Code for

“Hand Gesture Controller (Virtual Mouse) and

Voice Assistant using OpenCV, ML, Python”

app.py

import eel

import os

from queue import Queue

class ChatBot:

    started = False

    userinputQueue = Queue()

    def isUserInput():

        return not ChatBot.userinputQueue.empty()

    def popUserInput():

        return ChatBot.userinputQueue.get()

    def close\_callback(route, websockets):

        # if not websockets:

        #     print('Bye!')

        exit()

    @eel.expose

    def getUserInput(msg):

        ChatBot.userinputQueue.put(msg)

        print(msg)

    def close():

        ChatBot.started = False

    def addUserMsg(msg):

        eel.addUserMsg(msg)

    def addAppMsg(msg):

        eel.addAppMsg(msg)

    def start():

        path = os.path.dirname(os.path.abspath(\_\_file\_\_))

        eel.init(path + r'\web', allowed\_extensions=['.js', '.html'])

        try:

            eel.start('index.html', mode='chrome',

                                    host='localhost',

                                    port=27005,

                                    block=False,

                                    size=(350, 480),

                                    position=(10,100),

                                    disable\_cache=True,

                                    close\_callback=ChatBot.close\_callback)

            ChatBot.started = True

            while ChatBot.started:

                try:

                    eel.sleep(10.0)

                except:

                    #main thread exited

                    break

        except:

            pass

Gesture\_Controller.py

# Imports

import cv2

import mediapipe as mp

import pyautogui

import math

from enum import IntEnum

from ctypes import cast, POINTER

from comtypes import CLSCTX\_ALL

from pycaw.pycaw import AudioUtilities, IAudioEndpointVolume

from google.protobuf.json\_format import MessageToDict

import screen\_brightness\_control as sbcontrol

pyautogui.FAILSAFE = False

mp\_drawing = mp.solutions.drawing\_utils

mp\_hands = mp.solutions.hands

# Gesture Encodings

class Gest(IntEnum):

    # Binary Encoded

    """

    Enum for mapping all hand gesture to binary number.

    """

    FIST = 0

    PINKY = 1

    RING = 2

    MID = 4

    LAST3 = 7

    INDEX = 8

    FIRST2 = 12

    LAST4 = 15

    THUMB = 16

    PALM = 31

    # Extra Mappings

    V\_GEST = 33

    TWO\_FINGER\_CLOSED = 34

    PINCH\_MAJOR = 35

    PINCH\_MINOR = 36

# Multi-handedness Labels

class HLabel(IntEnum):

    MINOR = 0

    MAJOR = 1

# Convert Mediapipe Landmarks to recognizable Gestures

class HandRecog:

    """

    Convert Mediapipe Landmarks to recognizable Gestures.

    """

    def \_\_init\_\_(self, hand\_label):

        """

        Constructs all the necessary attributes for the HandRecog object.

        Parameters

        ----------

            finger : int

                Represent gesture corresponding to Enum 'Gest',

                stores computed gesture for current frame.

            ori\_gesture : int

                Represent gesture corresponding to Enum 'Gest',

                stores gesture being used.

            prev\_gesture : int

                Represent gesture corresponding to Enum 'Gest',

                stores gesture computed for previous frame.

            frame\_count : int

                total no. of frames since 'ori\_gesture' is updated.

            hand\_result : Object

                Landmarks obtained from mediapipe.

            hand\_label : int

                Represents multi-handedness corresponding to Enum 'HLabel'.

        """

        self.finger = 0

        self.ori\_gesture = Gest.PALM

        self.prev\_gesture = Gest.PALM

        self.frame\_count = 0

        self.hand\_result = None

        self.hand\_label = hand\_label

    def update\_hand\_result(self, hand\_result):

        self.hand\_result = hand\_result

    def get\_signed\_dist(self, point):

        """

        returns signed euclidean distance between 'point'.

        Parameters

        ----------

        point : list contaning two elements of type list/tuple which represents

            landmark point.

        Returns

        -------

        float

        """

        sign = -1

        if self.hand\_result.landmark[point[0]].y < self.hand\_result.landmark[point[1]].y:

            sign = 1

        dist = (self.hand\_result.landmark[point[0]].x - self.hand\_result.landmark[point[1]].x)\*\*2

        dist += (self.hand\_result.landmark[point[0]].y - self.hand\_result.landmark[point[1]].y)\*\*2

        dist = math.sqrt(dist)

        return dist\*sign

    def get\_dist(self, point):

        """

        returns euclidean distance between 'point'.

        Parameters

        ----------

        point : list contaning two elements of type list/tuple which represents

            landmark point.

        Returns

        -------

        float

        """

        dist = (self.hand\_result.landmark[point[0]].x - self.hand\_result.landmark[point[1]].x)\*\*2

        dist += (self.hand\_result.landmark[point[0]].y - self.hand\_result.landmark[point[1]].y)\*\*2

        dist = math.sqrt(dist)

        return dist

    def get\_dz(self,point):

        """

        returns absolute difference on z-axis between 'point'.

        Parameters

        ----------

        point : list contaning two elements of type list/tuple which represents

            landmark point.

        Returns

        -------

        float

        """

        return abs(self.hand\_result.landmark[point[0]].z - self.hand\_result.landmark[point[1]].z)

    # Function to find Gesture Encoding using current finger\_state.

    # Finger\_state: 1 if finger is open, else 0

    def set\_finger\_state(self):

        """

        set 'finger' by computing ratio of distance between finger tip

        , middle knuckle, base knuckle.

        Returns

        -------

        None

        """

        if self.hand\_result == None:

            return

        points = [[8,5,0],[12,9,0],[16,13,0],[20,17,0]]

        self.finger = 0

        self.finger = self.finger | 0 #thumb

        for idx,point in enumerate(points):

            dist = self.get\_signed\_dist(point[:2])

            dist2 = self.get\_signed\_dist(point[1:])

            try:

                ratio = round(dist/dist2,1)

            except:

                ratio = round(dist1/0.01,1)

            self.finger = self.finger << 1

            if ratio > 0.5 :

                self.finger = self.finger | 1

    # Handling Fluctations due to noise

    def get\_gesture(self):

        """

        returns int representing gesture corresponding to Enum 'Gest'.

        sets 'frame\_count', 'ori\_gesture', 'prev\_gesture',

        handles fluctations due to noise.

        Returns

        -------

        int

        """

        if self.hand\_result == None:

            return Gest.PALM

        current\_gesture = Gest.PALM

        if self.finger in [Gest.LAST3,Gest.LAST4] and self.get\_dist([8,4]) < 0.05:

            if self.hand\_label == HLabel.MINOR :

                current\_gesture = Gest.PINCH\_MINOR

            else:

                current\_gesture = Gest.PINCH\_MAJOR

        elif Gest.FIRST2 == self.finger :

            point = [[8,12],[5,9]]

            dist1 = self.get\_dist(point[0])

            dist2 = self.get\_dist(point[1])

            ratio = dist1/dist2

            if ratio > 1.7:

                current\_gesture = Gest.V\_GEST

            else:

                if self.get\_dz([8,12]) < 0.1:

                    current\_gesture =  Gest.TWO\_FINGER\_CLOSED

                else:

                    current\_gesture =  Gest.MID

        else:

            current\_gesture =  self.finger

        if current\_gesture == self.prev\_gesture:

            self.frame\_count += 1

        else:

            self.frame\_count = 0

        self.prev\_gesture = current\_gesture

        if self.frame\_count > 4 :

            self.ori\_gesture = current\_gesture

        return self.ori\_gesture

# Executes commands according to detected gestures

class Controller:

    """

    Executes commands according to detected gestures.

