Project 2 - FaceSwap Report

Ameya Patil
Department of Computer Science
University of Maryland
College Park, Maryland 20740

Sigurthor Bjorgvinsson
Department of Computer Science
University of Maryland
College Park, Maryland 20740

I. Introduction

The aim of this task was to swap faces in a seamless manner in a video having 2 or more faces, or simply to replace a face in a video with some static image of another face. This task has 4 stages - face detection, face warping and swapping, seamless face blending, and motion flicker filtering. Of these 3 stages, face detection was performed using a pre-trained model for detecting faces in images. The second stage of face warping and swapping has a classical approach and a deep learning based approach, the implementations for which have been described below in detail. Both the classical and deep learning approaches were combined in a single tool which accepts a 'swapType' argument to decide which warping and swapping algorithm to use. This is followed by a description of our implementation of the blending and motion flicker filtering. We also describe the problems that we faced along the way, and some results to show for the successful and failed cases of face swapping.

II. Phase 1: Classical Approach

The classical approach for face warping and swapping was implemented using 2 different techniques - triangulation and thin plate spline, both of which have been described below.

A. Finding Faces

We use dlib to get faces from frames and images (from now on 'image') using a pre-trained model retrieved from [5]. This model returns 68 face landmarks for each face in a image which we use in all our approaches.

B. Triangulation

Triangulation, as the name indicates, uses triangulation to map one face to another. To swap faces we begin with two faces where we have found 68 landmarks. We create a map between the landmarks, coordinates wise, in faceA (destination face) to faceB (source face). By applying Delaunay Triangulation to create a mesh of triangles connected with the landmarks using the SubDiv2d function in OpenCV. From this we get a list of triangles that are contained in the landmarks of faceA. We create a copy of this list where each coordinate is translated using the mapping to the corresponding landmark in faceB.

For each triangle, we want to find the mapping from the source image to the destination image. To do this we loop through the triangles and for each pixel in the faceA, we find out what pixel in the faceB should go to that pixel. Using Barycentric coordinates we figure out what pixels are in each triangle of faceA. Using the inverse Barycentric Matrix of the triangle in faceA, we get barycentric coordinates which we plug into the barycentric matrix of faceB. From this we get a list of pixel mapping, faceA pixel : faceB pixel.

Each pixel mapping from faceB is a float so by using interpolation of faceB we extract pixel values of float indexes.

We keep track of each pixel mapping we identified (more than 98% of total pixels) because there were cases where a few pixels were not identified to be part of any triangle. We expect that this is happening because of the approximation numpy inverse of the Barycentric matrix and that the pixels are on a edge case location of the triangle. These 2% of unmapped pixels are mostly attributed because of spacing created by the opencv convexhull/ploy fill function (uncomment code in line 93 in lib/triangulation.py to see mapping mask). The pixels that did not map but were in the faceA we filled by taking the median of 9 pixel box around the pixels in question after moving the mapped pixels.

After all face has been transferred we apply Poisson Blending which is described in later section

C. Thin Plate Spline

Thin plate spline is a mathematical model for shaping a plane thin metal plate to perfectly fit an arbitrary surface. Although, this is a task in 3D in real life, the mathematical model which describes such an operation can be used to warp a face to transform its structure in 2D. The actual TPS model takes as input, a set of control points on the original plane thin metal plate, and the corresponding set of control points on the arbitrary surface to which the plate has to be fit. Using these inputs, the model estimates its parameters such that the plate can be bent to match the arbitrary surface as smoothly as possible, i.e. with minimum bending energy. Having estimated the model parameters, the model now estimates the height that each point on the plate needs to be moved so as to match the structure of the arbitrary surface.

This model can be analogously applied to the case of 2D images. We have the corresponding pairs of facial landmarks in the source and destination face. Using these landmarks, we estimate the model parameters and then, instead of estimating

the height that each pixel needs to be moved, we estimate the displacement of the pixel from its original position along both the X and Y directions, to get the new location of that pixel as per the new face structure. We thus get a mapping for the pixels belonging to one face, to the pixels belonging to the other face.

For estimating the model parameters, we need to compute a square matrix containing the data of all the control points in one image. The parameters are then estimated by inverting this matrix and multiplying it with a matrix containing all the corresponding control points of the destination image. **Before inverting the square matrix however, we add a diagonal matrix to it with positive diagonal values very close to zero.** The reason behind doing this is that the matrix is symmetric with diagonal elements all zeros. The algorithm used to invert matrices is an iterative one with multiple row operations or column operations. Having zeros for the diagonal elements makes this task of finding inverse more time consuming, because we can only create ones in them by using some linear combination of the other rows or columns and not by simply scaling that particular row or column. This does not affect the answer much if the value added is very close to 0, it just speeds up the iterative process.

