Project 2: FaceSwap

Keshubh Sharma, Dushyant Patil

I. Introduction

The purpose of this project is to swap faces in a video file. The video files input are in 2 categories: One with a single face in them and second with 2 faces. One objective of the project is to swap the single face video with another face. Another objective of the project is to swap second face in 2 ace video by first face i.e. the output video should have two bodies with single face similar to the Jimmy Fallon video mentioned in the project guidelines notes. The objectives are to be achieved using classical approach of Delaunay triangulation and Thin plate spline approach with the help of 'dlib' library. The deep learning part uses the 2D images to generate a depth mesh of facial landmarks. The video used for single face in all frames is a video of Joe Biden giving a speech at the UN summit. The video with 2 faces is a recorded video of both the authors of this report.

II. Phase1: Classical Approach

The objective of Phase 1 is to implement Delaunay triangulation and Thin plate spline approach with the help of 'dlib' library to swap faces.The procedure to swap faces in classical approach is as follows:

1) Read frames from videos
2) Predict faces and generate facial landmarks using dlib
3) Implement correspondence between facial landmarks (using triangulation-Barycentric method or TPS weights method)
4) Find inverse mapping using the correspondences and interpolation.
5) Face replacement
6) Gaussian blending to get smooth output
7) Reduction of flickering

1. Delaunay Triangulation

The face landmarks detected using dlib are used to form triangles using Delaunay triangulation. The triangles are used to find correspondence between source image(video frame) and destination image. This correspondence is used for inverse mapping and face replacement. The in depth processes are as follows:

1) Face Detection and landmark creation: The face detector and shape predictor objects were created to with the help of *dlib.getfrontal_face_detector* and *dlib.shape_predictor*. The images are converted to grayscale and are fed to detector object to detect and store faces. These faces along with the grayscale image are fed to predictor objects to detect facial landmarks. The landmarks on a frame in video are shown in Fig 1
2) Triangle creation: The triangles are creates using facial landmarks in previous step. A bounding rectangle around the faces is created and input to a cv2.Subdiv2D function to create a subdiv object. Thee function *subdiv.getTrianglesList* is used to get triangles on the face connecting the landmarks. The output of triangulation is as shown in Fig 2
3) Triangle correspondence: The points detected in step 1 for both the faces are always generated in similar order and bear a one to one correspondence. But the function getTriangleList does not bear correspondence between triangles. To find mapping between triangles which detect similar regions on both the faces, a function *find_ind()*

Fig. 1: Face Detection and landmark creation

Fig. 2: Triangulation output

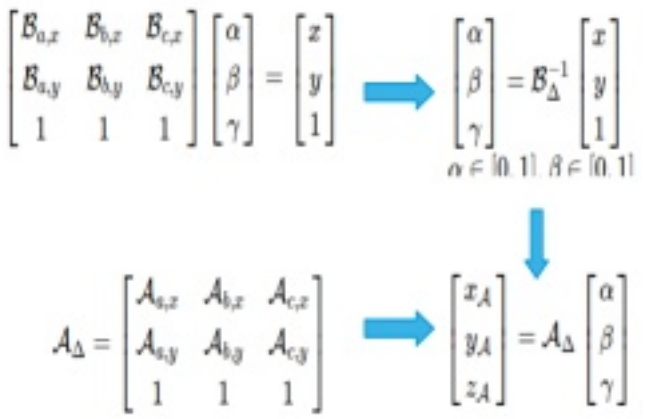

Fig. 3: Barycentric coordinates

Fig. 4: Destination Face cutout in source size

was created which finds indices of the landmarks. These indices are used to find triangles which have landmarks triplet with same indices. This triangle matching is used to find Barycentric coordinates of source and destination images.

