Facial Video Filter:

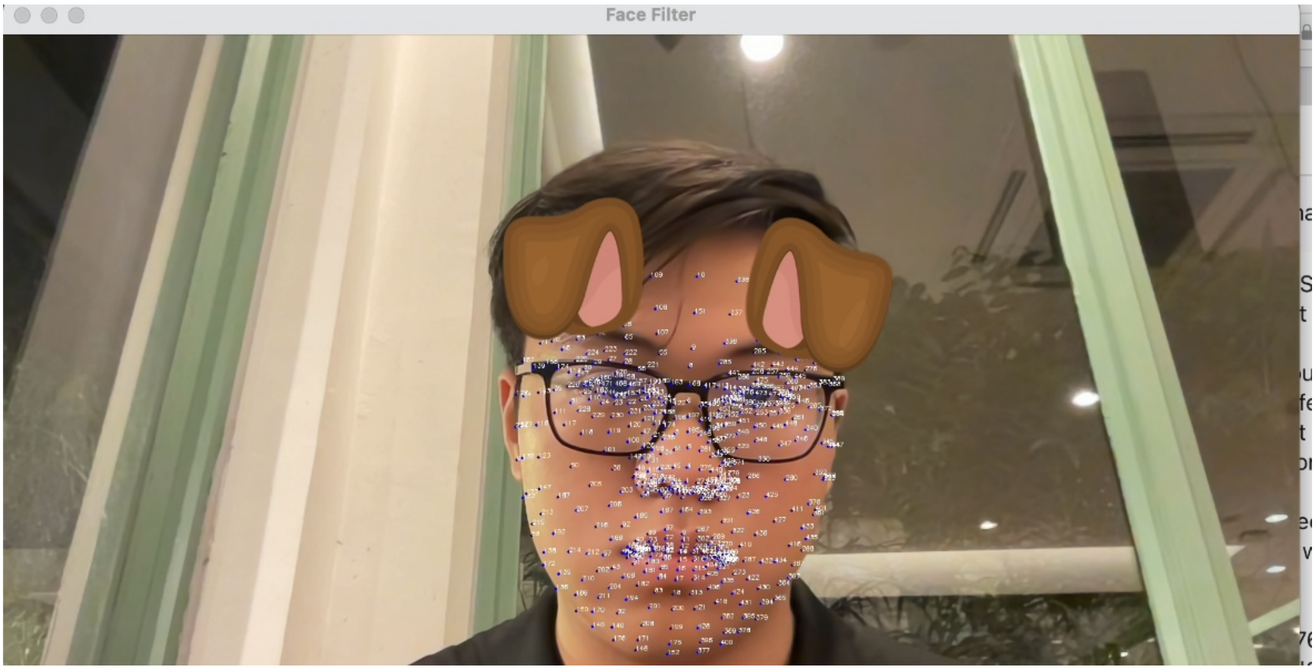

Table of Contents

1. Language and Tools

Language:

- Python 3.11

Libraries & Tools

- Opencv: Used for reading and extracting the video, image input
- Mediapipe: Used for face detection & extract facial landmark points
- Scipy: Used for calculate Delaunay triangulation
- Numpy, Math: Used for math calculation
- Hydra, Rootutils: Used for setup machine
- Csv: Used for read annotation file

2. Step by step

Step 1: Import library and Utils

Essential Library

```
from scipy.spatial import Delaunay
from omegaconf import DictConfig, OmegaConf
from math_utils import FBC
import mediapipe as mp
import cv2
import hydra
import rootutils
import csv
import numpy as np
import math
```

Face Detection Utils (Face Detection and Landmark)

- Facial Box Detection:

```
def face_detection(self, image):
    # Define mediapipe module
    mp_face_detection = mp.solutions.face_detection
    face_detection = mp_face_detection.FaceDetection(model_selection=1,
                                                     min_detection_confidence=0.7)

    # Input video frame into module
    result = face_detection.process(cv2.cvtColor(image, cv2.COLOR_BGR2RGB))
    face_detection.close()
    return result
```

