()

    cv2.imshow("Image", img)
    cv2.waitKey(1)
```

This code sets the groundwork for a hand-tracking application, continuously capturing and displaying video frames. To integrate MEDIAPIPE's hand tracking, additional steps are required to process the captured images and render the hand landmarks.



mpHands = mp.solutions.hands
This line initializes MEDIAPIPE's hand-tracking module, making its functionality available. It accesses the '**hands'** solution from the **MEDIAPIPE** library, which contains the pre-trained hand-tracking model.

hands = mpHands.Hands()
Here, an instance of the hand-tracking model has been created. This object, `**hands**`, will be used to process the images captured from the webcam and detect hands in them.

imgRGB = cv2.cvtColor(img, cv2.COLOR_BGR2RGB)
Since OpenCV captures images in BGR format, but **MEDIAPIPE** requires RGB format, this line converts the image from BGR to RGB. This conversion ensures that the colors in the image are correctly interpreted by **MEDIAPIPE** during hand detection.

results = hands.process(imgRGB)
This line is where the hand-tracking model processes the converted RGB image to detect hands. The `**process**` method of the `**hands**` object takes the RGB image as input and returns the hand-tracking results, which include the detection and landmarks of the hands in the image.

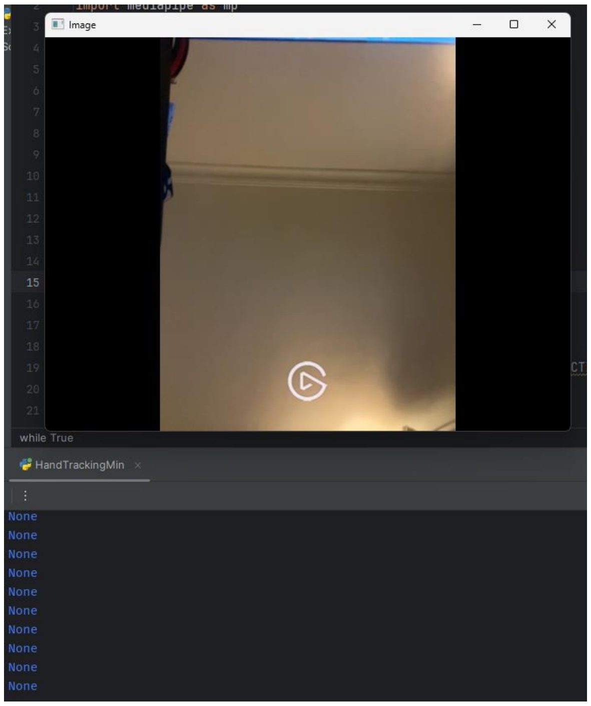
print(results.multi_hand_landmarks)
the program now outputs the hand landmark data when hands are detected in the webcam feed. If no hands are detected, it prints None. When hands are present, it prints the landmark data, which is a detailed representation of the hand's position and orientation in the form of coordinates for each landmark.

If **results.multi_hand_landmarks** is **None**, this means that no hands were detected in the frame, and the console will display None.

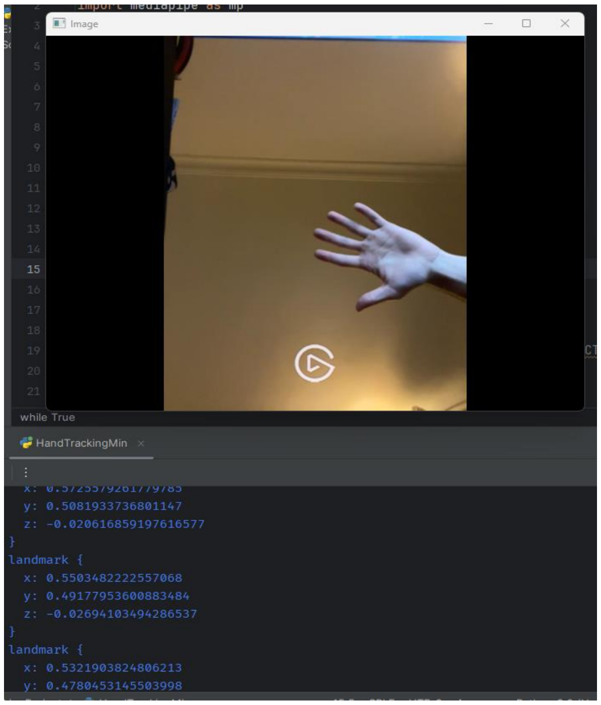
If hands are detected, **results.multi_hand_landmarks** contains the landmark data for each detected hand. In this case, the console will display complex data structures, each representing the 21 landmarks of a detected hand. This output includes the **x**, **y**, and **z** coordinates for each landmark, providing detailed information about the hand's pose and orientation in the captured image.

```
cap = cv2.VideoCapture(0)

mpHands = mp.solutions.hands
hands = mpHands.Hands()

while True:
    success, img = cap.read()
    imgRGB = cv2.cvtColor(img, cv2.COLOR_BGR2RGB)
    results = hands.process(imgRGB)
    print(results.multi_hand_landmarks)
```

These additions to the code enable real-time hand tracking by processing the video feed frame by frame, detecting hands, and providing landmarks that can be used for further analysis or applications. Additionally, the code continuously outputs the detection results to the console, confirming that hands are being detected.



```
None
None
None
None
None
None
None
None
None
None
```



```
  x: 0.5725579261779785
  y: 0.5081933736801147
  z: -0.020616859197616577
}
landmark {
  x: 0.5503482222557068
  y: 0.49177953600883484
  z: -0.02694103494286537
}
landmark {
  x: 0.5321903824806213
  y: 0.4780453145503998
```

---

mpDraw = mp.solutions.drawing_utils
This line initializes **MEDIAPIPE**'s drawing utilities, which are used to visualize the landmarks and connections between them on the image.

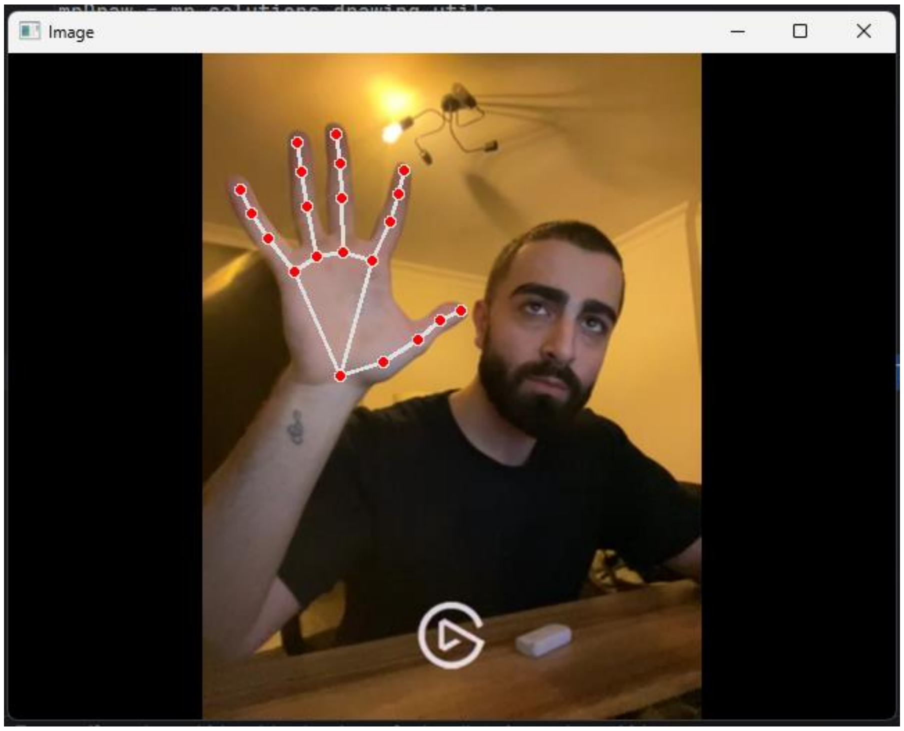
if results.multi_hand_landmarks
This conditional statement checks if any hand landmarks have been detected in the current frame. If there are detected landmarks (`**multi_hand_landmarks**` is not `**None**`), the code inside this block executes.

for handLms in results.multi_hand_landmarks
This loop iterates over each set of hand landmarks detected in the frame. If multiple hands are detected, it processes each hand individually.

mpDraw.draw_landmarks(img, handLms, mpHands.HAND_CONNECTIONS)
This function call is used to draw the detected hand landmarks (**handLms**) on the image (**img**). The third parameter, **mpHands.HAND_CONNECTIONS,** specifies the connections between the landmarks. This means that in addition to the landmarks (points on the hand), lines connecting these points (representing the hand's structure) are also drawn.

