TEK5030

View morphing

COURSE PROJECT

Ingrid Utseth
ingridut@fys.uio.no
Betina Høyer Wester
betiahw@student.matnat.uio.no
Henrik Løland Gjestang
henriklg@student.matnat.uio.no

Dato: May 19, 2019

Contents

1 Introduction

In this project in TEK5030 we aim to implement a fully functional model for warping images. All of our code can be found on GitHub. Link is found in the appendix.

When morphing different views of the same scene we will be able to produce new views of the scene not seen before. The model we will implement is based on existing image morphing techniques, and does not require any 3D images.

View morphing works by pre-warping two images, then compute a morph between the two pre-warped images and then lastly post-warping each in-between image produced by the morph. The first step, pre-warping, is done automatically but for the post-warping to work we need some control points. We have chosen to select these points first manually, by the cursor on the two images, and then by using a pre-trained machine learning model to automatically select 68 points around the face.

We will test out this method and then compare it with regular image morphing as well as to play around a bit and hopefully get some interesting results.

2 Method

a) Key point correspondences

In order to compute the fundamental matrix, we need a set of 8 point matches between the two images. These can easily be selected manually, but there are also methods that can find these points automatically. As we are focusing on facial view morphing, we implemented the facial landmark detector in the open source *dlib* toolkit for Python and C++. Dlib's detector uses Histogram of Oriented Gradients (HOG) features with a linear classifier, an image pyramid and sliding window for detection of objects. Trained on the iBUG 300-W face landmark dataset, the model will estimate the location of 68 facial landmarks [1].

Selecting a subset of the 68 facial landmarks detected should give the location of the same world points in each image.

b) Fundamental Matrix

The Fundamental matrix **F** is a tool used in analysis of scenes taken with uncalibrated cameras. The matrix **F** is defined by the equation:

$$\boldsymbol{u}'^{T}\boldsymbol{F}\boldsymbol{u} = 0 \tag{1}$$

Where **u** and **u'** is a pair of matching points in the two images I_0 and I_1. For two views **F** is a 3 x 3 matrix with rank 2. **Fu** describes a line (epipolar line) of where the corresponding points of **u'** will lie. We have followed a paper [2] written on a 8-point algorithm for finding **F**. One of the main disadvantages for using this method which is quite easy to implement is that it is susceptible to noise. However, this paper shows that by normalizing the points by translation and scaling, the results are comparable with some of the best iterative algorithms.

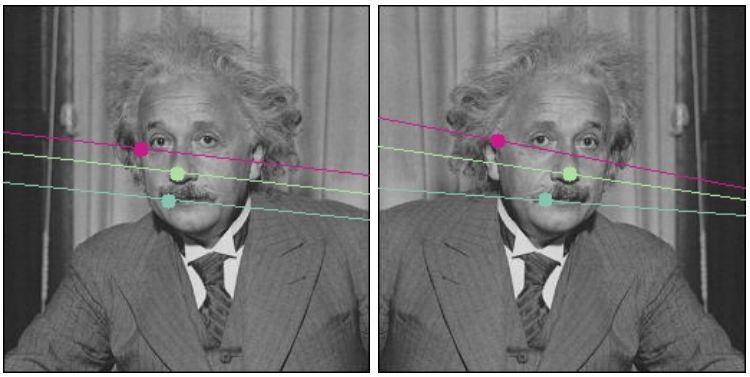


Figure 1: Epipolar Lines

c) Prewarp

The purpose of the prewarp is to align the image planes. We know that two image planes are parallel if the fundamental matrix have the following form:

$$\hat{F} = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & -1 \\ 0 & 1 & 0 \end{bmatrix} \tag{2}$$

