Project 2: Face Swap

Prateek Arora
Masters of Engineering in Robotics
University of Maryland, College Park
Email: abhi1625@umd.edu

Abhinav Modi
Masters of Engineering in Robotics
University of Maryland, College Park
Email: kmadhira@terpmail.umd.edu

Kartik Madhira
Masters of Engineering in Robotics
University of Maryland, College Park
Email: pratique@terpmail.umd.edu

Using 1 Late day

I. Introduction

The aim of this project is to implement an end-to-end pipeline to swap faces in a video just like Snapchats face swap filter. There are two phases to the project. Phase 1 deals with traditional face swap approach and Phase 2 deals with deep learning pipeline to swap faces. Each are explained in detail below.

Phase 1

II. Delaunay Triangulation

A. Facial Landmarks detection

The first step in the traditional approach is to find facial landmarks (important points on the face) so that we have one-to-one correspondence between the facial landmakrs. The reason to use facial landmarks instead of using all the points on the face is to reduce computational complexity. For detecting facial landmarks we have use dlib library built into OpenCV and python. Some sample outputs of Dlib is shown below.

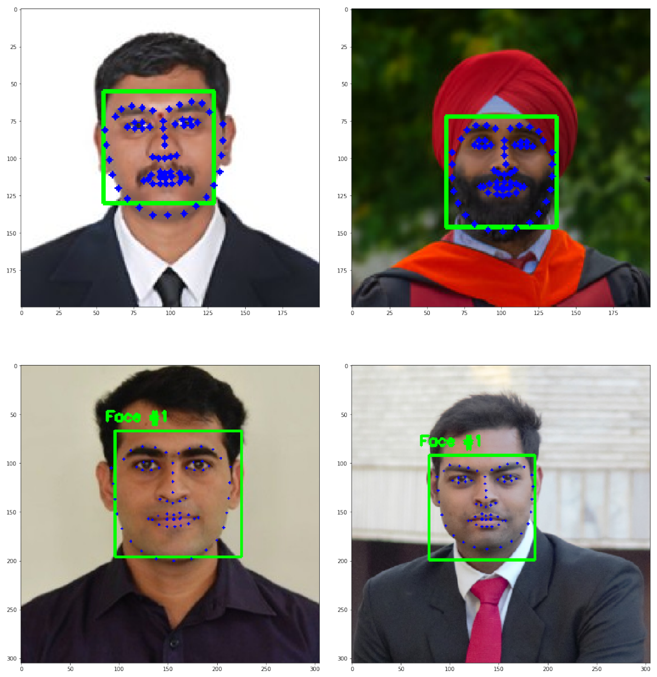

Figure 1: 68 Facial Landmark Detection using dlib OpenCV

B. Face Warping using Triangulation

After obtained facial landmarks, we need to ideally warp the faces in 3D, even though we don't have 3D information. In order to do so, we can consider a small area around each feature to be a 2D plane. These 2D plane can be transformed and into 2D planes of other face to approximate 3D information of face. Using the facial we perform triangulation. Triangulating or forming a triangular mesh over the 2D image is simple but we want to triangulate such that it's fast and has an "efficient" triangulation. We do so using Delaunay Triangulation and can be constructed in $(\mathcal{O}(n \log n))$ time. We want the triangulation to be consistent with the image boundary such that texture regions won't fade into the background while warping. We tried performing triangulation of the two images but in some cases the triangulation results were different i.e. triangle in one image didn't correspond to triangles in other image for corresponding features. To circumvent this issue we take the average of feature points and perform delaunay triangulation of them. Since we know the corresponding points which were averaged we get the same triangle correspondences in the same images. Delaunay Triangulation tries the maximize the smallest angle in each triangle. Triangulation output is shown in figure 2

We have used the getTriangleList() function in cv2.Subdiv2D class of OpenCV to implement Delaunay Triangulation. Now, we warp the destination face ($\mathcal{B}$) to the source face ($\mathcal{A}$) (we are using inverse warping so that we don't have any holes in the image) or to a mean face (obtained by averaging the triangulations (corners of triangles) of two faces). Warping is performed as explained in the following steps:

1) For each triangle in the target/destination face $\mathcal{B}$, the Barycentric coordinate for every pixel are calculated using the following equation.