- Facial Landmark Points:

```
def face_landmark_points(self, faces):
    # Define output list
    landmark_points = []

    # Define mediapipe module
    mp_face_mesh = mp.solutions.face_mesh
    face_mesh = mp_face_mesh.FaceMesh(static_image_mode=True,
                                      max_num_faces=1,
                                      refine_landmarks=True)

    # Get facial box detection
    face_detection = faces
    IS_FACE_DETECTION = len(face_detection.detections) >= 1
```

```
    # Extract landmark for each facial box
    if IS_FACE_DETECTION == True:
        for face in face_detection.detections:
            face_landmark_points = []
            face_box = face.location_data.relative_bounding_box
            ih, iw, _ = self.image.shape
            x, y = int(face_box.xmin * iw), int(face_box.ymin * ih)
            w, h = int(face_box.width * iw), int(face_box.height * ih)

            # Cropped facial box to detect
            face_cropped = self.image[y:y+h, x:x+w]

            # Input box into landmark points module
            temp_image = cv2.cvtColor(face_cropped, cv2.COLOR_BGR2RGB)
            landmark_points_cropped = face_mesh.process(temp_image)

            IS_FACE_LANDMARK_POINTS = len(landmark_points_cropped.multi_face_landmarks) >= 1
            if IS_FACE_LANDMARK_POINTS == True:
                for face_landmarks in landmark_points_cropped.multi_face_landmarks:
                    for points in face_landmarks.landmark:
                        x_origin = int(points.x * w) + x
                        y_origin = int(points.y * h) + y

                        face_landmark_points.append((x_origin, y_origin))
            landmark_points.append(face_landmark_points)

    face_mesh.close()
    return landmark_points
```

Filter Utils (Filter Loading)

- Load Image Of Filter

```
def __load_filter_image(self, image_directory: str, include_alpha: bool):
    image = cv2.imread(image_directory, cv2.IMREAD_UNCHANGED)
    alpha = None
    if include_alpha == True:
        b, g, r, alpha = cv2.split(image)
        image = cv2.merge((b, g, r))
    return image, alpha
```

- Load Annotation Of Filter:

```
def __load_filter_landmark(self, landmark_directory: str):
    with open(landmark_directory) as file:
        csv_reader = csv.reader(file, delimiter=",")
        points = {}
        for _, rowValue in enumerate(csv_reader):
            try:
                x, y = int(rowValue[1]), int(rowValue[2])
                points[rowValue[0]] = (x, y)
            except ValueError:
                continue
        return points
```

- Calculate Convex Hull:

```
def find_convex_hull(self, points):
    hull = []
    hullIndex = cv2.convexHull(np.array(list(points.values())), clockwise=False, returnPoints=Fa
    addPoints = [
        [0],[1],[2],[3],[4],[5],[6],[7],[8],[9],[10],[11],[61], //
        [146],[91],[181],[84],[17],[314],[405],[321],[375],[291],# MEDIAPIPE OUTER LIPS
        [78],[95],[88],[178],[87],[14],[317],[402],[318],[324],[308], # MEDIAPIPE INNNER LIPS
        [168],[197],[2],[326],# MEDIAPIPE NOSE
        [362],[382],[381],[380],[374],[373],[390],[249],[263],[33],[7], //
        [163],[144],[145],[153],[154],[155],[133],# MEDIAPIPE EYES
        [336],[296],[334],[293],[300],[70],[63],[105],[66],[107],# MEDIAPIPE EYEBROWS
    ]
    hullIndex = np.concatenate((hullIndex, addPoints))
    for i in range(0, len(hullIndex)):
        hull.append(points[str(hullIndex[i][0])])

    return hull, hullIndex
```

- And we define filter application as a class

Step 2: Facial Detection and Landmark Processing

Define utils (We use hydra to read setup)

```
facial_detection_tool = FaceDetectionUtils(cfg.facial_detection_configs)
filter_tool = FilterUtils(cfg.filter_configs)
```

- We extract each frame to process like an image.

```
self.capture = cv2.VideoCapture(self.videoPath)
while (self.capture.isOpened()):
    ret, frame = self.capture.read()
    if not ret:
        break
```

- Get the landmark points for filter usage.

```
# Get face boxes
face_box = self.facial_detection_tool.face_detection(frame)
if not face_box:
    continue

# Get landmark points
face_landmark_points = self.facial_detection_tool.face_landmark_points(face_box)[0]
if not face_landmark_points:
    continue
```

Step 3: Applying Affine Transform

- First, we need to calculate the last point that create the equilateral triangle for easy transforming purpose.

Creating an Equilateral Triangle

To find a third point **C(x, y)** that forms an equilateral triangle with two given points **A(x_1, y_1)** and **B(x_2, y_2)**, you can use trigonometric functions and a rotation matrix.