Any two images with a fundamental matrix F can be prewarped by finding two protective transforms H_0 and H_1 that satisfies the following equation:

$$(H_1^{-1})^T F H_0^{-1} = \hat{F} \tag{3}$$

The first step in finding H_0 and H_1 is to find a rotation matrix $R_{\theta_i}^{d_i}$ which rotates an image I along a given axis d_i of an angle θ_i. Applying the rotation matrix $R_{\theta_0}^{d_0}$ to I_0 and $R_{\theta_1}^{d_1}$ to I_1, makes the two image planes parallel [4]. Given the axis $d_0 = [d_0^x \ d_0^y \ 0]^T \in I_0$, we have that $[x \ y \ z]^T = F d_0$ and the corresponding axis in I_1 is given by $d_1 = [-y \ x \ 0]^T$. The angle of rotation is given by

$$\theta_i = \arctan\left(\frac{d_i^y e_i^x - d_i^x e_i^y}{e_i^z}\right) \tag{4}$$

where $e_0 = [e_0^x \ e_0^y \ e_0^z] \in I_0$ ans $e_1 = [e_1^x \ e_1^y \ e_1^z] \in I_1$ are epipoles. The next step is to find another rotation matrix which align scanlines [3]. We compute the new epipoles $[\bar{e}_1^x \ \bar{e}_1^y \ \bar{e}_1^z] = R_{\theta_i}^{d_i} e_i$ and denote the new angle of rotation to be

$$\phi_i = -\arctan\left(\frac{\bar{e}_i^y}{\bar{e}_i^x}\right) \tag{5}$$

We then get

$$H_0 = R_{\phi_0} R_{\theta_0}^{d_0} \tag{6}$$

$$H_1 = R_{\phi_1} R_{\theta_1}^{d_1} \tag{7}$$

d) Image morphing

After producing the prewarped images $\hat{I}_1$ and $\hat{I}_2$ we are ready to morph the images. The naive way of doing this is simply:

$$\hat{I}_s(x,y) = s\hat{I}_1(x,y) + (1-s)\hat{I}_2(x,y) \tag{8}$$

for some fraction s [5]. However, in order to get good results, we need to align the points in each image. Let $p_1 \in \hat{I}_1$ and $p_2 \in \hat{I}_2$ be projections of the same scene point P. Then

$$p_s = sp_1 + (1-s)p_2 \tag{9}$$

It is not practical to find the point correspondences for each pixel in the image. Instead, we will use our point correspondences from earlier to calculate a weighted mean of the points in the two sets. Let $N = \{n_1, n_2, ..., n_k\} \in \hat{I}_1$ and $M = \{m_1, m_2, ..., m_k\} \in \hat{I}_2$ be the two sets of k point that correspond one-to-one between the two images. With the following formula, we obtain a third set of points $S = \{s_1, s_2, ..., s_k\}$:

$$s_i = (1-\alpha)n_i + \alpha m_1 \tag{10}$$

These points will be use to build our morphed image, which will be a weighted average of our two input images. We use the OpenCV package to perform Delaunay triangulation on each set of points. The function outputs coordinates for the corners of a set of triangles such that no point in the input set is inside the circumcircle of any triangle. For each triangle in N, we calculate the affine transform that will convert the triangle to the corresponding triangle in S. After doing this for all triangles we have built M'. We do the same for each triangle in N to build N'.

Thus we are able to warp each triangle in our source images to the shape they will have in the morphed image. Our intermediate morphed image $\hat{I}_s$ is found by simply cross-disolving the colors [5].

$$\hat{I}_s = (1-\alpha)N' + \alpha M' \tag{11}$$

This morph can be used on images taken of different objects from the same angle. See figure 3

e) Postwarp

To complete the view morph, we need to postwarp the intermediate morphed image $\hat{I}_s$ to a desirable plane; that is we need determine and apply H_s to $\hat{I}_s$. Seitz [3] suggests that we determine the path of four control points through the entire view morph process. The position of these control points in $\hat{I}_s$ and in the final morph I_s can be used to determine H_s since each control point pair $p \in I_s$ and $\hat{p} \in \hat{I}_s$ is related by:

$$c\hat{p} = \begin{bmatrix} h_{11} & h_{12} & h_{13} \\ h_{21} & h_{22} & h_{23} \\ h_{31} & h_{32} & h_{33} \end{bmatrix} p \tag{12}$$

Figure 2: Image of Donald Trump and a bear [6].

