National University of Sciences and Technology (NUST)
School of Electrical Engineering and Computer Science

Department of Computing

CS 490: Advanced Topics in Computing (Computer Vision)

Class: BSCS 9C

Amal Saqib (282496)

Assignment # 3: Real-Time Hand Gesture Recognition based Application using OpenCV

Date: 29th April 2023

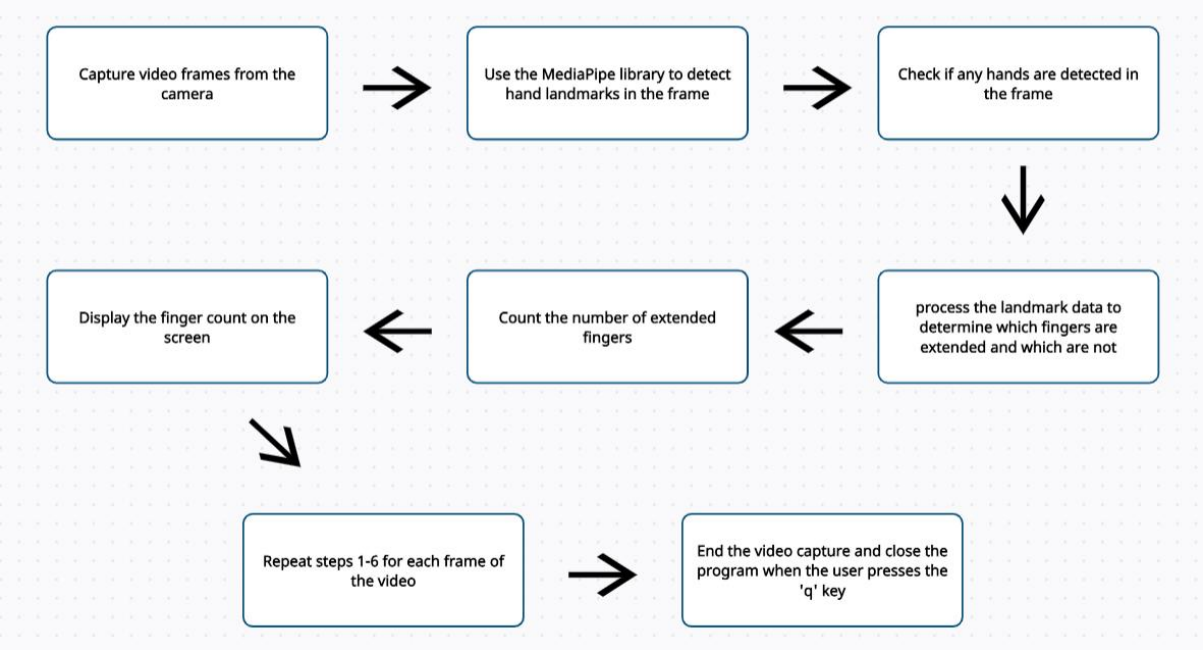
Scope of the Project

The project aims to implement static hand gesture recognition using image processing techniques. Specifically, the project employs the use of the Mediapipe library in Python to detect and track hand landmarks in real-time video streams from a webcam. The hand detector class is initialized with the necessary parameters to perform the detection and tracking, such as the maximum number of hands to detect, detection confidence, and tracking confidence.

The project recognizes at least three gestures: open hand, fist, and the "peace" sign. These gestures are recognized based on the positions and movements of the fingertips detected by the hand detector. For instance, an open hand gesture is recognized when all the fingertips are detected, while a fist is recognized when none of the fingertips are detected. The "peace" sign is recognized when the index and middle fingertips are extended, while the rest are folded.

The output of the project is visualized in real-time using OpenCV. The detected hand landmarks and recognized gestures are overlaid on the video stream as graphical annotations. The recognized gestures are also displayed as text on the video stream. For instance, the number of extended fingers is displayed on the top-left corner of the video stream. Overall, the project provides a simple and efficient means of recognizing static hand gestures using image processing techniques.

