A Face Morphing Application

Final Year Project Report

Yohanan Ben-Gad[*]
University of Birmingham

April 2017

Abstract

This report describes the creation of an interface which allows a user to create an image located at an intermediate point between two input images. To do so, the user inputs a series of control point pairs onto the input images, which are then triangulated to form identically structured lattices over the images. These lattices are linearly interpolated to form a new lattice with control points at a specific intermediate point between the two input images. The program then loops through each pixel in the new lattice and uses barycentric coordinates to find the equivalent pixel in each of the input lattices and, by extension, the pixel in each of the input images. The colours of these two pixels are then interpolated and this colour is assigned to the pixel in the new lattice. To assess the quality of my output images, I compared them to the output images formed from the same input images using existing morphing programs. My program succeeds in forming output images that are very similar and occasionally better than those produced by other programs.

All software for this project can be found at:
https://codex.cs.bham.ac.uk/svn/yxb389/FinalYearProject/

[*] I would like to thank Prof Ela Claridge for her guidance and supervision of this project.

Table of Contents

1 Introduction

Traditionally, morphing is a process in which the illusion of a seamless warp is created between two images by displaying in animation a series of frames, each depicting a slight variation from the previous image in the direction of the second image. These intermediate frames are created by interpolating the two images using a series of control coordinates to find visual intermediates between them.

This technique rose to prominence in the 1980's, famously being used films such as 'Willow' and 'Indiana Jones and the Last Crusade' [1].

Morphing sequence from 'Indiana Jones and the Last Crusade' [2]

Of course, if one is simply looking to create an image that acts as an intermediate of two input images, only one of these frames need be generated, and within this broader field, there has emerged a particular emphasis on morphing human faces. While there are multiple ways of doing this (as will be shown later), they all rely on selecting a fixed number of vertices or lines on each face in specific positions (forehead, chin, eyes, nose, etc) which then become the control coordinates around which interpolation is done. In recent years several programs have emerged online that are designed to allow regular people (as opposed to professional special effects artists) to do this themselves.

Arguably the most popular one is MorphThing [3]—this program requires users only input their images, select the top, bottom, right and left sides of the face, and then takes care of the rest of it for them- automatically inserting the control coordinates and carrying out interpolation to produce an image that is exactly midway between the inputs. While this technique has limitations that will be discussed shortly, the program's popularity can be attributed to the convenience its methodology affords to its users.

Then there is Moonjee [4]. That program will automatically assign control points for the user but still allows the user the opportunity to tweak them to fix any imperfections. However, the quality of the images produced is greatly inferior to that of the other two programs referenced here.

Finally, there is Morpheus Photo Morpher [5]. Unlike the examples cited above, this program was not specifically designed for face morphing and thus can only be used by people with some expertise. This is software designed for morphing any kinds of images, where the users is left to choose how to input the control coordinates themselves. A 15-frame animation is then produced showing the transition from the first image to the second, though screenshots from this can also be extracted. This software will be referred to frequently throughout this report, as a point of comparison, and as a way of testing alternative methods.

The aim of the project described in this report is to create a functional and user friendly piece of software that can create face morphs—be they of real humans, animals, drawn images or computer generated models, to produce results that are of as high a level of quality as those produced by any of these programs or any other existing software available on the web. Furthermore, my intention was to expand upon the central area that these programs lack: the ability to vary the ratios between the different inputs. Unlike existing programs, which only offer a single ratio between the input images (exactly halfway between them), or in the case of Morpheus—fifteen (two of which just being the input and output images), my program is based around a 100 point slider, allowing for as large a variety of morph rates as anyone could realistically need. Using this wider variety of images, users will be able to construct morph animations ranging up to one hundred frames, or depict these morphs in 'filmstrips' (multiple images positioned next to each-other such as each row in the above figure) of the same length. To do this, rather than using precise numerical values and percentages, I would need to construct my interpolation code around dummy values which would then be determined by the position on the slider.

2 Overview of existing morphing methods and algorithms

2.1 Introduction

Before looking at specific methods of image morphing, one needs to understand some of the broader characteristics. For the sake of simplicity, I will demonstrate these concepts using a single source image rather than two, although it will soon become clear how they can be expanded on to include more than one source of input.

All methods of image morphing are based on finding pairs of points, lines or pixels (and in most cases more than one) in the source and destination images. In the broadest sense, there are two possible ways of doing this—forward mapping and reverse mapping.

Forward mapping: In this method, the program goes through the points, lines or pixels in the source image uses information about them to find the locations of the corresponding points, lines or pixels in the destination image.

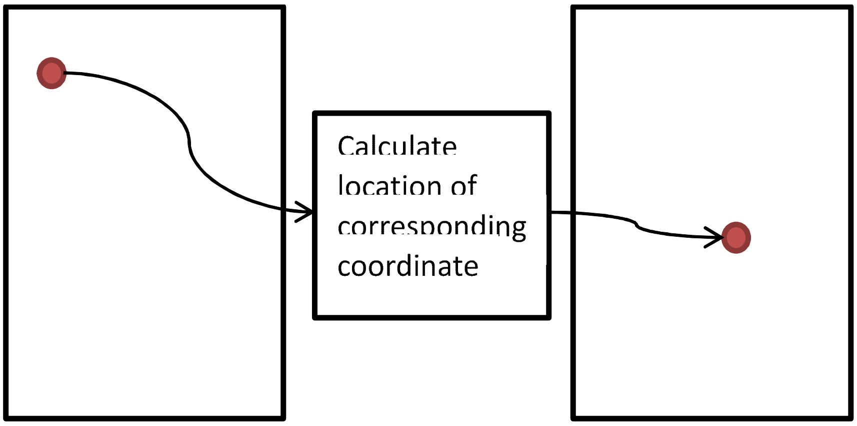


Diagram of forward mapping

Reverse mapping: The opposite of forward mapping. The process goes through the points, lines or pixels in the destination image and uses information about them to find the locations of the corresponding points, lines or pixels in the source image.

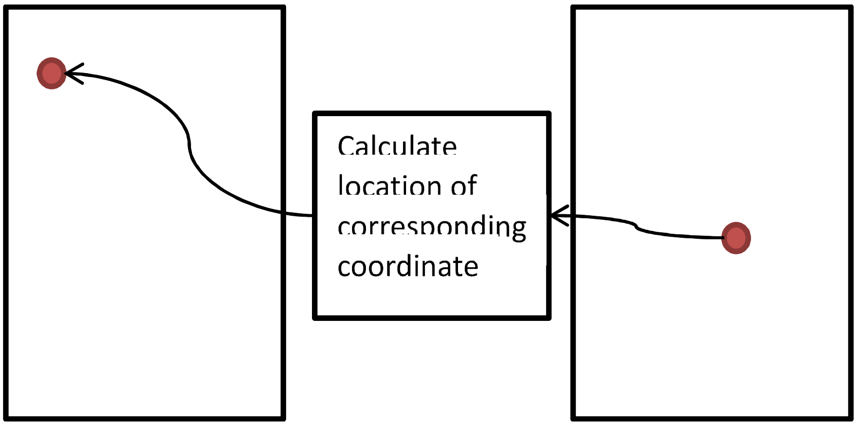
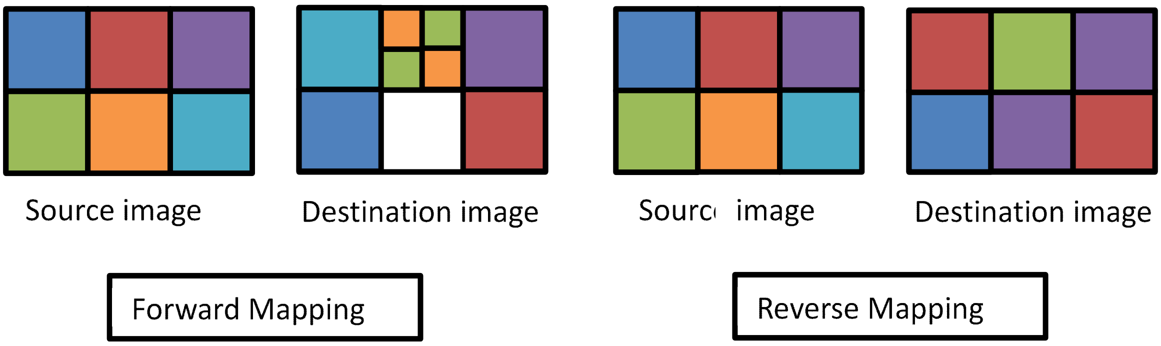


Diagram of reverse mapping

Reverse mapping has the advantage over forward mapping that all pixels in the destination image are assigned a corresponding pixel. In forward mapping this is not that case, as in some cases, more than one point in the source image will correspond to the same spot in the destination image.



An example of what can happen in reverse mapping. Each square represents a pixel, with pixels of the same colour being linked together by mapping. Note that the orange and green square represents that the pixel corresponds to both the orange and the green squares in the source image.

2.2 Field Morphing

This method is outlined by Beier and Neely [6]. Unlike the methods that will follow, field morphing uses lines rather than points as its control variables. One uses the geographic relationship between each pixel and these control lines in the destination image to calculate the location of the corresponding pixel in the source image.

TRANSFORMATION WITH ONE PAIR OF LINES:

This technique utilises reverse mapping. Begin by selecting a line PQ in the destination image and $P'Q'$ in the source image. For each pixel X in the destination image, find X' in the source image by carrying out this series of equations:

$$u = \frac{(X-P)\cdot(Q-P)}{\|Q-P\|^2}$$

$$v = \frac{(X-P)\cdot \text{Perpendicular}\,(Q-P)}{\|Q-P\|}$$

$$X' = P' + u\cdot\left(Q'-P'\right) + \frac{v\cdot \text{Perpendicular}\,(Q'-P')}{\|Q'-P'\|}$$

This transforms each pixel by rotation and/or scale.

TRANSFORMATION WITH MULTIPLE PAIRS OF LINES:

This is used for more complex transformations. A value X'_i is calculated for each pair of lines and this is used to find the distance D_i between X and X'_i. A weight is then assigned to each D_i based on

$$weight = \left(\frac{length^p}{a + dist}\right)^b$$

where the constants a, b, and p determine the relative effects of the lines. This is then used to calculate the weighted average for D and thus the value of X'.

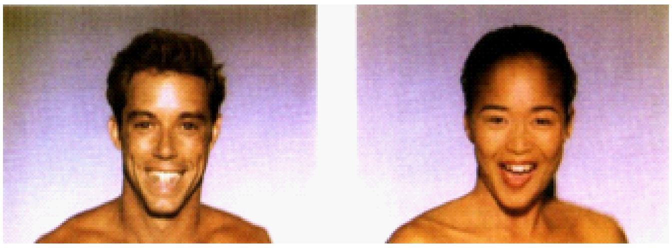
MORPHING BETWEEN TWO IMAGES:

Begin by interpolating the lines in Image 1 with the lines in Image 2 to form a midpoint.[1] This means that if we perform a transformation with multiple pairs of lines on both images with the midpoint as the destination image, these images:

Initial images

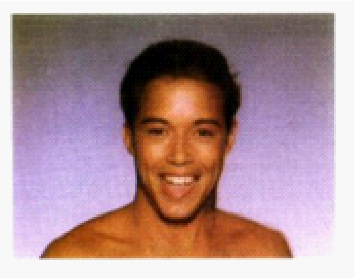
[1] In this project, the term 'midpoint' is used to mean any intermediate point between the two inputs. The position of this midpoint may be determined by the user.

are transformed into:



Intermediate stage

Taking the midpoint of the colours of each pixel then gives us the final image:



Final stage

Advantages:

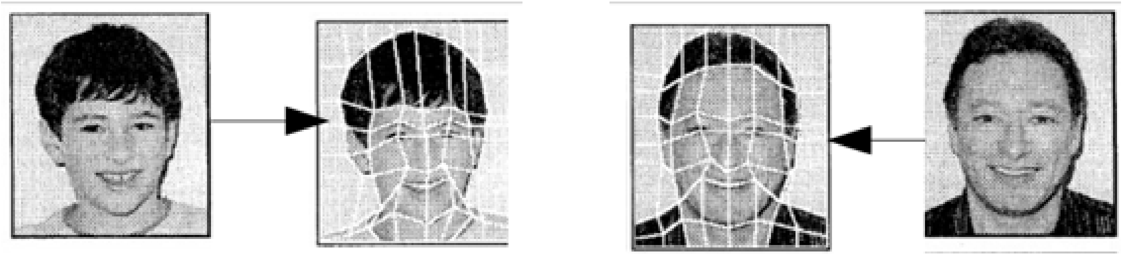
- Easy to change the variables (no. of lines and values of constraints can even be determined by the user without extra coding being required).
- Very simple to code.
- Natural, easy control of line segments.
- Always one-to one, i.e. there will never be two points in the destination image that map to the same point in the source image (this is a result of using reverse mapping).

Disadvantages:

- Slow
- May still require some debugging.

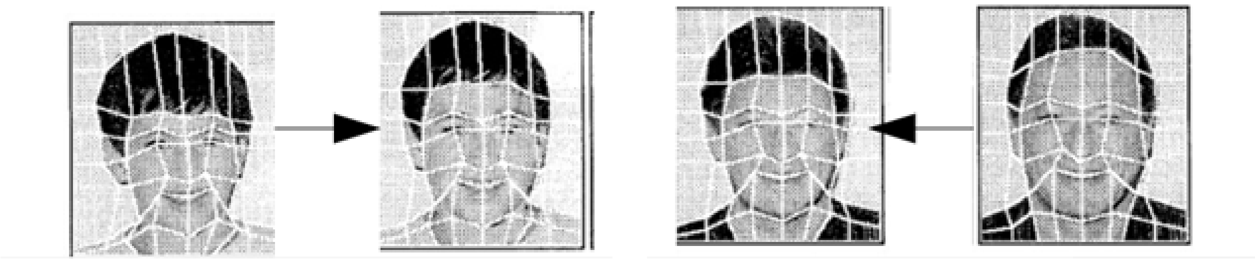
2.3 Mesh Warping

Described by Wolberg [7], mesh warping works by selecting a finite number of control points on each image such that each point on one image is paired with another (we will refer to these pairs by writing (letter, letter')). Lines are then drawn between these points so that if *ab* is a line in Image 1, $a'b'$ is a line in Image 2. Importantly, it is vital that no two lines ever cross paths.



First stage of mesh warping

The two meshes are then interpolated to a mid-point, and both Image 1 and Image 2 are then warped towards that midpoint. They will thus both have the same 'shape'. The eyes, mouth edges and/or any other points specified by the user will be in the same place for both images:

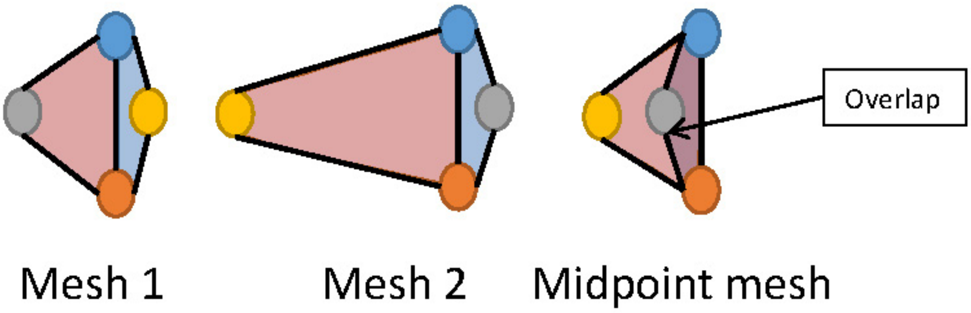


Second stage of mesh warping

Now all that remains is to interpolate the colour of each pixel to a corresponding midpoint:

Final stage of mesh warping

One obvious potential problem is that triangles will cross over themselves when finding the midpoint, creating an overlap. For instance, in the following diagram, we have two input meshes of four coordinates and two triangles each. When the midpoint is taken however, one of these triangles in the output mesh ends up directly on top of the other.