    Attributes

    ----------

    tx\_old : int

        previous mouse location x coordinate

    ty\_old : int

        previous mouse location y coordinate

    flag : bool

        true if V gesture is detected

    grabflag : bool

        true if FIST gesture is detected

    pinchmajorflag : bool

        true if PINCH gesture is detected through MAJOR hand,

        on x-axis 'Controller.changesystembrightness',

        on y-axis 'Controller.changesystemvolume'.

    pinchminorflag : bool

        true if PINCH gesture is detected through MINOR hand,

        on x-axis 'Controller.scrollHorizontal',

        on y-axis 'Controller.scrollVertical'.

    pinchstartxcoord : int

        x coordinate of hand landmark when pinch gesture is started.

    pinchstartycoord : int

        y coordinate of hand landmark when pinch gesture is started.

    pinchdirectionflag : bool

        true if pinch gesture movment is along x-axis,

        otherwise false

    prevpinchlv : int

        stores quantized magnitued of prev pinch gesture displacment, from

        starting position

    pinchlv : int

        stores quantized magnitued of pinch gesture displacment, from

        starting position

    framecount : int

        stores no. of frames since 'pinchlv' is updated.

    prev\_hand : tuple

        stores (x, y) coordinates of hand in previous frame.

    pinch\_threshold : float

        step size for quantization of 'pinchlv'.

    """

    tx\_old = 0

    ty\_old = 0

    trial = True

    flag = False

    grabflag = False

    pinchmajorflag = False

    pinchminorflag = False

    pinchstartxcoord = None

    pinchstartycoord = None

    pinchdirectionflag = None

    prevpinchlv = 0

    pinchlv = 0

    framecount = 0

    prev\_hand = None

    pinch\_threshold = 0.3

    def getpinchylv(hand\_result):

        """returns distance beween starting pinch y coord and current hand position y coord."""

        dist = round((Controller.pinchstartycoord - hand\_result.landmark[8].y)\*10,1)

        return dist

    def getpinchxlv(hand\_result):

        """returns distance beween starting pinch x coord and current hand position x coord."""

        dist = round((hand\_result.landmark[8].x - Controller.pinchstartxcoord)\*10,1)

        return dist

    def changesystembrightness():

        """sets system brightness based on 'Controller.pinchlv'."""

        currentBrightnessLv = sbcontrol.get\_brightness(display=0)/100.0

        currentBrightnessLv += Controller.pinchlv/50.0

        if currentBrightnessLv > 1.0:

            currentBrightnessLv = 1.0

        elif currentBrightnessLv < 0.0:

            currentBrightnessLv = 0.0

        sbcontrol.fade\_brightness(int(100\*currentBrightnessLv) , start = sbcontrol.get\_brightness(display=0))

    def changesystemvolume():

        """sets system volume based on 'Controller.pinchlv'."""

        devices = AudioUtilities.GetSpeakers()

        interface = devices.Activate(IAudioEndpointVolume.\_iid\_, CLSCTX\_ALL, None)

        volume = cast(interface, POINTER(IAudioEndpointVolume))

        currentVolumeLv = volume.GetMasterVolumeLevelScalar()

        currentVolumeLv += Controller.pinchlv/50.0

        if currentVolumeLv > 1.0:

            currentVolumeLv = 1.0

        elif currentVolumeLv < 0.0:

            currentVolumeLv = 0.0

        volume.SetMasterVolumeLevelScalar(currentVolumeLv, None)

    def scrollVertical():

        """scrolls on screen vertically."""

        pyautogui.scroll(120 if Controller.pinchlv>0.0 else -120)

    def scrollHorizontal():

        """scrolls on screen horizontally."""

        pyautogui.keyDown('shift')

        pyautogui.keyDown('ctrl')

        pyautogui.scroll(-120 if Controller.pinchlv>0.0 else 120)

        pyautogui.keyUp('ctrl')

        pyautogui.keyUp('shift')

    # Locate Hand to get Cursor Position

    # Stabilize cursor by Dampening

    def get\_position(hand\_result):

        """

        returns coordinates of current hand position.

        Locates hand to get cursor position also stabilize cursor by

        dampening jerky motion of hand.

        Returns

        -------

        tuple(float, float)

        """

        point = 9

        position = [hand\_result.landmark[point].x ,hand\_result.landmark[point].y]

        sx,sy = pyautogui.size()

        x\_old,y\_old = pyautogui.position()

        x = int(position[0]\*sx)

        y = int(position[1]\*sy)

        if Controller.prev\_hand is None:

            Controller.prev\_hand = x,y

        delta\_x = x - Controller.prev\_hand[0]

        delta\_y = y - Controller.prev\_hand[1]

        distsq = delta\_x\*\*2 + delta\_y\*\*2

        ratio = 1

        Controller.prev\_hand = [x,y]

        if distsq <= 25:

            ratio = 0

        elif distsq <= 900:

            ratio = 0.07 \* (distsq \*\* (1/2))

        else:

            ratio = 2.1

        x , y = x\_old + delta\_x\*ratio , y\_old + delta\_y\*ratio

        return (x,y)

    def pinch\_control\_init(hand\_result):

        """Initializes attributes for pinch gesture."""

        Controller.pinchstartxcoord = hand\_result.landmark[8].x

        Controller.pinchstartycoord = hand\_result.landmark[8].y

        Controller.pinchlv = 0

        Controller.prevpinchlv = 0

        Controller.framecount = 0

    # Hold final position for 5 frames to change status

    def pinch\_control(hand\_result, controlHorizontal, controlVertical):

        """

        calls 'controlHorizontal' or 'controlVertical' based on pinch flags,

        'framecount' and sets 'pinchlv'.

        Parameters

        ----------

        hand\_result : Object

            Landmarks obtained from mediapipe.

        controlHorizontal : callback function assosiated with horizontal

            pinch gesture.

        controlVertical : callback function assosiated with vertical

            pinch gesture.

        Returns

        -------

        None

        """

        if Controller.framecount == 5:

            Controller.framecount = 0

            Controller.pinchlv = Controller.prevpinchlv

            if Controller.pinchdirectionflag == True:

                controlHorizontal() #x

            elif Controller.pinchdirectionflag == False:

                controlVertical() #y

        lvx =  Controller.getpinchxlv(hand\_result)

        lvy =  Controller.getpinchylv(hand\_result)

        if abs(lvy) > abs(lvx) and abs(lvy) > Controller.pinch\_threshold:

            Controller.pinchdirectionflag = False

            if abs(Controller.prevpinchlv - lvy) < Controller.pinch\_threshold:

                Controller.framecount += 1

            else:

                Controller.prevpinchlv = lvy

                Controller.framecount = 0

        elif abs(lvx) > Controller.pinch\_threshold:

            Controller.pinchdirectionflag = True

            if abs(Controller.prevpinchlv - lvx) < Controller.pinch\_threshold:

                Controller.framecount += 1

            else:

                Controller.prevpinchlv = lvx

                Controller.framecount = 0

    def handle\_controls(gesture, hand\_result):

        """Impliments all gesture functionality."""

        x,y = None,None

        if gesture != Gest.PALM :

            x,y = Controller.get\_position(hand\_result)

        # flag reset

        if gesture != Gest.FIST and Controller.grabflag:

            Controller.grabflag = False

            pyautogui.mouseUp(button = "left")

        if gesture != Gest.PINCH\_MAJOR and Controller.pinchmajorflag:

            Controller.pinchmajorflag = False

        if gesture != Gest.PINCH\_MINOR and Controller.pinchminorflag:

            Controller.pinchminorflag = False

        # implementation

        if gesture == Gest.V\_GEST:

            Controller.flag = True

            pyautogui.moveTo(x, y, duration = 0.1)

        elif gesture == Gest.FIST:

            if not Controller.grabflag :

                Controller.grabflag = True

                pyautogui.mouseDown(button = "left")

            pyautogui.moveTo(x, y, duration = 0.1)

        elif gesture == Gest.MID and Controller.flag:

            pyautogui.click()

            Controller.flag = False

        elif gesture == Gest.INDEX and Controller.flag:

            pyautogui.click(button='right')

            Controller.flag = False

        elif gesture == Gest.TWO\_FINGER\_CLOSED and Controller.flag:

            pyautogui.doubleClick()

            Controller.flag = False

        elif gesture == Gest.PINCH\_MINOR:

            if Controller.pinchminorflag == False:

                Controller.pinch\_control\_init(hand\_result)

                Controller.pinchminorflag = True

            Controller.pinch\_control(hand\_result,Controller.scrollHorizontal, Controller.scrollVertical)

        elif gesture == Gest.PINCH\_MAJOR:

            if Controller.pinchmajorflag == False:

                Controller.pinch\_control\_init(hand\_result)

                Controller.pinchmajorflag = True

            Controller.pinch\_control(hand\_result,Controller.changesystembrightness, Controller.changesystemvolume)

'''

----------------------------------------  Main Class  ----------------------------------------

    Entry point of Gesture Controller

'''

class GestureController:

    """

    Handles camera, obtain landmarks from mediapipe, entry point

    for whole program.