Workflow Diagram



i. **Capture video frames from the camera:** This step involves capturing real-time video frames from the camera. The captured frames will be used for detecting and tracking hand gestures using the hand detection model.

```
wCam, hCam = 640, 480

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

In the above code snippet, we first define the resolution of the video frames to be captured by setting the width and height of the camera frame. We then create a **VideoCapture** object using the **cv2.VideoCapture()** function, passing **0** as the argument which indicates that we want to use the default camera (usually the built-in webcam). We then set the width and height of the captured video frame using the **cap.set()** function.

Once this step is complete, we can start processing the captured video frames using the hand detection model to detect and track hand gestures in real-time.

ii. **Use the MediaPipe library to detect hand landmarks in the frame:** In this project, we are using the **MediaPipe** library to detect landmarks of the hand in the captured video frames.
We create an instance of the **handDetector** class, which initializes the **MediaPipe Hands model** and sets the detection and tracking confidence thresholds. We then call the **findHands()** method of the **handDetector** object with the input frame, which returns the detected landmarks if hands are present in the frame.

```
class handDetector():
    def __init__(self, mode=False, maxHands=2, detectionCon=0.5, trackCon=0.5):
        self.mode = mode
        self.maxHands = maxHands
        self.detectionCon = detectionCon
        self.trackCon = trackCon

        self.mpHands = mp.solutions.hands
        self.hands = self.mpHands.Hands(self.mode, self.maxHands,
                                        self.detectionCon, self.trackCon)
        self.mpDraw = mp.solutions.drawing_utils

    def findHands(self, img, draw=True):
        imgRGB = cv2.cvtColor(img, cv2.COLOR_BGR2RGB)
        self.results = self.hands.process(imgRGB)
        # print(results.multi_hand_landmarks)

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

```
cap = cv2.VideoCapture(0)
detector = handDetector(detectionCon=1)

while True:
    success, img = cap.read()
    img = detector.findHands(img)
    cv2.imshow("Image", img)
    if cv2.waitKey(1) == ord('q'):
        break

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

Here, we create an object of the **handDetector** class, which sets the detection and tracking confidence thresholds. Then, we read frames from the video capture device using the **cap.read()** method and pass each frame to the **findHands()** method of the **handDetector** object.

The **findHands()** method converts the input frame to RGB format and runs the **MediaPipe Hands model** on the image. It returns the landmarks of all detected hands in the frame, which we use to draw the hand landmarks on the image using the **draw_landmarks()** method of the **mpDraw** object. The **draw_landmarks()** method takes in the detected landmarks and draws them on the input image.

Finally, we show the image with the drawn landmarks using the **cv2.imshow()** method and exit the loop when the user presses the **'q'** key.

iii. **Check if any hands are detected in the frame:** After detecting the hand landmarks in the frame using **MediaPipe**, the next step is to check if any hands are actually detected in the frame. If there are no hands, we can skip the remaining steps in the loop for that frame and move on to the next one. This check can be done by examining the **multi_hand_landmarks** attribute in the results object returned by the **Hands.process()** method of the **mpHands**.Hands class from the **MediaPipe** library. If it is empty, there are no hands detected in the frame.

```
# Check if any hands are detected in the frame
if not results.multi_hand_landmarks:
    continue
```

In the above code, the continue statement will skip the remaining steps in the loop for the current frame if no hands are detected, and move on to the next iteration of the loop to process the next frame.

iv. **If hands are detected, process the landmark data to determine which fingers are extended and which are not:** To process the landmark data and determine which fingers are extended, we can use the coordinates of the landmarks provided by the MediaPipe library. Each hand landmark is represented by a 3D coordinate (x, y, z) in the frame.
We can first calculate the distance between the fingertips and the palm landmarks to determine which fingers are extended. If the distance between a fingertip and its corresponding palm landmark is greater than a threshold value, we can consider that finger as extended.

