Image Morphing

CS 6643 Computer Vision Final Project, Spring 2020

Professor: James Fishbaugh

By Zhihao Zhang

Introduction:

Image morph is a special effect in motion pictures and animations that changes one image into another through a seamless transition. It is a very interesting application of computer vision that interests me to dive into deeply about how it works. In this project, I focus on how Image morph works and how to implement.

Data usage:

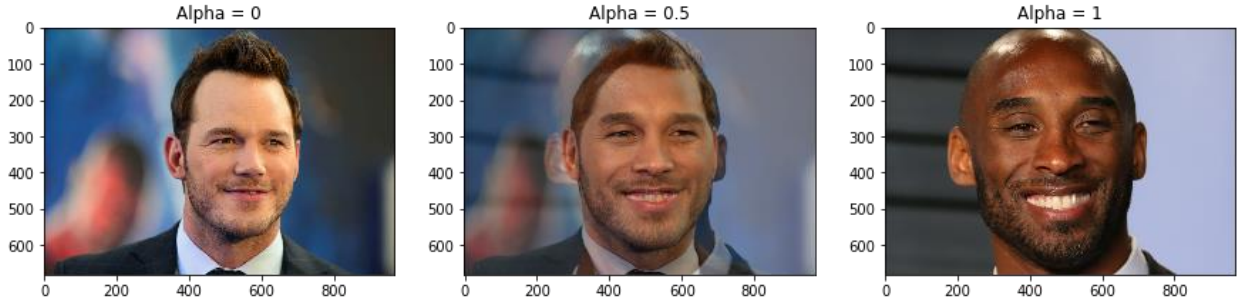
Two images: Source image is the starting image. Target image is the ending image.

Dlib python library for facial detection and making 68 facial landmarks.
A pre-train model for making landmarks: shape_predictor_68_face_landmarks.dat

Additional library: numpy, cv2, imageio.

Approach:

The basic idea is very simple. By finding the corresponding facial points on both source image and target image, we can use affine transformation to warp both images into a third image. This third image is composed of attributes of source and target images by a control parameter alpha. By changing this alpha value, we can obtain an image with different proportions of two different images. Alpha is between 0 and 1.
For instance, if alpha = 0, the third image is most like the source image. If alpha=1, the third image is most like the target image. If alpha=0.5, the third image is equally like both images.

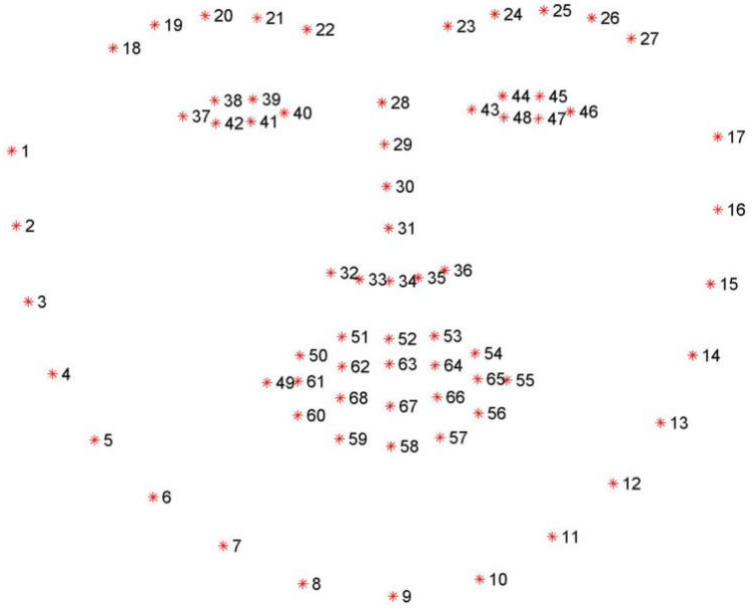


What is the procedure?

1. Find corresponding points of facial landmark for both images.
2. Find the Delaunay Triangulation for those corresponding points.
3. Compute Affine transformation between the Delaunay triangles of both faces
4. Perform warping and alpha blending for initial image and final image.
5. Create a video/gif from newly created frames to show the morphing effect.