Figure 3: Morphing of two images of Donald Trump and a Bear.

for some scalar c. Eliminating c, and adding the constraints $|H_s| = 1$, this equation can be solved with four point pairs. We used openCV's findHomography function for this step.

3 Results and discussion

a) Point correspondences

Our project focuses exclusively on view morphing applied to faces. We wanted the process to be as automatic as possible. Thus we decided to use a pre-trained model that picks out 68 points on human faces (see figure 4).

The model seems robust, although it does struggle with extreme angles. In the rightmost image of figure 5, a good number of the points are clearly outside the boundaries of the face.

It is of course possible to expand the automatic point correspondence locator to include other points in the image, for instance hair, clothing and the background. This could be done with, for instance, some feature detector combined with RANSAC for feature matching and would allow us the morph the entire image. Due to time constraints, however, we decided to focus on the face only.

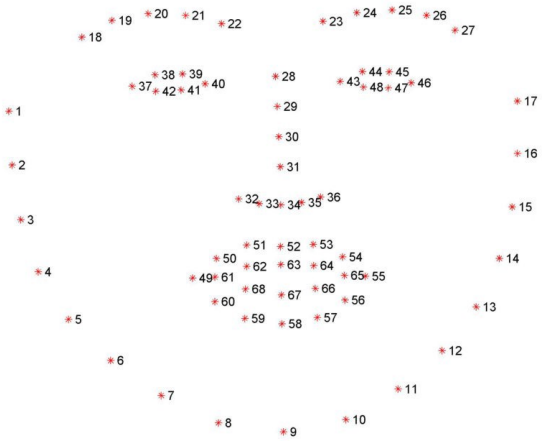


Figure 4: Position of landmark's found with dlib's facial landmark detector [8].

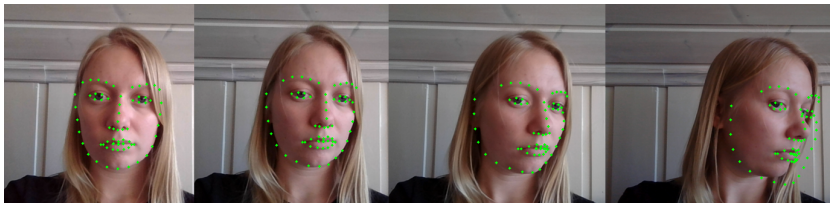


Figure 5: Dlib facial feature detector for different facial angles

b) Prewarp

The result after applying H_0 to I_0 and H_1 to I_1 is shown in fig 6. We see that the left image is slightly distorted. Calculating the fundamental matrix using point correspondences from estimated points leads to uncertain results. We could have improved this result by using stereo cameras, because then the projection matrices are known, and we can use this to calculated a projection matrix instead of using the homographies H_0 and H_1.

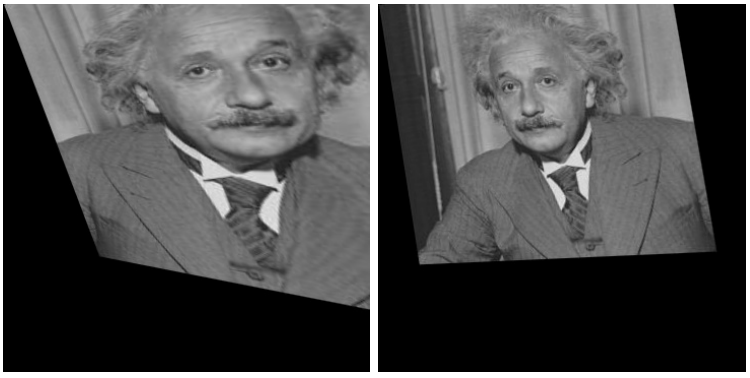
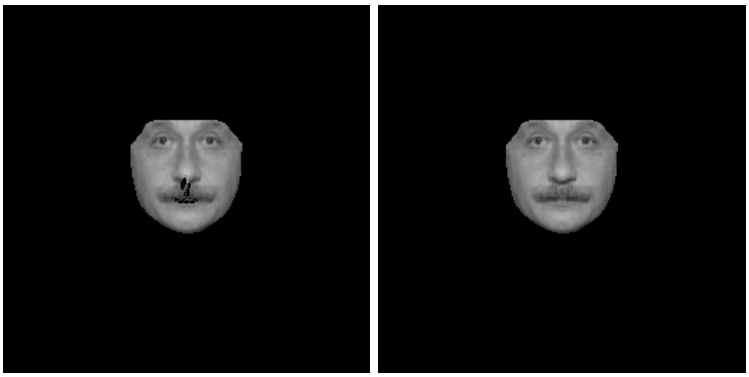