    Attributes

    ----------

    gc\_mode : int

        indicates weather gesture controller is running or not,

        1 if running, otherwise 0.

    cap : Object

        object obtained from cv2, for capturing video frame.

    CAM\_HEIGHT : int

        highet in pixels of obtained frame from camera.

    CAM\_WIDTH : int

        width in pixels of obtained frame from camera.

    hr\_major : Object of 'HandRecog'

        object representing major hand.

    hr\_minor : Object of 'HandRecog'

        object representing minor hand.

    dom\_hand : bool

        True if right hand is domaniant hand, otherwise False.

        default True.

    """

    gc\_mode = 0

    cap = None

    CAM\_HEIGHT = None

    CAM\_WIDTH = None

    hr\_major = None # Right Hand by default

    hr\_minor = None # Left hand by default

    dom\_hand = True

    def \_\_init\_\_(self):

        """Initilaizes attributes."""

        GestureController.gc\_mode = 1

        GestureController.cap = cv2.VideoCapture(0)

        GestureController.CAM\_HEIGHT = GestureController.cap.get(cv2.CAP\_PROP\_FRAME\_HEIGHT)

        GestureController.CAM\_WIDTH = GestureController.cap.get(cv2.CAP\_PROP\_FRAME\_WIDTH)

    def classify\_hands(results):

        """

        sets 'hr\_major', 'hr\_minor' based on classification(left, right) of

        hand obtained from mediapipe, uses 'dom\_hand' to decide major and

        minor hand.

        """

        left , right = None,None

        try:

            handedness\_dict = MessageToDict(results.multi\_handedness[0])

            if handedness\_dict['classification'][0]['label'] == 'Right':

                right = results.multi\_hand\_landmarks[0]

            else :

                left = results.multi\_hand\_landmarks[0]

        except:

            pass

        try:

            handedness\_dict = MessageToDict(results.multi\_handedness[1])

            if handedness\_dict['classification'][0]['label'] == 'Right':

                right = results.multi\_hand\_landmarks[1]

            else :

                left = results.multi\_hand\_landmarks[1]

        except:

            pass

        if GestureController.dom\_hand == True:

            GestureController.hr\_major = right

            GestureController.hr\_minor = left

        else :

            GestureController.hr\_major = left

            GestureController.hr\_minor = right

    def start(self):

        """

        Entry point of whole programm, caputres video frame and passes, obtains

        landmark from mediapipe and passes it to 'handmajor' and 'handminor' for

        controlling.

        """

        handmajor = HandRecog(HLabel.MAJOR)

        handminor = HandRecog(HLabel.MINOR)

        with mp\_hands.Hands(max\_num\_hands = 2,min\_detection\_confidence=0.5, min\_tracking\_confidence=0.5) as hands:

            while GestureController.cap.isOpened() and GestureController.gc\_mode:

                success, image = GestureController.cap.read()

                if not success:

                    print("Ignoring empty camera frame.")

                    continue

                image = cv2.cvtColor(cv2.flip(image, 1), cv2.COLOR\_BGR2RGB)

                image.flags.writeable = False

                results = hands.process(image)

                image.flags.writeable = True

                image = cv2.cvtColor(image, cv2.COLOR\_RGB2BGR)

                if results.multi\_hand\_landmarks:

                    GestureController.classify\_hands(results)

                    handmajor.update\_hand\_result(GestureController.hr\_major)

                    handminor.update\_hand\_result(GestureController.hr\_minor)

                    handmajor.set\_finger\_state()

                    handminor.set\_finger\_state()

                    gest\_name = handminor.get\_gesture()

                    if gest\_name == Gest.PINCH\_MINOR:

                        Controller.handle\_controls(gest\_name, handminor.hand\_result)

                    else:

                        gest\_name = handmajor.get\_gesture()

                        Controller.handle\_controls(gest\_name, handmajor.hand\_result)

                    for hand\_landmarks in results.multi\_hand\_landmarks:

                        mp\_drawing.draw\_landmarks(image, hand\_landmarks, mp\_hands.HAND\_CONNECTIONS)

                else:

                    Controller.prev\_hand = None

                cv2.imshow('Gesture Controller', image)

                if cv2.waitKey(5) & 0xFF == 13:

                    break

        GestureController.cap.release()

        cv2.destroyAllWindows()

# uncomment to run directly

#gc1 = GestureController()

#gc1.start()

Gesture\_Controller\_Gloved.py

import numpy as np

import cv2

import cv2.aruco as aruco

import os

import glob

import math

import pyautogui

import time

class Marker:

    def \_\_init\_\_(self, dict\_type = aruco.DICT\_4X4\_50, thresh\_constant = 1):

        self.aruco\_dict = aruco.Dictionary\_get(dict\_type)

        self.parameters = aruco.DetectorParameters\_create()

        self.parameters.adaptiveThreshConstant = thresh\_constant

        self.corners = None # corners of Marker

        self.marker\_x2y = 1 # width:height ratio

        self.mtx, self.dist = Marker.calibrate()

    def calibrate():

        criteria = (cv2.TERM\_CRITERIA\_EPS + cv2.TERM\_CRITERIA\_MAX\_ITER, 30, 0.001)

        objp = np.zeros((6\*7,3), np.float32)

        objp[:,:2] = np.mgrid[0:7,0:6].T.reshape(-1,2)

        objpoints = [] # 3d point in real world space

        imgpoints = [] # 2d points in image plane.

        path = os.path.dirname(os.path.abspath(\_\_file\_\_))

        p1 = path + r'\calib\_images\checkerboard\\*.jpg'

        images = glob.glob(p1)

        for fname in images:

            img = cv2.imread(fname)

            gray = cv2.cvtColor(img,cv2.COLOR\_BGR2GRAY)

            ret, corners = cv2.findChessboardCorners(gray, (7,6),None)

            if ret == True:

                objpoints.append(objp)

                corners2 = cv2.cornerSubPix(gray,corners,(11,11),(-1,-1),criteria)

                imgpoints.append(corners2)

                img = cv2.drawChessboardCorners(img, (7,6), corners2,ret)

        ret, mtx, dist, rvecs, tvecs = cv2.calibrateCamera(objpoints, imgpoints, gray.shape[::-1],None,None)

        #mtx = [[534.34144579,0.0,339.15527836],[0.0,534.68425882,233.84359493],[0.0,0.0,1.0]]

