Face Image Morphing (May 2017)

Conrad Appel[1], Danton Zhao[2], Logan Campbell[3]
Lyle School of Engineering, Computer Engineering[1], Electrical Engineering[2,3]
Southern Methodist University

***Abstract*—In this paper, we propose an implementation of Facial Image Morphing using openCV. Our framework uses the detection of facial features in an image and then warps and adjusts the pixel intensity values determined by a predefined alpha value. Alpha is a value between 0 and 1 that determines the presence of each original image in the morphed image.**

I. Introduction

AT THE simplest level, Image morphing is simply the blending of pixels of image I and J to create M using:

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

$$0 \leq \alpha \leq 1$$

When $\alpha = 0$, M will be identical to I, and when $\alpha = 1$, it will look like J. This simplistic method will blend the two images, but the faces will most likely not be aligned and this will result in a blurry mess of image M

Thus, we must establish pixel correspondence between images I and J. Only then can we calculate the pixel locations in M. The x and y locations in M can be represented with:

$$X_M = (1-\alpha)X_I + \alpha * X_J \tag{2}$$
$$Y_M = (1-\alpha)Y_I + \alpha * Y_J \tag{3}$$

Equation (1) is then applied to the shifted pixels to obtain our morphed image.

II. Image Morphing Technique

A. Landmark Detection

First, to find each corresponding points in images I and J, we used Dlib, a general-purpose C++-based toolkit, to detect facial landmarks. Dlib creates a bag of features using a Histogram of Ordered Gradients, then classifies using a sliding window detection. Dlibs model was trained to return 68 distinct points at specific points on a face (such as on the corners of the lips, the edge of the face, and more). In addition to Dlibs facial landmarks, we added anchor points to the corners of the images because we also want to morph the pieces of the image that fall outside of the main face.

B. Subdivision

Using our set of points, we subdivide the image using Delaunay Triangulation. This algorithm aims to create triangle subdivisions between all of the points while maximizing the minimum angle present in any triangle. By doing this maximization, we create the fattest triangles possible and minimize sliver triangles - ones that are extremely thin. Each of these triangles are represented within the application as sets of 3 point coordinates each. We also create a separate subdivision for each triangle to use within the next step - a rectangular subsection of the original image that is cropped using a bounding box around the triangle. Using this subdivision proves beneficial because many library functions expect a simple two dimensional array to perform operations on (such as cv2.wiarpAffine(), as described below).

C. Warping

To warp or blend the two images we must use equations (2) and (3) to locate the pixels in image M. Then we calculate the affine transform using OpenCVs getAffineTransform. This will map the three corners of a triangle in image I or J to a corresponding triangle in M. Then we must warp all of the pixels inside each triangle to M. This is achieved using the function warpAffine. It must be noted that warpAffine only takes in an image and not a triangle, so we had to create a bounding box for each triangle and then create a triangular mask using fillConvexPoly. Finally after some fine-tuning to hide the seams, the images can be blended using varying values of α.

D. Find Corresponding Points

a) Dlib
   i) dlib.get_frontal_face_detector() to find a bounding box over a face
      1) Uses sliding window classifier over a histogram of gradients pyramid
   ii) dlib.shape_predictor() to find key-points within the bounding box
      1) Model trained over the HELEN dataset for facial feature detection

E. Delaunay Triangulation

a) Algorithm used to maximize the minimum necessary angle within triangles within points (no slivers)
b) Used cv2.Subdiv2D class and .getTriangleList()
   i) This handled the triangulation algorithm for us
   ii) Returns a list of sets of 3 points (x & y coordinates)
   iii) We only had to clean up the results a little bit
      1) Discard triangles that extended beyond the image
      2) Establish the one-to-one correspondence between each image's triangles

F. Warping Images

a) Iterate through all available triangles
b) Calculate the affine transformations from I→M and J→M
c)

$$\begin{bmatrix} x_i' \\ y_i' \end{bmatrix} = Affine\_transform \cdot \begin{bmatrix} x_i \\ y_i \\ 1 \end{bmatrix} \tag{4}$$

d) Use warpAffine to apply the affine transform
e) Apply triangle mask to only retrieve morphed triangle