Figure 6: Prewarped images



(a) Morph with 68 points **(b)** Morph with 52 points

Figure 7: Morph of Einstein (figure 13 (a) and (b) in the appendix) with $s = 0.5$

c) Morph

Our implementation of the morphing algorithm with Delaunay triangles worked reasonably well. We get a decent performance when morphing images of two different people from a similar angle.

However, we did uncover some issues with the way we implemented the triangles. We used one of the images to calculate the Delaunay triangle and then simply connected the corresponding points in the other images to form corresponding triangles. In some cases, the corresponding triangles were too small or the points were not positioned in a way that allowed us to draw corresponding triangles, which led to information loss and holes in the morphed image. The facial landmark detector a lot of points around the mouth in particular, some of which are not visible when the face is at an angle. Figure 7 (a), a straight forward morph of two angled images illustrates this problem. Some pieces under Einstein's nose are missing.

Figure 8: Figure (a) displays the original 68 points, while figure (b) displays the 52 points we ended up using

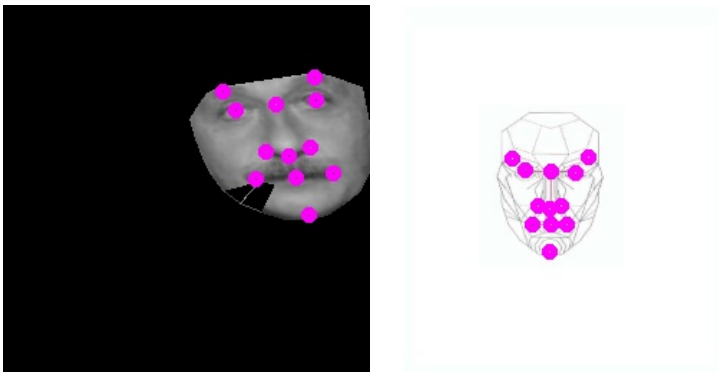
We might avoid this problem by ignoring some of the excessive points around the mouth. It seemed that the large number of points in the same area caused some triangles to disappear. We removed 16 detected points around the mouth

and noise and got a more stable result (see figure 7 (b)). Figure 8 illustrates the points we removed.

d) Postwarp

We struggled with the postwarp process. Seitz suggestion that we simply find the path of four control points was easier said than done since the plane of the final morphed image is unknown. We experimented with using a "mask" of a face turned towards the camera and mapping point correspondences manually between $\hat{I}_s$ and the mask.

The process is illustrated in figure 9. Figure 10 shows the final result.



(a) Point selection on $\hat{I}_s$ **(b)** Point selection on mask

Figure 9: Selecting point correspondences for postwarp process

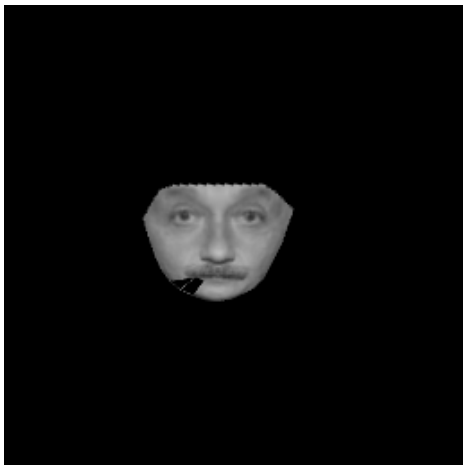


Figure 10: Final morph

There are obviously some issues with the final morph. The black rectangles on the left side on the face indicates that some information got lost in the morphing process. Most likely this could be solved by further limiting the

number of points used in the morph. Removing half the points surrounding the face solved this problem (see figure 11)

