CS445 Final Project: Image Morphing

Wenqi He (wenqihe2)

November 19, 2022

1 Motivation

Image morphing is a visual effect widely used in films, TV commercials and user-generated videos online. While seemingly complex, it is easy to produce an approximate solution using triangulation and affine transformation, and the quality can be improved by simply increasing the number of key points to create a finer triangulation. For this project I will provide my implementation of the algorithm and explore its applicability and limitations.

2 Approach

If we think of each point in the image as a vector in $\mathbb{R}^n$ that has both position and color coordinates, then we are trying to define a function $T : [0, 1] \to (\mathbb{R}^n \to \mathbb{R}^n)$ that takes every point in the source image to its new coordinates (both positional coordinates x_i and color channels c_i) at time t:

$$T : t \mapsto (x_0, x_1, ..., c_0, c_1, ...) \mapsto (x'_0, x'_1, ..., c'_0, c'_1, ...)$$

under the constraint that $T(0) = I$ and $T(1)(\mathbf{x}, M_0(\mathbf{x})) = (\mathbf{x}', M_1(\mathbf{x}'))$.

The main difficulty lies in explicitly defining the transformation for the spatial coordinates. Once that is determined, we'll be able to get the initial and final positions of each point, using which we can then look up its initial and final colors, then linearly interpolate them, i.e. $\mathbf{p}_t := (1 - t)\mathbf{p}_{initial} + t\mathbf{p}_{final}$.

A simple approach to this problem is to create identical triangulations for the two images based on corresponding key points, then solve for a piecewise affine transformation, that is, locally, we apply a linearly interpolated affine transformation for each triangle $\triangle abc$:

$$T_{\triangle abc}(t)(\mathbf{x}, M_0(\mathbf{x})) = (1 - t)\Big(\mathbf{x}, M_0(\mathbf{x})\Big) + t\Big(A_{abc}\mathbf{x}, M_1(A_{abc}\mathbf{x})\Big)$$

Intuitively, this function performs a combination of warping and cross dissolving.

3 Implementation Details

As with previous projects, this project is implemented in Python, and uses NumPy/SciPy for numerical computation, OpenCV and Matplotlib for image and video I/O, and Jupyter Notebook for interactivity. The following sections explain the key steps in my implementation as well as some important design choices.

3.1 Key Points

For finding corresponding key points, we can't use automatic feature matching algorithms such as SIFT because the two images are usually of different objects and therefore the descriptors generally won't match. My implementation takes the simple approach by asking the user to pick key points manually. Furthermore, since there is no obvious way to determine which pair of points should match, my implementation relies on the user picking key points in the exact same order for both images.

3.2 Delaunay Triangulation

Triangulation is done using library function `scipy.spatial.Delaunay`. It is only performed on the initial image and the resulting set of simplices is used for both images. The reason for applying the algorithm only to one image is that Delaunay triangulation depends on distances and the keypoint distances in two unrelated images are almost certainly different, which could potentially produce inconsistent results.

3.3 Affine Transformation

The affine coefficients for each pair of corresponding triangles can be solved exactly as a system of linear equations using library function `numpy.linalg.solve`. Specifically, for points $(x_1, y_1, 1)$, $(x_2, y_2, 1)$, $(x_3, y_3, 1)$, $(x_4, y_4, 1)$, $(x_5, y_5, 1)$, $(x_6, y_6, 1)$ and affine transformation

$$A = \begin{pmatrix} a_1 & a_2 & a_3 \\ a_4 & a_5 & a_6 \\ 0 & 0 & 1 \end{pmatrix}$$

we have the system

$$\begin{pmatrix} x_1 & y_1 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & x_1 & y_1 & 1 \\ x_2 & y_2 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & x_2 & y_2 & 1 \\ x_3 & y_3 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & x_3 & y_3 & 1 \end{pmatrix} \begin{pmatrix} a_1 \\ a_2 \\ a_3 \\ a_4 \\ a_5 \\ a_6 \end{pmatrix} = \begin{pmatrix} x_4 \\ y_4 \\ x_5 \\ y_5 \\ x_6 \\ y_6 \end{pmatrix}$$