4) Barycentric coordinates: The Barycentric coordinates are used to find inverse mapping between destination triangles and source triangles. The Barycentric coordinates of a point inside a triangle $[\alpha, \beta, \gamma]$ denote a appropriate mass distribution at the vertices to get center of mass of triangle at that point. To find these coordinates of points in source image we need to multiply the Cartesian coordinates of destination points by an Barycentric inverse matrix of destination triangles as shown in fig.3. These inverse multiplied Barycentric coordinates are multiplied by the Barycentic matrix of source image to find mapped coordinates in destination image for the source image points s shown in fig.3. This step is to be repeated for all the source points inside the source image triangles to get corresponding destination image point. Then the intensity values of all destination mapped points are assigned to source image copy. This copy is then used to get a cutout of the face which represents face of destination image in the size of source face as shown in fig 4.(face orientation changed than previos images for sake of presentation).
5) Face replacement: The image copy created in previous step in Fig4 is used to replace the face in source. The *cv.bitwise_and* operation is performed on source image using inverse of the mask in previous step. Then both these cutout are added using *cv2.add* as showin in Fig.5
6) Blending: The image obtained in previous step in Fig.5 is blended with the source using Gaussian blending to get a smooth swapped face as shown in Fig.6

2. Thin Plate Spline Interpolation method The face landmarks detected using dlib are used to find the warped mapping using interpolation as per thin plate spline (TPS) method. The step till landmark detection in same as triangulation. The image processing after the face cutout creation and face replacement are also same as that of the triangulation method.

Fig. 5: Face replacement

Fig. 6: Faceswap final output per frame

$$\begin{bmatrix} K & P \\ P^T & 0 \end{bmatrix} \begin{bmatrix} w_1 \\ w_2 \\ \vdots \\ w_p \\ a_x \\ a_y \\ a_1 \end{bmatrix} = \begin{bmatrix} v_1 \\ v_2 \\ \vdots \\ v_p \\ 0 \\ 0 \\ 0 \end{bmatrix}$$

Fig. 7: TPS equation

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

Fig. 8: TPS weight determination

In this section we will explain the TPS weight determination and inverse mapping part to find corresponding face points in destination image for the source face. The in depth processes are as follows:

1) Weights determination: The solution of the TPS model requires solving the following equation shown in Fig.7 Thus to find the warped coordinated of source face, the weights of the TPS equation are to be found using source landmarks and destination landmarks as per the equation shown in Fig. 8. Here $K_{ij} = U(\|(x_i, y_i) - (x_j, y_j)\|)$ and P is $(x_i, y_i, 1)$ where x_i, y_i belong to the source image landmark points and $v_i = (x_j, y_j)$ belong to destination landmark points. Here U for Euclidean distance 'r' is given by:

$$U(r) = r^2 log(r^2) \tag{1}$$

2) Mapping using TPS equation: The TPS weights found in previous step are used to find corresponding coordinate in destination face for each point in source face. The equation to find these mapped coordinates $(f_x(x, y), f_y(x, y))$ is as follows:

$$f(x, y) = a_1 + a_x x + a_y y + \Sigma w_i U(\|(x_i, y_i) - (x, y)\|) \tag{2}$$

This equation is used to find the mapped coordinates in destination image $(x^{'}, y^{'})$ for all the points (x,y) in source image face. Then the intensity values of points $(x^{'}, y^{'})$ in destination image are assigned to coordinates (x,y) in an image copy to get a face cutout as shown in Fig.9

3) Face replacement: The image copy created in previous step in Fig9 is used to replace the face in source. The *cv.bitwise_and* operation is performed on source image using inverse of the mask in previous step. Then both these cutout are added using *cv2.add* as showin in Fig.10

4) Blending: The image obtained in previous step in Fig.10 is blended with the source using Gaussian blending to get

Fig. 9: Face cutout using TPS

Fig. 10: TPS Face replacement

Fig. 11: TPS final output per frame

Fig. 12: Source video frame

a smooth swapped face as shown in Fig.11

3. Output images from classical approach

- The input frame (source) and destination are shown in Fig 12 and Fig 13
- The output of 2 frames for single face using triangulation is shown in Fig 14 and Fig 15
- The output of 2 frames for single face using TPS is shown in Fig 16 and Fig 17
- The input of 2 frames for 2 faces is shown in Fig 18
- The output of 2 frames for 2 faces using triangulation is shown in Fig 18 and Fig 19