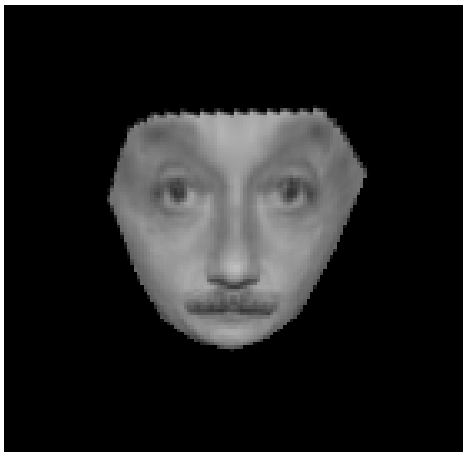


Figure 11: Final morph with fewer point correspondences

Another issue with the morph is the Einstein's distorted head. This is almost certainly due to the poor accuracy of the manual point selection. After some experimenting, we discovered that a more balanced selection of points in all areas of the face significantly helped reduce the distortion and finally produced a reasonably successful view morph (figure 12). The view morphing steps are illustrated in figure 13 in the appendix.

Figure 12: View morph with careful postwarp point selection

e) Morphing vs View Morphing

Comparing the view morphing in figure 12 and regular morphing without prewarp and postwarp in figure 7 (b), it is hard to say whether the view morph

looks more "real" or less distorted than regular morph. If anything, due to the many steps involved in the view morphing process where errors can occur and difficulty in accurately selecting points for the postwarp, the regular morph looks better.

View morphing might be more useful when morphing images of objects with straight lines and corners. Straight lines are prone to bending in regular morphing, but view morphing should help with this issue. The selection of postwarp point correspondences might also be easier. Since our model is using the a face feature point detector, it would need some adjustments before it is able to view morph images of something other than faces and due to time constraints, we were not able to implement the necessary changes.

4 Conclusions and perspectives

In conclusion we have found that both regular image morphing and our implemented view-morph will do quite well on faces. We suspect this would not be the case for more complex viewpoints due to distortions in view angles, such as in a case where parts of the object is hidden, or the angle is too great. Further experiments with objects with straight lines and corners (for instance a box) would likely produce better examples of how view morphing can be superior compared to regular image morphing. In our case, the extra steps and many approximations needed in order to produce a face view morph caused the final morph to be of similar or poorer quality compared to a regular morph.

We would have liked to calibrate two stereo cameras to be used in this project, but due to time constrains we were not able to. Knowing the projection matrices of the two cameras, we could have calculated a projection matrix for the morphed image rather than estimating the homographies using less accurate point correspondences. The projection matrix could also have been used in the postwarp step to get a more accurate result. We would also have liked to generalize our algorithm for other objects to show the benefits of view morphing.

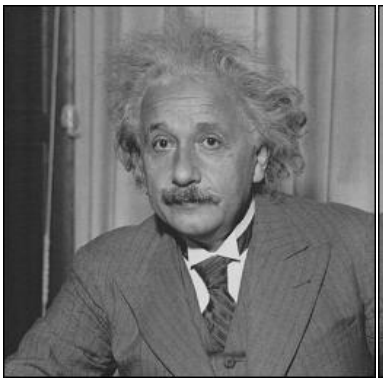
5 Appendix

The code we used can be found on our GitHub page:

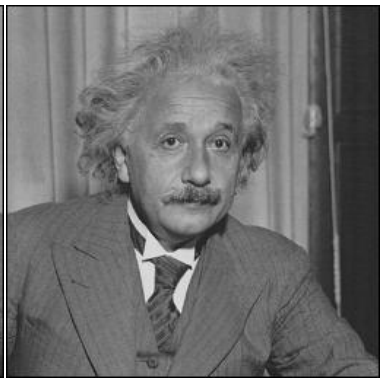
`https://github.com/henriklg/view-morphing`.

