mediapipe for face detection and landmark extraction.

cv2 (OpenCV) for image processing.

csv for reading annotation files.

faceBlendCommon (fbc) for face blending purpose.

filters\_config = {

'anonymous': [{'path': "filters/anonymous.png",

'anno\_path': "filters/anonymous\_annotations.csv",

'morph': True, 'animated': False, 'has\_alpha': True}],

'anime': [{'path': "filters/anime.png",

'anno\_path': "filters/anime\_annotations.csv",

'morph': True, 'animated': False, 'has\_alpha': True}],

'dog': [{'path': "filters/dog-ears.png",

'anno\_path': "filters/dog-ears\_annotations.csv",

'morph': False, 'animated': False, 'has\_alpha': True},

{'path': "filters/dog-nose.png",

'anno\_path': "filters/dog-nose\_annotations.csv",

'morph': False, 'animated': False, 'has\_alpha': True}],

'cat': [{'path': "filters/cat-ears.png",

'anno\_path': "filters/cat-ears\_annotations.csv",

'morph': False, 'animated': False, 'has\_alpha': True},

{'path': "filters/cat-nose.png",

'anno\_path': "filters/cat-nose\_annotations.csv",

'morph': False, 'animated': False, 'has\_alpha': True}],

'jason-joker': [{'path': "filters/jason-joker.png",

'anno\_path': "filters/jason-joker\_annotations.csv",

'morph': True, 'animated': False, 'has\_alpha': True}],

'gold-crown': [{'path': "filters/gold-crown.png",

'anno\_path': "filters/gold-crown\_annotations.csv",

'morph': False, 'animated': False, 'has\_alpha': True}],

'flower-crown': [{'path': "filters/flower-crown.png",

'anno\_path': "filters/flower-crown\_annotations.csv",

'morph': False, 'animated': False, 'has\_alpha': True}],

}

sets a configuration dictionary for different filters. Each filter has attributes such as path to the image, anno\_path to its landmark annotations, and properties like morph, animated, and has\_alpha.

def getLandmarks(img):

mp\_face\_mesh = mp.solutions.face\_mesh

selected\_keypoint\_indices = [127, 93, 58, 136, 150, 149, 176, 148, 152, 377, 400, 378, 379, 365, 288, 323, 356, 70, 63, 105, 66, 55,

285, 296, 334, 293, 300, 168, 6, 195, 4, 64, 60, 94, 290, 439, 33, 160, 158, 173, 153, 144, 398, 385,

387, 466, 373, 380, 61, 40, 39, 0, 269, 270, 291, 321, 405, 17, 181, 91, 78, 81, 13, 311, 306, 402, 14,

178, 162, 54, 67, 10, 297, 284, 389]

height, width = img.shape[:-1]

with mp\_face\_mesh.FaceMesh(max\_num\_faces=1, static\_image\_mode=True, min\_detection\_confidence=0.5) as face\_mesh:

results = face\_mesh.process(cv2.cvtColor(img, cv2.COLOR\_BGR2RGB))

if not results.multi\_face\_landmarks:

print('Face not detected!!!')

return 0

for face\_landmarks in results.multi\_face\_landmarks:

values = np.array(face\_landmarks.landmark)

face\_keypnts = np.zeros((len(values), 2))

for idx, value in enumerate(values):

face\_keypnts[idx][0] = value.x

face\_keypnts[idx][1] = value.y

# Convert normalized points to image coordinates

face\_keypnts = face\_keypnts \* (width, height)

face\_keypnts = face\_keypnts.astype('int')

relevant\_keypnts = []

for i in selected\_keypoint\_indices:

relevant\_keypnts.append(face\_keypnts[i])

return relevant\_keypnts

return 0

Using Mediapipe face mesh module, facial landmarks are detected.

The selected\_keypoint\_indices are manually chosen indices that correspond to particular facial landmarks. These indices are derived from the complete list of landmarks provided by the FaceMesh model, which detects 468 landmarks on the face.

