Computer Vision

View Morphing of Human Faces

Keio University

STEFAN Vlad 82123434
25/07/2022

1 Introduction

The purpose of this project was to implement a view morphing algorithm similar to the one detailed in [4]. Morphing techniques seek to seamlessly blend multiple views of the same scene to create unseen intermediary views. These techniques are appealing because they only use 2D transfomations and do not require 3D reconstruction of the scene.
Although view morphing can be applied to any scene, this project specifically focused on the morphing of human faces.
View morphing can be divided in three distinct steps. The first part of the process is the projection of both images on the same plane to be able to treat both images as if they were parallel views of the same scene. Afterwards, a linear interpolation is computed between both images. The final step of the algorithm is a projection of the blended image into the desired plane.
The remainder of this report is structured as follows: First, a detailed explanation is given for the implementation of every step of the algorithm. Afterwards, experimental results are shown and discussed.
The full code is available at the following link. It was implemented in Python 3.9 with the OpenCV, dlib, and numpy libraries.

2 Implementation

As discussed in the introduction, view morphing is composed of three distinct steps. In [4], they are referred to as prewarping, morphing and postwarping. The rest of this section details the implementation of each of these steps.

2.1 Prewarping

The purpose of this step is to align both images on the same plane. A similar process was applied in many of the previous reports. The procedure is performed as follows:

1. Find keypoint correspondences between both images;
2. Compute the fundamental matrix with the corresponding feature points;
3. Use the fundamental matrix to obtain two projective transforms to align the image planes;

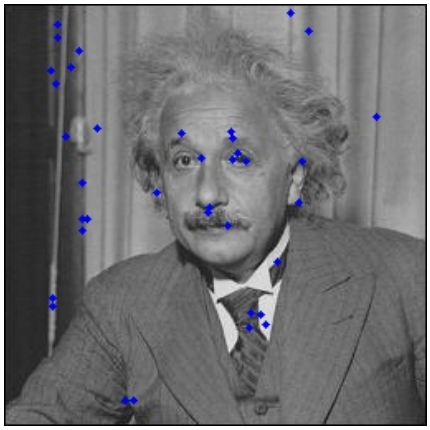
2.1.1 Keypoint matching

Multiple methods were tried for this step. Similarly to what was applied in previous reports, the first method consists in using the OpenCV implementation of the SIFT algorithm [3] to acquire feature points. Those feature points are then matched and nearest neighbour distance ratio is used to filter ambiguous matchings. An example is shown in figures 1a and 1b.

```
def find_features(gray1, gray2):
    """
    Find keypoint correspondences between 2 images
    :param gray1: input grayscale image
    :param gray2: input grayscale image
    :return: matching keypoints
    """
    # find the keypoints and descriptors with SIFT
    sift = cv.SIFT_create()
    kp1, des1 = sift.detectAndCompute(gray1, None)
    kp2, des2 = sift.detectAndCompute(gray2, None)

    # FLANN parameters
    FLANN_INDEX_KDTREE = 1
```

```
    index_params = dict(algorithm=FLANN_INDEX_KDTREE, trees=5)
    search_params = dict(checks=50)
    flann = cv.FlannBasedMatcher(index_params, search_params)
    matches = flann.knnMatch(des1, des2, k=2)
    pts1 = []
    pts2 = []

    # ratio test as per Lowe's paper
    for i, (m, n) in enumerate(matches):
        if m.distance < 0.8 * n.distance:
            pts2.append(kp2[m.trainIdx].pt)
            pts1.append(kp1[m.queryIdx].pt)

    pts1 = np.int32(pts1)
    pts2 = np.int32(pts2)

    return pts1, pts2
```