```
cap = cv2.VideoCapture(0)

mpHands = mp.solutions.hands
hands = mpHands.Hands()
mpDraw = mp.solutions.drawing_utils

while True:
    success, img = cap.read()
    imgRGB = cv2.cvtColor(img, cv2.COLOR_BGR2RGB)
    results = hands.process(imgRGB)
    #print(results.multi_hand_landmarks)

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

    cv2.imshow("Image", img)
    cv2.waitKey(1)
```

These new lines of code enhance the hand-tracking application by not only detecting the hand landmarks but also visually representing them on the output video feed. This visual representation helps in verifying the accuracy of the hand tracking and provides a clear, intuitive understanding of how the hand is being detected and interpreted by the system in real time.



To enhance the functionality of the hand-tracking project and provide a more interactive and informative experience, the next step is to integrate a feature that displays the frames per second (FPS) on the webcam image. This addition is crucial for monitoring the performance of the hand-tracking application in real-time, as it allows for the assessment of how smoothly the application is running. High FPS indicates a smooth and responsive experience, while low FPS may signal performance issues.

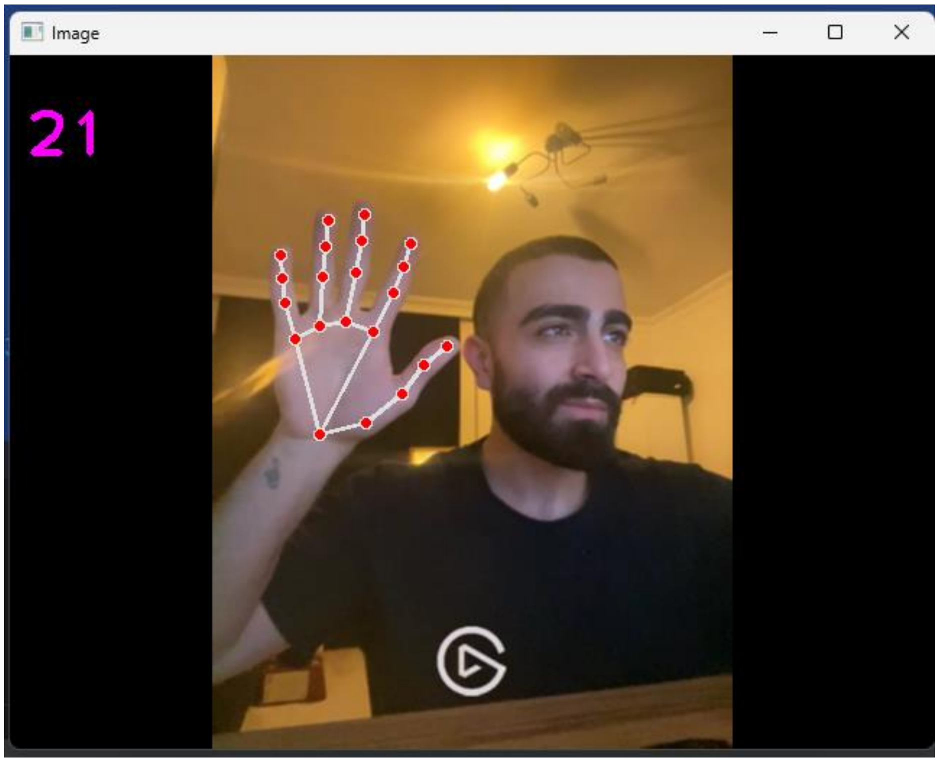
pTime = 0
cTime = 0
Two variables, **pTime** (previous time) and **cTime** (current time), are initialized outside the main loop. These are used to calculate the time elapsed between frames, which is necessary for calculating the FPS.

cTime = time.time()
fps = 1 / (cTime - pTime)
pTime = cTime
Inside the loop, after capturing and processing the image, **cTime** is updated with the current time using **time.time()**. The FPS is then calculated by taking the inverse of the difference between **cTime** and **pTime**. After the FPS calculation, **pTime** is set to **cTime** for use in the next iteration.

cv2.putText(img, str(int(fps)), (10, 70), cv2.FONT_HERSHEY_PLAIN, 3, (255, 0, 255), 3)
The calculated FPS is converted to an integer and displayed on the webcam image using **cv2.putText()**. The position, font, scale, color, and thickness of the FPS text are specified to ensure clear visibility.

```
cap = cv2.VideoCapture(0)

mpHands = mp.solutions.hands
hands = mpHands.Hands()
mpDraw = mp.solutions.drawing_utils

pTime = 0
cTime = 0

while True:
    success, img = cap.read()
    imgRGB = cv2.cvtColor(img, cv2.COLOR_BGR2RGB)
    results = hands.process(imgRGB)
    #print(results.multi_hand_landmarks)

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

    cTime = time.time()
    fps = 1/(cTime-pTime)
    pTime = cTime

    cv2.putText(img, str(int(fps)),(10,70), cv2.FONT_HERSHEY_PLAIN,
3, (255,0,255), 3)
```

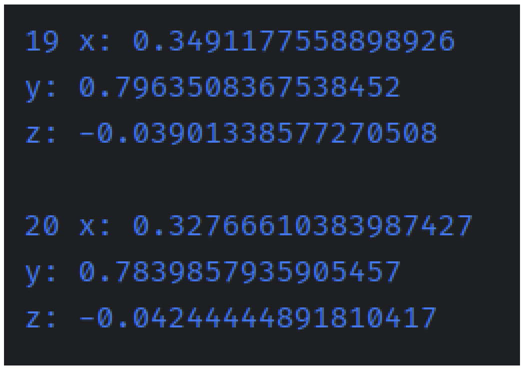
```
    cv2.imshow("Image", img)
    cv2.waitKey(1)
```



---

for id, lm in enumerate(handLms.landmark):
This line iterates over each detected landmark within a hand. **enumerate** provides both the index (**id**) and the landmark object (**lm**), enabling access to each landmark's specific data.

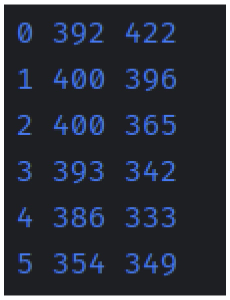
#print(id, lm)
Initially, this line was used to print the normalized coordinates of each landmark (**lm**) along with its identifier (**id**). However, normalized coordinates (values between 0 and 1) are not as intuitive to understand in the context of the image's actual size. Therefore, this line was commented out to stop printing these less readable values.



```
19 x: 0.3491177558898926
y: 0.7963508367538452
z: -0.03901338577270508

20 x: 0.32766610383987427
y: 0.7839857935905457
z: -0.04244444891810417
```

h, w, c = img.shape
cx, cy = int(lm.x * w), int(lm.y * h)
Here, the code calculates the actual pixel coordinates of each landmark on the image. **img.shape** gives the dimensions of the image, which are used to convert the normalized coordinates (**lm.x, lm.y**) to pixel coordinates (**cx, cy**). This conversion makes the landmark positions more understandable and practical for further processing or visualization.

print(id, cx, cy)
This line prints the identifier of each landmark along with its coordinates in pixels. By switching to pixel coordinates, the output becomes more meaningful and directly usable, especially for tasks like tracking or interacting with specific points on the image.



```
0 392 422
1 400 396
2 400 365
3 393 342
4 386 333
5 354 349
```

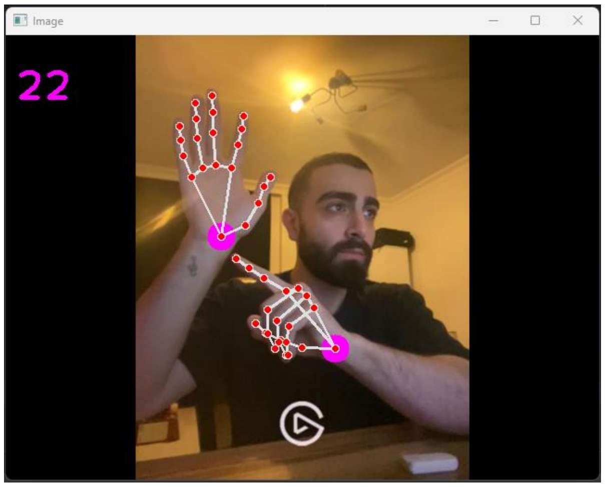
if id == 0:
This conditional statement checks if the current landmark being processed is the first landmark (id equal to 0). In the context of MediaPipe's hand tracking, each landmark on the hand has a unique identifier, where 0 usually refers to a specific point, such as the wrist or the tip of the thumb, depending on the model's landmark scheme. Similarly, if 4 was used instead (**if id == 4:**), it would target the landmark with the identifier 4, which might correspond to a different specific point on the hand, such as the tip of the thumb or another finger, again depending on how the landmarks are defined in the model.

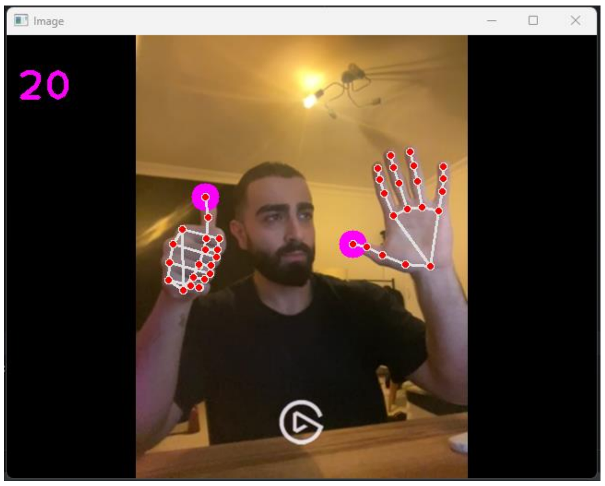
cv2.circle(img, (cx, cy), 15, (255, 0, 255), cv2.FILLED)
This line of code draws a circle on the image **img** at the coordinates (**cx, cy**), which are the pixel coordinates of the landmark. The circle has a radius of **15** pixels, is colored in magenta (255, 0, 255 in BGR color space), and is filled (**cv2.FILLED**).

