CS/EE 166 Computational Cameras **Spring 2024**

Final Project Report - Image Morphing

Eric Lee *Due: 2024-06-05*

Introduction

The central focus of the project is to perform image morphing, where one image will be seamlessly transitioned into another. In the context of this project, the goal will be to create a video from images of faces such that each face will smoothly morph into each other in sequence.

Given two images, the simplest way to blend two images would be to cross-dissolve them. However our goal is to create warping that may be local, not just global, leading to our main focus: Inverse Warping on Triangular Meshes. Two faces are marked with points corresponding to their lips, eyes, hair, and facial structure. This is then converted to a mesh of triangles (using Delaunay Triangulation). Then, the meshes are interpolated, preserving local shapes but allowing corresponding features to be mapped.

Once we have this image morphing, we can extend the transformations to easily transform the mesh of a face without transitioning to a different image, resulting in facing being morphed into caricatures. We also extend this to simpler photos like flowers being arbitrarily morphed in any mesh transformation we want, simply by defining the beginning and end states and making an animation.

We provide the script `morph` that allows us to design meshes and interpolate images (see appendix)

Algorithm and Methodology

Our algorithm takes 2 images and 2 meshes, but that means we first need to design the meshes. These meshes will highlight the important features of both images, which ideally look similar to one another. Drawing these triangles can be time consuming, so our algorithm allows for the selection of key points to act as the vertices of these triangles instead, then automatically triangulate the images.

Although there are many ways to split a plane into triangles, we will use Delaunay triangulation. This is because Delaunay triangulation maximizes the minimum angle made by any one triangle, which avoids any sliver simplexes (thin triangles) which can have unexpected behavior when stretched out. An equivalent definition of Delaunay triangulation is a triangulation given a set of points in a convex hull such that each triangles' circumcircle do not contain any of the points. This means we can (roughly) use an iterative greedy algorithm that considers all possible triangulations over a few points, then adding points one by one, removing triangles that conflict. However, since this problem is a convex hull optimization problem, we just use the library `qhull` that solves similar problems.

Our script allows for the execution of `--task draw_mesh`, which lets the user select points over both images. This result is stored as a list of coordinates in a csv file. When `--task interp_images` is executed, we first generate the average points by taking the pairwise average between the selected points. This means our two lists of selected points must have the same length. Then, we perform Delaunay triangulation on this average mesh to triangulate the plane, then map those same triangles back to our original meshes.

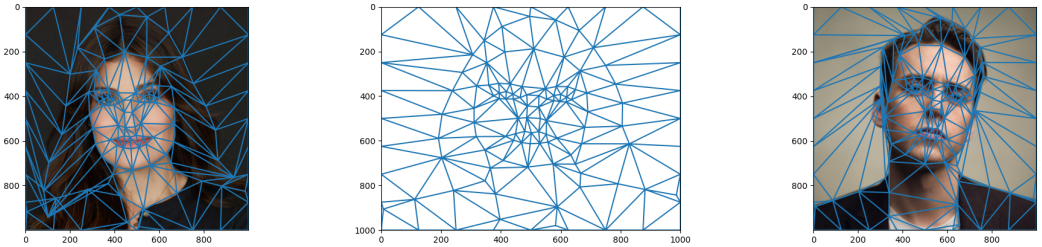

Figure 1: The average mesh (middle) is triangulated through Delaunay triangulation, which is then mapped to the original points corresponding to the two faces. This allows for features such as the hair to be mapped one to one. We add points around the border to ensure the points are in a convex hull.

Once we have this triangulation, we need to map each simplex to another in a linear map. As this is a affine linear transformation between 2 dimensional shapes, we need a 3 dimensional matrix. We solve this by treating the 2D vectors to each corner of a triangle as 3D vectors with $z = 1$. This makes the vectors always linearly independent, allowing for a change of basis between the corners of one triangle to another triangle. In other words, our transformation between two matrices can be represented by the following matrix:

$$\begin{pmatrix} x_1' & x_2' & x_3' \\ y_1' & y_2' & y_3' \\ 1 & 1 & 1 \end{pmatrix} \begin{pmatrix} x_1 & x_2 & x_3 \\ y_1 & y_2 & y_3 \\ 1 & 1 & 1 \end{pmatrix}^{-1}.$$

Once we calculate which pixel is in which simplex (conversion to barycentric coordinates), for each frame, we can sample the color at the pixel after the linear map. However, we have a problem: the transform is continuous, but our images are made of discrete pixels.

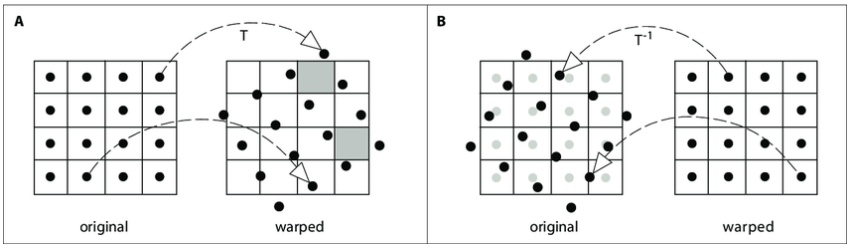

Figure 2: Inverse vs Forward color sampling

We deal with this issue through inverse sampling. If transform from our source image to the frame, the resulting pixels can be spread apart due to the transformation, attentionally leaving massive gaps that are not colored. By inverting the transformation and determining the color of each pixel in the frame by mapping the matrix backwards, we can ensure that each pixel of our frame is colored. If the transformation lands between pixels, we use bilinear interpolation (weighted average of nearest 4 pixels) to choose the color. We sample the source and destination images weighted linearly through our translation, approximating a cross-fade.