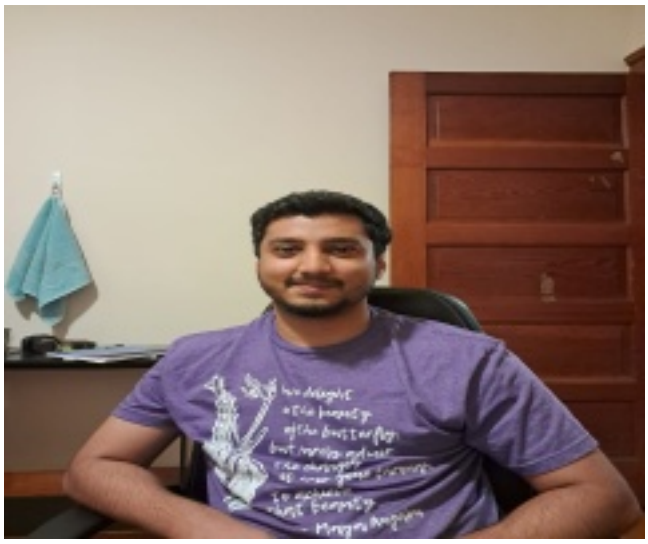

Fig. 13: Destination image

Fig. 14: Triangulation output 1

Fig. 15: Triangulation output 2

Fig. 16: TPS output 1

Fig. 17: TPS output 2

Fig. 18: Input for 2 face code

Fig. 19: Output1 TRI for 2 face

Fig. 20: Output2 TRI for 2 face

III. Phase2: Deep Learning Approach

The Deep learning implementation for face swap is as follows:

1. Face Swap implementation using Model output points For this approach we used the 3DDFA (3D Dense Face Alignment) face alignment method to align the 3D mesh of face to the 2D image on anchor points to obtain a 3D mapping of facial features irrespective of the actual face direction of subject. The actual implementation of said paper is given in the github repository provided to us. In the said code we cloned the code and the pre-trained model which were done on a really robust dataset using a much powerful system to achieve great accuracy in terms of calculating the fiducial points, pos of subject face relative depth on the input image. The output files consists of depth representation, pos representation, Projected Normalized Coordinate Code (PNCC) representation 3DDFA representation. We took the output from the PRNet stored in a txt file which contains the fiducial coordinates and relative

Fig. 21: 3D Dense Face Alignment

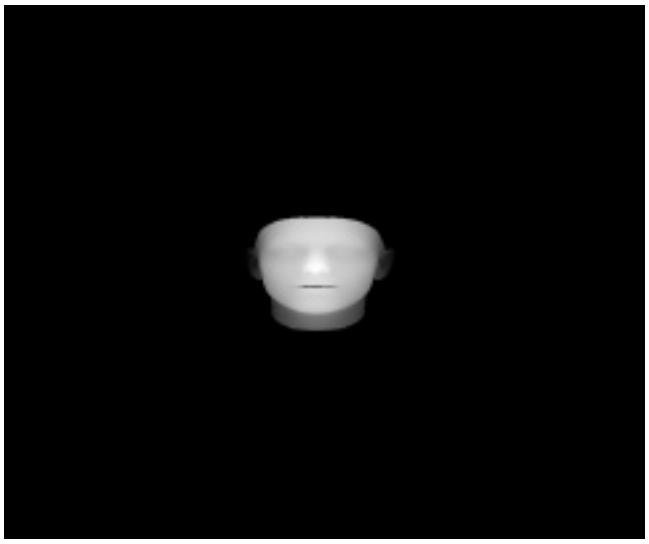

Fig. 22: Relative depth of input image

depth values and used the coordinates as input to a TPS wrapper scrpit such that the deepnet is used for all the frames of a video and then a destination face is swapped and blended in which is later compiled back into a video. The face fiducials, pose estimation, depth image and pncc are as shown in Fig.21, 22, 23, 24