```
if results.multi_hand_landmarks:
    for handLms in results.multi_hand_landmarks:
        for id, lm in enumerate(handLms.landmark):
            #print(id,lm)
            h, w, c = img.shape
            cx, cy, = int(lm.x*w,), int(lm.y*h)
            print(id, cx, cy)
            #if id == 0:
            if id == 4:
                cv2.circle(img, (cx,cy), 15, (255,0,255),
cv2.FILLED)
```

```
        mpDraw.draw_landmarks(img, handLms,
mpHands.HAND_CONNECTIONS)
```

The switch from normalized to pixel coordinates in the output was made to enhance readability and utility. Normalized coordinates are helpful for abstract, scale-independent processing but are less intuitive for direct spatial representation of an image. Conversely, pixel coordinates are more intuitive and practical for applications requiring knowledge of the actual position of landmarks on the displayed image. Additionally, the code enhancement to visually indicate the position of the first landmark, such as the wrist-in-hand tracking, on the webcam feed serves a crucial role. By drawing a filled circle at this landmark's location, it becomes easy to see where the landmark is on the hand in real-time. This visual aid is invaluable for debugging, enhancing visualization, and understanding the hand's movement and orientation as captured by the hand-tracking model, making the overall application more interactive and user-friendly.





In the evolution of my hand-tracking project, I abstracted the hand-tracking code into a separate Python file named HandTrackingModule.py. This modularization was aimed at creating a reusable, organized, and self-contained component that encapsulates all the hand-tracking functionality. This approach aligns with best practices in software development, promoting code reusability and maintainability.

class HandDetector:
Defines the **HandDetector** class, encapsulating all methods and properties needed for hand tracking.

def __init__(self, mode=False, max_hands=2, detection_confidence=0.5, tracking_confidence=0.5)
Initializes an instance of the **HandDetector** class with customizable parameters:
mode: If set to True, the hand detection is treated as a static image; False treats it as a video stream.
max_hands: Maximum number of hands to detect.
detection_confidence: Minimum confidence threshold for the detection to be considered successful.
tracking_confidence: Minimum confidence threshold for the tracking to be considered successful.

self.mp_hands = mp.solutions.hands
self.hands = self.mp_hands.Hands(static_image_mode=mode,
                    max_num_hands=max_hands,

```
                min_detection_confidence=detection_confidence,
                min_tracking_confidence=tracking_confidence)
self.mp_draw = mp.solutions.drawing_utils
```
self.mp_hands: Assigns the MediaPipe Hands solution to a variable for easier access.
self.hands: Creates a MediaPipe Hands object with the specified parameters, ready to perform hand tracking.
self.mp_draw: Assigns MediaPipe drawing utilities to a variable, which will be used to draw the hand landmarks and connections.

```
def find_hands(self, img, draw=True):
```
A method to process the image, detect hands, and optionally draw the landmarks on the image.

```
img_rgb = cv2.cvtColor(img, cv2.COLOR_BGR2RGB)
self.results = self.hands.process(img_rgb)
```
Converts the image from BGR to RGB color space as **MediaPipe** requires RGB images.
Processes the RGB image to detect hands and stores the result in **self.results**.

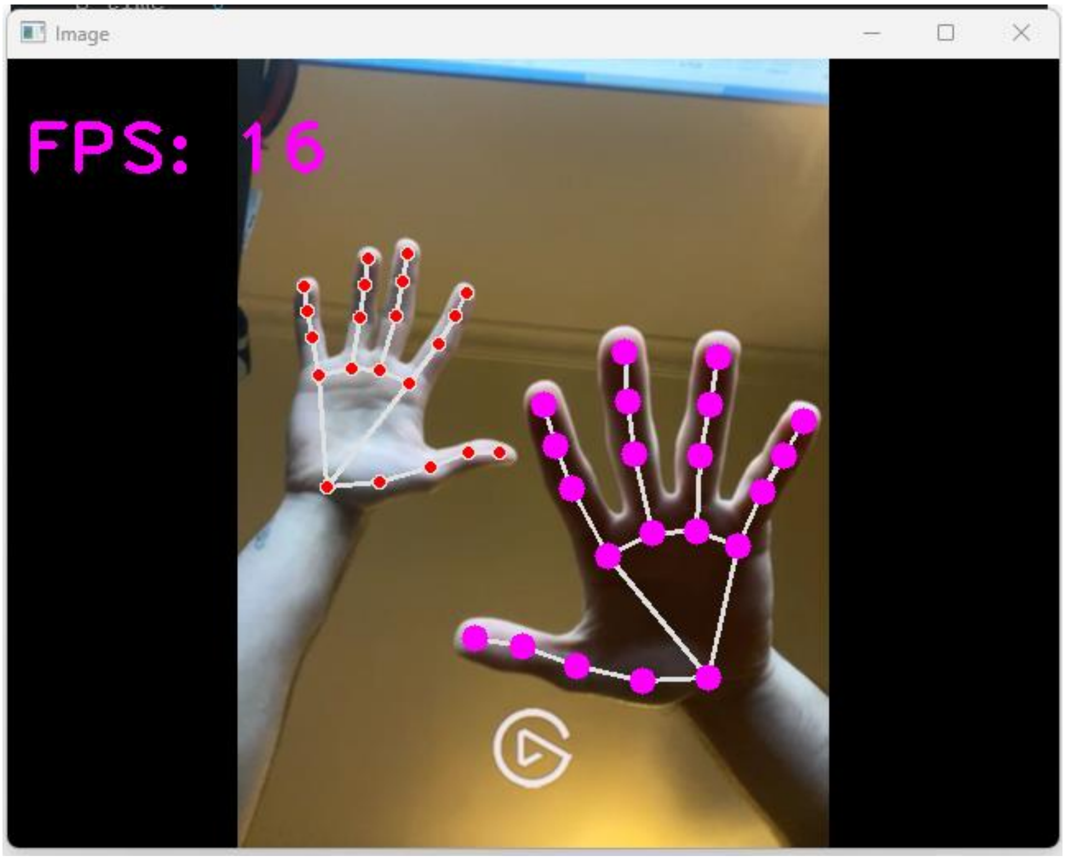
```
if self.results.multi_hand_landmarks:
   for hand_lms in self.results.multi_hand_landmarks:
      if draw:
         self.mp_draw.draw_landmarks(img, hand_lms, self.mp_hands.HAND_CONNECTIONS)
```
Checks if any hand landmarks are detected and iterates through each set of landmarks.
If **draw** is **True**, it draws the landmarks and their connections on the image.

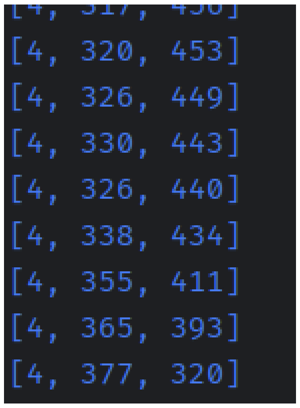
```
def find_position(self, img, hand_no=0, draw=True):
```
A method to find the position of hand landmarks in the image.

```
lm_list = []
if self.results.multi_hand_landmarks:
   my_hand = self.results.multi_hand_landmarks[hand_no]
   for id, lm in enumerate(my_hand.landmark):
      h, w, c = img.shape
      cx, cy = int(lm.x * w), int(lm.y * h)
      lm_list.append([id, cx, cy])
      if draw:
         cv2.circle(img, (cx, cy), 8, (255, 0, 255), cv2.FILLED)
```
Initializes an empty list **lm_list** to store landmark positions.
If landmarks are detected, it extracts and processes each landmark of the specified hand (**hand_no**).
Converts the normalized position of landmarks to pixel coordinates and appends them to **lm_list**.
If **draw** is **True**, it draws circles on the landmarks in the image.

Usage in main Function
The main function demonstrates how to use the HandDetector class in a live webcam feed to detect and display hand landmarks in real-time.

