Madeleine Elyea
Introduction to Computer Vision: Winter 2019
Final Project Report

# Final Project: Face Morphing

## Abstract:

The purpose of this project is to morph two faces into one. This paper details the introduction, methods used, and experiments for this project. The Methods section explains the details of the algorithm followed, including finding and matching feature points, an explanation of Delaunay Triangulation, calculating the affine transform, warping the triangles using the affine transform, blending the two images using alpha, and then repeating this process for consecutive values of alpha to create a final GIF video. The experiments section explains how to run this program and provides the final results achieved.

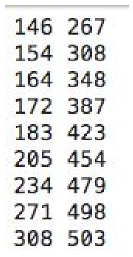
## Introduction

This project will take in two images of faces, and morph them into one. Rather than just blending two images, which creates a ghost-like appearance, computer vision techniques will be utilized to create a clean, morphed image. Both images will be warped using an affine transform calculation, and then blended into a single image. This project involves some techniques that were implemented in Project 2, including feature matching and warping concepts. However, it expands upon these techniques by adding region correspondence through Delaunay Triangulation.

## Methods

### 1. Find and match feature points.

The dlib library was used to detect facial features for both faces. This library is open-source and can be utilized in Python or C++. This library is used in the points.py file. Points.py is called to calculate the feature points for each face. The Dlib library has a function to predict 68 facial landmarks that was used. Then, twelve more points were added around the border of the image to give better final results. Without these additions, only the faces are morphed instead of the whole image. The feature points are then returned as a text file in the following format:

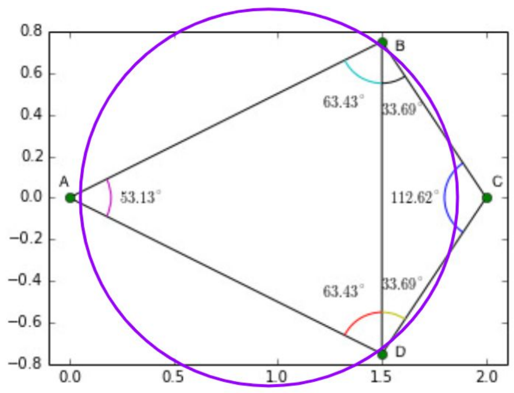


```
146 267
154 308
164 348
172 387
183 423
205 454
234 479
271 498
308 503
```

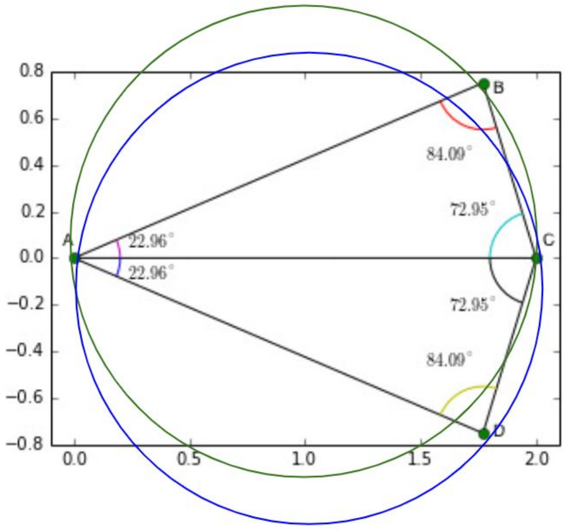
Then, the two sets of feature points are averaged together using the alpha value. This creates a weighted average that is then used in Delaunay Triangulation.

2. Utilize Delaunay Triangulation to expand point correspondence to region correspondence.

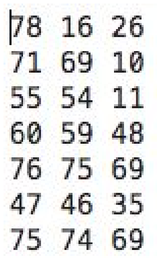
   The Delaunay Triangulation occurs in the delauney.py file. Delaunay triangulation takes in a list of feature points (the weighted average of two faces in step 1), and then determines the best triangulation of these points. To accomplish this, Delaunay triangulation avoids large angles, and ensures that no point of a triangle is in the circumcircle of another triangle. In the figure below, triangle ADB has small angles, and point C is not in the circumcircle.