The TPS model uses some kind of metric for measuring the distance of each pixel from every control point. There were some references which used the L1 norm metric for this distance, while some used the L2 norm. **We went ahead with the L2 norm as we did not get good results with the L1 norm.**

Again, as was done in triangulation, we performed inverse warping, i.e. we estimated the model parameters for warping the destination face to the source face. After this mapping was found, we copied the pixels from the source face to the corresponding locations in the destination face.

III. Phase 2: Deep Learning Approach

For the deep learning approach we used the pre-trained model from [6] to get a texture mesh. We decided to use our own landmarks retrieved from the dlib model used in phase 1. We used the demo-texture.py code as the baseline and stripped from it what we didn't need. We created a function in lib/prnet.py which called other code in lib/phase2Utils. We noticed some issues with the model, in our opinion made the inclusion of the ears made the results worse. We created a mask and cut out the face after the prnet had applied it on the destination image. This face applied to the final destination image with a mask that only included pixels within the landmarks. The final result what then blended using the Poisson blending algorithm from Phase 1.

IV. Blending

We used OpenCV API cv2.seamlessClone() to blend the source face with the destination face. This API internally implements the Poisson blend algorithm. This required creation of a mask which filters out only the destination face region. This mask was created using OpenCV API cv2.getConvexHull() to get a convex hull surrounding the facial landmarks around the destination face control points. **By noting the sequence in which control points are generated by the dlib face detector,** we were able to identify those facial landmarks which are most likely to decide the convex hull and thus pass only those points to cv2.getConvexHull() API. This helped in reducing the execution time. The next step was to mark those pixels in the image which were inside the convex hull mask which was done using the OpenCV API cv2.pointPolygonTest(). **Instead of checking every pixel in the image for inclusion in the mask, a bounding box for the convex hull was computed** so as to reduce the search space which further helped us to reduce the execution time. **The mask created in this way was padded on the inside, meaning the actual mask was slightly smaller than the convex hull of the face.** This was done to avoid bleeding of the face into the background which was found to cause hard edges at the jaw line.

V. De-flicker

The first step here was to identify what exactly caused the flicker, so that we could fix it. **The flicker was due to the inaccuracies of the face detector algorithm, combined with the face warping scheme used.** If the correspondences of the facial landmarks on both faces was not very accurate, or the transformation estimated by triangulation or tps method was not very accurate, the smooth movement of the original faces would be transformed to an abruptly changing movement of the swapped faces. The swapped regions of the individual video frames had flicker inside those regions only, so the flicker filtering had to be applied only inside those regions. We used a crude form of an algorithm described in [3] for motion flicker filtering which basically replaces the pixels in the flickering region of the current frame with a weighted average of the co-located pixels in the current frame and the previous frame. Once again, we created masks for each frame which marked out the faces in the frame, as these were the regions in the frames which had the flicker. **The masks were feathered so as to avoid having hard edges along the face contour.**

This approach towards flicker filtering does cause a little amount of blurriness, but overall the blurry appearance looks better than the flickering appearance. Another issue in this implementation was that **the 2 faces in the frame were not getting detected in a consistent sequence.** This was causing issues because the flicker filtering algorithm needed to average co-located pixels (in the co-located faces) in the current and previous frames. If the sequence of face detection was not consistent across frames, we map incorrect faces across the current and previous frames. This was handled by rearranging the list of detected biggest 2 faces as per their position in the frame, from left to right. The biggest 2 faces were identified by computing the face area using OpenCV API cv2.ContourArea()

VI. Results

A. Successful Swap Frames

Some of the frames from the successfully swapped videos are presented in Figures 1, 2 and 3

B. Failed Swap Frames

At times, the face detector could not detect faces in the video frames, in that case, we had no choice but to perform no face swap and simply write the original frame to the output video. We do not present screenshots from those videos here. For the cases where faces were detected but the faceswap failed, we have some examples from our own collected video data and also from the provided test videos, which are shown in Figures 4, 5 and 6.

The failure as shown in Figure 4 seem to have happened because of the small size of the faces compared to the video frame size. Our guess is that for the affine transformation or the TPS to work fine, the facial landmarks should be well spaced out which does not happen in Test2.mp4, hence the swap does not seem to have happened efficiently and only half the faces seem to have been swapped.

The failure as shown in Figure 5 happened because the source person turned his face away from the camera exposing only one half of his face profile to be mapped to the entire face of the destination person.

The failure as shown in Figure 6 happened because of the fact that the lady in the video opened her lips which made her teeth visible, while the static image of Rambo does not show his teeth. Thus the resulting swapped face had the lips stretched out unnaturally to fill up the space occupied by the teeth of the lady. This is similar to, and infact a sub-case of the issue manifesting in the previous example where half face profile of the source face was required to be mapped to a full face profile of the destination face.

VII. Conclusion

By working on this project, we have learned about face detection, different methods of warping a set of points to a different configuration. We also appreciate the way this task builds up on the previous project of finding a projective transform, by understanding that face warping can be implemented using such projective transforms on small parts (triangles) of the image.