```python
import cv2
import mediapipe as mp
import time

class HandDetector:
    def __init__(self, mode=False, max_hands=2,
detection_confidence=0.5, tracking_confidence=0.5):
        self.mode = mode
        self.max_hands = max_hands
        self.detection_confidence = detection_confidence
        self.tracking_confidence = tracking_confidence

        self.mp_hands = mp.solutions.hands
        self.hands = self.mp_hands.Hands(static_image_mode=mode,
                                         max_num_hands=max_hands,

min_detection_confidence=detection_confidence,

min_tracking_confidence=tracking_confidence)
        self.mp_draw = mp.solutions.drawing_utils

    def find_hands(self, img, draw=True):
        img_rgb = cv2.cvtColor(img, cv2.COLOR_BGR2RGB)
        self.results = self.hands.process(img_rgb)

        if self.results.multi_hand_landmarks:
            for hand_lms in self.results.multi_hand_landmarks:
                if draw:
                    self.mp_draw.draw_landmarks(img, hand_lms,
self.mp_hands.HAND_CONNECTIONS)
        return img

    def find_position(self, img, hand_no=0, draw=True):
        lm_list = []
        if self.results.multi_hand_landmarks:
            my_hand = self.results.multi_hand_landmarks[hand_no]
            for id, lm in enumerate(my_hand.landmark):
                h, w, c = img.shape
                cx, cy = int(lm.x * w), int(lm.y * h)
                lm_list.append([id, cx, cy])
                if draw:
                    cv2.circle(img, (cx, cy), 8, (255, 0, 255),
cv2.FILLED)
        return lm_list


def main():
    p_time = 0
    cap = cv2.VideoCapture(0)
    detector = HandDetector()

    while True:
        success, img = cap.read()
        img = detector.find_hands(img)
        lm_list = detector.find_position(img)
        if lm_list:
            print(lm_list[4])

        c_time = time.time()
        fps = 1 / (c_time - p_time)
        p_time = c_time

        cv2.putText(img, f'FPS: {int(fps)}', (10, 70),
cv2.FONT_HERSHEY_PLAIN, 3, (255, 0, 255), 3)
        cv2.imshow("Image", img)
        cv2.waitKey(1)

if __name__ == "__main__":
    main()
```





```
[4, 320, 453]
[4, 326, 449]
[4, 330, 443]
[4, 326, 440]
[4, 338, 434]
[4, 355, 411]
[4, 365, 393]
[4, 377, 320]
```

The **HandTrackingModule.py** transforms the intricate process of hand tracking into a user-friendly class, the **HandDetector**. This allows us to seamlessly include hand-tracking capabilities into diverse applications. By simplifying MediaPipe's advanced features, the module provides developers with straightforward tools to detect hands and pinpoint their key points in real time.

Our image demonstrates the module's robustness, highlighting two hands with a network of colored dots and lines that accurately follow the hands' movements. Particularly noteworthy is the thumb tip of the right hand, denoted by purple dots, which is isolated and its movements tracked in real-time. The corresponding output on the console, a series of (x, y) coordinates, represents the dynamic position of this thumb tip within the video feed, as identified by the landmark with an index of 4. This level of precision in tracking allows for precise gesture control and offers potential for innovative applications in user interface design, gaming, and even educational tools that rely on hand movements.

Furthermore, the module's design shines through the organized and maintainable nature of the code, making it a scalable resource for future projects. The ability to track multiple hands and discern specific landmarks like thumb tips in different colors lays the groundwork for more intricate hand-interaction applications. It exemplifies how developers can leverage the **HandTrackingModule.py** to build upon a reliable foundation without getting into the complexity of MediaPipe's underlying framework.

In conclusion, the creation of `**HandTrackingModule.py**` as an independent module significantly simplifies the integration of hand tracking into any project. It's an efficient, reusable solution tailored to fit a range of requirements, ensuring that projects are not only future-proof but also maintain a high standard of interactivity and user engagement. The visual output, with its clear delineation of hand landmarks and the ability to observe landmark tracking in action, is a testament to the practical applications and the high performance of the module. This makes the module a valuable asset in the development of responsive and interactive digital experiences.

Building upon the successful implementation of hand tracking, I have progressed to a more interactive application that utilizes hand gestures to control the volume on a device. A new file was created, VolumeHandControl.py, and is dedicated to this functionality. This file leverages the **HandTrackingModule** to detect and interpret hand gestures, converting them into volume control commands.

```python
import cv2
import time
import numpy as np
import HandTrackingModule as htm

###########################
wCam, hCam = 640, 480
###########################



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

detector = htm.HandDetector()

while True:
    success , img = cap.read()
    img = detector.find_hands(img)


    cTime = time.time()
    fps = 1/(cTime-pTime)
    pTime = cTime
    cv2.putText(img, f'FPS: {int(fps)}', (40,70),
cv2.FONT_HERSHEY_PLAIN, 2, (255,0,00), 2)

    cv2.imshow("Img", img)
    cv2.waitKey(1)
```

The **VolumeHandControl.py** script initiates with the standard camera setup, defining the dimensions (640x480 pixels) to ensure a consistent field of view for gesture recognition. The main operational loop of the script begins with activating the camera and employing the **HandDetector** class to continuously analyze the video feed for hand landmarks.

The primary focus of this script is to establish a real-time feedback loop that detects hand gestures and maps them to volume control commands. The process involves capturing each frame from the camera, using the **find_hands** method to identify and annotate hand landmarks, and then calculating and displaying the frame rate (FPS) to monitor performance.

This new phase of the project, represented by **VolumeHandControl.py**, marks a transition from basic hand tracking to a more sophisticated and interactive application. The script lays the groundwork for future enhancements, such as refining gesture recognition to include a wider range of controls and integrating more complex functionalities.

```
detector = htm.HandDetector(detection_confidence=0.7)

while True:
    success , img = cap.read()
    img = detector.find_hands(img)
    hand_lms = detector.find_position(img, draw=False)
    if len(hand_lms) != 0:
        print(hand_lms[2])

    cTime = time.time()
    fps = 1/(cTime-pTime)
    pTime = cTime
    cv2.putText(img, f'FPS: {int(fps)}', (40,70),
cv2.FONT_HERSHEY_PLAIN, 2, (255,0,00), 2)