We have used a number of python packages such as:

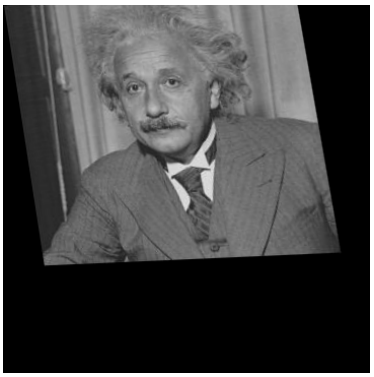
- dlib 19.17.0
- imutils 05.2
- numpy 1.16.2
- opencv-python 4.1.0.25
- python 3.6.8



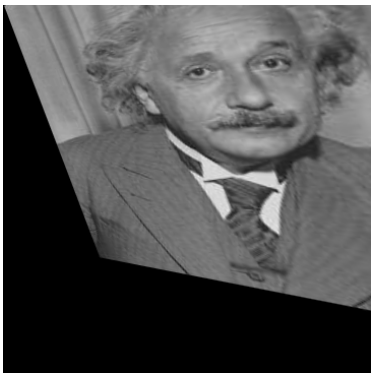
(a) Right view (image1) [11]



(b) Left view (image2) [11]

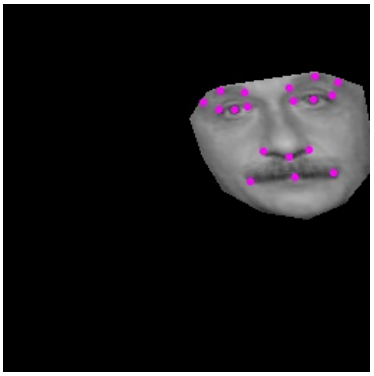


(c) Prewarp of image1

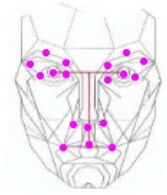


(d) Prewarp of image2

(e) Morphing c) and d)



(f) Point selection on morph



(g) Point selection on mask

Figure 13: Illustration of the view morphing process

Bibliography

[1] C++ toolkit containing machine learning algorithms. http://dlib.net/

[2] *In defence of 8-point Algorithm.* Richard L. Hartley http://www.cs.cmu.edu/afs/andrew/scs/cs/15-463/f07/proj_final/www/amichals/fundamental.pdf

[3] *Image-based Transformation of Viewpoint and Scene Appearance.* Steven Maxwell Seitz https://www.cs.cmu.edu/~seitz/papers/thesis.pdf

[4] *View Morphing.* Steven M. Seitz, Charles R. Dyer. https://homes.cs.washington.edu/~seitz/papers/sigg96.pdf

[5] *Face Morphing using Delaunay Triangulation.* Devendra Pratap Yadav. https://devendrapratapyadav.github.io/FaceMorphing/

[6] Image of Donald Trump and Bear. https://cheezburger.com/6757678336/donald-trump-totally-looks-like-grizzly-bear

[7] *Feature-Based Image Metamorphosis.* Thaddeus Beier, Shawn Neely. https://www.cs.princeton.edu/courses/archive/fall00/cs426/papers/beier92.pdf

[8] *Facial mapping landmarks with dlib.* https://towardsdatascience.com/facial-mapping-landmarks-with-dlib-python-160abcf7d672

[9] *Final Project: View Morphing.* Eugenio Dominguez, Anna Michalska. http://home.cse.ust.hk/~cstws/research/641D/morph/

[10] *Face Morph Using OpenCV — C++ / Python.* Satya Mallic. https://www.learnopencv.com/face-morph-using-opencv-cpp-python/

[11] Image of Einstein. http://www.dgp.toronto.edu/~gelkoura/csc2530/a2/index.html