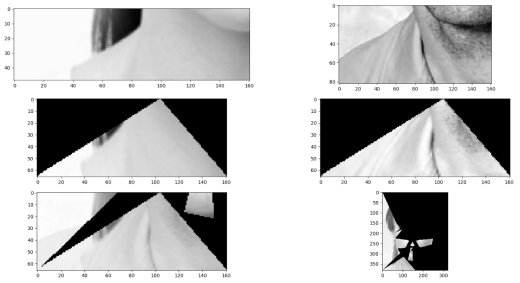

Fig. 1: Masked Triangles

G. GIF Creation

A GIF, at its most basic, is a series of static images joined and played in series. Its possible to use imageio Python packages ability to compile this information and output to a GIF format. After passing in an N-dimensional NumPy array of images, a back-and-forth effect demonstrates the morphing from image I to J as well as from J to I. The reverse playback can be created by appending the reverse of the image array to itself. Because each frame in a GIF can have an independent duration, a longer pause at the start and end frames should be created to emphasize each original image. Also, a compression in the number of available colors using the GIFs format ability will reduce the image file size.

H. Server

An in-class demonstration is a desirable goal and it is important that the server works as seamlessly as possible which includes a fast result. Generating the independent frames of the GIF is a highly-parallelizable and independent job. Thus, we spawn subprocesses, whose number depends on how many logical cores are available, to which frame-generation is offloaded.

a) Simple REST Server based on CherryPy with two end-points
   i) /static returns static files (HTML, JS, CSS, etc)
   ii) /upload accepts two images and redirects the user to a static GIF file after calling the image_morpher
b) Performance:
   i) Since an in-class demo was an end goal, we wanted to make sure it worked as seamlessly as possible - this included getting a fast result.
   ii) Generating the independent frames of the GIF is a highly-parallelizable and independent job. Thus, we spawn subprocesses (number depending on how many logical cores are available) to which frame-generation is offloaded

III. Results

The generated GIF results in a seamless transition from image I to J using varying values of alpha to morph the two images. Examples of this transition can be seen in Appendix B. While image morphing and GIF generation seems to perform well on a PC, users may have difficulty with results when accessing the server on a mobile device. If an image depicts a face with glasses, this may also cause problems with the program.

IV. Conclusion

In this paper, we provide a framework for implementing Facial Image Morphing using Python, Dlib, and OpenCV. By varying the value of alpha we are able to create an array of images that will demonstrate a transition from image I to J and can be viewed as a GIF. This is a proof of concept that if we are able to create one-to-one delaunay triangles in two images, seamless morphing is possible.

V. Appendix A: Code

See https://github.com/conradhappeliv/imagemorpher for the full code-base and instructions on running.

References

[1] http://docs.opencv.org/2.4/modules/imgproc/doc/geometric_transformations.html?highlight=getaffinetransform#warpaffine
[2] http://docs.opencv.org/2.4/modules/imgproc/doc/geometric_transformations.html#getaffinetransform
[3] http://dlib.net/imaging.html#get_frontal_face_detector
[4] http://dlib.net/imaging.html#shape_predictor
[5] http://imageio.readthedocs.io/en/latest/userapi.html#imageio.mimwrite
[6] https://github.com/