    cv2.imshow("Img", img)
    cv2.waitKey(1)
```

In the enhanced **VolumeHandControl.py** script, the **HandTrackingModule** is further utilized to not only detect hand gestures but also to pinpoint the precise position of specific landmarks on the hand. This advancement is crucial for interpreting gestures more accurately and mapping them to volume control actions.

The **HandDetector** is initialized with a **detection_confidence** parameter set to **0.7**, indicating a more stringent criterion for recognizing hands. This adjustment ensures that the hand tracking is more reliable and reduces false positives.

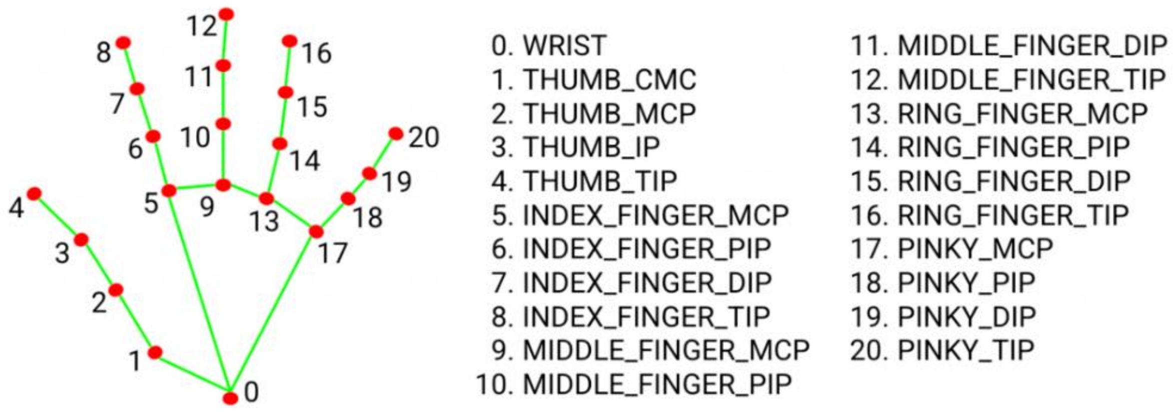
The script now includes a call to **detector.find_position(img, draw=False)**, which returns a list of hand landmarks (**hand_lms**). This method retrieves the spatial coordinates of each landmark on the detected hand. By setting draw to False, the landmarks are not drawn on the output image, suggesting that the focus is on processing the landmark data rather than visualizing it.

The script checks if any landmarks are detected **if len(hand_lms) != 0:** and then specifically prints the coordinates of the third landmark **print(hand_lms[2])**

This demonstrates the ability to access and potentially utilize individual landmarks for specific control actions, like adjusting volume based on the position and movement of certain fingers or hand parts.

To bridge the gap between the indices and their corresponding hand parts, one can refer to the **MediaPipe** hand tracking diagram which labels each landmark with a number and a descriptive name, like the WRIST for landmark 0 or INDEX_FINGER_TIP for landmark 8. The complete list and visual representation of these landmarks are provided by **MediaPipe** and can be accessed at their hand model documentation page: MediaPipe Hand Landmark Model.

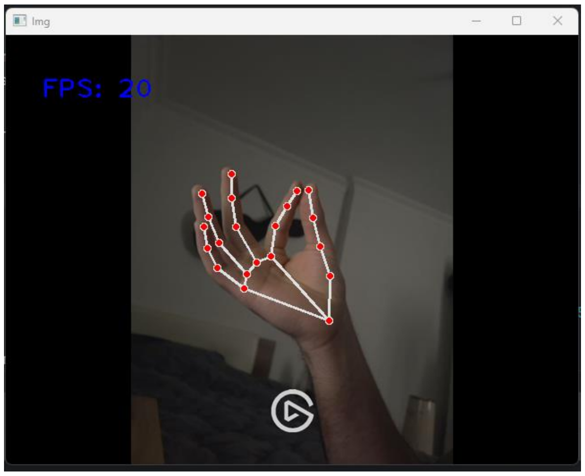
https://google.github.io/mediapipe/solutions/hands

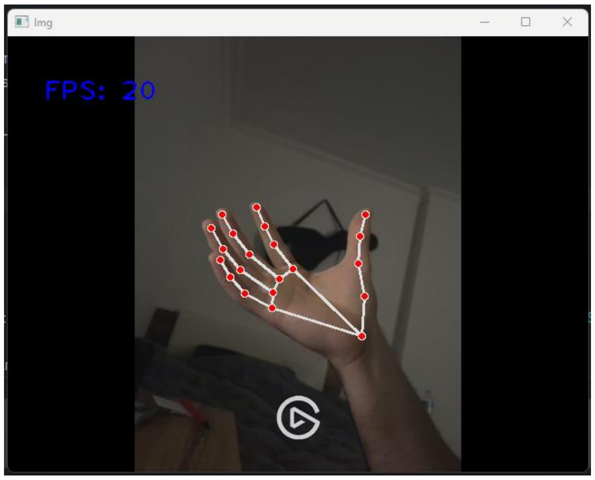


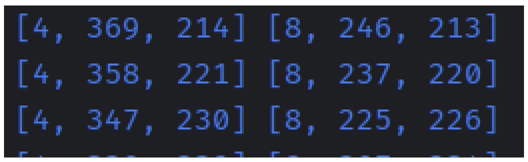
Considering this, the focus has shifted to landmarks **'4'** and **'8'**, which correspond to the tip of the thumb and the tip of the index finger, respectively. The script now includes a conditional statement that checks for the presence of detected landmarks:

```
if len(hand_lms) != 0:
    print(hand_lms[4], hand_lms[8])
```

When this condition is met, it signifies that the hand is in the frame and landmarks are successfully detected. The script then prints out the coordinates for the thumb tip and index fingertip. These specific landmarks are instrumental in performing pinch gestures. The output from the VolumeHandControl.py script now specifically provides the location data for just two fingertips: the thumb (landmark 4) and the index finger (landmark 8).







```
[4, 369, 214] [8, 246, 213]
[4, 358, 221] [8, 237, 220]
[4, 347, 230] [8, 225, 226]
```

Moving forward, I've enhanced the script to not only track the thumb and index finger tips but also to visually represent their positions. By extracting their coordinates:

x1, y1 = hand_lms[4][1], hand_lms[4][2]
Grabs the x and y coordinates for the tip of the thumb.

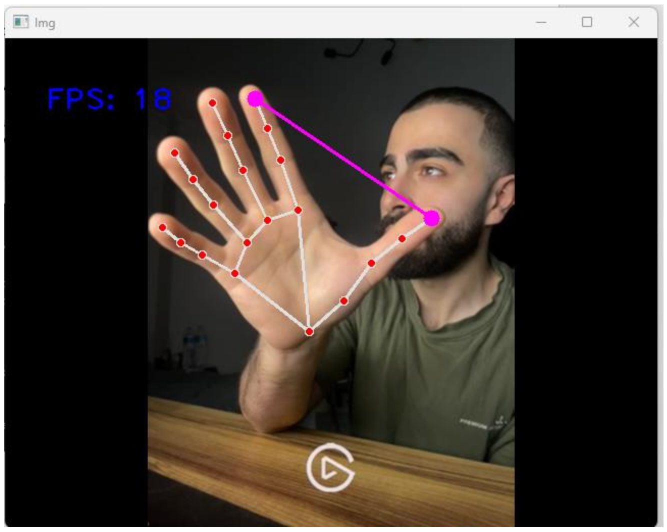
x2, y2 = hand_lms[8][1], hand_lms[8][2]
The same is applied to the tip of the finger.

Then, visual markers on the live feed are created

A filled purple circle at the thumb tip's coordinates and another at the index finger tip's coordinates. Each circle has a radius of 8 pixels, which makes the tips visibly marked for easy identification.

These two points with a purple line, which serves as a visual indicator of the gesture being made.

```
if len(hand_lms) != 0:
    print(hand_lms[4], hand_lms[8])

    x1, y1 = hand_lms[4][1], hand_lms[4][2]
    x2, y2 = hand_lms[8][1], hand_lms[8][2]

    cv2.circle(img, (x1,y1), 8, (255,0,255), cv2.FILLED)
    cv2.circle(img, (x2,y2), 8, (255,0,255), cv2.FILLED)
    cv2.line(img, (x1,y1), (x2,y2), (255,0,255), 2)

cTime = time.time()
fps = 1/(cTime-pTime)
```



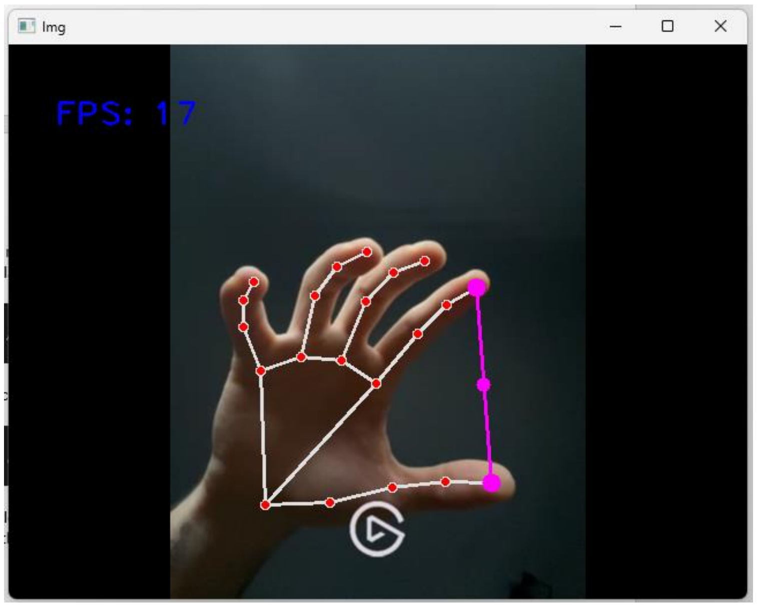
I've included a third circle in my visualization that marks the midpoint of the line connecting the thumb and index fingertips. By calculating the midpoint's coordinates (cx, cy) using

```
cx, cy = (x1 + x2) // 2, (y1 + y2) // 2
```

I then draw a smaller purple circle with a radius of 6 pixels at this midpoint using the code:

```
cv2.circle(img, (cx,cy), 6, (255,0,255), cv2.FILLED)
```

This filled circle provides a clear reference point for the center of the pinching gesture, enhancing the gesture's visual feedback on the live feed.

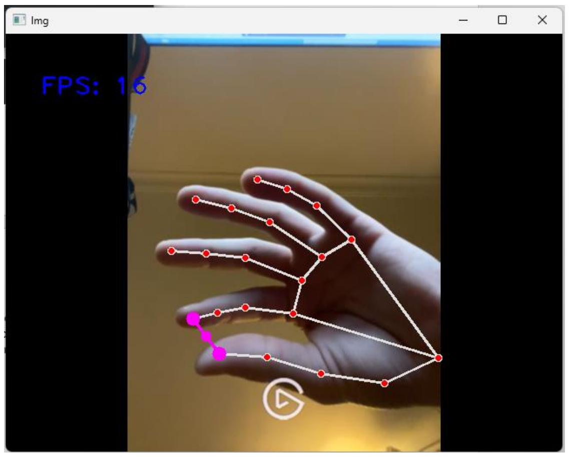


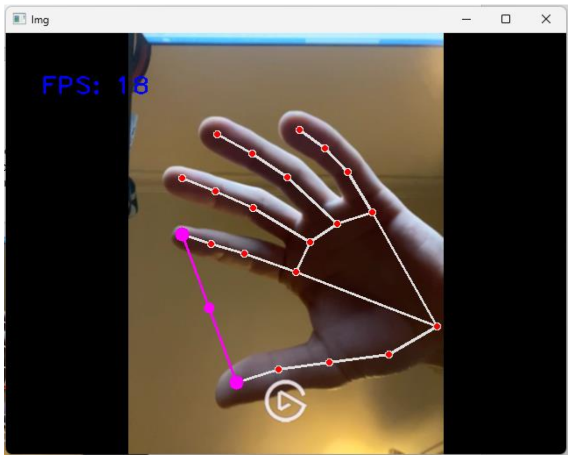
Incorporating the **math** module into my **VolumeHandControl.py** script, I'm now able to calculate the distance between the thumb and index finger, which I plan to use as a dynamic input for volume control. The distance or length is determined using Pythagoras' theorem, conveniently implemented with **math.hypot(x2 - x1, y2 - y1)**, which gives me the hypotenuse of the triangle formed by the line and its projections on the x and y axes — effectively, the straight-line distance between the two fingertips.

I've commented out the print statement for the landmarks since the console output of the landmark positions isn't needed anymore.

```
#print(hand_lms[4], hand_lms[8])
```

Now, with the length printed out, I can observe the changing distance as I bring my thumb and index finger closer or move them apart. This numerical length will be directly correlated to the volume level, allowing me to create a gesture-based volume control mechanism that feels natural and responsive.





```
41.048751503547585
41.048751503547585
41.340053217188775
```

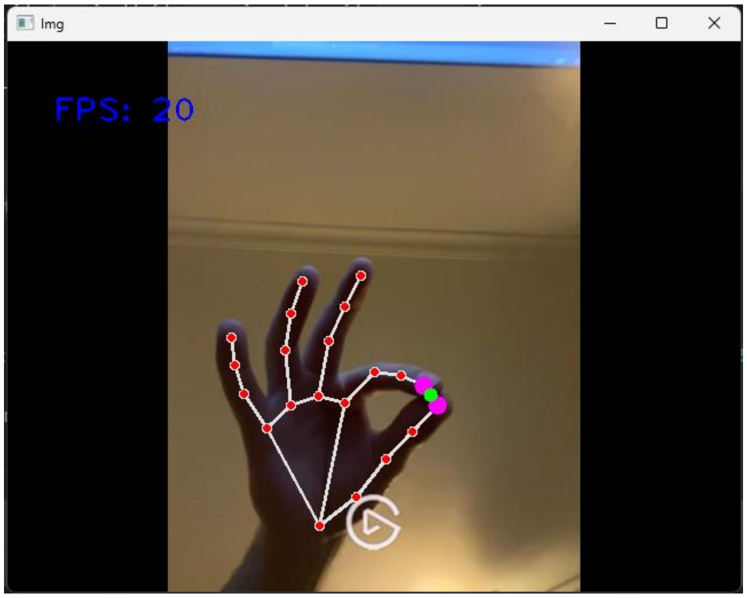
```
152.5417975507041
153.44705927452634
153.02614155757834
```

Now, by incorporating the math module and calculating the length of the line between the index and thumb, my project takes a significant step forward. The length measurement is crucial as it becomes the determining factor for adjusting the volume—the greater the distance between the thumb and index finger, the larger the value output by the script, and vice versa. This relationship is clearly observable: when I spread my fingers wider, the printed length value increases, indicating a volume increase, and when I close the gap between them, the value decreases, signaling a volume reduction.

```
length = math.hypot(x2 - x1, y2 - y1)
print(length)
```

To create a more interactive experience, I added a condition that checks the length of the line between these two points. If the distance is less than 20 — suggesting that the fingers are close together as if pressing a button — I draw a circle at the midpoint (cx, cy) in green. This visual feedback, a green circle, simulates a button effect to signify that the fingers are close enough to 'activate' or potentially change the volume.

```
if length<20:
    cv2.circle(img, (cx, cy), 6, (0, 255, 0), cv2.FILLED)
```



Moving forward, I utilized the **pycaw** library, which I discovered on GitHub. It provides a way to interact with the audio features of the system directly from Python. After installing it alongside the other packages, I incorporated its functionality into my script.

1. Integration of **pycaw**
   - I began by adding the necessary imports from **pycaw** at the top of my script.
   - I then set up the initialization code to interact with the audio device interfaces.

2. Understanding Volume Range
   - To understand the volume levels I could control, I retrieved the volume range using **volume.GetVolumeRange()** and printed it out.
   - The output **(-64.0, 0.0, 0.03125)** indicates that the volume levels I can set range from -64 (minimum) to 0 (maximum).

```
(-64.0, 0.0, 0.03125)
```

With this information, I can now proceed to map the distance between my thumb and index finger to this volume range. When the fingers are close together, I could set the volume to a lower level, and as they move apart, I would increase the volume accordingly. This intuitive interaction could allow for a smooth and dynamic volume control experience as part of my hand-tracking project.

Here’s the portion of the script that handles the audio interface:

```
from comtypes import CLSCTX_ALL
from pycaw.pycaw import AudioUtilities, IaudioEndpointVolume
```

```
#Initialization
devices = AudioUtilities.GetSpeakers()
interface = devices.Activate(
    IAudioEndpointVolume._iid_, CLSCTX_ALL, None)
volume = interface.QueryInterface(IAudioEndpointVolume)

#Fetching the volume range and printing it
print(volume.GetVolumeRange())

#Other functionalities are commented out for now
#volume.GetMute()
#volume.GetMasterVolumeLevel()
#volume.SetMasterVolumeLevel(-20.0, None)
```

Now that I have the volume range, I'll be able to create a function that maps the length of the line between the thumb and index finger to the volume range, effectively controlling the system's volume through hand gestures.

By reintegrating the command **volume.SetMasterVolumeLevel(-20.0, None)** into my script and executing it, I effectively changed the system volume to **26%** of its maximum. This action confirmed that the function was working and I had control over the volume level programmatically.

When I altered the parameter to **-5** in the **volume.SetMasterVolumeLevel(-5.0, None)** and ran the script again, the volume was set to **72%**, demonstrating a direct correlation between the numeric value provided to the function and the system's volume level.

This test is crucial as it establishes a baseline for how I can translate the physical distance between my thumb and index finger into precise volume levels. By mapping the distance to the volume range of **-64** to **0**, I can create a more natural and dynamic volume control mechanism. The next step would be to scale the distance measured from the hand landmarks to this volume range so that as I pinch my fingers closer or spread them apart, the volume decreases or increases correspondingly.

I've made some strategic adjustments to the volume control segment of my code to dynamically handle the audio levels:

volRange = volume.GetVolumeRange()
This line fetches the volume range from the system's audio settings, which returns a tuple with the minimum and maximum volume levels.

volume.SetMasterVolumeLevel(0, None)
Here, I'm setting the system volume to its maximum level as defined by the **pycaw** library, where **0** represents **100%** volume.

minVol = volRange[0]
I store the minimum volume level which is retrieved from the first position of the **volRange** tuple.

maxVol = volRange[1]
Similarly, I store the maximum volume level, which is **0** in the system, from the second position of the **volRange** tuple.

By setting these variables, I've created a framework to map the distance between my hand landmarks to the system's volume range. The minimum and maximum volume variables (**minVol** and **maxVol**) are crucial for establishing the bounds within which I can adjust the volume based on hand gestures. This means I can now create a formula that translates the distance between my thumb and index finger into a corresponding volume level within this range. As my hand gesture indicates a **pinch** or **spread**, the script can adjust the system volume down or up, respectively, within these set bounds.