Find the corresponding points

Instead of manually finding the corresponding points for source image and target image, we utilize dlib library to automatically detect facial points. The detector will take a grayscale image and return the bounding box of where the face is. If multiple faces, it will return a list of bounding boxes. In our case, both images only contain one face to morph.
Next step is to use predictor to find 68 facial points within the bounding box. Since we used a pre-train model: “shape_predictor_68_face_landmarks.dat”, there’s no need to spent time to train and predict. The prediction is very fast. In addition, my previous experiment on dlib has shown that the predictor works very well.
The following figure shows all 68 points starting from 1 to 68.

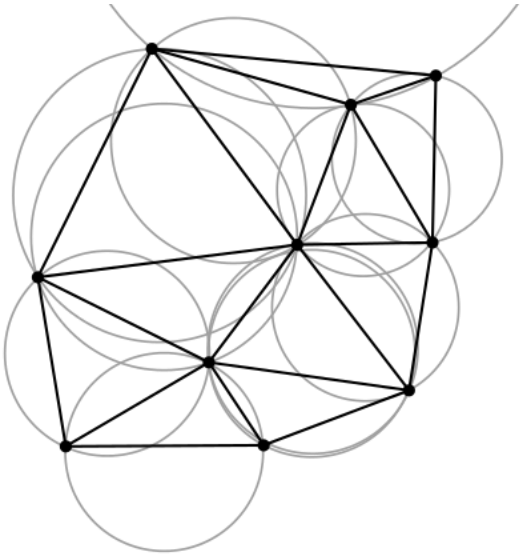


Lastly, we add 8 additional points to 68 corresponding points of source image and to 68 corresponding points of target image. Those 8 points are 4 corner points and 4 half-way points along the edges of image. The Reason is that we need to morph the entire image, not just the face part. In order to find the Delaunay triangles for a whole image, those additional points are needed to generate new triangles that cover the rest parts.

Now, we have 76 points on source image and 76 points on target image, and they are corresponding to each other.

Find the Delaunay Triangulation for those corresponding points

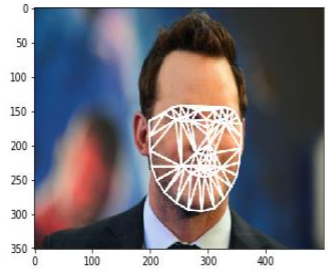
Definition from Wikipedia: a Delaunay triangulation for a given set P of discrete points in a plane is a triangulation DT such that no point in P is inside the circumcircle of any triangle in DT. What it means is that for a set of points to form Delaunay triangulation, no points fall into any circumcircle of formed triangles. The circumcircle is the circle where three vertices are tangent to.



Voronoi Diagram can be drawn from Delaunay Triangulation. The following experiment is just the demonstration of application of Delaunay.

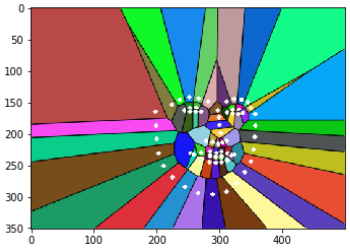
```
In [36]: delaunay_color = (255,255,255)
         draw_delaunay(img_tri,subdiv, delaunay_color)
         plt.imshow(img_tri)

Out[36]: <matplotlib.figure.AxesImage at 0x14101d4a8>
```




```
In [95]: # Allocate space for Voronoi Diagram
         img_voronoi = np.zeros(img_tri.shape, dtype = img_tri.dtype)
         # Draw Voronoi diagram
         draw_voronoi(img_voronoi,subdiv)
         plt.imshow(img_voronoi)

Out[95]: <matplotlib.figure.AxesImage at 0x142868a20>
```



In short, we have 76 points (each point stored as (x, y) coordinates) for source image. By utilizing OpenCV built-in functions cv2.Subdiv2D and subdiv.getTriangleList(), we can obtain 142 triangles. It stored vertex as this: [x1, y1, x2, y2, x3, y3]. (x1, y1) is the first vertex, (x2, y2) is the second vertex, (x3, y3) is the third vertex.

Notice: number of points has an impact on number of triangles. This is not important in morphing applications since we only need to transform those triangles. In my case, 76 points produced 142 triangles.