        #dist = [[-2.88320983e-01, 5.41079685e-02, 1.73501622e-03, -2.61333895e-04, 2.04110465e-01]]

        return mtx, dist

    def detect(self, frame):

        gray\_frame = cv2.cvtColor(frame, cv2.COLOR\_BGR2GRAY)

        self.corners, ids, rejectedImgPoints = aruco.detectMarkers(gray\_frame, self.aruco\_dict, parameters = self.parameters)

        if np.all(ids != None):

            rvec, tvec ,\_ = aruco.estimatePoseSingleMarkers(self.corners, 0.05, self.mtx, self.dist)

        else:

            self.corners = None

    def is\_detected(self):

        if self.corners:

            return True

        return False

    def draw\_marker(self, frame):

        aruco.drawDetectedMarkers(frame, self.corners)

def ecu\_dis(p1, p2):

    dist = np.sqrt((p1[0]-p2[0])\*\*2 + (p1[1]-p2[1])\*\*2)

    return dist

def find\_HSV(samples):

    try:

        color = np.uint8([ samples ])

    except:

        color = np.uint8([ [[105,105,50]] ])

    hsv\_color = cv2.cvtColor(color,cv2.COLOR\_RGB2HSV)

    #print( hsv\_color )

    return hsv\_color

def draw\_box(frame, points, color=(0,255,127)):

    if points:

        frame = cv2.line(frame, points[0], points[1], color, thickness=2, lineType=8) #top

        frame = cv2.line(frame, points[1], points[2], color, thickness=2, lineType=8) #right

        frame = cv2.line(frame, points[2], points[3], color, thickness=2, lineType=8) #bottom

        frame = cv2.line(frame, points[3], points[0], color, thickness=2, lineType=8) #left

def in\_cam(val, type\_):

    if type\_ == 'x':

        if val<0:

            return 0

        if val>GestureController.cam\_width:

            return GestureController.cam\_width

    elif type\_ == 'y':

        if val<0:

            return 0

        if val>GestureController.cam\_height:

            return GestureController.cam\_height

    return val

class ROI:

    def \_\_init\_\_(self, roi\_alpha1=1.5, roi\_alpha2=1.5, roi\_beta=2.5, hsv\_alpha = 0.3, hsv\_beta = 0.5, hsv\_lift\_up = 0.3):

        self.roi\_alpha1 = roi\_alpha1

        self.roi\_alpha2 = roi\_alpha2

        self.roi\_beta = roi\_beta

        self.roi\_corners = None

        self.hsv\_alpha = hsv\_alpha

        self.hsv\_beta = hsv\_beta

        self.hsv\_lift\_up = hsv\_lift\_up

        self.hsv\_corners = None

        self.marker\_top = None

        self.glove\_hsv = None

    def findROI(self, frame, marker):

        rec\_coor = marker.corners[0][0]

        c1 = (int(rec\_coor[0][0]),int(rec\_coor[0][1]))

        c2 = (int(rec\_coor[1][0]),int(rec\_coor[1][1]))

        c3 = (int(rec\_coor[2][0]),int(rec\_coor[2][1]))

        c4 = (int(rec\_coor[3][0]),int(rec\_coor[3][1]))

        try:

            marker.marker\_x2y = np.sqrt((c1[0]-c2[0])\*\*2 + (c1[1]-c2[1])\*\*2) / np.sqrt((c3[0]-c2[0])\*\*2 + (c3[1]-c2[1])\*\*2)

        except:

            marker.marker\_x2y = 999.0

        #mid-point of top line of Marker

        cx = (c1[0] + c2[0])/2

        cy = (c1[1] + c2[1])/2

        self.marker\_top = [cx, cy]

        l = np.absolute(ecu\_dis(c1,c4))

        try:

            slope\_12 = (c1[1]-c2[1])/(c1[0]-c2[0])

        except:

            slope\_12 = (c1[1]-c2[1])\*999.0 + 0.1

        try:

            slope\_14 = -1 / slope\_12

        except:

            slope\_14 = -999.0

        if slope\_14 < 0:

            sign = 1

        else:

            sign = -1

        bot\_rx = int(cx + self.roi\_alpha2 \* l \* np.sqrt(1/(1+slope\_12\*\*2)))

        bot\_ry = int(cy + self.roi\_alpha2 \* slope\_12 \* l \* np.sqrt(1/(1+slope\_12\*\*2)))

        bot\_lx = int(cx - self.roi\_alpha1 \* l \* np.sqrt(1/(1+slope\_12\*\*2)))

        bot\_ly = int(cy - self.roi\_alpha1 \* slope\_12 \* l \* np.sqrt(1/(1+slope\_12\*\*2)))

        top\_lx = int(bot\_lx + sign \* self.roi\_beta \* l \* np.sqrt(1/(1+slope\_14\*\*2)))

        top\_ly = int(bot\_ly + sign \* self.roi\_beta \* slope\_14 \* l \* np.sqrt(1/(1+slope\_14\*\*2)))

        top\_rx = int(bot\_rx + sign \* self.roi\_beta \* l \* np.sqrt(1/(1+slope\_14\*\*2)))

        top\_ry = int(bot\_ry + sign \* self.roi\_beta \* slope\_14 \* l \* np.sqrt(1/(1+slope\_14\*\*2)))

        bot\_lx = in\_cam(bot\_lx, 'x')

        bot\_ly = in\_cam(bot\_ly, 'y')

        bot\_rx = in\_cam(bot\_rx, 'x')

        bot\_ry = in\_cam(bot\_ry, 'y')

        top\_lx = in\_cam(top\_lx, 'x')

        top\_ly = in\_cam(top\_ly, 'y')

        top\_rx = in\_cam(top\_rx, 'x')

        top\_ry = in\_cam(top\_ry, 'y')

        self.roi\_corners = [(bot\_lx,bot\_ly), (bot\_rx,bot\_ry), (top\_rx,top\_ry), (top\_lx,top\_ly)]

    def find\_glove\_hsv(self, frame, marker):

        rec\_coor = marker.corners[0][0]

        c1 = (int(rec\_coor[0][0]),int(rec\_coor[0][1]))

        c2 = (int(rec\_coor[1][0]),int(rec\_coor[1][1]))

        c3 = (int(rec\_coor[2][0]),int(rec\_coor[2][1]))

        c4 = (int(rec\_coor[3][0]),int(rec\_coor[3][1]))

        l = np.absolute(ecu\_dis(c1,c4))

        try:

            slope\_12 = (c1[1]-c2[1])/(c1[0]-c2[0])

        except:

            slope\_12 = (c1[1]-c2[1])\*999.0 + 0.1

        try:

            slope\_14 = -1 / slope\_12

        except:

            slope\_14 = -999.0

        if slope\_14 < 0:

            sign = 1

        else:

            sign = -1

        bot\_rx = int(self.marker\_top[0] + self.hsv\_alpha \* l \* np.sqrt(1/(1+slope\_12\*\*2)))

        bot\_ry = int(self.marker\_top[1] - self.hsv\_lift\_up\*l + self.hsv\_alpha \* slope\_12 \* l \* np.sqrt(1/(1+slope\_12\*\*2)))

        bot\_lx = int(self.marker\_top[0] - self.hsv\_alpha \* l \* np.sqrt(1/(1+slope\_12\*\*2)))

        bot\_ly = int(self.marker\_top[1] - self.hsv\_lift\_up\*l - self.hsv\_alpha \* slope\_12 \* l \* np.sqrt(1/(1+slope\_12\*\*2)))

        top\_lx = int(bot\_lx + sign \* self.hsv\_beta \* l \* np.sqrt(1/(1+slope\_14\*\*2)))

        top\_ly = int(bot\_ly + sign \* self.hsv\_beta \* slope\_14 \* l \* np.sqrt(1/(1+slope\_14\*\*2)))

        top\_rx = int(bot\_rx + sign \* self.hsv\_beta \* l \* np.sqrt(1/(1+slope\_14\*\*2)))

        top\_ry = int(bot\_ry + sign \* self.hsv\_beta \* slope\_14 \* l \* np.sqrt(1/(1+slope\_14\*\*2)))

        region = frame[top\_ry:bot\_ry , top\_lx:bot\_rx]

        b, g, r = np.mean(region, axis=(0, 1))

        self.hsv\_glove = find\_HSV([[r,g,b]])

        self.hsv\_corners =  [(bot\_lx,bot\_ly), (bot\_rx,bot\_ry), (top\_rx,top\_ry), (top\_lx,top\_ly)]

    def cropROI(self, frame):

        pts = np.array(self.roi\_corners)

