DEPARTMENT OF ELECTRONICS AND COMMUNICATION

PES University

Outer Ring Rd, Banashankari 3rd stage, Banashankari,
Bengaluru, Karnataka 560085

2020-2021

Project Report on

"Face Swapping using OpenCV"

By

SRN	STUDENT NAME	EMAIL ID	CONTACT NO.
PES1201800488	Aryan Jain	aryan.jain227@gmail.com	9108673812

Under the guidance of
Ms. Lavanya Krishna
Faculty, ECE Department
Elective: DIP (UE18EC317)

Executive Summary

Face swapping is used to transfer a face from an image source to a target image while face reenacting or face puppeteering uses the facial movements and expression deformations of a control face in one photo to guide the motions and deformations of a face which is appearing in another photo. This is a very simple python program to demonstrate that application. A lot of work can be done especially when incorporated with Neural networks. The best application of such a face swapping program would be

1. Face de-identification
2. Switching faces
3. Composite group photographs

Hardware

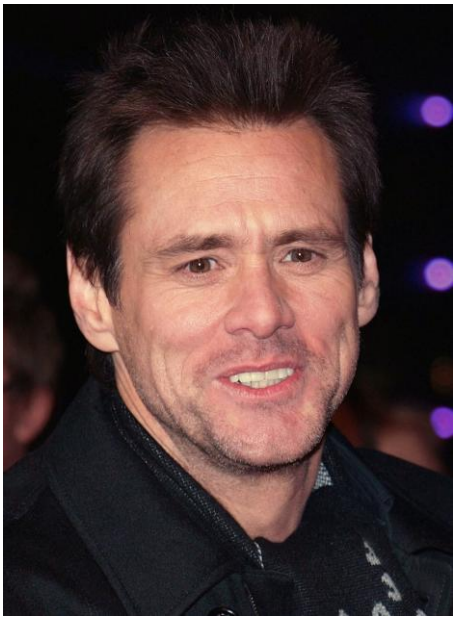
Operating System: Windows 7 or better
Processor: Intel or AMD x86 processor
Disk Space: 4GB or better
RAM: 2048 MB at least (recommended)

Software

1. Visual studio code (Or any Python3 compiler)
2. Dlib library
3. OpenCV library

Input Images

"Source Image"



"Destination Image"

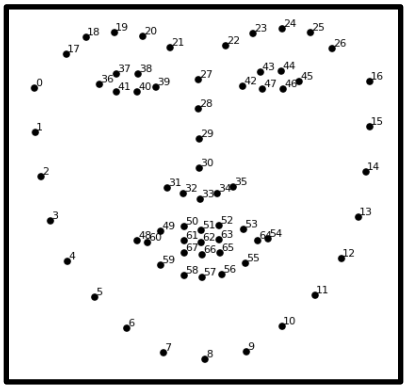
Process

Extracting the face from an image

Under this we mainly perform 3 steps:

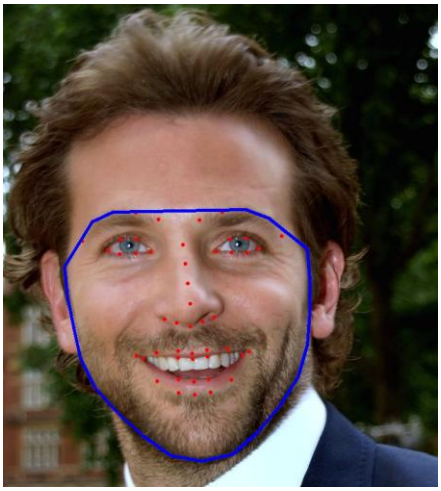
a. Facial landmark detection

For this we use the shape_predictor_68_face_landmarks.dat taken from github. This is a standard dataset used for landmark detection. It is distributed as jaw(0-16), right eyebrow(17-21), left eyebrow(22-26), nose(27-35), right eye(36-41), left eye(42-47) and mouth (48-67). These points are first plotted on to the image for our understanding.