Points Given

We start with two points:

- **A(x_1, y_1)**
- **B(x_2, y_2)**

Rotation Matrix

A rotation matrix for an angle θ allows you to rotate a point or a vector around the origin in 2D space. The general form of a 2D rotation matrix $R(\theta)$ is:

$$R(\theta) = \begin{bmatrix} \cos(\theta) & -\sin(\theta) \\ \sin(\theta) & \cos(\theta) \end{bmatrix}$$

Applying the Rotation Matrix for 60 Degrees

For an equilateral triangle, all internal angles are 60 degrees. To position point **C** relative to point **B**, we rotate the vector $\overrightarrow{AB}$ by 60 degrees counterclockwise.

Vector $\overrightarrow{AB}$

The vector from **A** to **B** is:

$$\overrightarrow{AB} = \begin{bmatrix} x_2 - x_1 \\ y_2 - y_1 \end{bmatrix}$$

Rotate $\overrightarrow{AB}$ by 60 Degrees

Applying the 60-degree counterclockwise rotation matrix:

$$R(60^\circ) = \begin{bmatrix} \frac{1}{2} & -\frac{\sqrt{3}}{2} \\ \frac{\sqrt{3}}{2} & \frac{1}{2} \end{bmatrix}$$

The rotated vector calculation is:

$$\begin{bmatrix} \frac{1}{2}(x_2 - x_1) - \frac{\sqrt{3}}{2}(y_2 - y_1) \\ \frac{\sqrt{3}}{2}(x_2 - x_1) + \frac{1}{2}(y_2 - y_1) \end{bmatrix}$$

Coordinates of Point C

Adding this vector to point **B** provides the coordinates for point **C**:

$$x = x_2 + \frac{1}{2}(x_2 - x_1) - \frac{\sqrt{3}}{2}(y_2 - y_1)$$
$$y = y_2 + \frac{\sqrt{3}}{2}(x_2 - x_1) + \frac{1}{2}(y_2 - y_1)$$

OR

$$x = x_2 + \cos(60) * (x_2 - x_1) - \sin(60) * (y_2 - y_1)$$
$$y = y_2 + \sin(60) * (x_2 - x_1) + \cos(60) * (y_2 - y_1)$$

This results in **C** forming an equilateral triangle with **A** and **B**.

```
def similarityTransform(self, inPoints, outPoints):
    s60 = math.sin(60*math.pi/180)
    c60 = math.cos(60*math.pi/180)

    inPts = np.copy(inPoints).tolist()
    outPts = np.copy(outPoints).tolist()

    # The third point is calculated so that the three points make an equilateral triangle
    xin = c60*(inPts[0][0] - inPts[1][0]) - s60*(inPts[0][1] - inPts[1][1]) + inPts[1][0]
    yin = s60*(inPts[0][0] - inPts[1][0]) + c60*(inPts[0][1] - inPts[1][1]) + inPts[1][1]

    inPts.append([int(xin), int(yin)])

    xout = c60*(outPts[0][0] - outPts[1][0]) - s60*(outPts[0][1] - outPts[1][1]) + outPts[1][0]
    yout = s60*(outPts[0][0] - outPts[1][0]) + c60*(outPts[0][1] - outPts[1][1]) + outPts[1][1]

    outPts.append([int(xout), int(yout)])
```

- Now we can use "estimateAffinePartial2D" for calculating the similarity transform.

```
tform = cv2.estimateAffinePartial2D(np.array([inPts]), np.array([outPts]), False)
return tform
```

- We will use tform to warp the original image to the filtered image.

```
for idx, filter in enumerate(filters):
    filter_runtime = multi_filter_runtime[idx]
    img1 = filter_runtime['image']
    points1 = filter_runtime['landmark_point']
    img1_alpha = filter_runtime['image_alpha']

    # Landmark-Based Transformation
    dst_points = [points2[int(list(points1.keys())[0])], points2[int(list(points1.keys())[1])]]
    tform = FBC().similarityTransform(list(points1.values()), dst_points)[0]

    # Image Transformation
    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]))

    # Mask Processing
    mask1 = cv2.merge((trans_alpha, trans_alpha, trans_alpha))
    mask1 = cv2.GaussianBlur(mask1, (3, 3), 10)
    mask2 = (255.0, 255.0, 255.0) - mask1

    # Image Blending
    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)
```

Step 4: Lucas-Kanade Optical Flow and Stablization

The Lucas-Kanade method estimates motion at specific points in an image, typically features like corners. It assumes that motion is small and nearly constant within a small window around these points.