2. Alternate FaceSwap implementation An alternative that we thought of was based on the fact that PRNet gives us the anchor points in 3D such that it simply corresponds to a cylinder. After figuring the anchor points we can create a feature patch map by essentially taking **d x d** patch along each anchor point and concatenating them into a **(N*d) x (N*d)** map. We can then impose this patch map onto the destination image mesh and warp it to fully cover the respective anchor points thus achieving a precise faceswap.

The results of DL face swap implementation are shown in Fig.25 and Fig.26

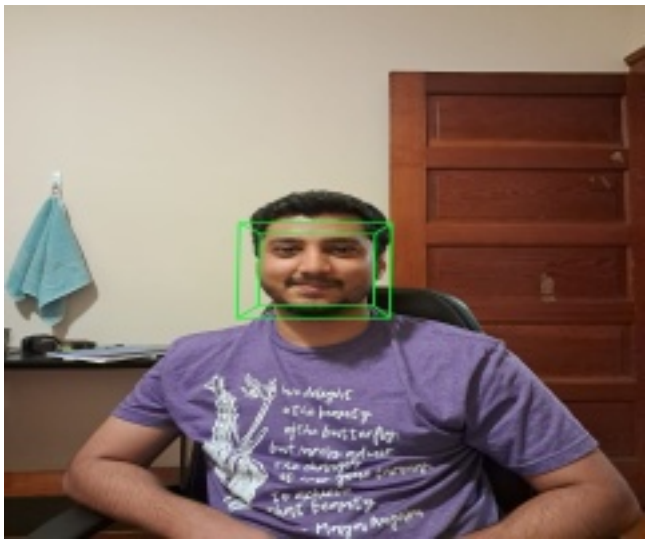

Fig. 23: pos representation of input image

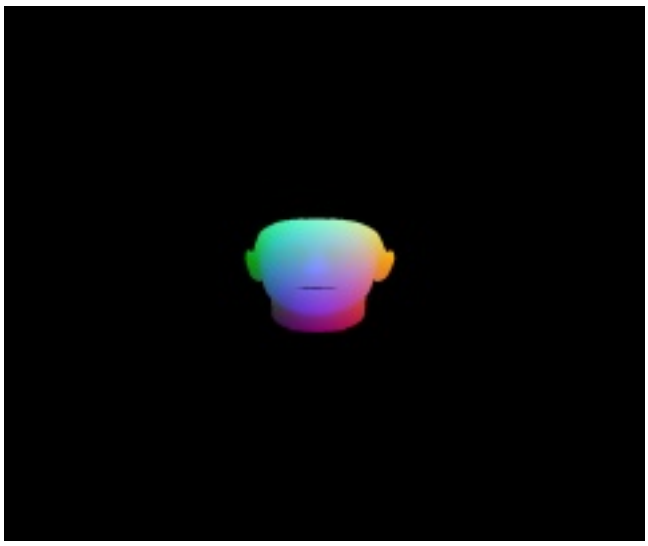

Fig. 24: Projected Normalized Coordinate Code

Fig. 25: DL Output1 for single face

Fig. 26: DL Output2 for single face

IV. Result Comparison

1. The output videos for triangulation show least amount of flickering while output for deep learning approach shows comparatively higher amount of flickering
2. The output quality of swapped video using TPS is better than triangulation and DL swapping
3. The triangulation code swaps the faces faster tha nrest of the two methods (Execution speed is better for triangulation). For 2 face swap code, TPS is significantly slower than TPS

V. Failure cases - Areas of concern

1. When matrix operations were performed to achieve mapping using TPS weights and K matrix, the coordinate values would be out of bound for a few points. This was resolved by replacing a few matrix operations by looping over all matrix elements
2. The DL output shows a lot of flickering