$$\begin{bmatrix} \alpha_1 & \alpha_2 & . & . \\ \beta_1 & \beta_2 & . & . \\ \gamma_1 & \gamma_2 & . & . \end{bmatrix} = \begin{bmatrix} \mathcal{B}_{a,x} & \mathcal{B}_{b,x} & \mathcal{B}_{c,x} \\ \mathcal{B}_{a,y} & \mathcal{B}_{b,y} & \mathcal{B}_{c,y} \\ 1 & 1 & 1 \end{bmatrix}^{-1} \begin{bmatrix} x_{\mathcal{B}_1} & x_{\mathcal{B}_2} & . & . \\ y_{\mathcal{B}_1} & y_{\mathcal{B}_2} & . & . \\ 1 & 1 & . & . \end{bmatrix}$$

Here, the Barycentric coordinate is given by $\begin{bmatrix} \alpha_i & \beta_i & \gamma_i \end{bmatrix}^T$ where i$\in$ $[1, 2, 3..]$ is the pixel number. The Barycentric coordinate are calculated

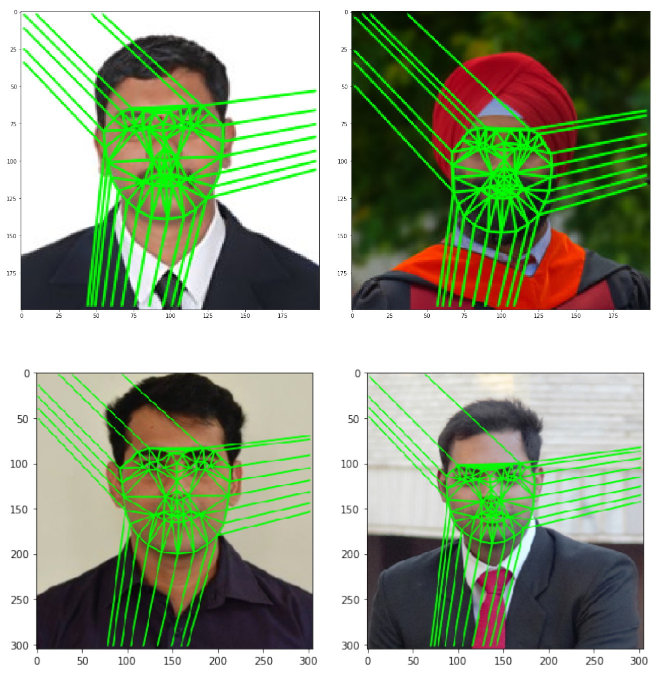
Figure 2: Triangulation output on both images

for every pixels in the image. The matrix after the equal is computed only once per triangle. In this matrix, abc represent the corners of the triangle and coordinates of the particular triangle corner respectively.

2) Since we have baycentric coordinates of all the pixels, we next determine which points in the image $\mathcal{B}$ lie inside the triangle by applying the following constraint:
   - $\alpha \in [0,1]$, $\beta \in [0,1]$, $\gamma \in [0,1]$,
   - $\alpha + \beta + \gamma = 1$

   We now have the filtered baycentric coordinates corresponding to the points lying inside the triangle.
3) Further we compute the corresponding pixel position in the source image $\mathcal{A}$ using the filtered barycentric equation shown in the last step but with a different triangle coordinates. This is computed as follows:

$$\begin{bmatrix} x_{\mathcal{A}} \\ y_{\mathcal{A}} \\ z_{\mathcal{A}} \end{bmatrix} = \mathcal{A}_{\Delta} \begin{bmatrix} \alpha \\ \beta \\ \gamma \end{bmatrix}$$

   Here, $\mathcal{A}_{\Delta}$ is given as follows:

$$\mathcal{A}_{\Delta} = \begin{bmatrix} \mathcal{A}_{a,x} & \mathcal{A}_{b,x} & \mathcal{A}_{c,x} \\ \mathcal{A}_{a,y} & \mathcal{A}_{b,y} & \mathcal{A}_{c,y} \\ 1 & 1 & 1 \end{bmatrix}$$

   After we obtain $\begin{bmatrix} x_{\mathcal{A}} & y_{\mathcal{A}} & z_{\mathcal{A}} \end{bmatrix}^T$,these values are converted to homogeneous coordinates as follows:

$$x_{\mathcal{A}} = \frac{x_{\mathcal{A}}}{z_{\mathcal{A}}} \text{ and } y_{\mathcal{A}} = \frac{y_{\mathcal{A}}}{z_{\mathcal{A}}}$$

4) Now, we copy back the value of the pixel at $(x_{\mathcal{A}}, y_{\mathcal{A}})$ to the target location. We have used scipy.interpolate.interp2d to perform this operation. The warped images are shown below.