Listing 1: Obtain keypoints with the SIFT algorithm



(a) Keypoints found by SIFT on a picture of Albert Einstein

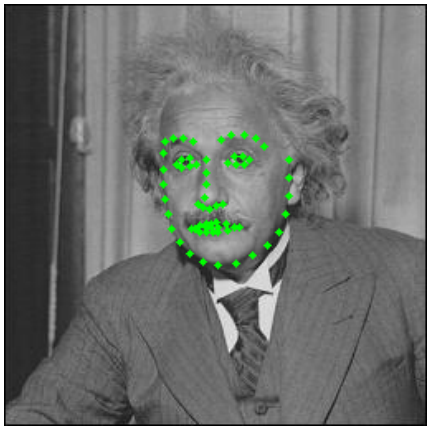
(b) Keypoints found by SIFT on a picture of Donald Trump

Since the objective is to apply view morphing to human faces, the second method to acquire keypoints uses a pretrained machine learning model available through the dlib library. This model is capable of detecting 68 facial landmarks. An example of output generated by this model is shown in figures 2a and 2b.

```
def find_face_features(gray1, gray2):
    """
    Find the facial correspondences between 2 images
    :param gray1: input grayscale image
    :param gray2: input grayscale image
    :return: lists of correspondent feature points
    """

    # Load facial features detector
    path = "shape_predictor_68_face_landmarks.dat"
    detector = dlib.get_frontal_face_detector()
    predictor = dlib.shape_predictor(path)

    # detect the faces present in each image
    rects1 = detector(gray1)
    rects2 = detector(gray2)

    # Predict the facial features and transform them to numpy arrays
    shape1 = []
    for i, rect in enumerate(rects1):
```

```
        shape1 = predictor(gray1, rect)
        shape1 = face_utils.shape_to_np(shape1)

    shape2 = []
    for i, rect in enumerate(rects2):
        shape2 = predictor(gray2, rect)
        shape2 = face_utils.shape_to_np(shape2)
    return shape1, shape2
```

Listing 2: Obtain keypoints with the dlib model



(a) Facial landmarks on a picture of Albert Einstein

(b) Facial landmarks on a picture of Donald Trump

2.1.2 Fundamental matrix

The fundamental matrix contains a combination of the intrinsic parameters of the camera and information about the second position of the camera compared to the first one.
It can easily be obtained with the `findFundamentalMat` function from OpenCV, which takes the previously found feature points as input.

```
# Compute the fundamental matrix
F, mask = cv.findFundamentalMat(pts1, pts2, cv.FM_LMEDS)
```

Listing 3: Compute the fundamental matrix

2.1.3 Projective transforms

A projective transform, also known as a homography, is defined by a 3×3 matrix which specifies the transformation of pixel coordinates.
OpenCV provides the `stereoRectifyUncalibrated` function, which takes the fundamental matrix and keypoints as input, and outputs homography matrices required to align both images.

```
def prewarp(gray1, gray2):
    """
    Compute the homography matrices to rectify 2 grayscale images
    :param gray1: input grayscale image
    :param gray2: input grayscale image
    :return: Homography matrices to rectify the images
    """
    # Find feature points
    pts1, pts2 = find_features(gray1, gray2)

    # Compute the fundamental matrix
    F, mask = cv.findFundamentalMat(pts1, pts2, cv.FM_LMEDS)
```

```
    # We keep only inlier points
    pts1 = pts1[mask.ravel() == 1]
    pts2 = pts2[mask.ravel() == 1]

    # Compute the homography matrices
    h1, w1 = gray1.shape
    h2, w2 = gray2.shape
    _, H1, H2 = cv.stereoRectifyUncalibrated(np.float32(pts1), np.float32(pts2), F, imgSize=(
    w1, h1))

    return H1, H2
```

Listing 4: Compute homography matrices

2.2 Morphing

A simple way to blend two images I and J would be to take a weighted mean of every pixel value:

$$M(x, y) = (1 - \alpha)I(x, y) + \alpha J(x, y) \tag{1}$$

The problem with this approach is that the results are quite unnatural, since it is unlikely for corresponding pixels to be blended together.
Thus, to improve the result, it is necessary to first obtain correspondences between pixels. These correspondences can be used to determine intermediary positions:

$$x_m = (1 - \alpha)x_i + \alpha x_j \tag{2}$$
$$y_m = (1 - \alpha)y_i + \alpha y_j \tag{3}$$