```
devices = AudioUtilities.GetSpeakers()
interface = devices.Activate(
    IAudioEndpointVolume._iid_, CLSCTX_ALL, None)
volume = interface.QueryInterface(IAudioEndpointVolume)
#volume.GetMute()
#volume.GetMasterVolumeLevel()
volRange = volume.GetVolumeRange()
volume.SetMasterVolumeLevel(0, None)
minVom = volRange[0]
maxVom = volRange[1]
```

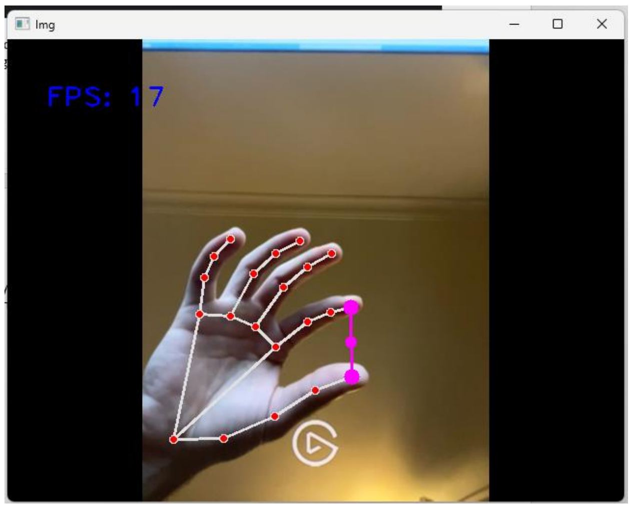
To seamlessly integrate hand gestures with volume control, I've employed the concept of range mapping. First, I established the physical distance my hand gestures would cover — a span from 50 (fingers close together) to 300 (fingers fully extended). This range represents the hand gesture's input values.

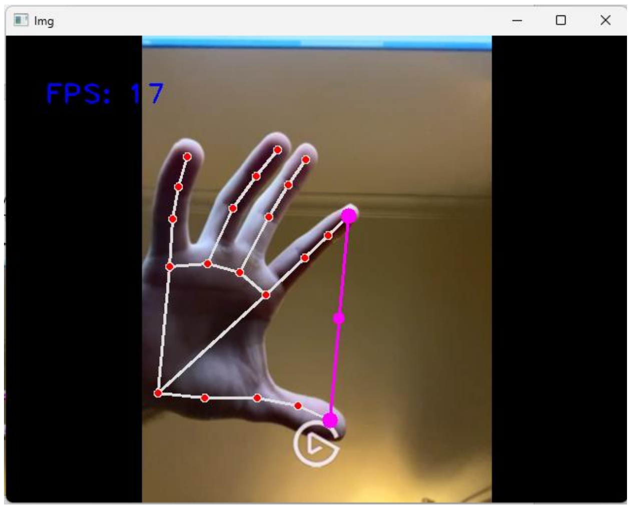
Next, I determined the system's volume control range using **pycaw**, which I found to be from -64 (muted) to 0 (maximum volume).

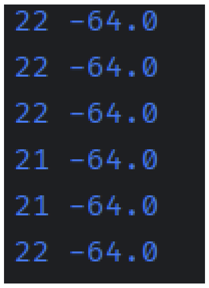
Now, the crucial part is to translate the hand gesture input into the volume control output. For this, I used numpy's **interp** function, which interprets the length variable — the distance between my thumb and index finger — within the hand gesture range and maps it to the corresponding volume level in the volume range.

```
vol = np.interp(length, [50,300],[minVol, maxVol])
print(int(length), vol)
```

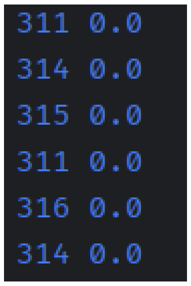
By printing the vol variable, I can observe that as the distance between my fingers decreases, the volume value approaches -65, moving towards the lower end of the volume range, simulating a reduction in volume or muting. Conversely, spreading my fingers apart drives the volume towards 0, indicating an increase in volume towards the maximum level.







```
22 -64.0
22 -64.0
22 -64.0
21 -64.0
21 -64.0
22 -64.0
```



```
311 0.0
314 0.0
315 0.0
311 0.0
316 0.0
314 0.0
```

In the accompanying images, the relationship between hand distance and volume level is visually demonstrated. As the distance between the thumb and index finger increases, which is captured by the range of 20 to 300, the corresponding volume value approaches 0 — this signifies the volume is being turned up, reaching its maximum capacity. Conversely, as the distance decreases, the volume level moves closer to -64, indicating a decrease in volume. These images effectively illustrate how a simple hand gesture can translate into a nuanced control mechanism, allowing for an intuitive adjustment of the device's audio levels.

I've optimized the script to adjust the system volume dynamically, directly in response to the calculated length from the hand landmarks. Here's what I did to streamline the process:

Initially, I had the line **volume.SetMasterVolumeLevel(0, None)** immediately following the retrieval of the volume range. This was effectively setting the volume to its maximum level every time the script ran, which wasn't the desired behavior for dynamic control.

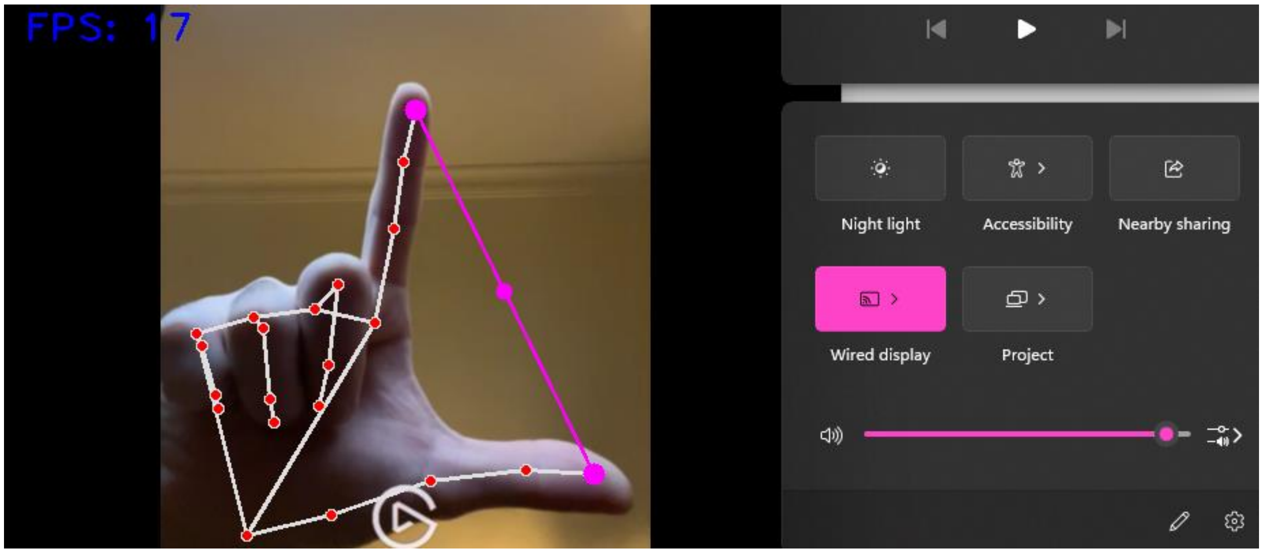
To refine this, I moved the command to a new location in the script, placing it after the volume level is determined by the distance between my thumb and index finger.

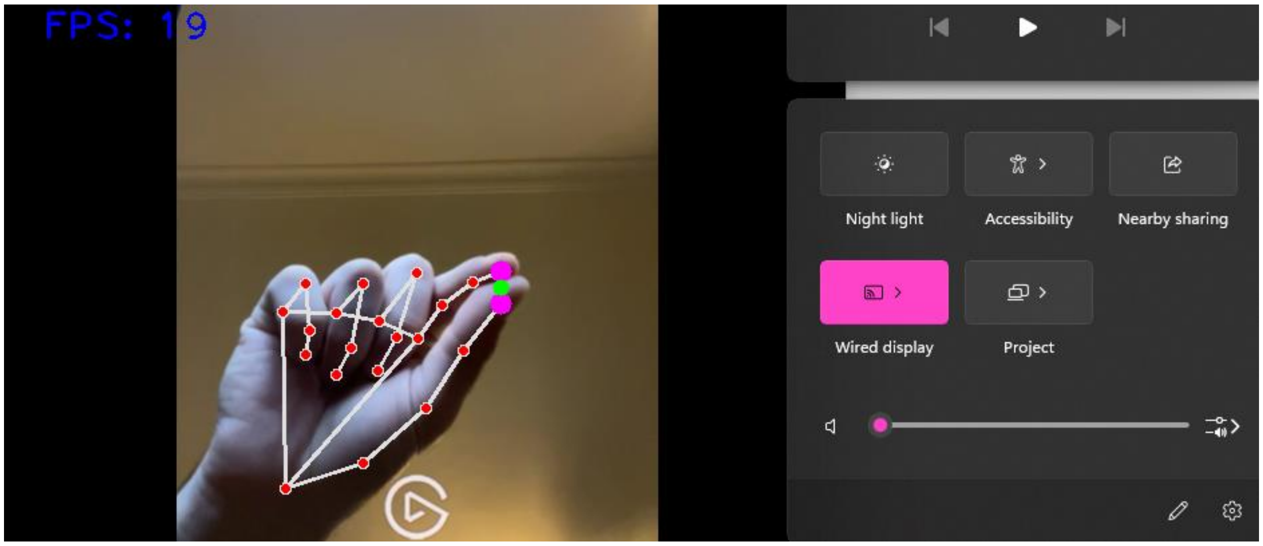
```
vol = np.interp(length, [50,300],[minVol, maxVol])
print(int(length), vol)
volume.SetMasterVolumeLevel(vol, None)
```

Now, **vol** is a variable that represents the desired volume level, mapped from the distance of the hand gesture. The **SetMasterVolumeLevel** function is called with **vol** as a parameter, which means the volume is now being set to a level that corresponds to the current hand position, rather than automatically going to the maximum. This adjustment provides real-time volume control based on the actual gesture, creating a responsive and interactive experience.