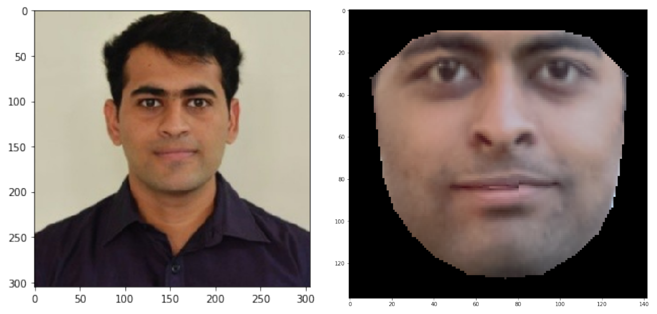
Figure 3: Left: Image A, Right: Image B warped to A

TRIANGULATON FACE SWAP RESULTS

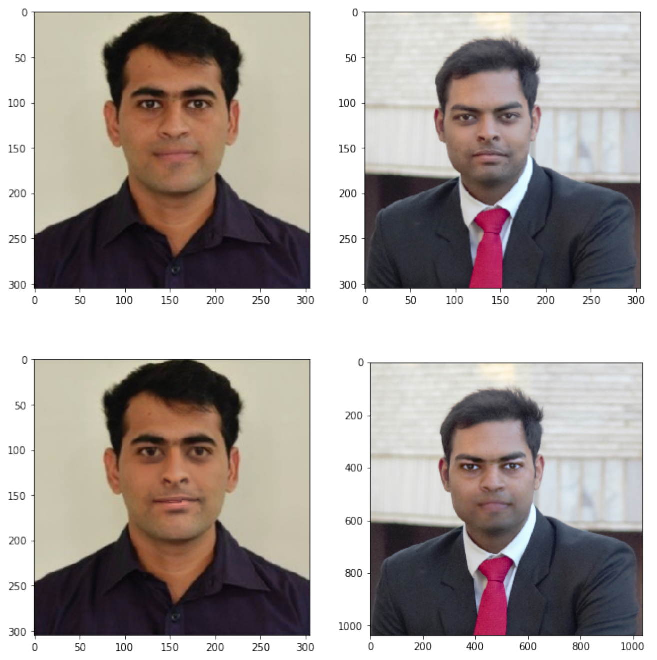
Figure 4: Clockwise from top left: Image A, Image B, Face swaped Images for custom set 1 using Triangulation

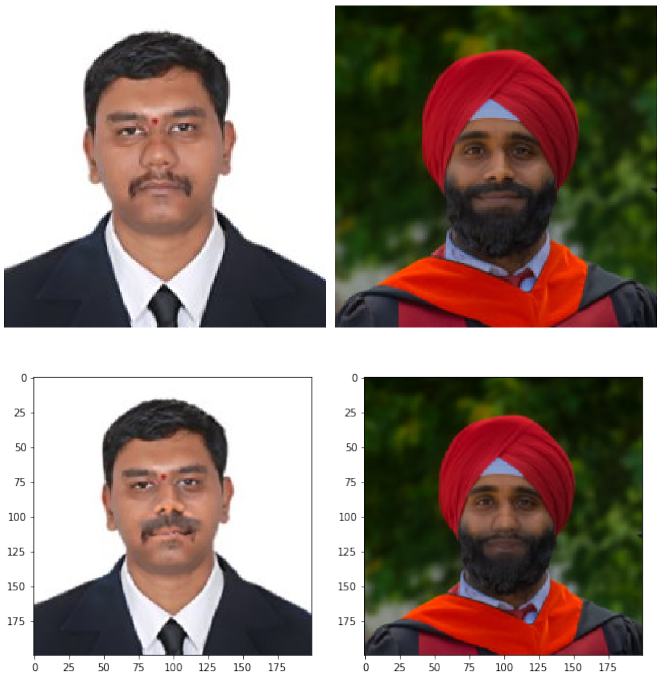

Figure 5: Clockwise from top left: Image A, Image B, Face swaped Images for custom set 2 using Triangulation