The pixel value at (x_m, y_m) is obtained by blending the images:

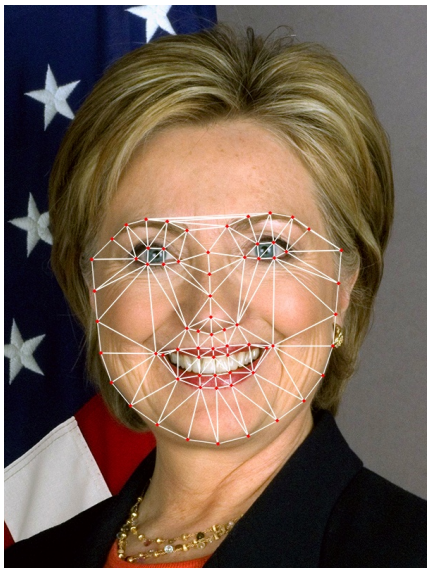
$$M(x_m, y_m) = (1 - \alpha)I(x_i, y_i) + \alpha J(x_j, y_j) \tag{4}$$

Finding correspondences between all pixels in both images is not realistic. As such, the method detailed in [1] was implemented. Starting from the two sets of facial landmarks detected previously, a third set is computed by taking the weighted average. Delaunay triangulation is then performed on the previously computed sets of keypoints. This operation outputs a list of triangles, with the input points as corners and with no point from the input set in the circumcircle of any triangle. The triangles created in both images capture roughly the same areas of the face as can be seen in images 3a and 3b. To perform the blending, triangles from both images are mapped to the corresponding triangle in the morphed image with an affine transformation and their pixel values are alpha blended with equation 4. The result of such an operation is illustrated in figure 4.

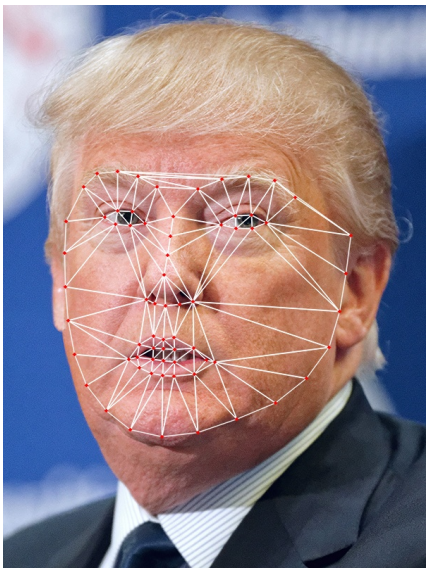
```
def morph_triangle(img1, img2, img, t1, t2, t, alpha):
    """
    Blend corresponding triangular regions in 2 image
    :param img1: input image
    :param img2: input image
    :param img: output image
    :param t1: trianle in img1
    :param t2: triangle in img2
    :param t: triangle in output image
    :param alpha: blending parameter (float between 0.0 and 1.0)
    """
    # Find bounding rectangle for each triangle
    r1 = cv.boundingRect(np.float32([t1]))
    r2 = cv.boundingRect(np.float32([t2]))
    r = cv.boundingRect(np.float32([t]))