Next step is to find the indices of vertex for a given triangle. This index is where the vertex point is stored in our 76 points list.
For example, [x1, y1, x2, y2, x3, y3] is our triangle. We know (x1, y1) represents one point and the same for the rest. Therefore, this triangle list can be represented as [v1, v2, v3], where v1 is the index of (x1, y1) stored in our points list, v2 is the index of (x2, y2) stored in our points list, v3 is the index of (x3, y3) stored in our points list.
Since our points lists for source image and target image are computed in corresponding order, the triangles they can generate are also following the same order! We use this index information to retrieve vertex points for both images!
Code snapshot:

```
# Gets delaunay 2D segmentation and return a list with the the triangles' indexes
def get_delaunay_indexes(image, points) :

    rect = (0, 0, image.shape[1], image.shape[0])
    subdiv = cv2.Subdiv2D(rect);
    for p in points :
        subdiv.insert( tuple(p) )

    triangleList = subdiv.getTriangleList();
    triangles = []
    for p in triangleList:
        vertexes = [0, 0, 0]
        for v in range(3) :
            vv = v * 2
            for i in range(len(points)) :
                if p[vv] == points[i][0] and p[vv+1] == points[i][1] :
                    vertexes[v] = i

        triangles.append(vertexes)

    return triangles
```

Compute Affine transformation between the Delaunay triangles of both faces and Perform warping and alpha blending for initial image and final image

The idea to do Affine transformation, warp and alpha blending is based on one online post (See reference for details):

We create a third image M. Given two images I and J. We want to blend I and J into M by a controlled parameter alpha, which is between 0 and 1. When alpha is 0, the morph M looks like I, when alpha is 1, M looks like J.
Two equations are needed:

1. Find the location in image M

$$x_m = (1-\alpha)x_i + \alpha x_j$$
$$y_m = (1-\alpha)y_i + \alpha y_j$$

2. Find intensity value at (x_m, y_m)

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

Equation 1, we can compute corresponding points for the third image M. Now we have 3 different lists of points, and we now the index of vertices for the Delaunay triangles. We can compute the Affine transformation.

Equation 2, Alpha blends the pixel intensity of both images. Alpha controls the proportion of how many pixels intensity value comes from source image or target image.

Affine transformation:

1. According to Delaunay triangle indices, we extract out the first Delaunay triangle for source image, target image, and third image M.
2. Find the bounding rectangle for each triangle. The reason is that cv2.warpAffine only take image as input, we need to mask our triangle in order to utilize OpenCV warpAffine function.
3. Use cv2.getAffineTransform to obtain the transformation matrix of Source triangle to image M triangle. Use cv2.getAffineTransform again to obtain transformation matrix of Target triangle to image M triangle. The idea is to know how to warp source triangle to third image's triangle and how to warp target triangle to third image's triangle.
4. Warp source image triangle and target image triangle. Use the bounding rectangle information above, we can slice the original image into a small piece that contains the triangle. Now we input this small piece of image and its transformation matrix into the cv2.warpAffine. Finally we output the warped version of triangle. The same idea applies to Target image.
5. Now we obtain warped source image and warped target image (image refers to the small rectangle piece that contains the Delaunay triangle). Use equation 2 to alpha blend both warped images into a single image.
6. Final step is to copy triangular region of the rectangular patch to the output image. This is an unmasking step. Extract out of triangle from the mask and only store the triangle into the output image.
7. Repeat Step 1- 6 for each triangle. In my experiment, I have 142 triangles. Therefore I iterate 142 times to compute all triangles' transformation.

Notice: we apply affine transformation one triangle at a time.

