At the simplest level, Image morphing is simply the blending of pixels of image I and J to create M using the equation M(x,y) = (1-alpha)I(x,y) + alpha\*J(x,y) where alpha is a value between 0 and 1. When alpha is zero, M will look like I, and when alpha is 1, it will look like J. This simplistic method will blend the two images, but the faces will most likely not be aligned and this will result in a blurry mess of image M.

Instead, we must establish pixel correspondence between images I and J. Then we can calculate the pixel locations in M. The x and y locations in M can be represented by the following two equations:

Xm = (1-alpha)xi + alpha\*xj (1)

Ym = (1-alpha)yi + alpha\*yj (2)

Then we can use the blending equation (3): M(x,y) = (1-alpha)I(x,y) + alpha\*J(x,y) to find the intensity of each pixel.

First, to find each corresponding points in images I and J, we used dlib to detect facial feature points. Then we added points to the corners of the images.

Using the corresponding points we want to perform Delaunay Triangulation on a set of averaged points for the two images. This will give us an indexed list of triangles that we can use to warp the images for morphing.

To warp or blend the two images we must use equations 1 and 2 to locate the pixels in image M. Then we calculate the affine transform using OpenCV’s getAffineTransform. This will map the three corners of a triangle in image I or J to a corresponding triangle in M.

Then we must warp all of the pixels inside each triangle to M. This is achieved using the function warpAffine. It must be noted that warpAffine only takes in an image and not a triangle, so we had to create a bounding box for each triangle and then create a triangular mask using fillConvexPoly.

Finally after some fine-tuning to hide the seams, the images can be blended using varying values of alpha with 0 looking more like image I, and 1 being closer to J.