```

```
    # Offset points by left top corner of the respective rectangles
    t1_rect = []
    t2_rect = []
    t_rect = []

    for i in range(0, 3):
        t_rect.append(((t[i][0] - r[0]), (t[i][1] - r[1])))
        t1_rect.append(((t1[i][0] - r1[0]), (t1[i][1] - r1[1])))
        t2_rect.append(((t2[i][0] - r2[0]), (t2[i][1] - r2[1])))

    # Get mask by filling triangle
    mask = np.zeros((r[3], r[2], 3), dtype=np.float32)
    cv.fillConvexPoly(mask, np.int32(t_rect), (1.0, 1.0, 1.0), 16, 0)

    # Apply affine transformations to small rectangular patches
    img1_rect = img1[r1[1]:r1[1] + r1[3], r1[0]:r1[0] + r1[2]]
    img2_rect = img2[r2[1]:r2[1] + r2[3], r2[0]:r2[0] + r2[2]]

    size = (r[2], r[3])
    warp_image1 = apply_affine_transform(img1_rect, np.float32(t1_rect), np.float32(t_rect),
    size)
    warp_image2 = apply_affine_transform(img2_rect, np.float32(t2_rect), np.float32(t_rect),
    size)

    # Alpha blend rectangular patches
    img_rect = (1.0 - alpha) * warp_image1 + alpha * warp_image2

    # Copy triangular region of the rectangular patch to the output image
    img[r[1]:r[1] + r[3], r[0]:r[0] + r[2]] = img[r[1]:r[1] + r[3], r[0]:r[0] + r[2]] * (1 -
    mask) + img_rect * mask
```

Listing 5: Blend two triangular regions from different images



(a) Delauny triangulation on the face of Hillary Clinton



(b) Delauny triangulation on the face of Donald Trump

Figure 3: Delauny triangulation

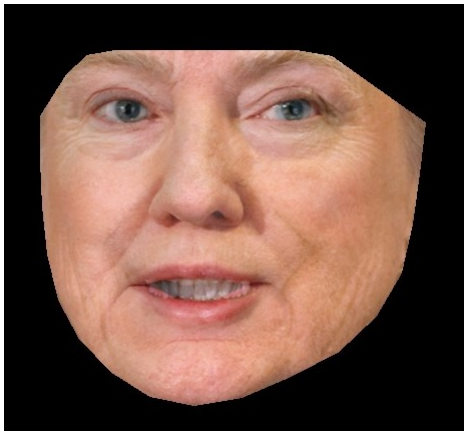


Figure 4: Face morph

2.3 Postwarping

The last step of the view morphing process is the postwarping. For this part of the process, a projective transform is applied on the image to bring it to a desired plane. This step is quite complex, and no method was found to perform it automatically in a reliable way. As such, inspired by the method used in [2], a homography is generated by manually selecting points and generating a homography matrix with the `findHomography` function of OpenCV.

```
def homography_points(m_points, p_points):
    """
    Obtain postwarp homography using selected points
    :param m_points: list of points in the morphed image (at least 4 points required)
    :param p_points: list of corresponding points after postwarp
    :return: postwarp homography matrix
    """
    m_points = np.asarray(m_points, dtype=np.uint8)
    p_points = np.asarray(p_points, dtype=np.uint8)

    pts_src = []
    pts_dest = []
    for i in range(0, len(m_points)):
        pts_src.append([m_points[i, 0], m_points[i, 1]])
        pts_dest.append([p_points[i, 0], p_points[i, 1]])

    pts_src = np.asarray(pts_src, dtype=np.uint8)
    pts_dest = np.asarray(pts_dest, dtype=np.uint8)

    H_s, _ = cv2.findHomography(pts_src, pts_dest)
    return H_s
```

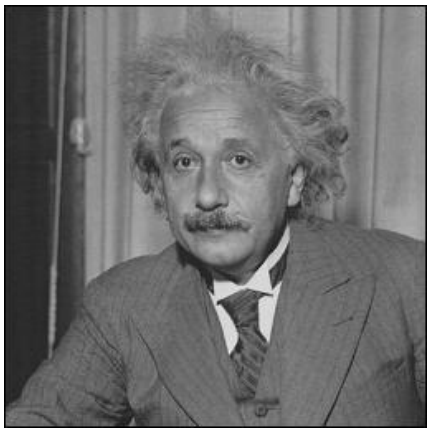
Listing 6: Obtain a homography matrix from two arrays of point coordinates

