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*—Abstract: In this paper, we propose an implementation of Facial Image Morphing using Python, Dlib, and OpenCV. Our application detects facial landmarks in an image and then warps and blends the pixels between the landmarks to create a smooth transition between two faces.**

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 need to establish pixel correspondence between images I and J. Only then can we calculate the pixel intensity values in M.

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, using the HELEN dataset, 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 Dlib's 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. Some of these triangles extend beyond the bounds of the image, resulting in invalid triangles. We remove these invalid triangles so that one-to-one relationships between triangles in image I,J, and M are preserved. 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.warpAffine(), as described below).

C. Warping

To warp or blend the two images we must use equations (3) and (4) 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. Following that, we must warp all of the pixels inside each triangle to their corresponding triangle in 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.

$$\begin{bmatrix} x_i' \\ y_i' \end{bmatrix} = Affine\_transform \cdot \begin{bmatrix} x_i \\ y_i \\ 1 \end{bmatrix} \tag{2}$$

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

$$Y_M = (1-\alpha)Y_I + \alpha * Y_J \tag{4}$$

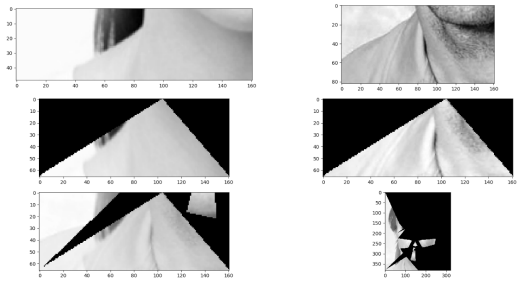

Fig. 1: Masked Triangles

D. GIF Creation

A GIF, at its most basic, is a series of static images joined and played in series. By pushing each still into a NumPy array,

the GIF generated from the array appears to be a seamless morph. 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. Similar to video, creation of a GIF has an inherent tradeoff of filesize and fluidity/frame rate. For our design, we opted for fewer frames (8 frames) because we wanted to service as many server requests as possible. Using Python's imageio, we further quantized the color spectrum to only 32 shades of grey for each frame and create a compressed GIF from that array of frames.

E. Server

An in-class demonstration is a desirable goal and it is important that the server works as smooth and efficient as possible. This implies that fast results are desired so that a user does not have to wait for a GIF to be created.We use a simple REST server based on CherryPy with two endpoints. On one, we return static files such as the HTML and CSS of a page. On the other, we accept two images from the user and redirect the user to the static GIF generated from those two images. Each frame of the GIF can be generated independent of other frames, so we decided to parallelize the process. Thus we create a Python pool object to handle the parallelization of frame generation.

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. After landmark detection and subdivision, we are able to warp the images to produce a morphed image M. The morphed image can then be manipulated by alpha to show varying degrees of each original image. 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.

V. Appendix A: Code

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

VI. Appendix C: Examples

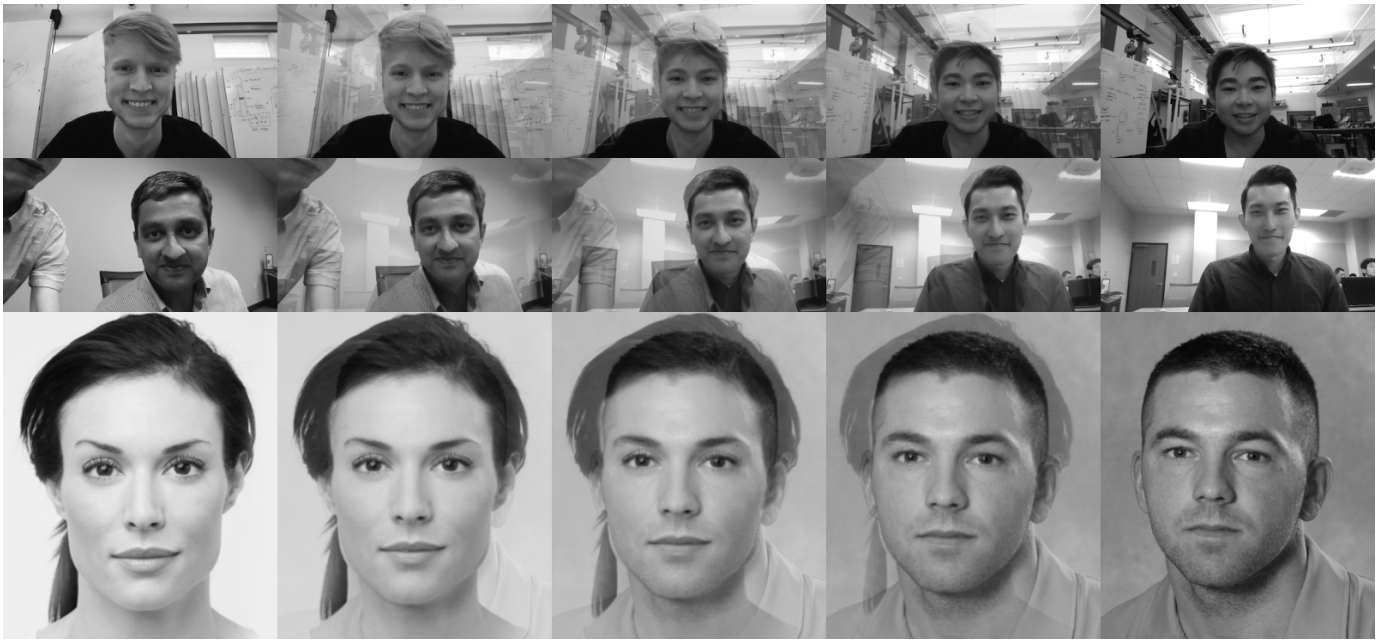

Fig. 2: Sample Merge Sequences