**Purpose of selected\_keypoint\_indices**

1. **Filter Application**: By using specific keypoints, the filter can be accurately placed and adjusted on facial features. For example, applying a filter to the eyes or mouth requires precise landmarks for proper positioning.
2. **Tracking and Stabilization**: In motion tracking or face stabilization tasks, only relevant keypoints are tracked to ensure the filter moves correctly with the face, providing a smoother and more realistic effect.

**Landmark Detection**:

* The FaceMesh model detects 468 facial landmarks and stores their normalized coordinates.

**Conversion to Pixel Coordinates**:

* The normalized coordinates are converted to pixel coordinates relative to the image size.

**Select Relevant Key Points**:

* From the full list of detected landmarks, only the ones at indices specified in selected\_keypoint\_indices are selected.
* These indices are carefully chosen to represent important facial features such as the eyes, mouth, and jawline.

**Return the Selected Key Points**:

* The function returns only the coordinates of the selected keypoints, which are then used to apply filters or track facial movements.

def load\_filter\_img(img\_path, has\_alpha):

# Read the image

img = cv2.imread(img\_path, cv2.IMREAD\_UNCHANGED)

alpha = None

if has\_alpha:

b, g, r, alpha = cv2.split(img)

img = cv2.merge((b, g, r))

return img, alpha

**Load the Image**:

* cv2.imread(img\_path, cv2.IMREAD\_UNCHANGED) loads the image from the specified path without changing it, meaning it retains the alpha channel (transparency) if it exists.

**Split Channels**:

* cv2.split(img) separates the image into its four channels: Blue (b), Green (g), Red (r), and Alpha (a).

**Merge RGB Channels**:

* cv2.merge((r, g, b)) combines the red, green, and blue channels back into an RGB image (without the alpha channel).

**Return the RGB Image and Alpha Channel**:

* The function returns the RGB image (img\_rgb) and the alpha channel (a) separately.

**Load Filter Image and Frame**:

* filter\_img, alpha = load\_filter\_img('filter.png') loads the filter image and its alpha channel.
* frame = cv2.imread('frame.jpg') loads the frame where the filter will be applied.
* Normalize the Alpha Channel - for blending the images properly
* The for loop in the code blends the filter image with the frame image by applying the alpha channel (transparency) to each color channel (Red, Green, Blue) individually.

def load\_landmarks(annotation\_file):

with open(annotation\_file) as csv\_file:

csv\_reader = csv.reader(csv\_file, delimiter=",")

points = {}

for i, row in enumerate(csv\_reader):

# skip head or empty line if it's there

try:

x, y = int(row[1]), int(row[2])

points[row[0]] = (x, y)

except ValueError:

continue

return points

This function loads landmark annotations from a CSV file and returns them as a dictionary of points.

Typically, the CSV file contains the landmark coordinates for specific facial features. Each row in the CSV file represents a landmark and includes:

* **Landmark Identifier**: A name or index for the landmark.
* **X Coordinate**: The X position of the landmark.
* **Y Coordinate**: The Y position of the landmark.

An empty dictionary named points is created. This will store the landmark data, with landmark identifiers as keys and (x, y) coordinates as values

for loop iterates over each row of the CSV file. Each row is a list of values corresponding to a single landmark.

**Extract and Convert Coordinates**

* For each row, the function tries to convert the second and third values (row[1] and row[2], which are expected to be the x and y coordinates) into integers.

If the conversion is successful, it stores the coordinates in the points dictionary with the landmark identifier (row[0]) as the key.

If there is a ValueError (for instance, if the coordinates are not valid integers), the function skips that row.