        ## (1) Crop the bounding rect

        rect = cv2.boundingRect(pts)

        x,y,w,h = rect

        croped = frame[y:y+h, x:x+w].copy()

        ## (2) make mask

        pts = pts - pts.min(axis=0)

        mask = np.zeros(croped.shape[:2], np.uint8)

        cv2.drawContours(mask, [pts], -1, (255, 255, 255), -1, cv2.LINE\_AA)

        ## (3) do bit-op

        dst = cv2.bitwise\_and(croped, croped, mask=mask)

        ## (4) add the white background

        bg = np.ones\_like(croped, np.uint8)\*255

        cv2.bitwise\_not(bg,bg, mask=mask)

        kernelOpen = np.ones((3,3),np.uint8)

        kernelClose = np.ones((5,5),np.uint8)

        hsv = cv2.cvtColor(dst, cv2.COLOR\_BGR2HSV)

        lower\_range = np.array([self.hsv\_glove[0][0][0]//1-5,50,50])

        upper\_range = np.array([self.hsv\_glove[0][0][0]//1+5,255,255])

        mask = cv2.inRange(hsv, lower\_range, upper\_range)

        #mask = cv2.dilate(mask,kernelOpen,iterations = 1)

        Opening =cv2.morphologyEx(mask,cv2.MORPH\_OPEN,kernelOpen)

        Closing =cv2.morphologyEx(Opening,cv2.MORPH\_CLOSE,kernelClose)

        FinalMask = Closing

        return FinalMask

class Glove:

    def \_\_init\_\_(self):

        self.fingers = 0

        self.arearatio = 0

        self.gesture = 0

    def find\_fingers(self, FinalMask):

        conts,h=cv2.findContours(FinalMask,cv2.RETR\_EXTERNAL,cv2.CHAIN\_APPROX\_NONE)

        hull = [cv2.convexHull(c) for c in conts]

        try:

            cnt = max(conts, key = lambda x: cv2.contourArea(x))

            #approx the contour a little

            epsilon = 0.0005\*cv2.arcLength(cnt,True)

            approx= cv2.approxPolyDP(cnt,epsilon,True)

            #make convex hull around hand

            hull = cv2.convexHull(cnt)

            #define area of hull and area of hand

            areahull = cv2.contourArea(hull)

            areacnt = cv2.contourArea(cnt)

            #find the percentage of area not covered by hand in convex hull

            self.arearatio=((areahull-areacnt)/areacnt)\*100

            #find the defects in convex hull with respect to hand

            hull = cv2.convexHull(approx, returnPoints=False)

            defects = cv2.convexityDefects(approx, hull)

        except:

            print("No Contours found in FinalMask")

        # l = no. of defects

        l=0

        try:

            #code for finding no. of defects due to fingers

            for i in range(defects.shape[0]):

                s,e,f,d = defects[i,0]

                start = tuple(approx[s][0])

                end = tuple(approx[e][0])

                far = tuple(approx[f][0])

                # find length of all sides of triangle

                a = math.sqrt((end[0] - start[0])\*\*2 + (end[1] - start[1])\*\*2)

                b = math.sqrt((far[0] - start[0])\*\*2 + (far[1] - start[1])\*\*2)

                c = math.sqrt((end[0] - far[0])\*\*2 + (end[1] - far[1])\*\*2)

                s = (a+b+c)/2

                ar = math.sqrt(s\*(s-a)\*(s-b)\*(s-c))

                #distance between point and convex hull

                d=(2\*ar)/a

                # apply cosine rule here

                angle = math.acos((b\*\*2 + c\*\*2 - a\*\*2)/(2\*b\*c)) \* 57

                # ignore angles > 90 and ignore points very close to convex hull(they generally come due to noise)

                if angle <= 90 and d>30:

                    l += 1

                    #cv2.circle(frame, far, 3, [255,255,255], -1)

                #draw lines around hand

                cv2.line(FinalMask,start, end, [255,255,255], 2)

            l+=1

        except:

            l = 0

            print("No Defects found in mask")

        self.fingers = l

    def find\_gesture(self, frame):

        font = cv2.FONT\_HERSHEY\_SIMPLEX

        self.gesture = 0

        if self.fingers==1:

            #cv2.putText(frame, str(int(arearatio)), (10,50), font, 2, (0,0,255), 3, cv2.LINE\_AA)

            if self.arearatio<15:

                cv2.putText(frame,'0',(0,50), font, 2, (0,0,255), 3, cv2.LINE\_AA)

                self.gesture = 0

            elif self.arearatio<25:

                cv2.putText(frame,'2 fingers',(0,50), font, 2, (0,0,255), 3, cv2.LINE\_AA)

                self.gesture = 2

            else:

                cv2.putText(frame,'1 finger',(0,50), font, 2, (0,0,255), 3, cv2.LINE\_AA)

                self.gesture = 1

        elif self.fingers==2:

            cv2.putText(frame,'2',(0,50), font, 2, (0,0,255), 3, cv2.LINE\_AA)

            self.gesture = 3

        '''

        elif self.fingers==3:

            #cv2.putText(frame,'3',(0,50), font, 2, (0,0,255), 3, cv2.LINE\_AA)

        elif self.fingers==4:

            #cv2.putText(frame,'4',(0,50), font, 2, (0,0,255), 3, cv2.LINE\_AA)

        elif self.fingers==5:

            #cv2.putText(frame,'5',(0,50), font, 2, (0,0,255), 3, cv2.LINE\_AA)

        else :

           # cv2.putText(frame,'reposition',(10,50), font, 2, (0,0,255), 3, cv2.LINE\_AA)

        '''

class Tracker:

    def \_\_init\_\_(self):

        self.tracker\_started = False

        self.tracker = None

        self.start\_time = 0.0

        self.now\_time = 0.0

        self.tracker\_bbox = None

    def corners\_to\_tracker(self, corners):

        csrt\_minX = int( min( [corners[0][0][0][0], corners[0][0][1][0], corners[0][0][2][0], corners[0][0][3][0]] ))

        csrt\_maxX = int( max( [corners[0][0][0][0], corners[0][0][1][0], corners[0][0][2][0], corners[0][0][3][0]] ))

        csrt\_minY = int( min( [corners[0][0][0][1], corners[0][0][1][1], corners[0][0][2][1], corners[0][0][3][1]] ))

        csrt\_maxY = int( max( [corners[0][0][0][1], corners[0][0][1][1], corners[0][0][2][1], corners[0][0][3][1]] ))

        self.tracker\_bbox = [csrt\_minX, csrt\_minY, csrt\_maxX-csrt\_minX, csrt\_maxY-csrt\_minY]

    def tracker\_to\_corner(self, final\_bbox):

        if self.tracker\_bbox == None:

            return None

        final\_bbox = [[[1,2],[3,4],[5,6],[7,8]]]

        final\_bbox[0][0] = [self.tracker\_bbox[0],self.tracker\_bbox[1]]

        final\_bbox[0][1] = [self.tracker\_bbox[0]+ self.tracker\_bbox[2],self.tracker\_bbox[1]]

        final\_bbox[0][2] = [self.tracker\_bbox[0]+ self.tracker\_bbox[2],self.tracker\_bbox[1] + self.tracker\_bbox[3]]

        final\_bbox[0][3] = [self.tracker\_bbox[0],self.tracker\_bbox[1] +self.tracker\_bbox[3]]

        return [np.array(final\_bbox, dtype = 'f')]

    def CSRT\_tracker(self, frame):

        if self.tracker\_bbox == None and self.tracker\_started == False:

            return

        if self.tracker\_started == False:

            if self.tracker == None:

                self.tracker = cv2.TrackerCSRT\_create()

        if self.tracker\_bbox != None:

            try:

                self.start\_time = time.time()

                ok = self.tracker.init(frame, self.tracker\_bbox)

                self.tracker\_started = True

            except:

                print("tracker.init failed")

        try:

            ok, self.tracker\_bbox = self.tracker.update(frame)

        except:

            ok = None

            print("tracker.update failed")

        self.now\_time = time.time()

        if self.now\_time-self.start\_time >= 2.0 :