3.4 Rasterization and Interpolation

To compute an interpolated image, my implementation first computes the position of each triangle, then applies the digital differential analyzer (DDA) algorithm to compute each pixel covered by the triangle. For each pixel $\mathbf{x}_t = \big[(1-t)I + tA_{abc}\big]\mathbf{x}$ within each triangle $\triangle abc$:

$$(\mathbf{x}_t, M_t(\mathbf{x}_t)) = (1-t)\Big(\mathbf{x}, M_0(\mathbf{x})\Big) + t\Big(A_{abc}\mathbf{x}, M_1(A_{abc}\mathbf{x})\Big)$$

Substitute $\mathbf{x}_t$ for $\mathbf{x}$ on the RHS,

$$\boxed{M_t(\mathbf{x}_t) = (1-t)M_0\left(\big[(1-t)I + tA_{abc}\big]^{-1}\mathbf{x}_t\right) + tM_1\left(A_{abc}\big[(1-t)I + tA_{abc}\big]^{-1}\mathbf{x}_t\right)}$$

Image functions M_0 and M_1 are implemented using bilinear interpolation:

$$M(x,y) := \begin{pmatrix} 1-(y-\lfloor y \rfloor) \\ y - \lfloor y \rfloor \end{pmatrix}^T \begin{pmatrix} M\left[\lfloor x \rfloor, \lfloor y \rfloor\right] & M\left[\lceil x \rceil, \lfloor y \rfloor\right] \\ M\left[\lfloor x \rfloor, \lceil y \rceil\right] & M\left[\lceil x \rceil, \lceil y \rceil\right] \end{pmatrix} \begin{pmatrix} 1-(x-\lfloor x \rfloor) \\ x - \lfloor x \rfloor \end{pmatrix}$$

4 Results

- **Face morphing**: `https://youtube.com/shorts/Uq8NgQOqN_8`
- **Face morphing with translation**: `https://youtube.com/shorts/_irBuBUF7kc`
- **View synthesis from mirrored image**: `https://youtube.com/shorts/_mKlzxkB7Lg`
- **View synthesis from two images**: `https://youtube.com/shorts/kpIpy6_gppc`

For faces, smooth morphing results could be produced by using main facial features such as pupils and nasal tips as key points. Interestingly, despite the relatively coarse triangulation, boundaries of triangles are hardly visible, probably due to neighboring triangles having similar transformations.

A surprising discovery is that this triangulation-based-interpolation technique can also be used for view synthesis given two views of a scene comprising simple polygonal objects (e.g. laptop and shelf in the last example) by placing key points at edges and vertices.

See Appendix I for the triangulations used to produce these results and Appendix II for an illustration of warping and cross dissolving happening at different steps in the morphing process.