* After processing all rows, the function returns the points dictionary, which contains the landmarks and their coordinates.

def find\_convex\_hull(points):

hull = []

hullIndex = cv2.convexHull(np.array(list(points.values())), clockwise=False, returnPoints=False)

addPoints = [

[48], [49], [50], [51], [52], [53], [54], [55], [56], [57], [58], [59], # Outer lips

[60], [61], [62], [63], [64], [65], [66], [67], # Inner lips

[27], [28], [29], [30], [31], [32], [33], [34], [35], # Nose

[36], [37], [38], [39], [40], [41], [42], [43], [44], [45], [46], [47], # Eyes

[17], [18], [19], [20], [21], [22], [23], [24], [25], [26] # Eyebrows

]

hullIndex = np.concatenate((hullIndex, addPoints))

for i in range(0, len(hullIndex)):

hull.append(points[str(hullIndex[i][0])])

return hull, hullIndex

This function finds the convex hull for given points, which is used to determine the outer boundary of the filter to be applied.

* + including additional points for key facial features (lips, nose, eyes, eyebrows), the function ensures that these features are accurately represented and can be properly warped or transformed.
  + cv2.convexHull calculates the convex hull of the given set of points.
  + returnPoints=False means that the function will return the indices of the points forming the convex hull rather than the coordinates themselves.
  + The convex hull indices are combined with the additional facial landmark indices to form a comprehensive set of points.
  + For loop iterates through the combined indices, converts each index to a string (as the keys in points are strings), and appends the corresponding coordinates to the hull list.
  + unction returns the list of coordinates (hull) and the indices (hullIndex) forming the convex hull and including additional points.

def load\_filter(filter\_name="anonymous"):

filters = filters\_config[filter\_name]

multi\_filter\_runtime = []

for filter in filters:

temp\_dict = {}

img1, img1\_alpha = load\_filter\_img(filter['path'], filter['has\_alpha'])

temp\_dict['img'] = img1

temp\_dict['img\_a'] = img1\_alpha

points = load\_landmarks(filter['anno\_path'])

temp\_dict['points'] = points

if filter['morph']:

# Find convex hull for delaunay triangulation using the landmark points

hull, hullIndex = find\_convex\_hull(points)

# Find Delaunay triangulation for convex hull points

sizeImg1 = img1.shape

rect = (0, 0, sizeImg1[1], sizeImg1[0])

dt = fbc.calculateDelaunayTriangles(rect, hull)

temp\_dict['hull'] = hull

temp\_dict['hullIndex'] = hullIndex

temp\_dict['dt'] = dt

if len(dt) == 0:

continue

if filter['animated']:

filter\_cap = cv2.VideoCapture(filter['path'])

temp\_dict['cap'] = filter\_cap

multi\_filter\_runtime.append(temp\_dict)

return filters, multi\_filter\_runtime

filters\_config: A global dictionary containing configurations for various filters.

filters: Fetches the configuration for the specified filter name.

multi\_filter\_runtime: Initializes an empty list to store runtime data for each filter.

Loops through each filter configuration to handle multiple filters if necessary.

temp\_dict: A temporary dictionary to store the filter image, alpha channel, landmarks, and other necessary data

load\_filter\_img: Uses this method to read the filter image from the specified path and retrieves its alpha channel if it has one.

* img1: The filter image.
* img1\_alpha: The alpha channel of the filter image.

Stores the filter image and its alpha channel in temp\_dict.

load\_landmarka() used:

reads the annotation file (a CSV file) and retrieves the landmark points.

Stores the landmark points in temp\_dict.

Next,

Checks if the filter requires morphing, if so

If the filter requires morphing, the function calculates the convex hull and Delaunay triangulation of the landmark points.

The convex hull (hull) is used to find the boundary of the face.

The Delaunay triangulation (dt) divides the convex hull into triangles, which facilitates smooth and accurate transformations of the filter image to fit the target face.

Step By Step:

find\_convex\_hull: Calculates the convex hull of the landmark points. The convex hull is the smallest polygon that encloses all the given points.

* hull: The points forming the convex hull.
* hullIndex: The indices of these points.

sizeImg1: Retrieves the dimensions of the filter image.

rect: Defines the bounding rectangle for the Delaunay triangulation.

fbc.calculateDelaunayTriangles: Calculates the Delaunay triangulation for the given convex hull points.

* **Delaunay triangulation**: A technique to divide the convex hull into triangles, which helps in transforming the filter image to match the target face.
* dt: The Delaunay triangles.

Stores the convex hull (hull), its indices (hullIndex), and Delaunay triangles (dt) in temp\_dict.

If no Delaunay triangles are found (len(dt) == 0), the loop continues to the next filter configuration.

Next,

If the filter is animated, it opens the video file using cv2.VideoCapture and stores the video capture object in temp\_dict.

Appends the prepared filter data (temp\_dict) to the multi\_filter\_runtime list.

Returns the filter configurations and the runtime data for the filters.

#After the Convex Hull Calculation and Delaunay Triangulation Transformation is done where

Each triangle in the filter image is mapped to a corresponding triangle on the target face using affine transformations.

This ensures that the filter image deforms to fit the shape and features of the target face accurately.

ap = cv2.VideoCapture(0)

# Some variables

count = 0

isFirstFrame = True

sigma = 50

# Use the 'anonymous' filter

filters, multi\_filter\_runtime = load\_filter('anonymous')

# ////////////////////////////////////////////////////////////////////

# The main loop

while True:

ret, frame = cap.read()

if not ret:

break

else:

points2 = getLandmarks(cv2.cvtColor(frame, cv2.COLOR\_BGR2RGB))

# if face is partially detected

if not points2 or (len(points2) != 75):

continue

################ Optical Flow and Stabilization Code #####################

img2Gray = cv2.cvtColor(frame, cv2.COLOR\_BGR2GRAY)

if isFirstFrame:

points2Prev = np.array(points2, np.float32)

img2GrayPrev = np.copy(img2Gray)

isFirstFrame = False

lk\_params = dict(winSize=(101, 101), maxLevel=15,

criteria=(cv2.TERM\_CRITERIA\_EPS | cv2.TERM\_CRITERIA\_COUNT, 20, 0.001))

points2Next, st, err = cv2.calcOpticalFlowPyrLK(img2GrayPrev, img2Gray, points2Prev,

np.array(points2, np.float32),

\*\*lk\_params)

# Final landmark points are a weighted average of detected landmarks and tracked landmarks

for k in range(len(points2)):

d = cv2.norm(np.array(points2[k]) - points2Next[k])

alpha = math.exp(-d \* d / sigma)

points2[k] = (1 - alpha) \* np.array(points2[k]) + alpha \* points2Next[k]

points2[k] = fbc.constrainPoint(points2[k], frame.shape[1], frame.shape[0])

points2[k] = (int(points2[k][0]), int(points2[k][1]))

# Update variables for next pass

points2Prev = np.array(points2, np.float32)

img2GrayPrev = img2Gray

################ End of Optical Flow and Stabilization Code ###############

if VISUALIZE\_FACE\_POINTS:

for idx, point in enumerate(points2):

cv2.circle(frame, point, 2, (255, 0, 0), -1)

cv2.putText(frame, str(idx), point, cv2.FONT\_HERSHEY\_SIMPLEX, .3, (255, 255, 255), 1)

cv2.imshow("landmarks", frame)

filter\_runtime = multi\_filter\_runtime[0] # Use the first (and only) filter

img1 = filter\_runtime['img']

points1 = filter\_runtime['points']

img1\_alpha = filter\_runtime['img\_a']

if filters[0]['morph']:

hullIndex = filter\_runtime['hullIndex']

dt = filter\_runtime['dt']

hull1 = filter\_runtime['hull']

# create copy of frame

warped\_img = np.copy(frame)

# Find convex hull

hull2 = []

for i in range(len(hullIndex)):

hull2.append(points2[hullIndex[i][0]])

mask1 = np.zeros((warped\_img.shape[0], warped\_img.shape[1]), dtype=np.float32)

mask1 = cv2.merge((mask1, mask1, mask1))

img1\_alpha\_mask = cv2.merge((img1\_alpha, img1\_alpha, img1\_alpha))