            #cv2.putText(frame, "Please posture your hand correctly", (10,50), cv2.FONT\_HERSHEY\_SIMPLEX, 1,(0,0,255),1)

            cv2.putText(frame,'Posture your hand correctly',(10,10), cv2.FONT\_HERSHEY\_SIMPLEX, 0.75, (0,0,255), 1, cv2.LINE\_AA)

            #print("tracking timeout")

            self.tracker\_started = False

            self.tracker\_bbox = None

            return

        if ok:

            # Tracking success

            p1 = (int(self.tracker\_bbox[0]), int(self.tracker\_bbox[1]))

            p2 = (int(self.tracker\_bbox[0] + self.tracker\_bbox[2]), int(self.tracker\_bbox[1] + self.tracker\_bbox[3]))

            cv2.rectangle(frame, p1, p2, (80, 255, 255), 2, 1)

        else :

            # Tracking failure

            self.tracker\_started = False

            cv2.putText(frame, "Tracking failure detected", (100,80), cv2.FONT\_HERSHEY\_SIMPLEX, 0.75,(0,0,255),2)

            print("Tracking failure detected")

            #reintiallize code to tackle tracking failure

class Mouse:

    def \_\_init\_\_(self):

        self.tx\_old = 0

        self.ty\_old = 0

        self.trial = True

        self.flag = 0

    def move\_mouse(self,frame,position,gesture):

        (sx,sy)=pyautogui.size()

        (camx,camy) = (frame.shape[:2][0],frame.shape[:2][1])

        (mx\_old,my\_old) = pyautogui.position()

        Damping = 2 # Hyperparameter we will have to adjust

        tx = position[0]

        ty = position[1]

        if self.trial:

            self.trial, self.tx\_old, self.ty\_old = False, tx, ty

        delta\_tx = tx - self.tx\_old

        delta\_ty = ty - self.ty\_old

        self.tx\_old,self.ty\_old = tx,ty

        if (gesture == 3):

            self.flag = 0

            mx = mx\_old + (delta\_tx\*sx) // (camx\*Damping)

            my = my\_old + (delta\_ty\*sy) // (camy\*Damping)

            pyautogui.moveTo(mx,my, duration = 0.1)

        elif(gesture == 0):

            if self.flag == 0:

                pyautogui.doubleClick()

                self.flag = 1

        elif(gesture == 1):

            print('1 Finger Open')

class GestureController:

    gc\_mode = 0

    pyautogui.FAILSAFE = False

    f\_start\_time = 0

    f\_now\_time = 0

    cam\_width  = 0

    cam\_height = 0

    aru\_marker = Marker()

    hand\_roi = ROI(2.5, 2.5, 6, 0.45, 0.6, 0.4)

    glove = Glove()

    csrt\_track = Tracker()

    mouse = Mouse()

    def \_\_init\_\_(self):

        GestureController.cap = cv2.VideoCapture(0)

        if GestureController.cap.isOpened():

            GestureController.cam\_width  = int( GestureController.cap.get(cv2.CAP\_PROP\_FRAME\_WIDTH) )

            GestureController.cam\_height = int( GestureController.cap.get(cv2.CAP\_PROP\_FRAME\_HEIGHT) )

        else:

            print("CANNOT OPEN CAMERA")

        GestureController.gc\_mode = 1

        GestureController.f\_start\_time = time.time()

        GestureController.f\_now\_time = time.time()

    def start(self):

        while (True):

            #mode checking

            if not GestureController.gc\_mode:

                print('Exiting Gesture Controller')

                break

            #fps control

            fps = 30.0

            GestureController.f\_start\_time = time.time()

            while (GestureController.f\_now\_time-GestureController.f\_start\_time <= 1.0/fps):

                GestureController.f\_now\_time = time.time()

            #read camera

            ret, frame = GestureController.cap.read()

            frame = cv2.flip(frame, 1)

            #detect Marker, find ROI, find glove HSV, get FinalMask on glove

            GestureController.aru\_marker.detect(frame)

            if GestureController.aru\_marker.is\_detected():

                GestureController.csrt\_track.corners\_to\_tracker(GestureController.aru\_marker.corners)

                GestureController.csrt\_track.CSRT\_tracker(frame)

            else:

                GestureController.csrt\_track.tracker\_bbox = None

                GestureController.csrt\_track.CSRT\_tracker(frame)

                GestureController.aru\_marker.corners = GestureController.csrt\_track.tracker\_to\_corner(GestureController.aru\_marker.corners)

            if GestureController.aru\_marker.is\_detected():

                GestureController.hand\_roi.findROI(frame, GestureController.aru\_marker)

                GestureController.hand\_roi.find\_glove\_hsv(frame, GestureController.aru\_marker)

                FinalMask = GestureController.hand\_roi.cropROI(frame)

                GestureController.glove.find\_fingers(FinalMask)

                GestureController.glove.find\_gesture(frame)

                GestureController.mouse.move\_mouse(frame,GestureController.hand\_roi.marker\_top,GestureController.glove.gesture)

            #draw call

            if GestureController.aru\_marker.is\_detected():

                GestureController.aru\_marker.draw\_marker(frame)

                draw\_box(frame, GestureController.hand\_roi.roi\_corners, (255,0,0))

                draw\_box(frame, GestureController.hand\_roi.hsv\_corners, (0,0,250))

                cv2.imshow('FinalMask',FinalMask)

            #display frame

            cv2.imshow('frame',frame)

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

                break

        # When everything done, release the capture

        GestureController.cap.release()

        cv2.destroyAllWindows()

Proton.py

import pyttsx3

import speech\_recognition as sr

from datetime import date

import time

import webbrowser

import datetime

from pynput.keyboard import Key, Controller

import pyautogui

import sys

import os

from os import listdir

from os.path import isfile, join

import smtplib

import wikipedia

import Gesture\_Controller

#import Gesture\_Controller\_Gloved as Gesture\_Controller

import app

from threading import Thread

# -------------Object Initialization---------------

today = date.today()

r = sr.Recognizer()

keyboard = Controller()

engine = pyttsx3.init('sapi5')

engine = pyttsx3.init()

voices = engine.getProperty('voices')

engine.setProperty('voice', voices[0].id)

# ----------------Variables------------------------

file\_exp\_status = False

files =[]

path = ''

is\_awake = True  #Bot status

# ------------------Functions----------------------

def reply(audio):

    app.ChatBot.addAppMsg(audio)

    print(audio)

    engine.say(audio)

    engine.runAndWait()

def wish():

    hour = int(datetime.datetime.now().hour)

    if hour>=0 and hour<12:

        reply("Good Morning!")

    elif hour>=12 and hour<18:

        reply("Good Afternoon!")

    else:

        reply("Good Evening!")

    reply("I am Proton, how may I help you?")