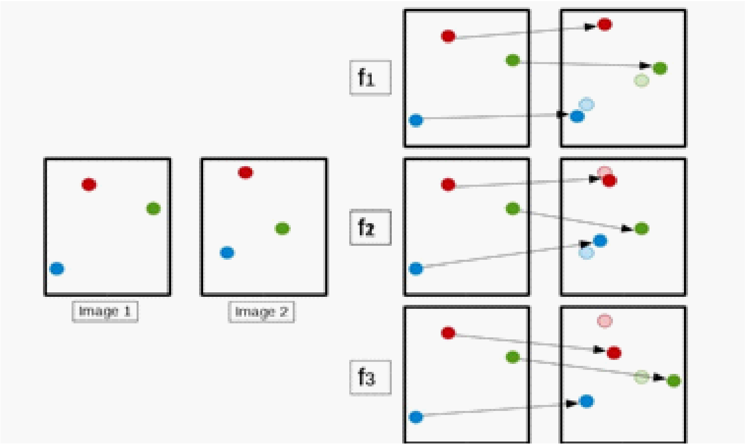


An example of what can go wrong in mesh warping.

Thus, it is necessary that both the horizontal and vertical order of the control points be the same for both images. Unlike field morphing, which can only be done with reverse mapping, mesh warping can utilise both forward and reverse mapping to fill in the pixels between control points.

2.4 Inverse-Distance Weighted Interpolation

In their paper, Ruprecht and Muller proposed another way of utilizing control coordinates as a basis around which to interpolate the rest of the pixels [8]. Here, a single, uniform function is found that can be applied to every pixel. This is generated by generating a different function for every set of control points that perfectly maps one to the other. Then, an 'average' value is calculated over all the functions by assigning each one a weight based on how far off the other control points are projected to be from their required destination when this function is applied to them.

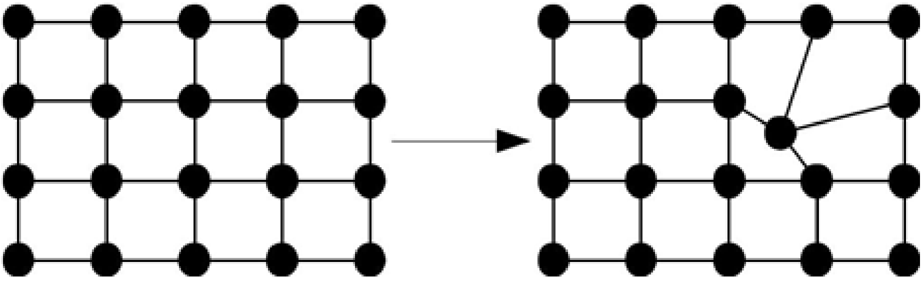


The functions for each of the points on the two input images are illustrated on the right, with f_1 being the function for the red point, f_2 for the green point, and f_3 for the blue point. The paper goes on to describe several different methods for finding these weight functions.

A variation of this can also be generated, which, for every pixel, will take the weighted average of the distances of each control point from that pixel and apply these weights to the functions associated with them. This is not dissimilar to field morphing, though obviously slightly more complex.

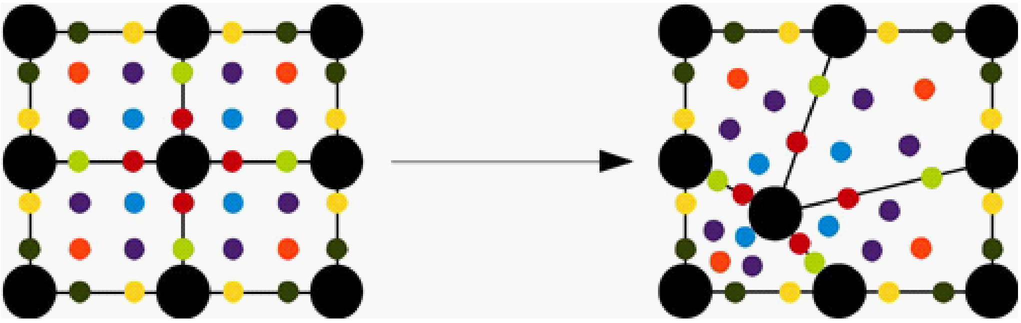
2.5 Free-Form Deformation

Described by Calvin and Maurer [9]. This technique involves overlaying a rectangular lattice of points onto the image so that each corresponding point in the image has coordinates in the xy plane. The lattice is then manipulated by a point being dragged to a new location (in some instances, this is done by hand by the user).



A control point being dragged inside a lattice.

The pixels within the mesh spaces around this point are then manipulated, with the distance at which each pixel is moved corresponding with that pixel's distance from the source and destination coordinates of this control point.



The pixels (small dots) are displaced along with the control point (large dots).

As a result of this transformation, pixels that overlap can be easily combined into a single pixel and any empty spaces left behind can be easily filled by interpolation.

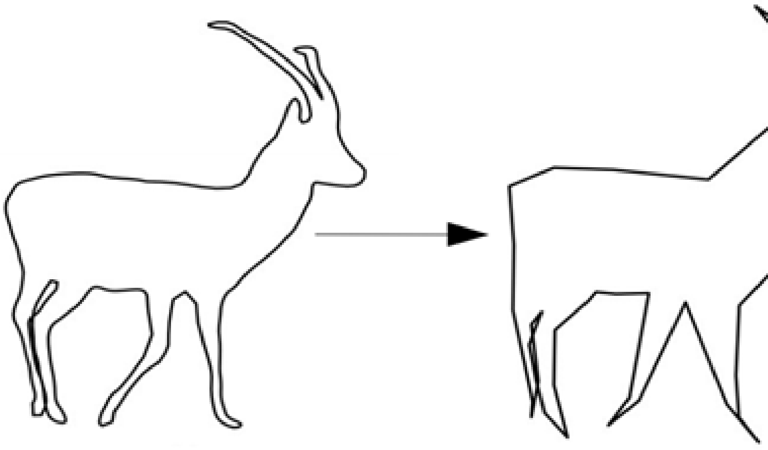
As with regular mesh warping however, this has the clear disadvantage that points can only be moved a limited distance without further techniques being needed—in this case, within the square grid-space bounded by the eight surrounding vertices.

One possible solution to this problem is to perform multiple deformations on multiple lattices. In other words, after carrying out a deformation on a point, discard the lattice and create a new one with a different set of intervals appropriate for the next movement that needs to be made. Then simply repeat the process until the point has reached its final destination.

2.6 Feature Preserving Texture

The image preserving texture algorithm begins by tracing the edge of the two objects the user wishes to morph using a series of connected lines. A correspondence is then established between these two sets, with lines cast over the corresponding points in each image being paired together.

This, however, is not a one-to-one matching. There is no guarantee that, unless the user goes out of their way to do so, every line will be matched with a corresponding one in the other image. In fact, there is no reason to expect that for each image the same number of lines will form its outline. To fix this, a finite number of paired lines are selected which, when extended and linked together, create a much cruder but still recognisable outline of each image.



The outline of an image is reformed.

Triangulations can now be performed, followed by texture mapping—which once again simply involves finding the appropriate midpoint of the colours of the corresponding pixels in both original images. In Tal and Elber [10] the process described will automatically create traces and matchings between the two images. For the purpose of relevance, this procedure is simplified here so that it is entirely user-controlled.

2.7 Conclusion

The next decision was to choose which method to actually implement. The first option I rejected was Feature Preserving Texture. This is a method that is only really suited for morphing shapes with highly distinct outlines and features that are vastly different to each-other. The differences between human faces are, by contrast, extremely subtle, and these differences would simply not be detected. Free-form Deformation was not much better. While this would potentially be the easiest and most convenient for the user, the amount of work that would be required for coding simply impractical. Not only would all the morphing functions themselves have to be encoded like for all the others, but on top of that I would have needed to create a program that could let the user manipulate a virtual lattice, something for more complex than simply having them enter points or lines. Field morphing was also rejected when I realised how frustrating it was to try to create perfect lines across a specific space with a computer. I would often realise that I had begun the line in the wrong place after extending it out and attempting to find the correct ending point for it—leading to a waste of time.

Thus, it came down to Mesh Warping and Inverse Distance-Weighted Interpolation, and between these there was a clear and obvious choice. Mesh Warping is much simpler to code and there is no evidence of it being any less effective. The only potential problem was that of overlap as described earlier, but this was not likely to cause too much trouble if the user inputs the control points correctly (all faces have a similar structure, even those of animals and cartoons).