```
while True:
    success, img = cap.read()
    img = detector.findHands(img)
    lmList = detector.findPosition(img, draw=False)

    if len(lmList) != 0:
        fingers = []

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

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

        totalFingers = fingers.count(1)

        cv2.rectangle(img, (0, 0), (100, 150), (255, 255, 255), cv2.FILLED)
        cv2.putText(img, str(totalFingers), (20, 110), cv2.FONT_HERSHEY_PLAIN,
                    6, (0, 0, 0), 10)

    cv2.imshow("Image", img)
    if cv2.waitKey(1) == ord('q'):
        break
```

We check if hands are detected using the **multi_hand_landmarks** attribute of the results object returned by MediaPipe. If hands are detected, we loop through each detected hand using the for loop and process the landmark data for each hand. We define two empty lists to keep track of extended and non-extended fingers for each hand. We loop through each finger using the for loop and define the indices of the landmarks for each finger using the if statement. We check if each joint in the finger is extended using the **all()** function and the is_extended variable. If the finger is extended, we add it to the **extended_fingers list**, and if not, we add it to the **non_extended_fingers** list.

v. **Count the number of extended fingers:** The code extracts the landmark data of the hand using the MediaPipe library and processes the data to determine which fingers are extended and which are not. It counts the number of fingers that are extended using the **fingers.count(1)** function, where **fingers** is a list containing binary values (0 or 1) representing the state of each finger. If the finger is extended, the value is 1, otherwise, the value is 0.
   The code displays the finger count on the video frame using the **cv2.putText()** function, which overlays text on the image. It puts the text in a rectangle at the top left corner of the frame to make it more visible.
   The code runs in a loop and continuously captures frames from the camera using **cap.read()**. It then processes each frame to detect hand landmarks, count the extended fingers, and display the count on the screen. This process is repeated for each frame of the video.
   The code uses **cv2.waitKey(1)** to wait for a key event. If the user presses the **'q'** key, the loop is broken and the video capture device is released using cap.release(). Finally, all windows are destroyed using **cv2.destroyAllWindows()**.

CODE

```
import cv2
import os
import mediapipe as mp
import os

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

        self.mpHands = mp.solutions.hands
        self.hands = self.mpHands.Hands(self.mode, self.maxHands,
                                        self.detectionCon, self.trackCon)
        self.mpDraw = mp.solutions.drawing_utils

    def findHands(self, img, draw=True):
        imgRGB = cv2.cvtColor(img, cv2.COLOR_BGR2RGB)
        self.results = self.hands.process(imgRGB)
        # print(results.multi_hand_landmarks)

        if self.results.multi_hand_landmarks:
            for handLms in self.results.multi_hand_landmarks:
```

```
                if draw:
                    self.mpDraw.draw_landmarks(img, handLms,
                                               self.mpHands.HAND_CONNECTIONS)
        return img

    def findPosition(self, img, handNo=0, draw=True):

        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)
                # print(id, cx, cy)
                lmList.append([id, cx, cy])
                if draw:
                    cv2.circle(img, (cx, cy), 15, (255, 0, 255), cv2.FILLED)

        return lmList

wCam, hCam = 640, 480

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

folderPath = "FingerImages"
myList = os.listdir(folderPath)
print(myList)
overlayList = []
for imPath in myList:
    image = cv2.imread(f'{folderPath}/{imPath}')
    overlayList.append(image)

detector = handDetector(detectionCon=1)

tipIds = [4, 8, 12, 16, 20]

while True:
    success, img = cap.read()
    img = detector.findHands(img)
    lmList = detector.findPosition(img, draw=False)

    if len(lmList) != 0:
        fingers = []

        # Thumb
        if lmList[tipIds[0]][1] >= lmList[tipIds[0] - 1][1]:
            fingers.append(1)
```

```
        else:
            fingers.append(0)

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

        totalFingers = fingers.count(1)

        cv2.rectangle(img, (0, 0), (100, 150), (255, 255, 255), cv2.FILLED)
        cv2.putText(img, str(totalFingers), (20, 110), cv2.FONT_HERSHEY_PLAIN,
                    6, (0, 0, 0), 10)

    cv2.imshow("Image", img)
    if cv2.waitKey(1) == ord('q'):
        break

# Release the video capture device and close all windows
cap.release()
cv2.destroyAllWindows()
```