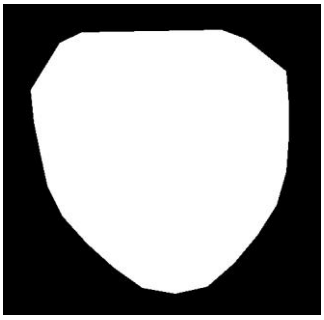
b. Convex Hull

The convex hull is the external area of the face. It can be directly taken from the cv2 library. It is to be noted that the line angles do not exceed 180^{o} which results in the exclusion of 4 eyebrow points (which technically form the exterior of the face).



c. Mask detection and image extraction

We use the convex hull polygon to extract a mask from the image. First, we use a grayscale image and completely darken it, then the points inside the polygon are assigned a grayscale value of 255 (White). We then bitwise and the original image and the mask to obtain our final extracted face.

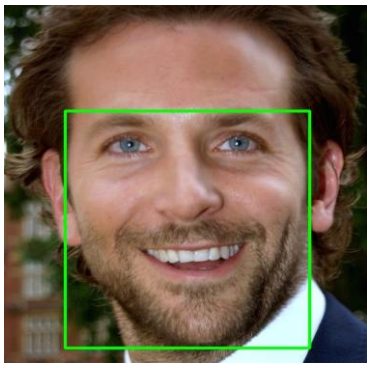


```
img = cv2.imread(r"E:/DIP/face_swapping_in_8_steps/bradley_cooper.jpg")
img_gray = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)
mask = np.zeros_like(img_gray)

detector = dlib.get_frontal_face_detector()
predictor = dlib.shape_predictor("shape_predictor_68_face_landmarks.dat")
faces = detector(img_gray)
for face in faces:
    landmarks = predictor(img_gray, face)
    landmarks_points = []
    for n in range(0, 68):
        x = landmarks.part(n).x
        y = landmarks.part(n).y
        landmarks_points.append((x, y))

        #cv2.circle(img, (x, y), 3, (0, 0, 255), -1)

    points = np.array(landmarks_points, np.int32)
    convexhull = cv2.convexHull(points)
    #cv2.polylines(img, [convexhull], True, (255, 0, 0), 3)
    cv2.fillConvexPoly(mask, convexhull, 255)

    face_image_1 = cv2.bitwise_and(img, img, mask=mask)
```

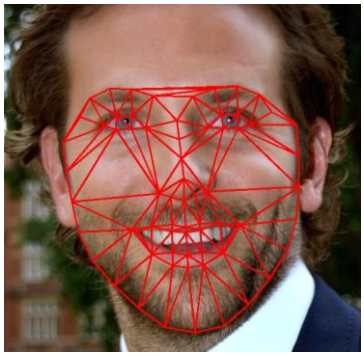
"Code snippet 1"

Face segmentation using Delaunay Triangulation

The Delaunay triangulation is a triangulation of the convex hull of the points in the diagram in which every circumcircle of a triangle is an empty circle. We are essentially cutting the face into smaller triangles which will then be replaced in the other image. We first find the neighboring rectangles of the image using opencv functions.



We create a subdiv for this rectangle and also insert the landmark points we obtained previously into that subdiv. We then try to obtain the coordinates of the triangles and plot them on the face (for representation purposes).



```
rect = cv2.boundingRect(convexhull)
#(x,y,w,h) = rect
#cv2.rectangle(img,(x,y),(x+w,y+h),(0,255,0),3)
subdiv = cv2.Subdiv2D(rect)
subdiv.insert(landmarks_points)
triangles = subdiv.getTriangleList()
triangles = np.array(triangles, dtype=np.int32)
for t in triangles:
    pt1 = (t[0], t[1])
    pt2 = (t[2], t[3])
    pt3 = (t[4], t[5])
    cv2.line(img, pt1, pt2, (0, 0, 255), 2)
    cv2.line(img, pt2, pt3, (0, 0, 255), 2)
    cv2.line(img, pt1, pt3, (0, 0, 255), 2)
```

"Code snippet 2"

Triangulation on the other face

We cannot perform the same steps of delaunay triangulation on the other face as we did previously. This is because each face has different landmark points which leads to varied sizes of triangles. If the landmark points used to create the triangles differ, then superimposition of one triangle onto another will be a very shabby process.

We try to find not only the coordinates but also the indices of the triangles formed.