3 Results

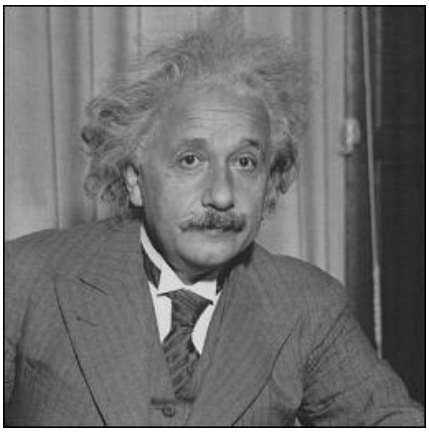
3.1 Prewarping

When feature points obtained by the SIFT algorithm are used to perform the prewarp, pictures are often oriented in the same direction after the operation, as shown in figures 6a and 6b, or 9b and 9a. As such, the morphing step is unable to generate intermediary views. However, the second method of acquiring feature points is not necessarily superior. Indeed, there are cases where the prewarping step distorts images, as shown in figures 7a

and 7b. Sometimes, this renders images unusable for future steps of the algorithm. Such a situation is illustrated in figures 12a and 12b. Maybe the result could have been improved if the intrinsic parameters of the camera were known instead of having to estimate a fundamental matrix.

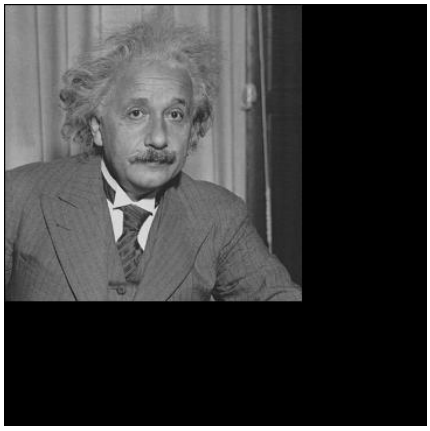


(a) Original image

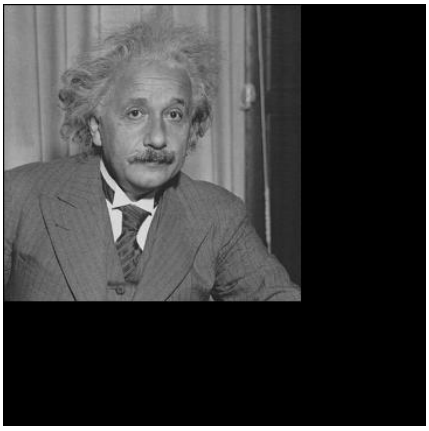


(b) Mirrored image

Figure 5: Images before prewarp



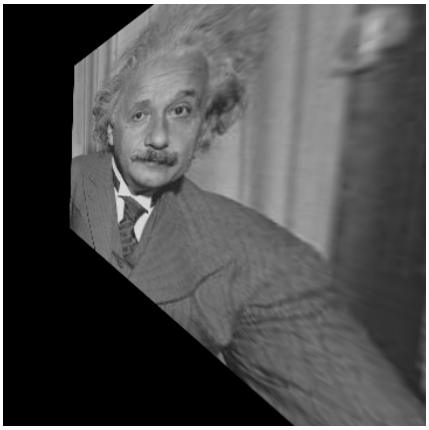
(a) Original image



(b) Mirrored image

Figure 6: Images after prewarp with SIFT feature

(a) Original image



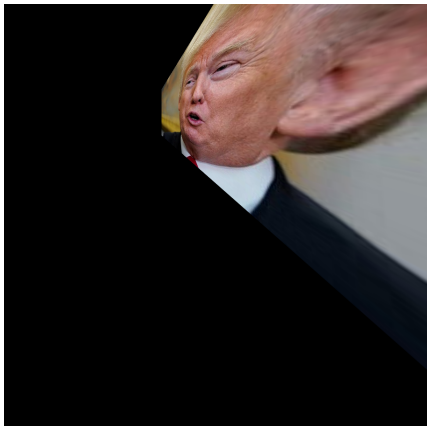
(b) Mirrored image

Figure 7: Images after prewarp with facial features

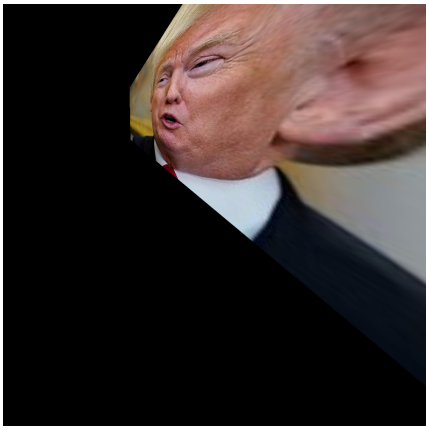
(a) Original image

(b) Mirrored image

Figure 8: Images before prewarp

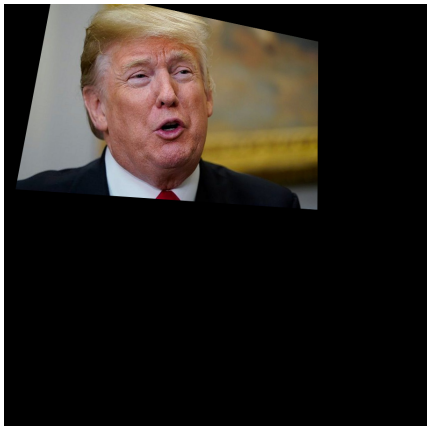


(a) Original image



(b) Mirrored image

Figure 9: Images after prewarp with SIFT features