Equation Setup

The fundamental assumption is that pixel intensities remain constant despite motion:

$$I(x, y, t) = I(x + \delta x, y + \delta y, t + \Delta t)$$

where (x, y) are the coordinates at time t, and δx and δy are displacements in x and y directions at time $t + \Delta t$.

Because of we assume that intensity is constant. So that the derivatives of it will be zero.

$$I_x \delta x + I_y \delta y = -I_t$$

System of Equations

This forms an underdetermined system. To resolve it, Lucas-Kanade uses a window of points, resulting in:

$$\mathbf{A}\mathbf{u} = \mathbf{b}$$

Where:

- $$\mathbf{u} = [\begin{smallmatrix}\delta x \\ \delta y\end{smallmatrix}]$$
- $$\mathbf{A} = \begin{bmatrix} I_x(p_1) & I_y(p_1) \\ \vdots & \vdots \\ I_x(p_n) & I_y(p_n) \end{bmatrix}$$
- $$\mathbf{b} = \begin{bmatrix} -I_t(p_1) \\ \vdots \\ -I_t(p_n) \end{bmatrix}$$

Solving for Motion

The motion vector $\mathbf{u}$ is found by solving:

$$\mathbf{u} = (\mathbf{A}^T\mathbf{A})^{-1}\mathbf{A}^T\mathbf{b}$$

The purpose of this algorithm when applied in this code will used to estimate the next position point.

```
lk_params = dict(winSize=(101, 101), maxLevel=15,
                 criteria=(cv2.TERM_CRITERIA_EPS | cv2.TERM_CRITERIA_COUNT, 20, 0.001))
points2Next, _, __ = cv2.calcOpticalFlowPyrLK(img2GrayPrev, img2Gray, points2Prev,
                                              np.array(points2, np.float32),
                                              **lk_params)
```

- `lk_params`:
    - **winSize**: The size of the search window at each pyramid level. `(101, 101)` specifies that each window is 101x101 pixels. This window defines the neighborhood around each point that is used to find the optical flow.
    - **maxLevel**: The number of pyramid layers used to calculate the optical flow. `15` indicates that the image pyramid (a multi-scale representation of the original image) will have 15 levels. More levels can help in capturing motion at various scales but increases computation.

  - **criteria**: This parameter sets the termination criteria of the iterative search algorithm. It combines two conditions:
    - **cv2.TERM_CRITERIA_EPS**: The algorithm will stop if the specified accuracy, `0.001`, is reached.
    - **cv2.TERM_CRITERIA_COUNT**: The algorithm will also stop if the number of iterations reaches the limit, `20` in this case.

- `cv2.calcOpticalFlowPyrLK`

  - **img2GrayPrev and img2Gray**: These are the two consecutive frames between which the optical flow is to be calculated. Typically, they should be grayscale images, hence the `Gray` in the variable names suggests that these images are already converted to grayscale.

  - **points2Prev**: This is an array of points in the previous image (`img2GrayPrev`) for which the flow needs to be found in the second image (`img2Gray`). These points are usually features detected in the first image, such as corners.

  - Output:

    - **points2Next**: The calculated new positions of the input points in the second image.

Stablization

Assuming:

- p_k is the point at index k in the current frame (`points2[k]` in the code).
- $p_k^{'}$ is the point at index k in the next frame predicted by optical flow (`points2Next[k]` in the code).
- d_k is the Euclidean distance between p_k and $p_k^{'}$.
- σ is a parameter affecting the smoothness of the tracking.
- α_k is the weight for updating the point's position.

The operations performed in the loop for each point can be mathematically described as follows:

1. Calculate the Euclidean distance d_k between the points:

$$d_k = \|p_k - p_k^{'}\|$$

2. Compute the weight α_k using the distance:

$$\alpha_k = e^{-\frac{d_k^2}{\sigma}}$$

3. Update the position of p_k using a weighted average of the current and new positions:

$$p_k = (1 - \alpha_k) \cdot p_k + \alpha_k \cdot p_k^{'}$$

4. Constrain the updated point within the image frame. If (w, h) are the width and height of the frame and $p_k = (x_k, y_k)$, then:

$$x_k = \max(\min(x_k, w - 1), 0)$$

$$y_k = \max(\min(y_k, h - 1), 0)$$

5. Convert the coordinates of p_k to integers, as image coordinates are integers:

$$p_k = (\lfloor x_k \rfloor, \lfloor y_k \rfloor)$$

```
for k in range(0, 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]))
```