VI. Appendix C: Examples

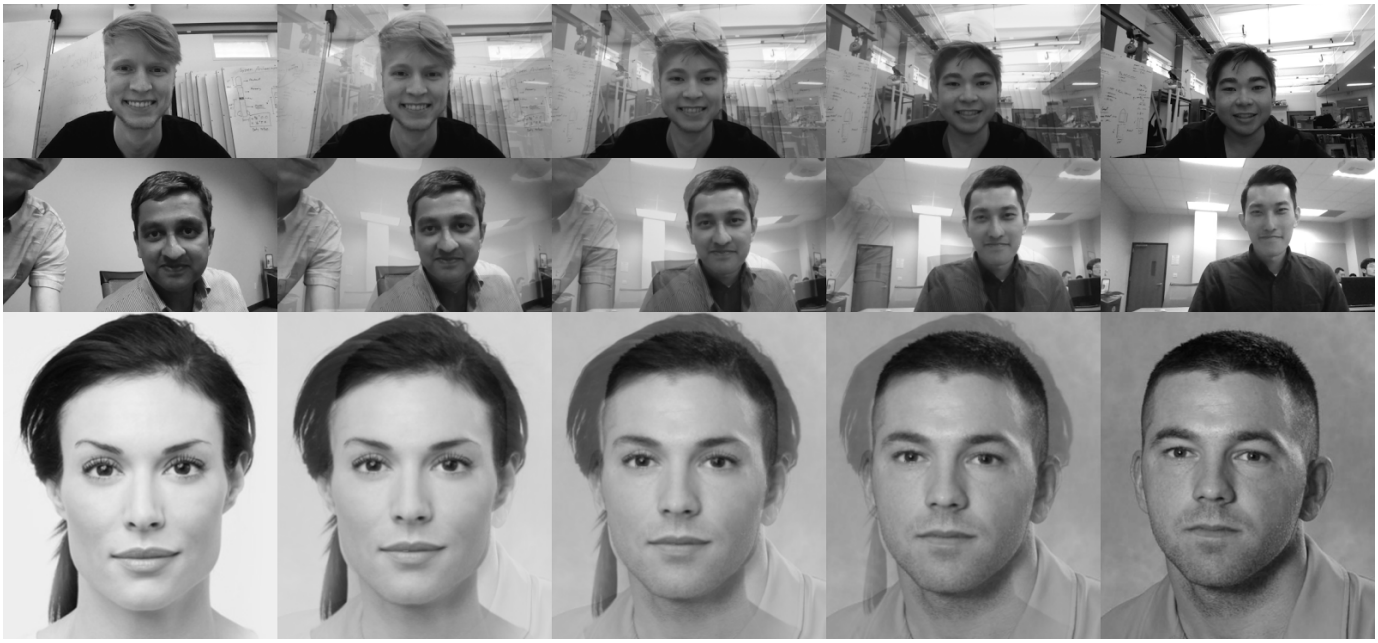

Fig. 2: Sample Merge Sequences

The following code was used to generate these images

```
face1 = io.imread('data/face5.jpg')
face2 = io.imread('data/face6.jpg')
face1gray = cv2.cvtColor(face1, cv2.COLOR_RGB2GRAY)
face2gray = cv2.cvtColor(face2, cv2.COLOR_RGB2GRAY)
res = None
for key, amt in enumerate([0, .25, .5, .75, 1]):
    r = morph(face1, face2, amt, False)
    if res is None:
        res = np.zeros((r.shape[0], r.shape[1]*5), np.float32)
    res[:, key*r.shape[1]:(key+1)*r.shape[1]] = r
from PIL import Image
img = Image.fromarray(res, 'F')
img = img.convert('RGB')
img.save('out.png')

        To generate the 6-tiled pic up above:
        plt.subplot(3, 2, 1)
plt.imshow(subimg1, cmap=plt.cm.gray)
plt.subplot(3, 2, 2)
plt.imshow(subimg2, cmap=plt.cm.gray)
plt.subplot(3, 2, 3)
plt.imshow(warped1*mask, cmap=plt.cm.gray)
plt.subplot(3, 2, 4)
plt.imshow(warped2*mask, cmap=plt.cm.gray)
combined = newimg_sq + (alpha_blend * mask)[:newimg_sq.shape[0], :newimg_sq.shape[1]]
plt.subplot(3, 2, 5)
plt.imshow(combined, cmap=plt.cm.gray)
newimg[bb_morph[1]:bb_morph[1] + bb_morph[3], bb_morph[0]:bb_morph[0] + bb_morph[2]]
        = newimg_sq + (alpha_blend * mask)[:newimg_sq.shape[0], :newimg_sq.shape[1]]
plt.subplot(3, 2, 6)
plt.imshow(newimg, cmap=plt.cm.gray)
plt.show()
```