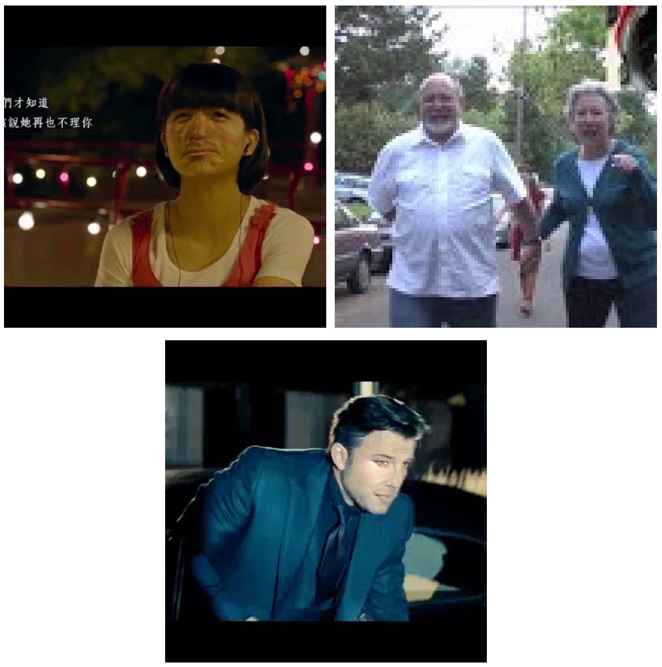

Figure 6: Clockwise from top left: Test set 1, Test set 2, Test set 3 result using Triangulation

III. Thin Plate Splines

In the previous section we used affine transformations to warp each corresponding triangle. This is not the best way to warp a human face due to its complexity. In this section we do the transformations using Thin-Plate Splines. Thin-Plate Splines are a spline based technique used for data interpolation and smoothing. The process is physically analogous to bending a thin sheet of metal. The goal is to find a mapping $f_x(x, y)$ and $f_y(x, y)$: $\mathbf{R}^2 \rightarrow \mathbf{R}^2$ such that:

$$f_x(x_i, y_i) = x_i^{'} \tag{1}$$

and

$$f_x(x, y) = a_1 + a_x x + a_y y + \sum_{i=1}^{p} w_i U(||(x_i, y_i) - (x, y)||_1) \tag{2}$$

The steps involve solving a system of linear equations of the form:

$$\begin{bmatrix} w_1 \\ w_2 \\ \vdots \\ w_p \\ a_x \\ a_y \\ a_1 \end{bmatrix} = \left(\begin{bmatrix} K & P \\ P^T & 0 \end{bmatrix} + \lambda I(p+3, p+3) \right)^{-1} \begin{bmatrix} v_1 \\ v_2 \\ \vdots \\ v_p \\ 0 \\ 0 \\ 0 \end{bmatrix} \tag{3}$$

where $K_{ij} = U(||(x_i, y_i) - (x_j, y_j)||_1)$, and is of the size $p \times p$. K is a symmetric matrix with the diagonal elements 0. K is plotted in the image shown below: $v_i = f(x_i, y_i)$ and

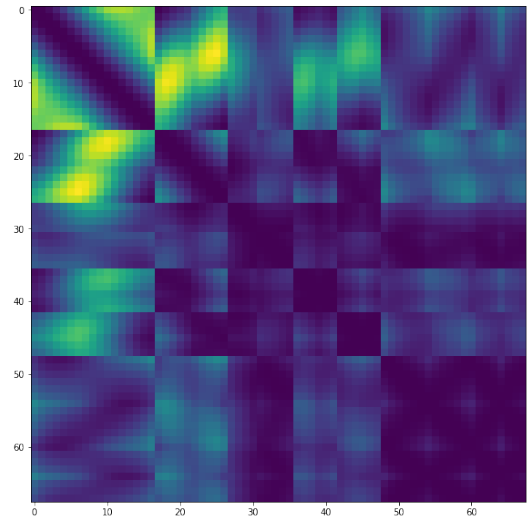

Figure 7: K matrix

i^{th} row of P is $(x_i, y_i, 1)$ and λ is a regularization term very close to 0.
The second step is to transform the pixels of the face in image $\mathcal{B}$ using the above developed TPS model and replace them in the image $\mathcal{A}$. The weights obtained after solving the above set of equations are used to find the warped location of the desired point using the equation(2).