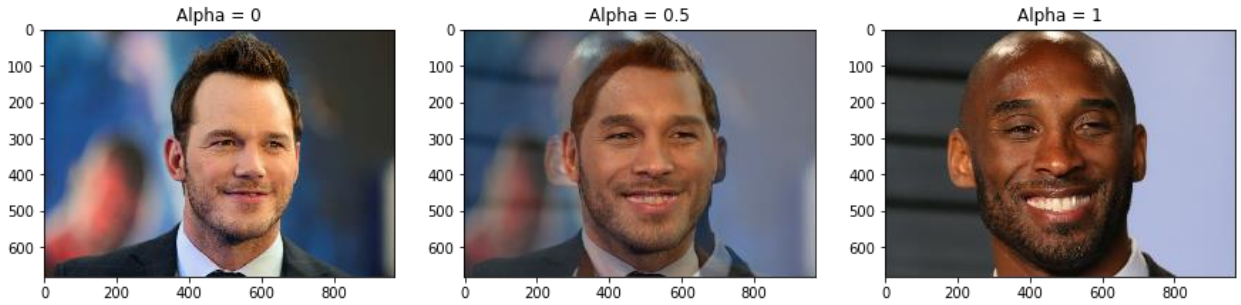
Create a video/gif from newly created frames to show the morphing effect

In order to achieve an animation effect, we need to generate multiple frames of morphed image with different alpha values. Because we know when alpha is close to 0, the morphed image is more like the source image, when alpha is closed to 1, the morphed image is more like the target image. By generating a series of images with increasing order of alpha value, we obtain video frames.
Final step is to convert frames into video format so that we can see the visual effect!

The method I practiced is a python library imageio (See reference for details).

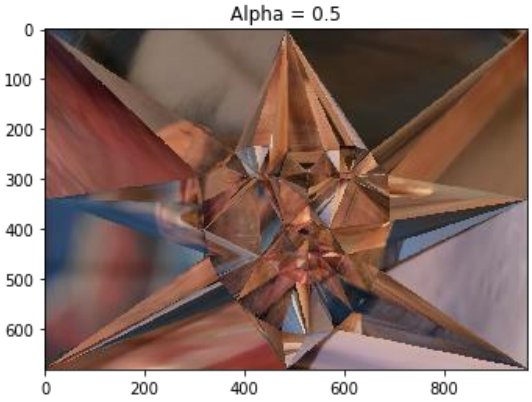
Results:

Here is the result of three different alpha value.



Difficulties and Challenges:

1. Delaunay Triangulation. I'm not familiar with this concept and spent a lot of time doing the research even though the implementation is simple by just calling the OpenCV functions.
2. Mask the Triangle. Before applying cv2.warpAffine, we need to mask our triangle so that it can be used by warpAffine function. After warping, we need to unmask our triangle out and store it in the output image. The idea of mask is to work around the requirement for warpAffine taking image as input. However, the actual implementation needs some time to figure out.
3. The idea of finding the indices of triangles in terms of vertex points. We only need to compute this list of indices once and apply them to all three lists of points. My first attempt is to compute the triangle list separately. That means I calculated all triangles for the list of points of source image, all triangles for the list of points of target image, and all triangles for the list of points of third morphed image. Then I applied affine transformation to those three triangle lists. The result wasn't what I want but very interesting (See below). In fact, I think this wrong version shows how triangulation works under the hood. However, I still don't know the reason behind this result. Fortunately, by changing the way I computed the triangle list, I overcame this problem.



4. Because of Dlib, we don't have to do face detection and facial features extraction. However, it's not very easy to implement such tasks from scratch. We might need techniques such as Faster R-CNN to do object detection.

Strengths and Weakness:

Strengths:

1. The morphing effect is promising. We can simply morph one person into another person smoothly.
2. The processing speed is fast. We don't need to train a model to detect a face. Moreover, the computation of affine transformation is reduced by Delaunay Triangulation.
3. By altering the output format, user can easily select gif or mp4 format. If we use library FFmpeg, we can even change more settings such as frames per second.

Weakness:

1. It cannot handle the case where multiple faces appeared in one image. The dlib detector can find multiple bounding boxes; however, it's not 100% guaranteed.
2. If the dlib failed to predict facial landmarks, our algorithm won't be able to do the following morphing step.
3. We need to download pre-train model for our dlib predictor.
4. When generating gif format output, the image file is large. However, we can optimize a gif using pygifsicle according to imageio documentation.

Future work:

1. Apply this morphing algorithm to multiple faces in a given image.
2. Instead of people's faces, experiment on non-living objects to see if the morphing can still work.

Reference:

1. Affine transformation and alpha blending. https://www.learnopencv.com/face-morph-using-opencv-cpp-python/
2. Convert frames to video format. https://imageio.readthedocs.io/en/stable/userapi.html#imageio.mimwrite