# Warp the triangles

for i in range(len(dt)):

t1 = []

t2 = []

for j in range(3):

t1.append(hull1[dt[i][j]])

t2.append(hull2[dt[i][j]])

fbc.warpTriangle(img1, warped\_img, t1, t2)

fbc.warpTriangle(img1\_alpha\_mask, mask1, t1, t2)

# Blur the mask before blending

mask1 = cv2.GaussianBlur(mask1, (3, 3), 10)

mask2 = (255.0, 255.0, 255.0) - mask1

# Perform alpha blending of the two images

temp1 = np.multiply(warped\_img, (mask1 \* (1.0 / 255)))

temp2 = np.multiply(frame, (mask2 \* (1.0 / 255)))

output = temp1 + temp2

else:

dst\_points = [points2[int(list(points1.keys())[0])], points2[int(list(points1.keys())[1])]]

tform = fbc.similarityTransform(list(points1.values()), dst\_points)

# Apply similarity transform to input image

trans\_img = cv2.warpAffine(img1, tform, (frame.shape[1], frame.shape[0]))

trans\_alpha = cv2.warpAffine(img1\_alpha, tform, (frame.shape[1], frame.shape[0]))

mask1 = cv2.merge((trans\_alpha, trans\_alpha, trans\_alpha))

# Blur the mask before blending

mask1 = cv2.GaussianBlur(mask1, (3, 3), 10)

mask2 = (255.0, 255.0, 255.0) - mask1

# Perform alpha blending of the two images

temp1 = np.multiply(trans\_img, (mask1 \* (1.0 / 255)))

temp2 = np.multiply(frame, (mask2 \* (1.0 / 255)))

output = temp1 + temp2

frame = output = np.uint8(output)

cv2.putText(frame, "Press ESC to exit", (10, 20), cv2.FONT\_HERSHEY\_SIMPLEX, .5, (255, 0, 0), 1)

cv2.imshow("Face Filter", output)

keypressed = cv2.waitKey(1) & 0xFF

if keypressed == 27:

break

cap.release()

cv2.destroyAllWindows()

Capture Video Stream and Loads the filter that is passed

**Optical Flow** and **Filter Application**. Each part ensures that the filter stays accurately placed on the face, even as the face moves or changes expression.

**1. Optical Flow and Stabilization**

The optical flow method helps track how facial landmarks move from frame to frame.

* **Convert Frame to Grayscale**:

Reads frames from the video stream.

Converts the frame to RGB and extracts facial landmarks using getLandmarks.

===> points2

Optical flow calculations are typically done on grayscale images to simplify the computations.On the first frame, store the detected landmarks(points2Prev ) and grayscale image(img2GrayPrev) for future comparisons.

cv2.calcOpticalFlowPyrLK computes how the landmarks have moved from the previous frame to the current frame using the Lucas-Kanade method. This function outputs the new positions of the landmarks (points2Next).

for k in range(len(points2)):

d = cv2.norm(np.array(points2[k]) - points2Next[k])

alpha = math.exp(-d \* d / sigma)

points2[k] = (1 - alpha) \* np.array(points2[k]) + alpha \* points2Next[k]

points2[k] = fbc.constrainPoint(points2[k], frame.shape[1], frame.shape[0])

points2[k] = (int(points2[k][0]), int(points2[k][1]))

This part combines the detected landmarks (points2) with the tracked landmarks (points2Next). Updates each landmark point with a weighted average of detected and tracked points, influenced by sigma.The blending helps smooth out sudden movements and keeps the landmarks stable.

Next is the filter application:

Extracts filter images, landmarks, and alpha channels from multi\_filter\_runtime.

Once you have the stable landmark positions, you apply the filter to the face. Depending on the type of filter, the approach differs: MORPHING OR TRANSFORMATION : This part based on the approach finally combine the warped filter img/transformed filter img to the original frame