# Set Microphone parameters

with sr.Microphone() as source:

        r.energy\_threshold = 500

        r.dynamic\_energy\_threshold = False

# Audio to String

def record\_audio():

    with sr.Microphone() as source:

        r.pause\_threshold = 0.8

        voice\_data = ''

        audio = r.listen(source, phrase\_time\_limit=5)

        try:

            voice\_data = r.recognize\_google(audio)

        except sr.RequestError:

            reply('Sorry my Service is down. Plz check your Internet connection')

        except sr.UnknownValueError:

            print('cant recognize')

            pass

        return voice\_data.lower()

# Executes Commands (input: string)

def respond(voice\_data):

    global file\_exp\_status, files, is\_awake, path

    print(voice\_data)

    voice\_data.replace('proton','')

    app.eel.addUserMsg(voice\_data)

    if is\_awake==False:

        if 'wake up' in voice\_data:

            is\_awake = True

            wish()

    # STATIC CONTROLS

    elif 'hello' in voice\_data:

        wish()

    elif 'what is your name' in voice\_data:

        reply('My name is Proton!')

    elif 'date' in voice\_data:

        reply(today.strftime("%B %d, %Y"))

    elif 'time' in voice\_data:

        reply(str(datetime.datetime.now()).split(" ")[1].split('.')[0])

    elif 'search' in voice\_data:

        reply('Searching for ' + voice\_data.split('search')[1])

        url = 'https://google.com/search?q=' + voice\_data.split('search')[1]

        try:

            webbrowser.get().open(url)

            reply('This is what I found Sir')

        except:

            reply('Please check your Internet')

    elif 'location' in voice\_data:

        reply('Which place are you looking for ?')

        temp\_audio = record\_audio()

        app.eel.addUserMsg(temp\_audio)

        reply('Locating...')

        url = 'https://google.nl/maps/place/' + temp\_audio + '/&amp;'

        try:

            webbrowser.get().open(url)

            reply('This is what I found Sir')

        except:

            reply('Please check your Internet')

    elif ('bye' in voice\_data) or ('by' in voice\_data):

        reply("Good bye Sir! Have a nice day.")

        is\_awake = False

    elif ('exit' in voice\_data) or ('terminate' in voice\_data):

        if Gesture\_Controller.GestureController.gc\_mode:

            Gesture\_Controller.GestureController.gc\_mode = 0

        app.ChatBot.close()

        #sys.exit() always raises SystemExit, Handle it in main loop

        sys.exit()

    # DYNAMIC CONTROLS

    elif 'launch gesture recognition' in voice\_data:

        if Gesture\_Controller.GestureController.gc\_mode:

            reply('Gesture recognition is already active')

        else:

            gc = Gesture\_Controller.GestureController()

            t = Thread(target = gc.start)

            t.start()

            reply('Launched Successfully')

    elif ('stop gesture recognition' in voice\_data) or ('top gesture recognition' in voice\_data):

        if Gesture\_Controller.GestureController.gc\_mode:

            Gesture\_Controller.GestureController.gc\_mode = 0

            reply('Gesture recognition stopped')

        else:

            reply('Gesture recognition is already inactive')

    elif 'copy' in voice\_data:

        with keyboard.pressed(Key.ctrl):

            keyboard.press('c')

            keyboard.release('c')

        reply('Copied')

    elif 'page' in voice\_data or 'pest'  in voice\_data or 'paste' in voice\_data:

        with keyboard.pressed(Key.ctrl):

            keyboard.press('v')

            keyboard.release('v')

        reply('Pasted')

    # File Navigation (Default Folder set to C://)

    elif 'list' in voice\_data:

        counter = 0

        path = 'C://'

        files = listdir(path)

        filestr = ""

        for f in files:

            counter+=1

            print(str(counter) + ':  ' + f)

            filestr += str(counter) + ':  ' + f + '<br>'

        file\_exp\_status = True

        reply('These are the files in your root directory')

        app.ChatBot.addAppMsg(filestr)

    elif file\_exp\_status == True:

        counter = 0

        if 'open' in voice\_data:

            if isfile(join(path,files[int(voice\_data.split(' ')[-1])-1])):

                os.startfile(path + files[int(voice\_data.split(' ')[-1])-1])

                file\_exp\_status = False

            else:

                try:

                    path = path + files[int(voice\_data.split(' ')[-1])-1] + '//'

                    files = listdir(path)

                    filestr = ""

                    for f in files:

                        counter+=1

                        filestr += str(counter) + ':  ' + f + '<br>'

                        print(str(counter) + ':  ' + f)

                    reply('Opened Successfully')

                    app.ChatBot.addAppMsg(filestr)

                except:

                    reply('You do not have permission to access this folder')

        if 'back' in voice\_data:

            filestr = ""

            if path == 'C://':

                reply('Sorry, this is the root directory')

            else:

                a = path.split('//')[:-2]

                path = '//'.join(a)

                path += '//'

                files = listdir(path)

                for f in files:

                    counter+=1

                    filestr += str(counter) + ':  ' + f + '<br>'

                    print(str(counter) + ':  ' + f)

                reply('ok')

                app.ChatBot.addAppMsg(filestr)

    else:

        reply('I am not functioned to do this !')

# ------------------Driver Code--------------------

t1 = Thread(target = app.ChatBot.start)

t1.start()

# Lock main thread until Chatbot has started

while not app.ChatBot.started:

    time.sleep(0.5)

wish()

voice\_data = None

while True:

    if app.ChatBot.isUserInput():

        #take input from GUI

        voice\_data = app.ChatBot.popUserInput()

    else:

        #take input from Voice

        voice\_data = record\_audio()

    #process voice\_data

    if 'proton' in voice\_data:

        try:

            #Handle sys.exit()

            respond(voice\_data)

        except SystemExit:

            reply("Exit Successfull")

            break

        except:

            #some other exception got raised

            print("EXCEPTION raised while closing.")

            break

tempCodeRunnerFile.py

                        gest\_name = handmajor.get\_gesture()

                        Controller.handle\_controls(gest\_name, handmajor.hand\_result)

                    for hand\_landmarks in results.multi\_hand\_landmarks:

                        mp\_drawing.draw\_landmarks(image, hand\_landmarks, mp\_hands.HAND\_CONNECTIONS)

                else:

                    Controller.prev\_hand = None

                cv2.imshow('Gesture Controller', imag

Web

Index.html

<!DOCTYPE html>

<html>

<head>

    <title>Proton</title>

    <script type="text/javascript" src="/eel.js"></script>

    <link rel="stylesheet" type="text/css" href="css/jquery.convform.css">

</head>

<body>

    <!-- header -->

    <header>

        <div class = "header-wrapper">

          <p class="awesome">PROTON Welcomes you!</p>

            <div class = "logo">

                <img src= "images/icon.png"  width="35" height="33">

            </div>

        </div>

    </header>

    <!-- ChatBot -->

    <div class="chat\_icon">

        <i class="fa fa-comments" aria-hidden="true"></i>

    </div>

    <div class="chat\_box">

            <div class="conv-form-wrapper" style="height:auto;">

                <div id="messages" style="overflow-y:auto;height:400px;">

                    </br></br>

                </div>

            </div>

            <div id="convForm" class="convFormDynamic" style="height:50px;position:absolute;left:10px;bottom:0px;margin:5px;">

                <div class="options dragscroll"></div>

                <input type="text" id="userInput" placeholder="Type Here" class="userInputDynamic" style="margin: 7px 7.575px 7px 10px; width: 219px; height: 24px;">

                <button id="userInputButton" type="button" class="submit" style="margin-top:9px">▶</button>

                <span class="clear"></span>

            </div>

        </div>

</body>

    <script type="text/javascript" src="js/main.js"></script>

</html>

jquery.convform.css

\* {

    margin: 0px;

    padding: 0px;

    font-family: sans-serif;

 }

 div.convFormDynamic {

     width: calc(100% - 33px);

     margin: 10px auto 15px;

     padding: 0 !important;

     position: relative;