5 Challenges / Innovation

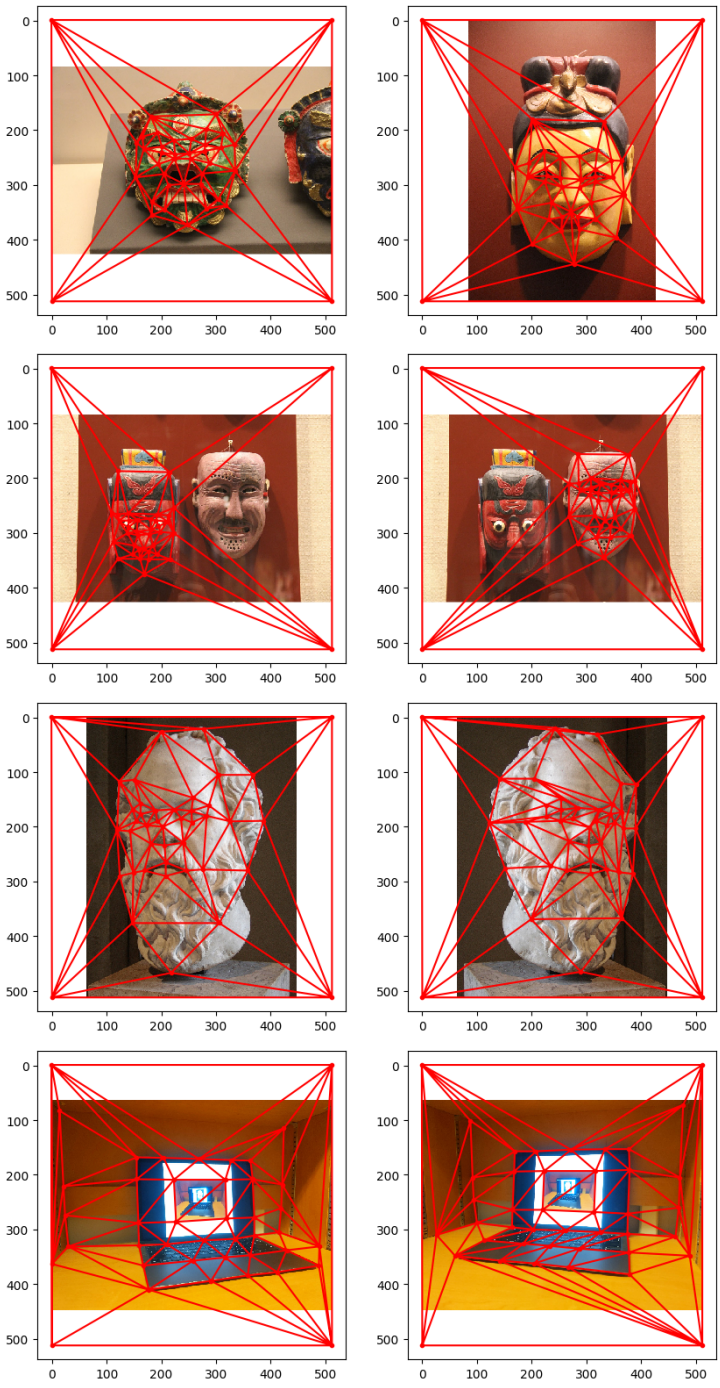
Expected points: 10/20

- The original algorithm described in class requires solving for two affine projection at each step. My algorithm only solves for affine coefficients once for each triangle, based on the observation that at each step t, $A_{t\to 0}$ and $A_{t\to 1}$ are both just linear combinations $aI + bA_{0\to 1}$. However, my formula involves a matrix inverse and computing inverse is no faster than solving a system of linear equations, so the only benefit is just that the boxed formula above uses one less variable than it otherwise would require.
- My initial solution uses $(0, 0)$, $(w-1, 0)$, $(0, h-1)$, $(w-1, h-1)$ as extra key points for triangulation, but DDA doesn't cover any right vertical edges or bottom horizontal edges, resulting in 1-pixel wide black margins on the bottom right. To eliminate this artifact, I shifted the points outside the image to $(-1, -1)$, $(w, -1)$, $(-1, h)$, (w, h)

6 Attribution

1. Gary Todd from Xinzheng, China (`https://commons.wikimedia.org/wiki/File:Nuo_Opera_Mask_(13942516064).jpg`), Nuo Opera Mask (13942516064), modified, `https://creativecommons.org/publicdomain/zero/1.0/legalcode`
2. Gary Todd from Xinzheng, China (`https://commons.wikimedia.org/wiki/File:Nuo_Opera_Mask_(13918920261).jpg`), Nuo Opera Mask (13918920261), modified, `https://creativecommons.org/publicdomain/zero/1.0/legalcode`
3. Gary Todd from Xinzheng, China (`https://commons.wikimedia.org/wiki/File:Nuo_Opera_Masks_(13918944696).jpg`), Nuo Opera Masks (13918944696), modified, `https://creativecommons.org/publicdomain/zero/1.0/legalcode`

7 Appendix I: Triangulations



8 Appendix II: Intermediate Steps