(a) Original image

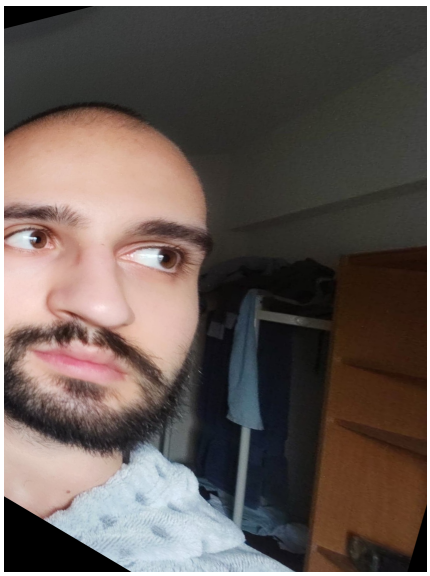
(b) Mirrored image

Figure 10: Images after prewarp with facial features

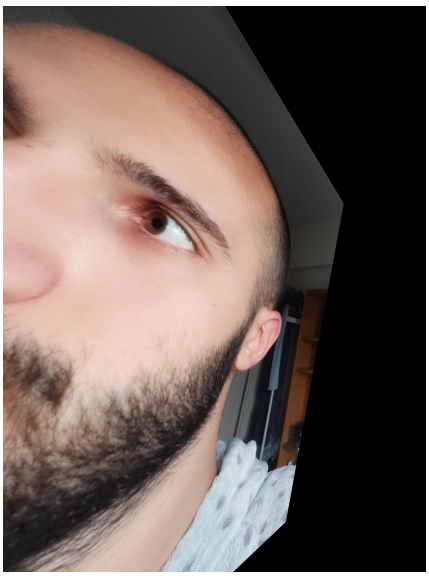
(a) Image taken from the left

(b) Image taken from the right

Figure 11: Original images



(a) Left image

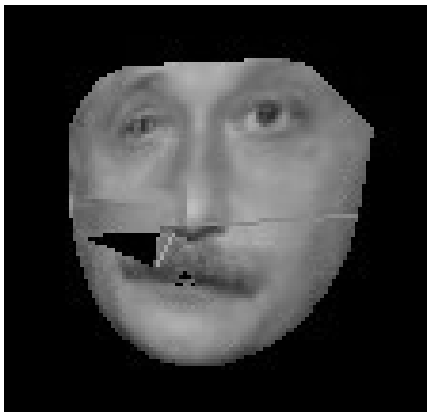


(b) Right image

Figure 12: Images after prewarp with facial features

3.2 Morphing

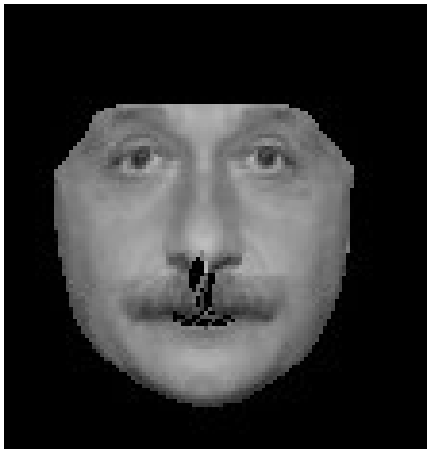
The morphing algorithm performs quite well and is often capable of generating convincing intermediary views without requiring any prewarping or postwarping as is shown in figures 14a, 14b, and 16. It is however not perfect, and can sometimes leave unwanted artefacts in the results, which can be seen in images 13a and 14a. If the original pictures are taken from angles that are too far apart, the final product can look unnatural, as shown in image 17.



(a) Morph of images 7a and 7b

(b) Morph of images 11a and 11b

Figure 13: Morph after prewarp



(a) Morph of images 6a and 6b

(b) Morph of images 9a and 9b

Figure 14: Morph without prewarp

(a) Picture taken from the left

(b) Picture taken from the right

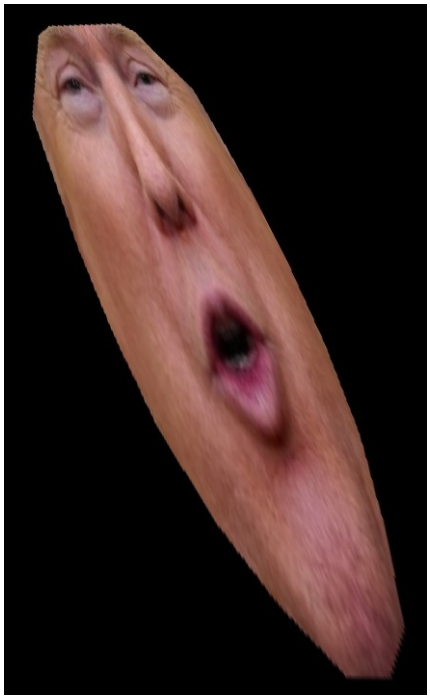
Figure 15: Original pictures

Figure 16: Morph of pictures 15a and 15b without any prewarping

Figure 17: Morph of pictures 11a and 11b without any prewarping

3.3 Postwarping