The results obtained for transforming the face using TPS are shown below:

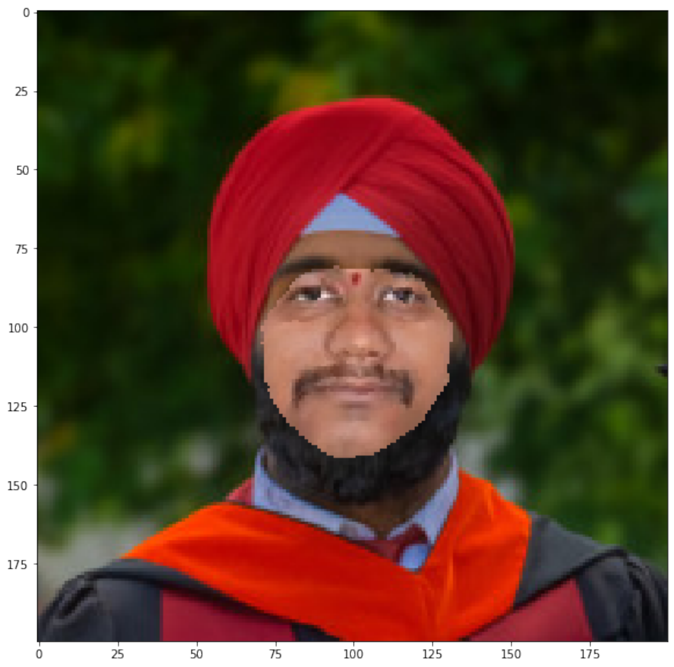
Figure 8: Warped image using Thin Plate Splines

IV. BLENDING

For blending we used third party code in [1], that corrects color of the warped face according to the destination image. We use **cv2.seamlessClone** for this operation. Figure 4 shows the effect after blending the target image with the destination image.

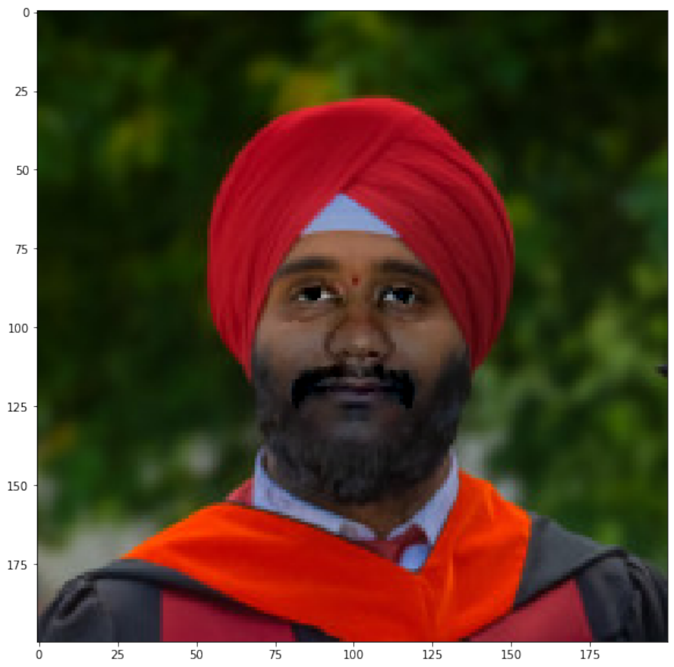
Figure 9: Final output of TPS after blending

The test sets given were passed through the given pippeline of TPS and the results are shown in the figs (10). Each test set was aimed towards testing dfferent aspects of the pipeline. The test 1 output is good in terms of raw facial landmark detection in each frame. There is less motion blur as compared to other test videos. Test 2 was to swap the two bigger faces in the image. The pipeline also gives satisfying results for this test set. The only problem we face is that facial landmark is not very robust. In some frames when only one side of the face is visible the dlib facial detection fails. A similar case occurs for very illumination of the face in test 3 video. The outputs for these videos can be improved by implementing a motion filtering or feature tracking algorithms like EKF(Estimated Kalman Filter) on top of the current pipeline.