References

[1] https://profs.etsmtl.ca/hlombaert/thinplates/

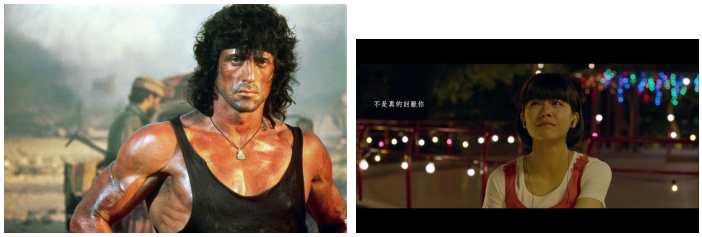
[2] https://www.learnopencv.com/seamless-cloning-using-opencv-python-cpp/

[3] Standard-Compliant Low-Pass Temporal Filter to Reduce the Perceived Flicker Artifact
Amaya Jimnez-Moreno, Eduardo Martnez-Enrquez, Vinay Kumar, Fernando Daz-de-Mara
Available: https://ieeexplore.ieee.org/document/6876206/authorsauthors

[4] https://nofilmschool.com/2018/02/flickering-ruining-your-video-fix-will-only-take-you-30-seconds

[5] http://dlib.net/files/shape_predictor_68_face_landmarks.dat.bz2

[6] https://github.com/YadiraF/PRNet



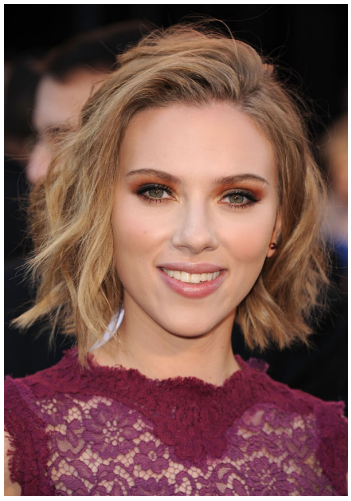
(a) Inputs for Test1.mp4

(b) Result with Triangulation faceswap

(c) Result with TPS faceswap

(d) Result with PRNet faceswap

Fig. 1. Successful results for Test1.mp4. All the frames are at the same timestamp



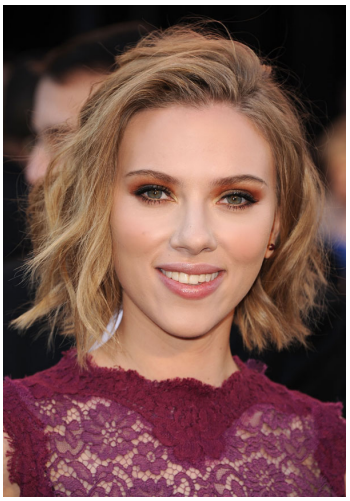
(a) Inputs for Test3.mp4

(b) Result with Triangulation faceswap

(c) Result with TPS faceswap

(d) Result with PRNet faceswap

Fig. 2. Successful results for Test3.mp4. All the frames are at the same timestamp



(a) Inputs for Test3.mp4

(b) Result with Triangulation faceswap

(c) Result with TPS faceswap

(d) Result with PRNet faceswap

Fig. 3. Successful results for Test3.mp4. All the frames are at the same timestamp

(a) Input for Test2.mp4

(b) Result with Triangulation faceswap

(c) Result with TPS faceswap

(d) Result with PRNet faceswap

Fig. 4. Failed results for Test2.mp4 - only one half portion of the faces seem to be swapped. All the frames are at the same timestamp

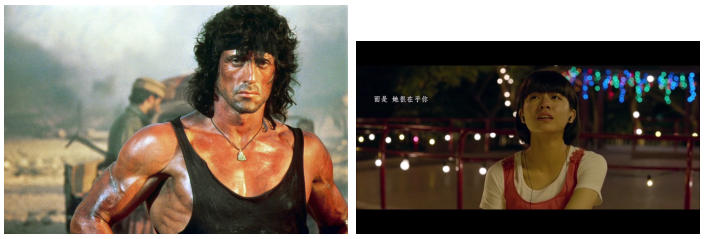
(a) Inputs for Local Data.mp4

(b) Result with Triangulation faceswap

(c) Result with TPS faceswap

(d) Result with PRNet faceswap

Fig. 5. Failed results for local data - the source person turned his face away from the camera, exposing only half the face profile leading to messed up mapping between matching facial feature points. All the frames are at the same timestamp



(a) Inputs for Test1.mp4

(b) Result with Triangulation faceswap

(c) Result with TPS faceswap

(d) Result with PRNet faceswap

Fig. 6. Failed results for Test1.mp4 - the actual video frame has the teeth of the lady showing while the swapped frame has no teeth showing. All the frames are at the same timestamp