```
import cv2
import time
import numpy as np
import HandTrackingModule as htm
import math
from comtypes import CLSCTX_ALL
from pycaw.pycaw import AudioUtilities, IAudioEndpointVolume

###########################
wCam, hCam = 640, 480
###########################



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

detector = htm.HandDetector(detection_confidence=0.7)



devices = AudioUtilities.GetSpeakers()
interface = devices.Activate(
    IAudioEndpointVolume._iid_, CLSCTX_ALL, None)
volume = interface.QueryInterface(IAudioEndpointVolume)
#volume.GetMute()
#volume.GetMasterVolumeLevel()
volRange = volume.GetVolumeRange()
minVol = volRange[0]
maxVol = volRange[1]
```

```python
while True:
    success , img = cap.read()
    img = detector.find_hands(img)
    hand_lms = detector.find_position(img, draw=False)
    if len(hand_lms) != 0:
        #print(hand_lms[4], hand_lms[8])

        x1, y1 = hand_lms[4][1], hand_lms[4][2]
        x2, y2 = hand_lms[8][1], hand_lms[8][2]
        cx, cy = (x1 + x2) // 2, (y1 + y2) // 2

        cv2.circle(img, (x1,y1), 8, (255,0,255), cv2.FILLED)
        cv2.circle(img, (x2,y2), 8, (255,0,255), cv2.FILLED)
        cv2.line(img, (x1,y1), (x2,y2), (255,0,255), 2)
        cv2.circle(img, (cx,cy), 6, (255,0,255), cv2.FILLED)

        length = math.hypot(x2 - x1, y2 - y1)
        #print(length)


        # Hand Range 50 - 300
        # Volume Range -64 - 0

        vol = np.interp(length, [50,300],[minVol, maxVol])
        print(int(length), vol)
        volume.SetMasterVolumeLevel(vol, None)

        if length<30:
            cv2.circle(img, (cx, cy), 6, (0, 255, 0), cv2.FILLED)

    cTime = time.time()
    fps = 1/(cTime-pTime)
    pTime = cTime
    cv2.putText(img, f'FPS: {int(fps)}', (40,70),
cv2.FONT_HERSHEY_PLAIN, 2, (255,0,00), 2)

    cv2.imshow("Img", img)
    cv2.waitKey(1)
```





The above images illustrate a clear demonstration of how the volume control in my application corresponds directly to the distance between my thumb and index finger. As the distance increases, the volume slider on the system interface moves towards the higher end, and as my fingers draw closer together, the slider decreases, reflecting a lower volume setting. This visual feedback confirms the effective mapping of hand gestures to audio output control in real time.

In the initial implementation of the hand-tracking volume control application, I encountered a problem where the volume could not be locked at 100%. This issue arose because the volume would inadvertently adjust when the hand moved out of the camera's view or when the finger distance changed. To address this, a volume-locking mechanism was introduced.

vol_lock = False
This line initializes a boolean variable **vol_lock** to **False**. It's used to keep track of whether the volume should be locked or adjustable. Initially, the volume is not locked, allowing for adjustments.

if length < 30:
This conditional statement checks if the distance between the thumb and index finger (denoted by **length** is less than 30. This specific gesture (fingers close together) is chosen as a trigger to unlock the volume control.

cv2.circle(img, (cx, cy), 6, (0, 255, 0), cv2.FILLED)
Inside the **if length < 30:** block, this line draws a filled green circle at the center point between the thumb and index finger. This serves as a visual indicator that the volume control is unlocked.

vol_lock = False (within the **if length < 30:** block):
This sets **vol_lock** to **False**, unlocking the volume control so it can be adjusted again.

if not vol_lock:
This condition checks if **vol_lock** is **False**. If so, it proceeds to adjust the volume based on the distance between the thumb and index finger. This ensures that volume adjustments only occur when the volume control is unlocked.

if vol >= maxVol * 0.95:
This line checks if the volume level (**vol**) has reached or exceeded 95% of the maximum volume (**maxVol**). If true, it implies the user intends to set the volume at or near 100%, and thus the volume control should be locked.

vol_lock = True
This sets **vol_lock** to **True**, effectively locking the volume control and preventing further adjustments until explicitly unlocked.

if cv2.waitKey(1) & 0xFF == ord('q'):
This line checks if the 'q' key has been pressed. If so, it breaks out of the while loop, allowing the program to proceed to closure.

cap.release()
After breaking out of the loop, this line releases the webcam, ensuring that the resource is properly freed and available for other applications.

cv2.destroyAllWindows()
This line closes all OpenCV windows that were opened during the execution of the program, preventing any leftover GUI windows from lingering on the screen.

Each of these lines of code adds to the functionality, user control, and robustness of the application, addressing the initial issue and improving the overall user experience.

The Final version of the code in the VolumeHandControl file was the following:

```
import cv2
import time
import numpy as np
```

```python
import HandTrackingModule as htm
import math
from ctypes import cast, POINTER
from comtypes import CLSCTX_ALL
from pycaw.pycaw import AudioUtilities, IAudioEndpointVolume

###########################
wCam, hCam = 640, 480
###########################

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

detector = htm.HandDetector(detection_confidence=0.7)

devices = AudioUtilities.GetSpeakers()
interface = devices.Activate(
    IAudioEndpointVolume._iid_, CLSCTX_ALL, None)
volume = cast(interface, POINTER(IAudioEndpointVolume))
volRange = volume.GetVolumeRange()
minVol = volRange[0]
maxVol = volRange[1]
vol_lock = False  # Variable to track the volume lock status

while True:
    success, img = cap.read()
    img = detector.find_hands(img)
    hand_lms = detector.find_position(img, draw=False)

    if len(hand_lms) != 0:
        x1, y1 = hand_lms[4][1], hand_lms[4][2]
        x2, y2 = hand_lms[8][1], hand_lms[8][2]
        cx, cy = (x1 + x2) // 2, (y1 + y2) // 2

        cv2.circle(img, (x1, y1), 8, (255, 0, 255), cv2.FILLED)
        cv2.circle(img, (x2, y2), 8, (255, 0, 255), cv2.FILLED)
        cv2.line(img, (x1, y1), (x2, y2), (255, 0, 255), 2)
        cv2.circle(img, (cx, cy), 6, (255, 0, 255), cv2.FILLED)

        length = math.hypot(x2 - x1, y2 - y1)

        if length < 30:  # Gesture to potentially unlock volume
control
            cv2.circle(img, (cx, cy), 6, (0, 255, 0), cv2.FILLED)  #
Green button effect
            vol_lock = False
        if not vol_lock:
            vol = np.interp(length, [50, 300], [minVol, maxVol])
            volume.SetMasterVolumeLevel(vol, None)

            if vol >= maxVol * 0.95:  # Consider it as max volume if
it's very close to 100%
                vol_lock = True

    cTime = time.time()
    fps = 1 / (cTime - pTime)
    pTime = cTime
    cv2.putText(img, f'FPS: {int(fps)}', (40, 70),
cv2.FONT_HERSHEY_PLAIN, 2, (255, 0, 0), 2)

    cv2.imshow("Img", img)
    if cv2.waitKey(1) & 0xFF == ord('q'):
        break

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

In my hand-tracking project, I've successfully utilized the MediaPipe framework to create an application that interprets hand gestures as commands to control the system's volume. Through rigorous coding and integration of OpenCV and MediaPipe libraries, I established a real-time tracking system that identifies key points on the hands to determine gestures.

I've structured the project into a modular format for clarity and maintainability, centralizing the hand-tracking functionality into a dedicated Python file. By drawing on the video feed, I enhanced the user interface to visually indicate the thumb and index finger's positions, as well as their midpoint, enhancing the user's understanding of the system's interpretation of their gestures.

Further, I have fine-tuned the volume control by employing the pycaw library, which interfaces with the system's audio settings. I established a hand gesture range and mapped it to the system's volume range. This allowed me to adjust the volume dynamically by changing the distance between two fingertips, creating an intuitive and interactive user experience.

With visuals to accompany the explanation, I've shown how the volume level correlates directly with the hand movements—spreading the fingers apart increases the volume and bringing them together decreases it. Each movement is reflected in the system's volume slider, confirming the precision of my application.

Overall, this project stands as a testament to the potential of gesture control in user interfaces, showcasing the ability to bridge the gap between complex hand-tracking technology and practical applications in a user-friendly manner. My system not only tracks hand movements with high accuracy but also translates them into meaningful control over the device's functions, offering a glimpse into the future of human-computer interaction.