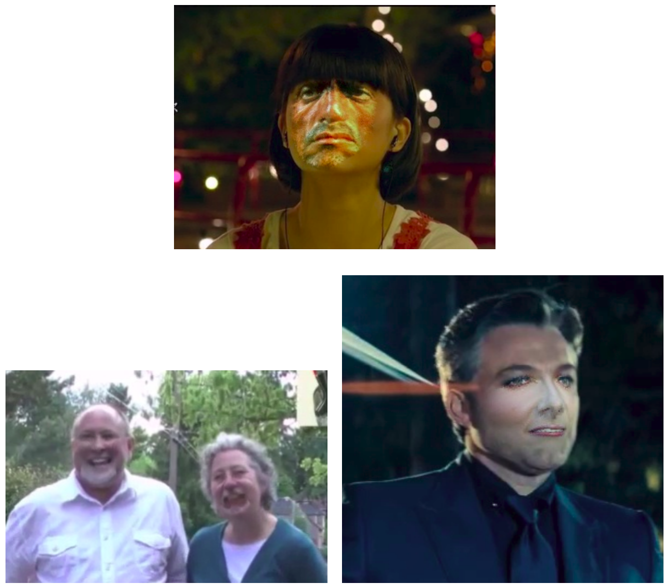
Figure 10: Output images showing swaped faces for Test1, Test2, Test3

V. PHASE 2: DEEP LEARNING

For this phase, an off-the-shelf model is trained to obtain face fiducials using deep learning. The approach used here implements a supervised encoder-decoder model to obtain the full 3D mesh of the face. Wr ran the code to obtain face fiducials/full 3D mesh.

A. *PRNet results for Test cases*

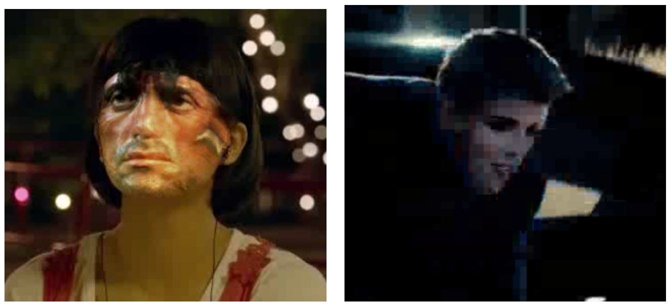

Figure 11: Clockwise from top left: Image A, Image B, Face swaped Images for Test sets using PRNet

VI. Conclusion

As compared to TPS we get better results in terms of transforming the face from one image to another. There is a slight issue while transforming the images using TPS. The transformation equation is highly nonlinear and thus we need to process each point(x,y) individually which is computationally very expensive. A way to model the process in another way which ensures faster runtime is needed. The value of λ is usually close to 0. But in our case, we have used λ = 50, which works better. As mentioned in the instructions the norm used to calculate K, using the function $U = -r^2 log(r^2)$, is first order. But according to Bookstein[[2]], 2^{nd} order norm works better provides a better intuition in the physical sense. Also the paper [[3]] provides a method of approximating the whole process of transforming points using Thin Plate Splines. The process uses matrix approximation to find the mapping for a larger number of points(Nystrom method). This approach utilizes the full set of basis functions to approximate the full set of target values. This approximation technique was not tried because of time constraints but was worth mentioning here and can be tried in the future. In the Deep learning based face swapping using PRnet, we get better results as compared to either of the traditional approaches. This is because, the faceswap using PRnet uses full 3D mesh grid of the face mask and then warps on this surface. TPS approach is similar to the 3D mesh creation but is approximation to a 3D surface than create 3D mesh itself. Since the traditional approaches use dlib facial features, not all the features of the face are represented in the warp. But 3D mesh created from the PRnet uses a 3D construction of the face on which the warped face image face is fully conserved.

References

[1] Faceswap - github. https://github.com/wuhuikai/FaceSwap.

[2] Fred L. Bookstein. Principal warps: Thin-plate splines and the decomposition of deformations. *IEEE Transactions on pattern analysis and machine intelligence*, 11(6):567–585, 1989.

[3] Gianluca Donato and Serge J Belongie. *Approximation methods for thin plate spline mappings and principal warps*. Citeseer, 2003.