The postwarping process was the most difficult part of the implementation. The resulting images are rather distorted. The deformation is probably caused by the lack of control points and the imprecision of manual selection.



(a) Postwarp applied on image 13b

(b) Postwarp applied on image 13a

Figure 18: Original pictures

4 Conclusion

This report goes over the implementation of a view morphing algorithm by first detailing the exact methods used for each step of the process, and then showing examples of obtained results.
Overall, the results produced by a regular morph are often less distorted than those made by view morphing. The many steps involved in the view morphing process increase the chance for errors and imprecisions to affect the final result.

References

[1] Henrik Gjestang. *Face Morph Using OpenCV — C++ / Python*. URL: `https://learnopencv.com/face-morph-using-opencv-cpp-python/` (visited on 07/02/2022).

[2] Henrik Gjestang. *View Morphing*. URL: `https://github.com/henriklg/view-morphing` (visited on 07/06/2022).

[3] OpenCV. *Introduction to SIFT (Scale-Invariant Feature Transform)*. URL: `https://docs.opencv.org/4.x/da/df5/tutorial_py_sift_intro.html` (visited on 07/02/2022).

[4] Steven M. Seitz and Charles R. Dyer. "View Morphing". In: *Proceedings of the 23rd Annual Conference on Computer Graphics and Interactive Techniques*. SIGGRAPH '96. New York, NY, USA: Association for Computing Machinery, 1996, pp. 21–30. ISBN: 0897917464. DOI: `10.1145/237170.237196`. URL: `https://doi.org/10.1145/237170.237196`.