Example: [0,17,56] is a triangle that contains those 3 landmark points. We then form the same triangles on the second face and only use them.

```
def extract_index_nparray(nparray):
    index = None
    for num in nparray[0]:
        index = num
        break
    return index
```

"Code snippet 3"

```
for t in triangles:
    pt1 = (t[0], t[1])
    pt2 = (t[2], t[3])
    pt3 = (t[4], t[5])
    index_pt1 = np.where((points == pt1).all(axis=1))
    index_pt1 = extract_index_nparray(index_pt1)
    index_pt2 = np.where((points == pt2).all(axis=1))
    index_pt2 = extract_index_nparray(index_pt2)
    index_pt3 = np.where((points == pt3).all(axis=1))
    index_pt3 = extract_index_nparray(index_pt3)
    if index_pt1 is not None and index_pt2 is not None and index_pt3 is not None:
        triangle = [index_pt1, index_pt2, index_pt3]
        indexes_triangles.append(triangle)
```

"Code snippet 4"

Warping triangles

We need to adjust the sizes of the triangles obtained from the first image to a similar size triangle in the second image. In order to do this, we take one triangle from the second image and mask it (Similar to our previous masking operation we get a black and white image). We then use bitwise operators on this masked triangle and the corresponding triangle in the first image.

We then warp the first triangle to fit the same size as the second triangle. We use the affine transformation for this. In affine transformation, all parallel lines in the original image will still be parallel in the output image. To find the transformation matrix, we need three points from the input image and their corresponding locations in the output image. Then cv2.getAffineTransform will create a 2×3 matrix which is to be passed to cv2.warpAffine.

```
# Warp triangles
points = np.float32(points)
points2 = np.float32(points2)

M = cv2.getAffineTransform(points, points2)

warped_triangle = cv2.warpAffine(cropped_triangle, M, (w, h))

break
```

"Code Snippet 5"

Swapping faces

We first try to just fit those triangles directly onto the second face. But that leads to errors as the black part from the masked triangles overlaps the image in some cases and as a reason we can see only some triangles. To go around this we keep on adding the triangles in a continuous counter so there is no overlapping.

We now use the second image and perform inverse masking on it (The face has a grayscale value of 0 and everything else in the image is retained). This is done by using THRESH_BINARY_INVERSE and setting a threshold value of 1.

We then add the two images together and use the built-in opencv function, “SeamlessClone” which helps us put the finishing touches and adjusts the colors automatically. Our final image is now ready.

```
    # Reconstructing destination face
    img2_new_face_rect_area = img2_new_face[y: y + h, x: x + w]
    img2_new_face_rect_area_gray = cv2.cvtColor(img2_new_face_rect_area, cv2.COLOR_BGR2GRAY)
    _, mask_triangles_designed = cv2.threshold(img2_new_face_rect_area_gray, 1, 255, cv2.THRESH_BINARY_INV)
    warped_triangle = cv2.bitwise_and(warped_triangle, warped_triangle, mask=mask_triangles_designed)

    img2_new_face_rect_area = cv2.add(img2_new_face_rect_area, warped_triangle)
    img2_new_face[y: y + h, x: x + w] = img2_new_face_rect_area



# Face swapped (putting 1st face into 2nd face)
img2_face_mask = np.zeros_like(img2_gray)
img2_head_mask = cv2.fillConvexPoly(img2_face_mask, convexhull2, 255)
img2_face_mask = cv2.bitwise_not(img2_head_mask)


img2_head_noface = cv2.bitwise_and(img2, img2, mask=img2_face_mask)
result = cv2.add(img2_head_noface, img2_new_face)

(x, y, w, h) = cv2.boundingRect(convexhull2)
center_face2 = (int((x + x + w) / 2), int((y + y + h) / 2))

seamlessclone = cv2.seamlessClone(result, img2, img2_head_mask, center_face2, cv2.NORMAL_CLONE)
```

“Code snippet 6”

Results

We now have a swapped image of both the actors.

References

1. Mthworld.wolfram.com (https://mathworld.wolfram.com/DelaunayTriangulation.html)
2. GeeksForGeeks (General queries)
3. OpenCV documentation (https://docs.opencv.org/master/)
4. Github for landmark detector (https://github.com/davisking/dlib-models/)
5. Face Swapping (https://www1.cs.columbia.edu/CAVE/publications/pdfs/Bitouk_SIGGRAPH08.pdf)