   If we were to shift points B and D right by 0.25, the triangulation would have to be reconfigured. This is because point C would be inside the circumcircle of points ABD. So, the triangles are redrawn to account for this (See figure below).



The Bowyer-Watson algorithm was utilized to accomplish this. It essentially reconfigures the triangulation for each feature point that is added. This is accomplished one point at a time until all feature points have been added to the triangulation. It begins by first finding all the triangles that are no longer valid due to the point insertion, by checking if they are in the circumcircle of any triangle. These triangles are added to a list of “bad triangles.” Any triangle whose circumcircle contains the new point is deleted, and this leave a polygonal “hole” that must be re-triangulated using the new point. This retriangulation occurs for every new point that is added, until a completed triangulation is generated. This is then saved to a .txt file called “tri.txt” that contains a list of the points of the Delaunay triangles in the following format:



```
78 16 26
71 69 10
55 54 11
60 59 48
76 75 69
47 46 35
75 74 69
```

## 3. Calculate the affine transform for each triangle mapping

The openCV function getAffineTransform was used to calculate the affine transform for each triangle in the list created in part 2. Originally I planned to code this myself, but ran out of time. The affine transform is calculate to transform one triangle to its respective triangle in the second image. This is repeated for all triangles in image 1, and again in image 2.

4. Warp all triangles in image 1 and image 2 using the affine transform that was calculated.

   The openCV function warpAffine was used to warp the triangle to its respective place in the morphed image. I also planned to code this function myself, but ran out of time. Both images are warped using the affine transform calculated in step 4. At this point there are two warped images, one for each face.

5. Blend these two warped images into the morphed image

   The two warped images are then blended into the final morphed image using the following calculation:

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

6. This process is repeated for consecutive alpha values to create a gif video of the blend from one face to another.

   I decided to create a gif that morphed forwards and then backwards continuously. To do this, the entire calculation is repeated for consecutive values of alpha, from 0 to 1 in steps of 0.1. Then each image is blended in a gif.

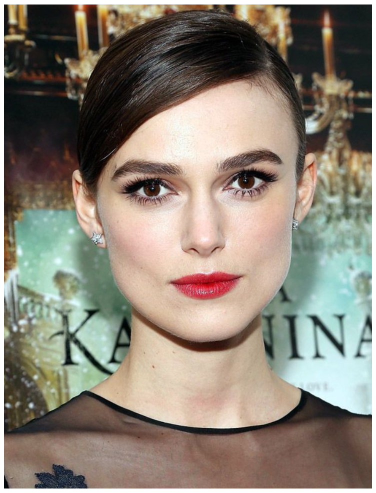
## Experiments:

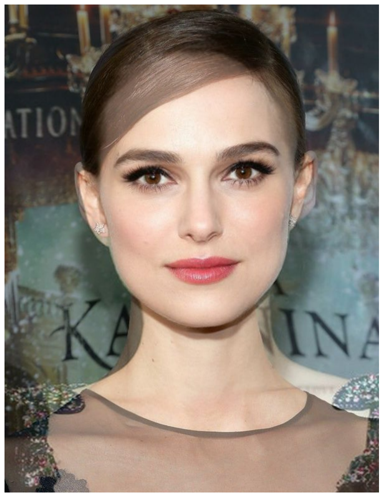
I used images of Natalie Portman and Keira Knightley because I've always thought they look very similar. Morphing their faces makes this even more apparent. This project takes in images of 600 x 800 and could be repeated with any faces in this format. To run, the following format is used:

```
python faceMorph.py face0.jpg face1.jpg
```

This project requires dlib to be installed, so if this is not installed, the previously generated .txt files can be used instead of generating them in points.py. However, these .txt files are specific to the images of Natalie Portman and Keira Knightley.

Madeleine Elyea
Introduction to Computer Vision: Winter 2019
Final Project Report


Keira Knightley


Keiralie Knightman (alpha 0.5)

Natalie Portman

See the completed GIF below: