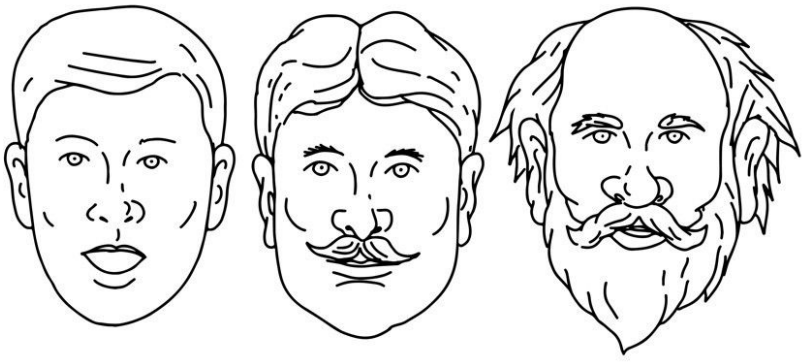
MPA PROJECT REPORT

"Developing a program for Image morphing with Python 3 and OpenCV"

Faculty Mentor: Prof. Anukriti Bansal



Team Members

Anshu Musaddi - 17UCS185

Arnab Sinha - 17UCS034

Table Of Contents

Overview

This project focuses on applying affine transformation to implement **image morphing**, **an image processing technique used for the metamorphosis from one image to another**. The process consists of two parts: first being the triangulation of the control points in all frames and the subsequent linear interpolation of the intermediate frames to form the morphed images.

Ever since its formulation, image morphing has found its applications in various fields, few of them being:

- **Medical field**
    - Capturing the physiological healing process
    - Capturing changes in cells
- **Forensic Science**
    - Recognizing physical features that a criminal may have
- **FIlm industry**
    - Create special animations
- **Environmental science**
    - Help analyze changes in the environment

However, this project does not focus on any one field but covers the basic concept of how it is done. At the end of this report, one would become familiar with the working of morphing.

Languages and Frameworks used:

The project has been carried out by the help of following languages/libraries -

- Python 3
- OpenCV
- Numpy
- PyPlot from Matplotlib

Executing the code

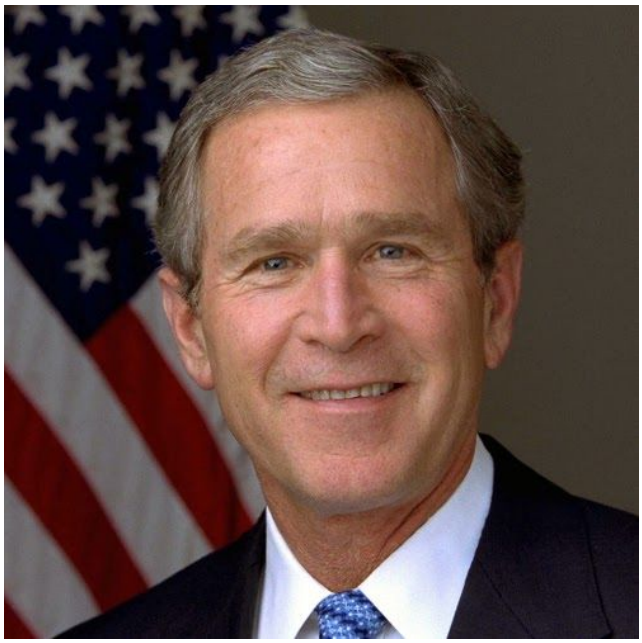
We have created a script that can be run by executing the following statement in the command line -

```
>> python morphing.py <sourceimage> <destimage> -frames <frames>
```

All the <...> are place holders and have to be changed as follows:

- sourceimage - The location of the source image from the current directory.
- destimage - The location of the destination image from the current directory.
- frame - The total number of frames we want after the source image. This is an optional parameter and the default has been set to 5.

For the purpose of illustration in this document we have used the following two images :



Source Image (500 * 500)

Destination Image (500 * 500)

Algorithm

The algorithm can be divided into the following subparts:

Importing Images

- The path of the images are loaded from arguments passed while executing the code. It is loaded into the system using OpenCV. The channels are rearranged into BGR for uniformity.
- If the images are not of the same shape, they are resized as per following formulae:
    - width = **min**(width of source, width of destination)
    - height = **min**(height of source, height of destination)

Choosing Control Points

After the images are loaded two dialog boxes appear. We need to select the control points for the purpose of morphing. The steps to do so are as follows:

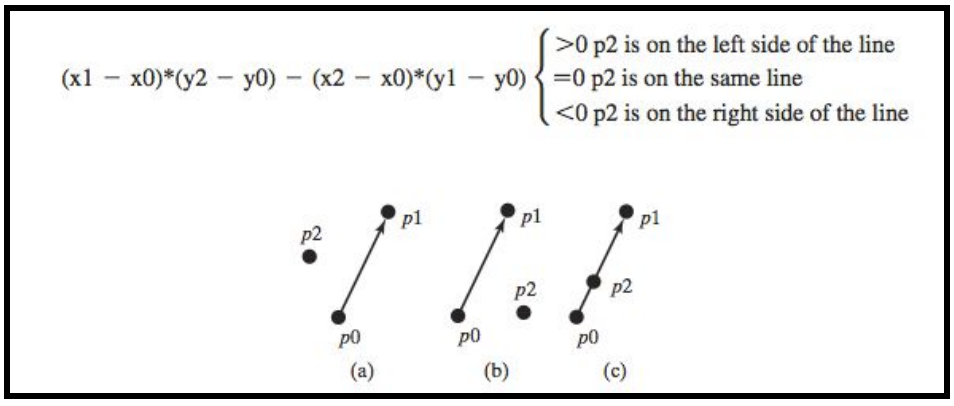
- The user needs to choose a point in the source image and then press Enter. Enter marks that the choice is final. As soon as enter is pressed the user can view his selection.
- A corresponding point needs to be selected in the destination image and then press Enter to confirm the selection.
- Once the user is satisfied with all the inputs wanted by him he can end this process by pressing ‘0’.

For the purpose of illustration we have used both the eyes and the center of the mouth as control points.

Delaunay Triangulation

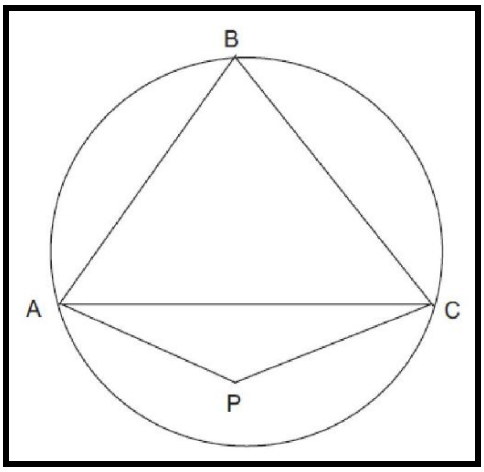
We have implemented Delaunay triangulation to get the triangles from the control points. The major steps of Delaunay Triangulation is as follows -

- Initialize Delaunay with super triangles. We have divided the region of the image shape into two by using one of the diagonal to start with. We have used a random diagonal as both the diagonal will give the same result. We will need to include these two as we want to triangulate the complete image.
- The points are then inserted one by one as we keep receiving inputs of control points from the user. We insert it parallely for Delaunay Triangulation of the source image.
- It finds the existing triangle in which the point lies. To detect whether it belongs to the triangle or not it calculates the side on which it belongs to with respect to an edge of the triangle. If for all the three lines it lies on the same side we can infer that it belongs to the triangle. The method for the same is:



- It deletes that triangle and adds three new triangles that are formed using the new point.

- Then for all the triangles formed it checks it's adjacent triangles to check for Delaunay's criteria. The condition states that we need to change the triangles if for any one of the triangles the circumcircle encloses the 4^{th} point.



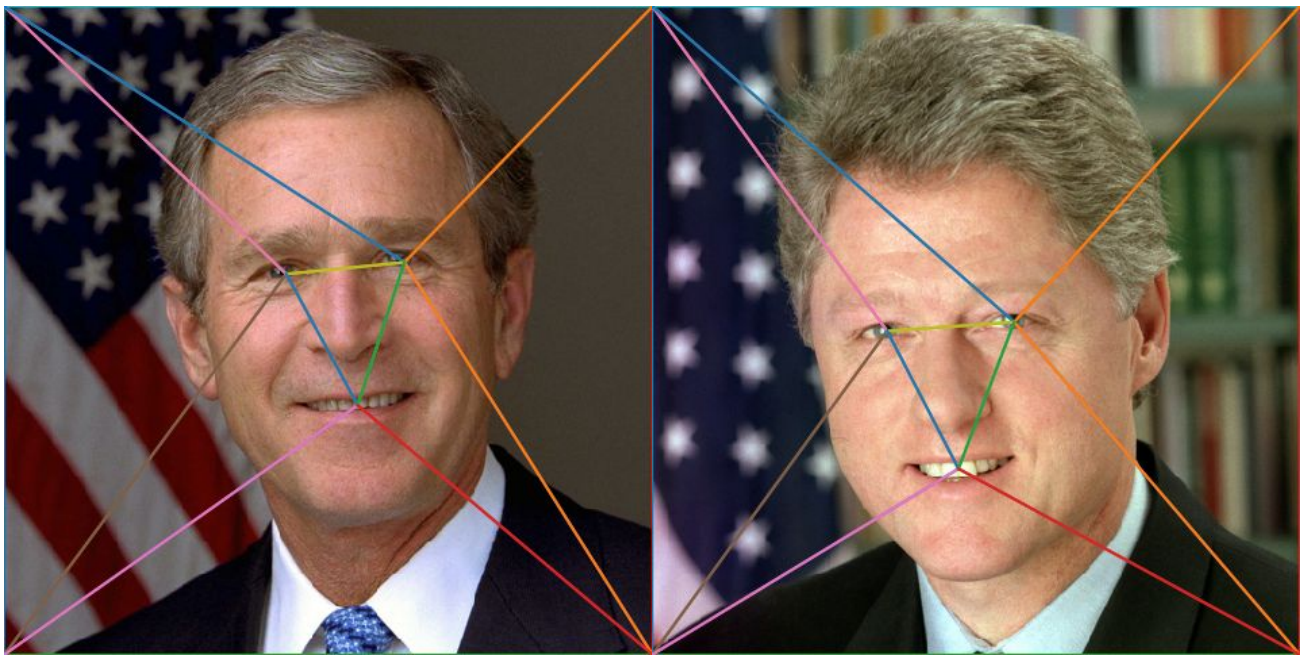
An example of violation of Delaunay' Criteria

- The above mentioned criteria can be reduced to the criteria that the sum of opposite angles to the common edge should sum up to be greater than Π. The angles are calculated using the length of the edges of the triangles and then applying the cosine rule. The rule is as follows -

$$\cos A = \frac{b^2 + c^2 - a^2}{2bc}$$

- Once we have triangulated the source image we map the triangulation to the destination image as we want the triangles in destination image same as the source image and even in the same order.

The output after Delaunay Triangulation is as follows



Source Image *Destination Image*

Linear Interpolation

There are two formulas that must be shown before moving to linear interpolation.

$$\vec{p}_i^{[k]} = \vec{p}_i^{[0]} * (\frac{N-k}{N}) - \vec{p}_i^{[N]} * (\frac{k}{N}) \quad \text{---(1)}$$

$p_i^{[k]}$ = the i^{th} point in the k^{th} intermediate frame ($0 \leq k \leq N$)

$$\vec{p} - \vec{p}_0 = \alpha.\vec{e}_1 + \beta.\vec{e}_2 \qquad \vec{e}_1 = \vec{p}_1 - \vec{p}_0 \qquad \vec{e}_2 = \vec{p}_2 - \vec{p}_0 \quad \text{---(2)}$$

α and β are the affine coefficients for point p

p_0, p_1 and p_2 are the coordinates of the triangle containing p

- Once we receive the triangles for the source image, we use formulae (1) to calculate the control points for the k frames and the destination frame.
- We then calculate the delaunay triangle coordinates for each frame using formula (1), where $p^{[0]}$ corresponds to the source image and $p^{[N]}$ corresponds to the destination image.
- For each point $\bar{p}$ in the kth frame, formulae (2) is used to find their affine coefficients and the interpolated point in the source and destination image.
    - To find the alpha and beta, the first box of formula set (2) is divided into two parts, x and y coordinates. Then the system of linear equations method is used to find the α and β for the 2 linear equations obtained using the known values ($\bar{p}$, $\bar{p}_0$, $\bar{p}_1$ and $\bar{p}_2$)
    - Once that is done, the same formula is used to find the unknown point in the source image (say $\bar{p'}$) and destination image ($\bar{p''}$) using all the known values in the rest of the equation corresponding to the respective images ($\alpha, \beta, \bar{p}_0, \bar{p}_1$ and $\bar{p}_2$). We use the fact that the affine coefficients remain constant for point $\bar{p}$, $\bar{p'}$ and $\bar{p''}$.
- The final frame is then written into a subfolder in the current directory as described below.

Video Generation

After all frames are constructed, the video is generated using **FFMPEG** at 40 FPS and saved in the current directory.

Results

Intermediate frames along with source and destination (L -> R & T->B)

Conclusion

Issues Handled

There were certain issues that we handled during the implementation part -

- We initially tried with the coordinate condition of Delaunay Condition but failed miserably thus modified it to the basic criteria of the angles one.
- The coordinate system of numpy and OpenCV are different. This was solved by converting all the coordinates into a numpy system.

Current Issues

Our algorithm works for most of the cases but have certain issues with few of the following cases -

- Delaunay Triangulation fails if any points lie on the current edges of the triangles. This is a limitation on the algorithm part of delaunay triangulation's triangle detection part. Since it is a rare case we did not try to find a work around.
- Issue of holes. If we take too many control points there is an issue of holes. We could not find the reason behind it.

Further Improvements

The code can be optimized further using certain data structures -

- Currently we are applying brute-force to find adjacent triangles. This can be optimized using queue and graphs.
- A better way to find alpha-beta can be implemented using the help of the vectorization system of numpy.
- We can use/make some libraries that can detect the control points by itself.

References

- Lecture notes pertaining to linear interpolation
- https://www.pyimagesearch.com/2015/03/09/capturing-mouse-click-events-with-python-and-opencv/ for implementing mouse clicking events to choose control points
- https://stackoverflow.com/questions/28327020/opencv-detect-mouse-position-clicking-over-a-picture for implementing mouse clicking events to choose control points
- https://www.youtube.com/watch?v=tWf1z9i-ORg for Delaunay Triangulation algorithm
- https://github.com/jmespadero/pyDelaunay2D for code inspiration but no part was copied
- OpenCV for Python documentation
- NumPy Documentation