     /\* box-shadow: 0 0 5px 5px rgba(222, 222, 222, 0.4); \*/

 }

 div.convFormDynamic input.userInputDynamic {

     border: none;

     padding: 6px 6px;

     outline: none;

     font-size: 0.905rem;

     float: left;

     width: 72%;

     height: 25%;

     line-height: 1.3em;

     min-height: 1.7em;

     max-height: 10rem;

     display: block;

     max-width: 72%;

     margin-right: 2.5%;

 }

 div.conv-form-wrapper:after {

     content: '';

     display: block;

     clear: both;

     width: 100%;

     height: 1px;

 }

 div.conv-form-wrapper div#messages {

     max-height: 71vh;

     padding-left: 10px;

     padding-right: 10px;

     height: auto !important;

     overflow-y: scroll;

     scrollbar-width: none;

     -ms-overflow-style: none; /\* IE 11 \*/

 }

 div.conv-form-wrapper \* {

     scrollbar-width: none;

     -ms-overflow-style: none;

 }

 div.conv-form-wrapper div#messages:after {

     content: '';

     display: table;

     clear: both;

 }

 div.conv-form-wrapper {

     position: relative;

 }

 div.conv-form-wrapper div.wrapper-messages {

     position: relative;

     height: 600px;

     overflow-y: scroll;

     transition: margin 0.1s;

 }

 div.conv-form-wrapper:before {

     content: '';

     position: absolute;

     width: 100%;

     display: block;

     height: 10px;

     top: 0;

     left: 0;

     z-index: 2;

     background: linear-gradient(#000000, transparent);

 }

 @media (max-width: 767px) {

     div.conv-form-wrapper div.wrapper-messages, div.conv-form-wrapper div#messages {

         max-height: 71vh;

     }

 }

 div.conv-form-wrapper div.wrapper-messages::-webkit-scrollbar, div#feed ul::-webkit-scrollbar, div.conv-form-wrapper div.options::-webkit-scrollbar {

     width: 0px;

     height: 0px;

     /\* remove scrollbar space \*/

     background: transparent;

     /\* optional: just make scrollbar invisible \*/

 }

 input.userInputDynamic.error {

     color: #ac0000 !important;

 }

 input.userInputDynamic {

     border-radius: 3px;

     margin: 7px 10px;

 }

 div.conv-form-wrapper div#messages {

     position: relative;

     bottom: 0;

     margin-left: -5px;

     height: auto !important;

     width: 97%;

     padding-bottom: 20px;

 }

 div.conv-form-wrapper div.options {

     word-wrap: normal;

     white-space: nowrap;

     overflow-x: scroll;

     position: absolute;

     bottom: 100%;

     width: 100%;

     transform: translateY(-5px);

 }

 div.conv-form-wrapper div.message:after {

     content: '';

     display: table;

     clear: both;

 }

 div.conv-form-wrapper  div.message.ready.rtol {

     animation: slideRtoLIn 0.5s ease;

     transform-origin: 0 0 0;

 }

 div.conv-form-wrapper div.message.ready.ltor {

     animation: slideLtoRIn 0.5s ease;

     transform-origin: 0 0 0;

 }

 div.conv-form-wrapper  div#messages div.message {

     border-radius: 20px;

     padding: 12px 22px;

     font-size: 0.905rem;

     color: #333;

     display: inline-block;

     padding: 10px 15px 8px;

     border-radius: 20px;

     margin-bottom: 5px;

     float: right;

     clear: both;

     max-width: 65%;

     word-wrap: break-word;

 }

 div.conv-form-wrapper  div#messages div.message.to {

     background: linear-gradient(to right, #388eff 29%, #70aeff 81%);

     color: #fff;

     float: left;

     border-top-left-radius: 0;

 }

 div.conv-form-wrapper div#messages div.message.from {

     background: linear-gradient(to left, #38ff84 0%, #98fbbe 100%);

     color: #000000;

     border-top-right-radius: 0;

 }

 .message.to+.message.from, .message.from+.message.to {

     margin-top: 15px;

 }

 @keyframes slideRtoLIn {

     0% {

         margin-right: -50px;

     }

     100% {

         margin-right: 0px;

     }

 }

 @keyframes slideLtoRIn {

     0% {

         margin-left: -50px;

     }

     100% {

         margin-left: 0;

     }

 }

 div.convFormDynamic button.submit {

     padding: 3px;

     border: none;

     float: left;

     margin: 5px;

     color: #06c5a6;

     cursor: pointer;

     border-radius: 8px;

     font-size: 1.1rem;

     width: 36px;

     height: 35px;

     margin-top: 8px;

     background: #fff;

     outline: none !important;

 }

 div.convFormDynamic button.submit:hover {

     background: #06b79a;

     color: #fff;

 }

 button.submit.glow {

     box-shadow: 0 0 10px 5px rgba(6, 197, 166, 0.4);

 }

 .no-border {

     border: none !important;

 }

 .dragscroll {

     cursor: grab;

 }

 div.conv-form-wrapper div#messages::-webkit-scrollbar, div#feed ul::-webkit-scrollbar {

     width: 0px;

     /\* remove scrollbar space \*/

     background: transparent;

     /\* optional: just make scrollbar invisible \*/

 }

 span.clear {

     display: block;

     clear: both;

 }

 .header-wrapper {

   background-color: #000000;

   color: white;

   width: auto;

   height: 20px;

   top: 0px;

   left: 0px;

   padding-left: 10px;

   padding-top: 5px;

   padding-bottom: 25px;

   padding-right: 20px;

   text-align: right;

 }

 .logo{

     position: fixed;

     top: 0px;

     left: 5%;

     padding: 10px;

     padding-bottom: 20px;

 }

 body {

     overflow: hidden;

     background: #1a1a2e;

 }

 .awesome {

   font-family: futura;

   width:100%;

   margin-top: 12px;

   margin-left: 12px;

   text-align: center;

   color:#313131;

   font-size:16px;

   font-weight: bold;

   position: absolute;

   -webkit-animation:colorchange 20s infinite alternate;

 }

 @-webkit-keyframes colorchange {

   0% {

     color: lightblue;

   }

   10% {

     color: #8e44ad;

   }

   20% {

     color: #1abc9c;

   }

   30% {

     color: #7afad8;

   }

   40% {

     color: lightblue;

   }

   50% {

     color: #70b8ff;

   }

   60% {

     color: lightblue;

   }

   70% {

     color: #2980b9;

   }

   80% {

     color: #67d962;

   }

   90% {

     color: #62c7d9;

   }

   100% {

     color: #9ee84f;

   }

 }

Main.js

//user clicked button

document.getElementById("userInputButton").addEventListener("click", getUserInput, false);

//user pressed enter '13'

document.getElementById("userInput").addEventListener("keyup", function (event) {

    if (event.keyCode === 13) {

        //cancel the default action

        event.preventDefault();

        //process event

        getUserInput();

    }

});

eel.expose(addUserMsg);

eel.expose(addAppMsg);

function addUserMsg(msg) {

    element = document.getElementById("messages");

    element.innerHTML += '<div class="message from ready rtol">' + msg + '</div>';

    element.scrollTop = element.scrollHeight - element.clientHeight - 15;

    //add delay for animation to complete and then modify class to => "message from"

    index = element.childElementCount - 1;

    setTimeout(changeClass.bind(null, element, index, "message from"), 500);

}

function addAppMsg(msg) {

    element = document.getElementById("messages");

    element.innerHTML += '<div class="message to ready ltor">' + msg + '</div>';

    element.scrollTop = element.scrollHeight - element.clientHeight - 15;

    //add delay for animation to complete and then modify class to => "message to"

    index = element.childElementCount - 1;

    setTimeout(changeClass.bind(null, element, index, "message to"), 500);

}

function changeClass(element, index, newClass) {

    console.log(newClass +' '+ index);

    element.children[index].className = newClass;

}

function getUserInput() {

    element = document.getElementById("userInput");

    msg = element.value;

    if (msg.length != 0) {

        element.value = "";

        eel.getUserInput(msg);

    }

}