Now that we have our interpolated images, we could extend this to only interpolate meshes instead of images. We implement this by only sampling our colors from the source image, resulting in our starting frame looking like our original image, but the image being slowly morphed into the destination mesh in any shape we want. We show two examples of this. The first is the creation of realistic caricatures by morphing the face mesh into the mesh of an existing caricature. The second application is simple animation, where we cause a flower to rotate and shrivel up simply by defining its initial and final mesh shapes.

Results

Figure 3: Image morphing between Idina Menzel (Actor, Elphaba from *Wicked*) and Jamie Muscato (Actor, JD from *Heathers: The Musical*). Both meshes contain 94 points and results in 215 triangles. Images are both 1000×1000 pixels large. Each frame takes 15 seconds to generate.

Figure 4: Image morphing between Brady Bhalla (Student in CS 166) and Jesse Eisenberg (Actor, Brady's Doppelgänger). Both meshes contain 61 points, and images are both 500×500 pixels large.

Figure 5: Image morphing between images of 5 cats. The meshes were made by selecting a total 38 points around each of the cats' face, eyes, and tongue, resulting in a simple triangulation.

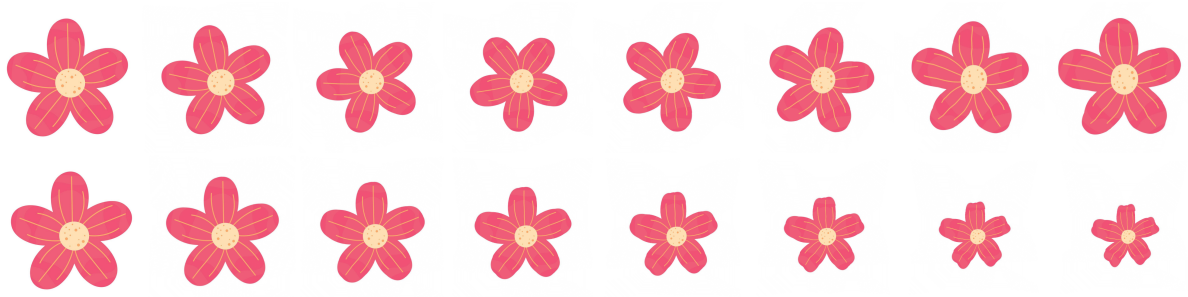

Figure 6: This time, color sampling is only done backwards to test the transitions of meshes between different images rather than interpolating between two images themselves. The flower has a simple mesh of 20 points rotated in the first row, then shrunk in the second row. This effectively creates an animation where we can warp the image as we wish in an intuitive manner.

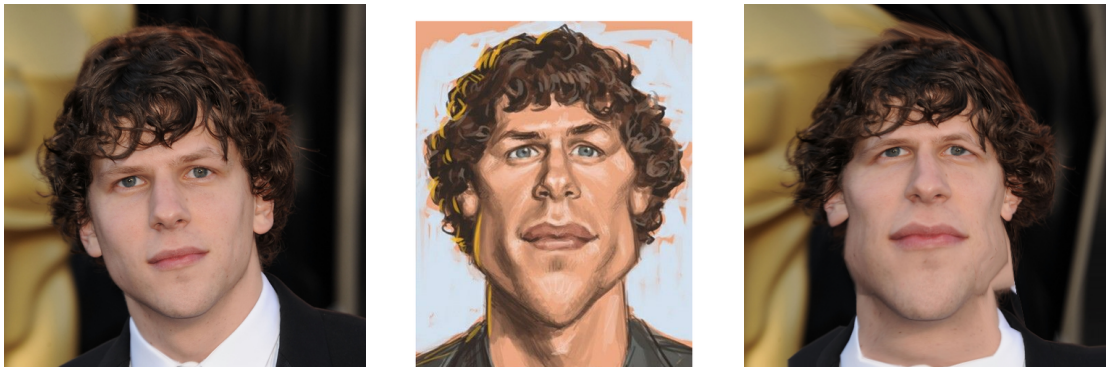

Figure 7: Morphing of meshes can also be used for just its final frame. By translating the mesh of Jesse Eisenberg's face into this caricature of Jesse Eisenberg, we can create a scary but realistic caricature of the actor. The mesh around both images are fairly simple, leading to a somewhat blocky look.

Challenges and Future Extensions

Our algorithm is simple, which makes it very performant. The main purpose of this is to simplify the process of making meshes, but manually selecting points can still cause unexpected issues. For example in *Figure 7*, we can see the right side of Jesse Eisenberg's face seems to be stretch out a bit because the triangulation decided the neck that was too close to the border was in fact part of the border. Although this can be mitigated by selecting better points for the mesh as well as extending the points in the border beyond the frame of the image, this can be a time consuming process.

One solution could be to introduce an algorithm that can recognize key features that correlate between two images to generate these points for us. For example, we could use a convolutional neural network trained on faces to highlight contours of the face and specific features, automating everything.

Although our algorithm works between images of different dimensions, it still scales linearly with the pixel count of the image as well as linearly the number of simplexes in our resulting mesh. This can result in possible slow downs for larger images. However, we propose that more complex algorithms such as Beier-Neely are a better fit for such projects.

It was challenging to debug many of the issues arising through the project, as there was no metric we could use to define what the "correct" results looked like, but everything worked out in the end. Overall, the project was interesting and gave some fun results.

Appendix: Code, Images, Meshes

All code, input images, and generated meshes / images / gifs are stored in a Github repository: https://github.com/ericlovesmath/image-morphing/tree/main