3 Design and implementation

3.1 Image Input

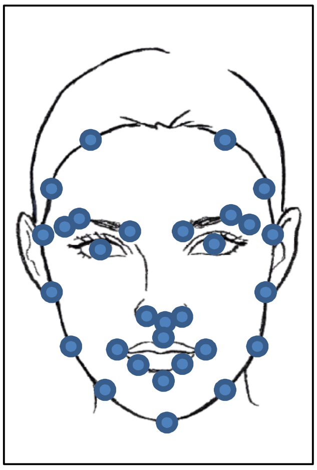
After the user inputs the two images they are displayed on the GUI and stored in the system as Java objects. This is somewhat unusual for programs, which typically convert the images into 2D arrays of colour values. Although this process is known to theoretically make spatial interpolation easier, I found that it made it more difficult to visualize the process on a practical level, and thus to code it.

3.2 Control Points

3.2.1 Input

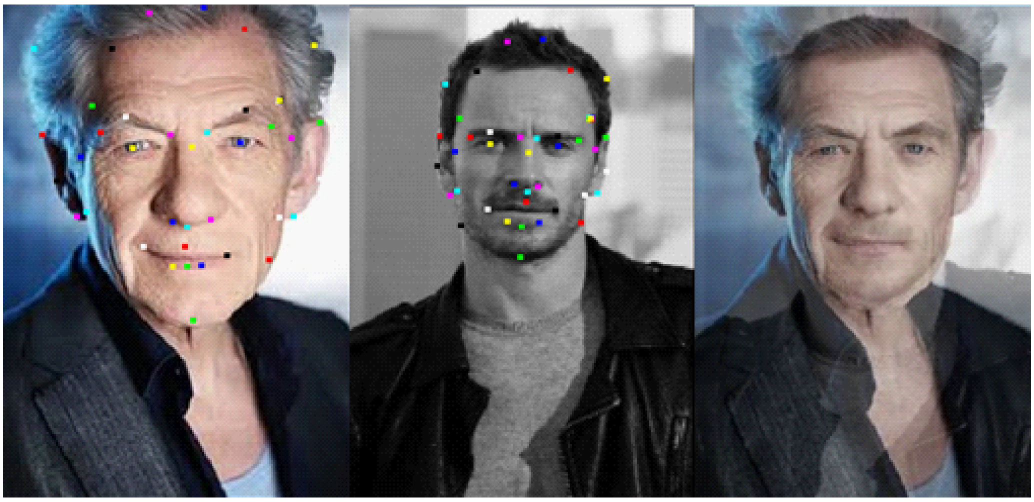
After deciding on mesh warping, the question became—how would the control points be found? The easiest answer from a usability standpoint would be to have the program detect them automatically. While implementing the code for this myself would have been a task the size of a whole other project, various packages exist around the internet that do this automatically. However, most of these are only designed to work for real human faces, and while some pre-made programs for detecting animal faces also exist, this technology is still in its infancy when in terms of detecting drawn characters [11]. This was made particularly apparent to me when MorphThing, which as stated previously detects control points automatically, required me to manually enter control points when I attempted to upload a cartoon. Thus, in order to allow for as much diversity in the input images as possible, I opted to make the user enter the control points themselves.

Once this was determined, I had to decide how many control points I would use and where I was going to place them. Given that one of my main sources of research was Wolberg's paper, I chose to use his method as a starting point.



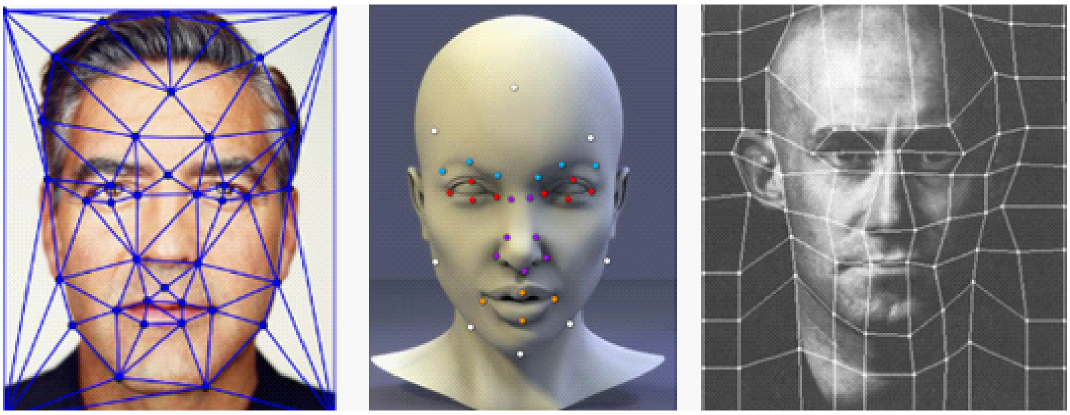
The configuration used in Wolberg's paper

To test this method out for myself, I downloaded Morpheus Photo Merger and morphed two photos together by selecting these points:



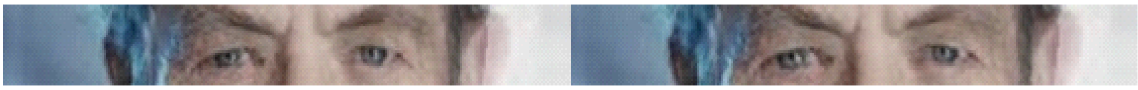
Screenshots from the Morpheus program. Left and centre—after completing the point insertion. Right—the resultant morph

This certainly doesn't look bad, but is it possible to do better? Searching through scholarly articles and just the internet in general, I found no examples of a radical departure from this template. However, there were a few variations:



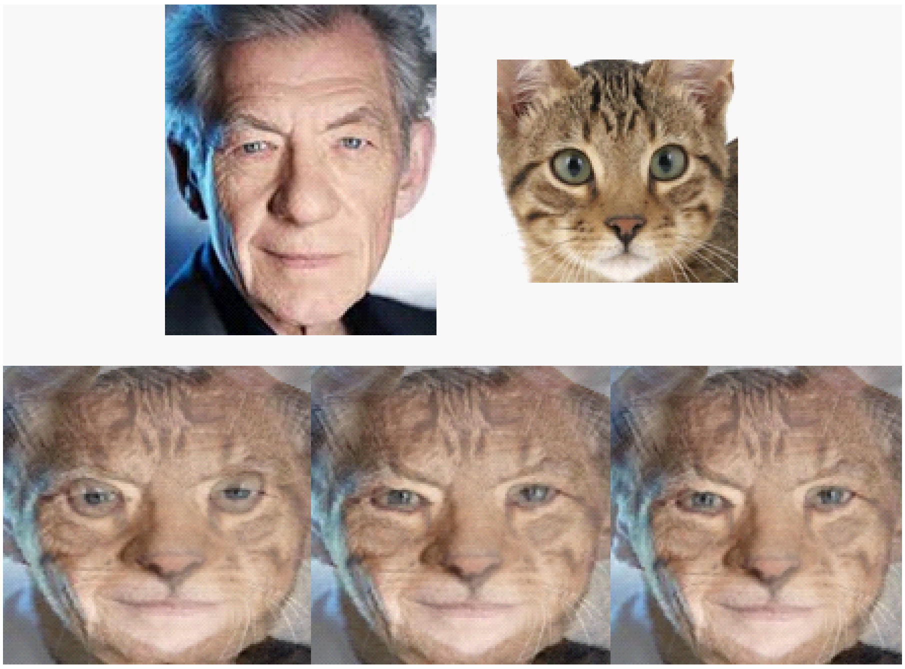
Left: https://inst.eecs.berkeley.edu/~cs194-26/fa15/upload/files/proj5/cs194-ao/, Center: http://faceonface.net/en/morph/, Right: https://link.springer.com/article/10.3758/BF03207733

This certainly does not look bad, but is it possible to do better? Searching through scholarly articles and just the internet in general, I found no examples of a radical departure from this template. However, there were a few variations:



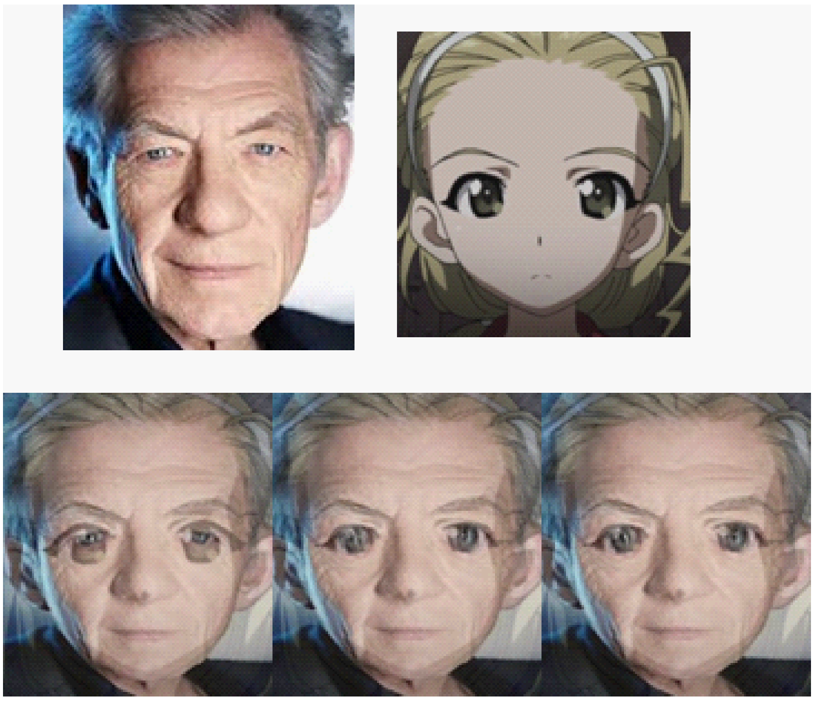
Left: Output from using one point in centre of each eye. Right: Output from using four points around each eye.

Clearly, depending on one's ability to spot detail, the differences between these two photos range from completely invisible to practically invisible, and it would therefore be tempting to simply write off the four-point method as an unnecessary complication. However, the results only came out this way because human eyes are so similar in both shape and size. The same cannot, however, be said for animals:



Bottom left: One point in centre of each eye. Bottom centre: Four points around each eye. Bottom left: Both.

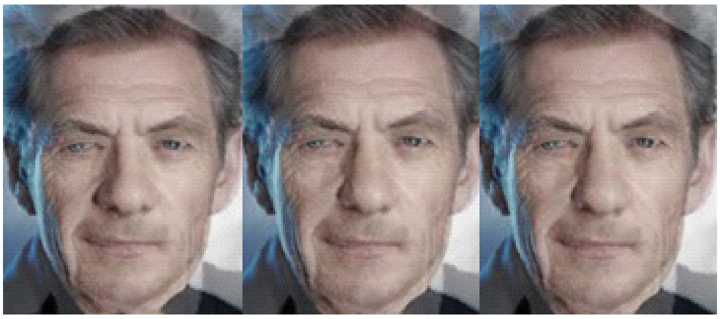
Or cartoon characters:



Bottom left: One point in centre of each eye. Bottom centre: Four points around each eye. Bottom left: Both.

The four-point method is clearly a vast improvement over the one-point method in both of these cases, with the combination of the two also showing a marginal improvement over just the former. Thus, for the sake of expanding the range of things the user could do with the program, I chose the combination of the two as the configuration I would use.

Notice also that several of the above examples also offer a slightly different method for defining the nose. Rather then just placing a single vertex underneath it and a vertex below each nostril, the latter two vertices are now moved to the side of the nose, with two extra vertices now being added directly above the point where the nose begins to stick out. In addition, the single vertex at the centre point between the eyes has now been replaced by two vertices on either side. Implementing this produces the following:

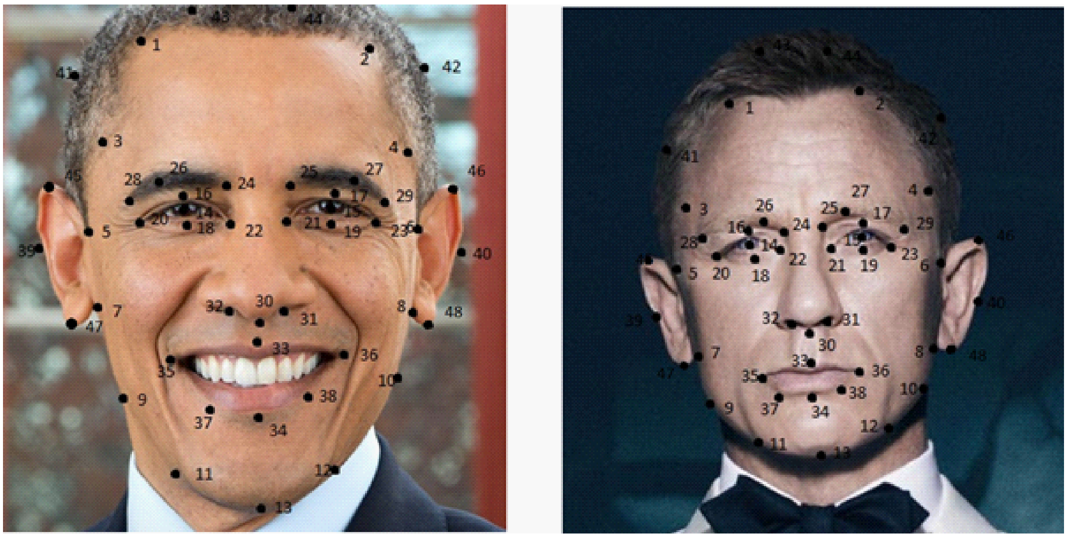


The outputs for the three configurations of the nose. Left: original method. Centre: new method. Right: Both.

Given that the new method uses more coordinates, it was surprising that this made the image look worse, with the edges between the nose and the rest of the face suddenly blurrier and less well defined. After repeating this several times with different sets of images and showing all three combinations to several people, the unanimous agreement was that the one using the original configuration method always looked the best and most life-like. While it's entirely possible that a 'perfect' positioning of the points would result in a better image in the latter two cases, this is not something one could realistically count on the user to do, and thus it made no sense to change the configuration of the coordinates outlining the nose from what they already were. I continued to test various other variations found in these examples and others like them, but ultimately none of them offered any improvement on the original model, leaving the four-point selection process as the only change that was implemented.

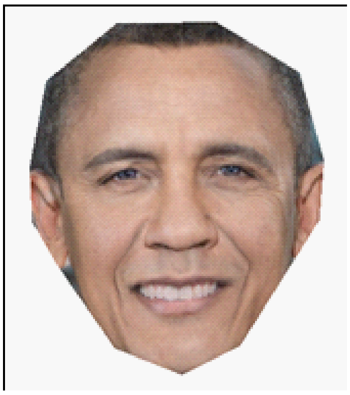
After the images are selected, the user must input 48 control points onto each of the images. This process will be expanded on in detail below.

After inputting has been completed, for each of the two input images, the 48 coordinate sets will be stored in an array, and because of the way the point selection process works, the ordering of these two arrays will be the same. (In other words, the coordinates for the middle of the left eyebrow on the first image are stored in the same position as the coordinates for the middle of the left eyebrow on the second image.)

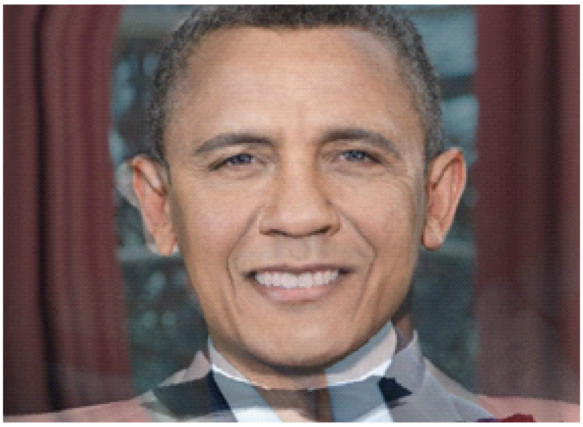


The order in which the user is directed to enter the control points.

The program will then automatically add a coordinate at each of the corners of both images. This is so the entire images will be included in the triangulation, and not just the faces themselves. This might seem like an unnecessary step. After all, why not just leave the background out? The answer is that doing so would produce this:



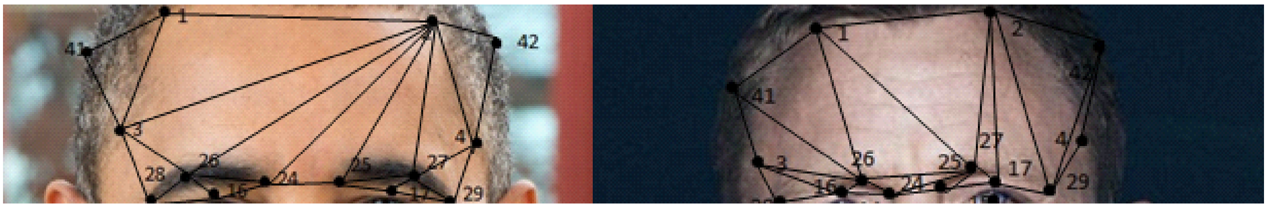
As can be seen, the outline of the image is formed from the outer triangle edges, creating a jagged, unappealing result. However, when the background is included as part of the triangulation and, thus, the image, this is the result:



Clearly, a significant improvement.

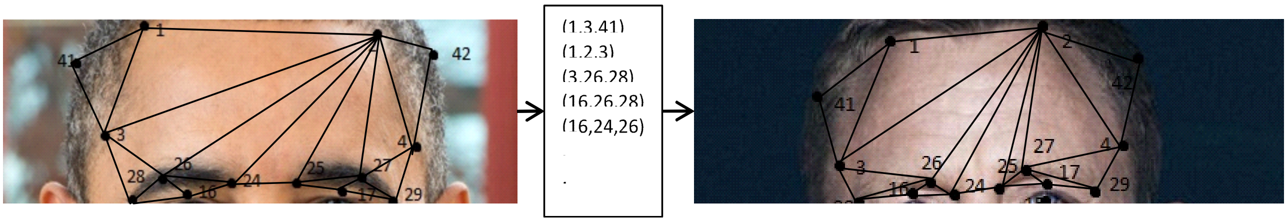
3.2.2 Triangulation

Triangulation is the process by which a selection of vertices are all connected to each other by edges (which cannot cross over each other) to form an elaborate, interconnected set of triangles. To save on time, I decided to download an existing software from the internet that did this automatically (given a list of the control coordinates).[2] The obvious next step at this stage would be to triangulate both images to form the lattices. However, when testing it out, it became clear that this software had not been designed to fit the requirements of this project, and consequently the triangles formed in each lattice were often inconstant with each other.



Note that the triangles connecting the lattice over the face to the corners of the image have been left out for the sake of simplicity.

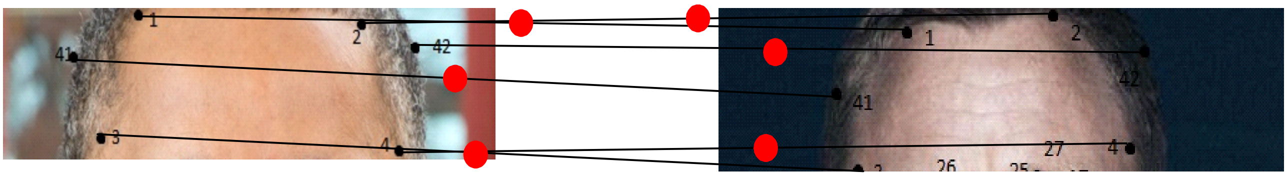
To get around this problem, I set the program to first only triangulate the first image. In doing so, the triangulation software automatically created a list of the triangles formed and their control points. Using this, it was then easy to simply make the program loop through this list and recreate the same set of triangles, only this time with the coordinates for each control point corresponding to those in the second image.



The stages of triangulating the input images.

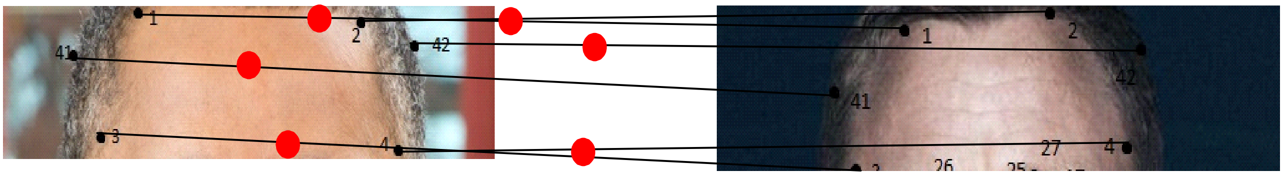
3.2.3 Interpolation

Next, the program creates a new set of coordinates. This is done by linearly interpolating the two sets of input coordinates, as dictated by the choice of mesh warping as a method. The specific positioning of these points is determined by the ratio as set by the user.



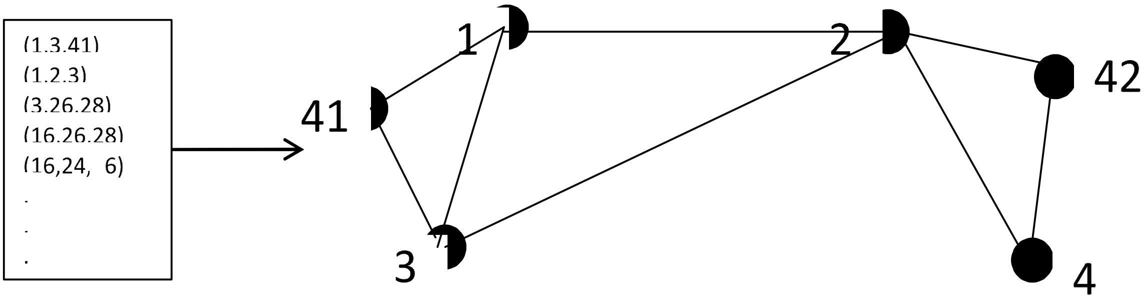
[2] Delunay_Triangulation. Author- Boaz Ben Moshe. Applet available at- http://www.pi6.fernuni-hagen.de/GeomLab/VoroGlide/

Ratio = 0.5.



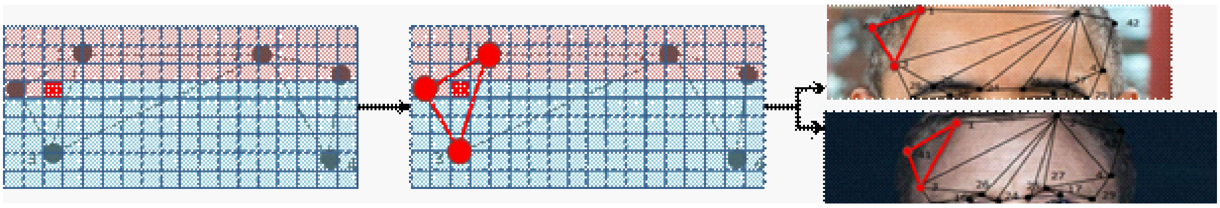
Ratio=0.25.

Once this has been done, the coordinates are triangulated from the list in the same fashion as for the second input image.



Like the lattices for the input images, this resultant lattice will be in the shape of a rectangle (again, due to the coordinates selected at the corners of the input images). The program creates a 2D colour array with the dimensions of this rectangle. This will be used for storing pixel values of the output image so that it can be saved.

The program will now loop through all pixels in the newly created lattice. For each pixel, the program will use an inbuilt method from the triangulation software to find the triangle it is inside from the list. The program then selects the two triangles in the same position from the lists corresponding to the lattices for the input images. Somewhere in these two triangles are the two pixels corresponding to the selected pixel in the new lattice. Now it is simply a matter of finding them.



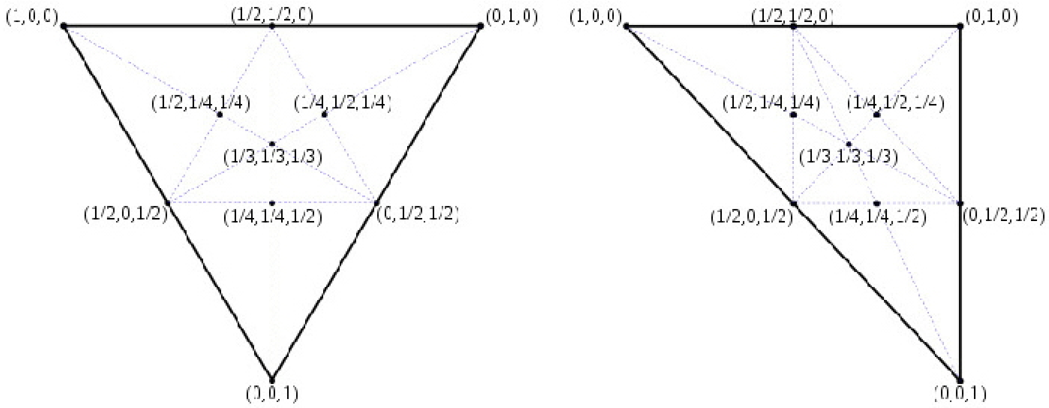
If a pixel overlaps two triangles, it will be assigned to the one the majority of it falls within.

3.3 Images

3.3.1 Spatial Interpolation

There are multiple ways of interpolating pixels around control coordinates, but the best method I could find was to use barycentric coordinates. This is both simple and elegant to code, while suffering from no drawbacks in terms of accuracy.

To understand how the barycentric coordinate system works, imagine a triangle with a weight attached to each vertex. Let W_1, W_2 and W_3 represent the value of each weight as a percentage of the total weight (i.e. $W_1+W_2+W_3 = 1$). For any point on this triangle, there will be a set of values for W_1, W_2 and W_3 for which that point is the centre of mass. The set of these values is that point's Barycentric coordinates. What's more, any other triangle will feature a point on it with these exact same Barycentric coordinates.



Source: Wikipedia—Barycentric Coordinates

Amazingly, mapping every point in one triangle to the equivalent point in the other with the same barycentric coordinates gives a perfect spatial interpolation. The beauty of it is that one can dispose of the complexity that comes with using Cartesian coordinates of having to keep track of the location of each pixel in the images themselves. Rather, we can now whittle this down to their location in just the triangles themselves.

Here is how the process works: one simply takes an output pixel and converts it from x,y coordinates to barycentric coordinates using the following sets of equations:

$$\lambda_1 = \frac{(y_2 - y_3)(x - x_3) + (x_3 - x_2)(y - y_3)}{\det(\mathbf{T})},$$

$$\lambda_2 = \frac{(y_3 - y_1)(x - x_3) + (x_1 - x_3)(y - y_3)}{\det(\mathbf{T})},$$

$$\lambda_3 = 1 - \lambda_2 - \lambda_3,$$

where x_1, x_2, x_3, y_1, y_2, and y_3 are used to represent the coordinates of the three input points that form the triangle surrounding the pixel and the matrix $\mathbf{T}$ is defined as:

$$\mathbf{T} \equiv \begin{pmatrix} x_1 - x_3 & x_2 - x_3 \\ y_1 - y_3 & y_2 - y_3 \end{pmatrix}.$$

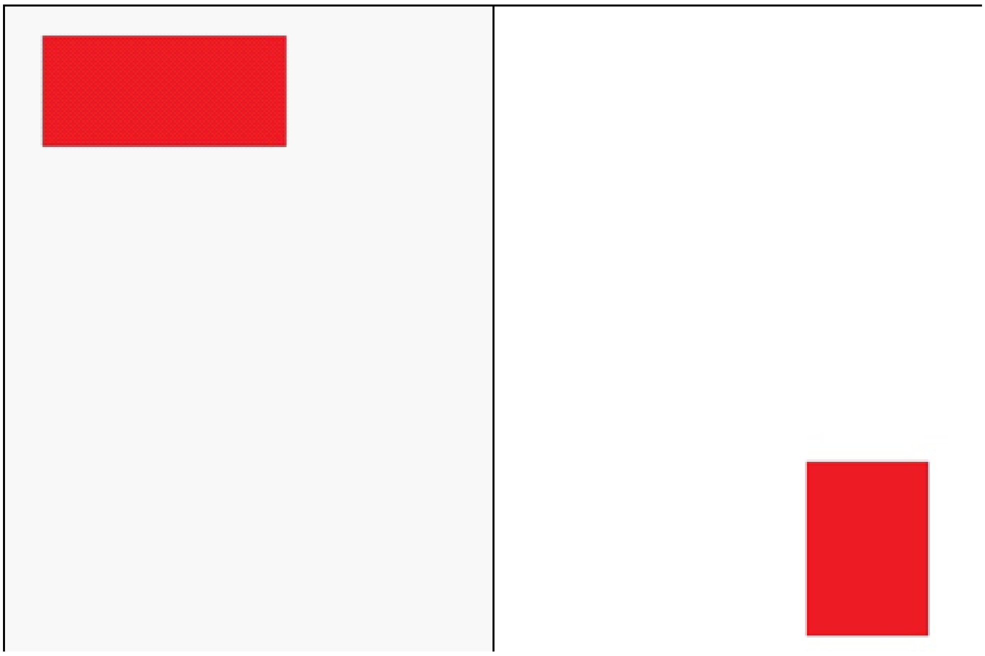
Now input the λ values calculated above into these equations:

$$x = \lambda_1 x_1 + \lambda_2 x_2 + \lambda_2 x_2,$$

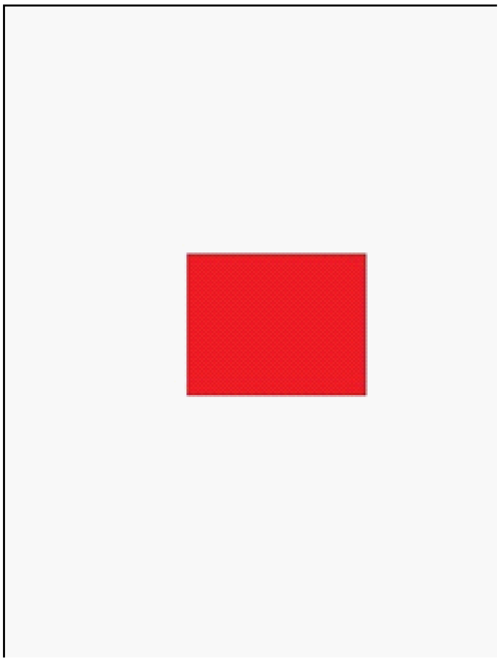
$$y = \lambda_1 y_1 + \lambda_2 y_2 + \lambda_2 y_2.$$

Note that x_1, x_2, x_3, y_1, y_2, and y_3 first represent the coordinates for the input points in the triangle from the first lattice, and then from the second lattice. After rounding the resultant x and y values to the nearest integers, we end up with the coordinates for the two pixels we need.

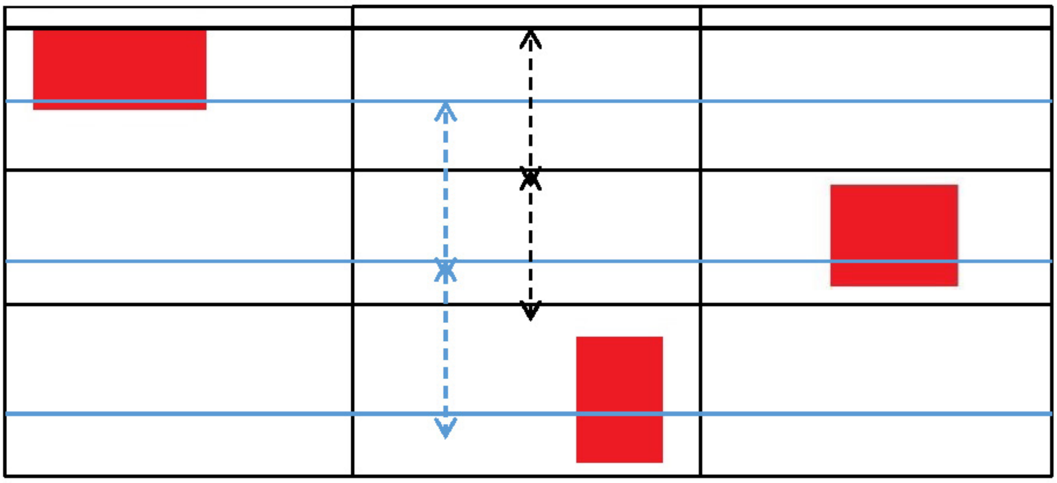
Due to the complexity of the morphing process, it was important to continuously test the code while I was writing it to avoid a heavy amount of backtracking later. And while running the program on two human faces may make it appear as if the program is working, how can one be sure that the outputted image is truly an accurate amalgamation of the inputs? It was therefore important that I first begin the testing process with images that were as simple as possible. Thus, after completing the spatial interpolation part of the code with the ratio set to 0.5, I inputted these two images, marking the corners of each square and each image with a control point:

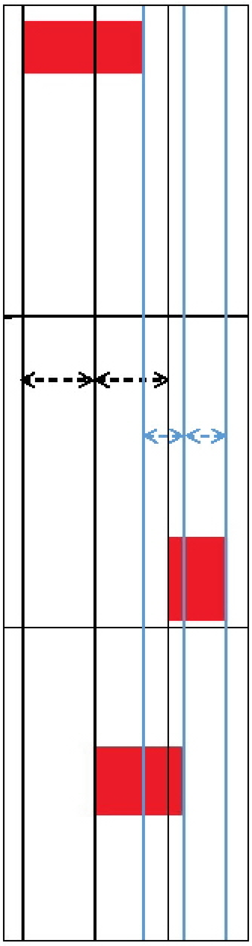


And the program outputted the following:



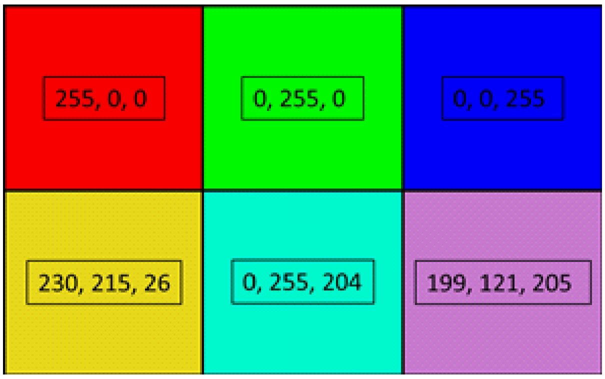
The following diagrams confirm that the dimensions and location of the square in the output image are indeed exactly halfway between the two input images:





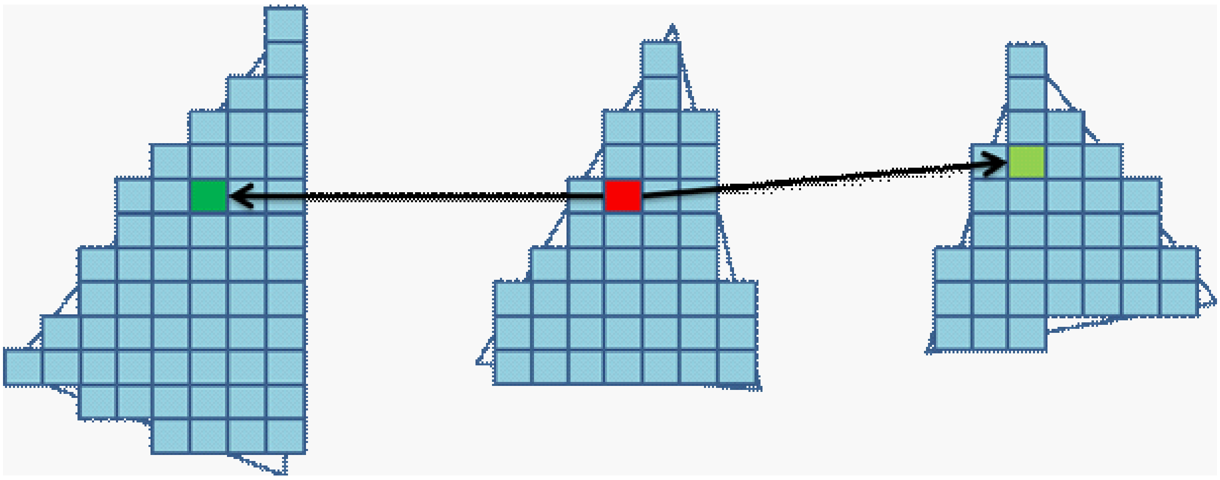
3.3.2 Colour Interpolation

The colour of each pixel in the input images is retrieved by transforming it into an RGB vector.



Examples of RGB values.

An RGB vector is a set of three numerals, each between 0 and 255, that represent the red, green and blue values that go into a colour. Now the program must find the find the correct colour for the output pixel. To do this, it simply takes the colour of the pixel it found from the first image, translates it into a number, and multiplies this number by one minus the ratio you are working with (this will be between 0 and 1). Then it takes the colour of the pixel from the second image and multiplies its numerical value by this same ratio. The two results are added together and the result is converted back into a colour. This is then stored in the 2D array in the position corresponding to the x and y coordinates of the pixel.



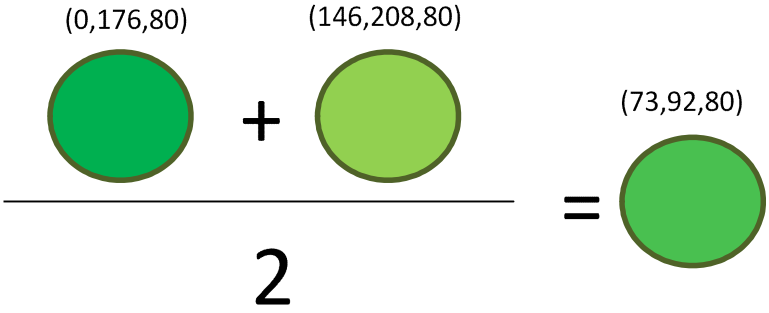
In this example, the ratio is 0.5, meaning that one can simply take the average of the two colour values to get the result:

$$K = \text{ratio}\,(\text{parameter in }\ [0,1])$$

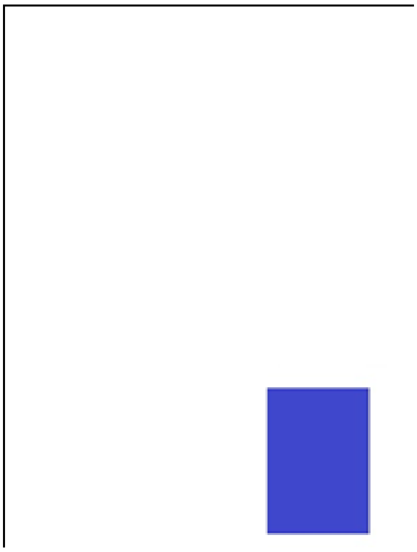
$$C_1 = (r_1, g_1, b_1)$$

$$C_2 = (r_2, g_2, b_2)$$

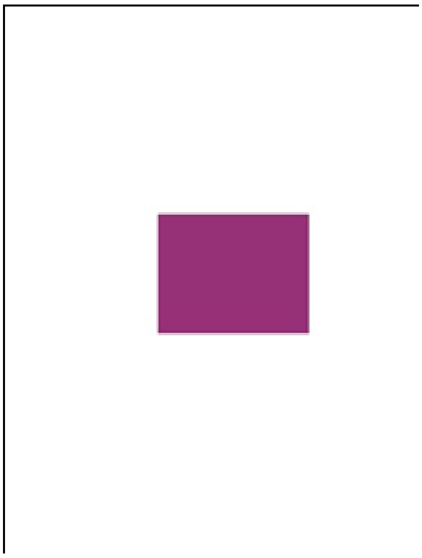
$$C_{interpolated} = (1 - K) * C_1 + K * C_2$$



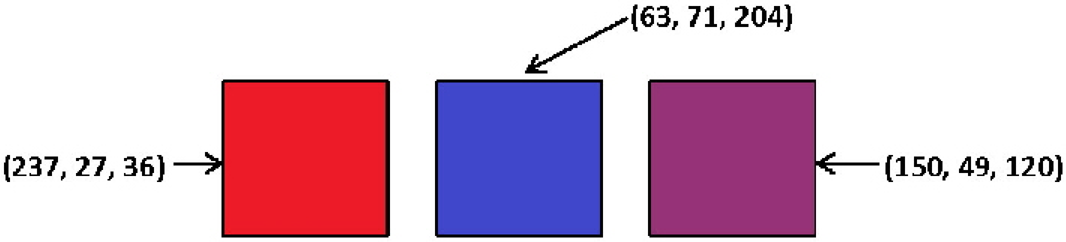
To test this out, I changed the colour of the box in the second input image from before to blue:



And the resultant output was:



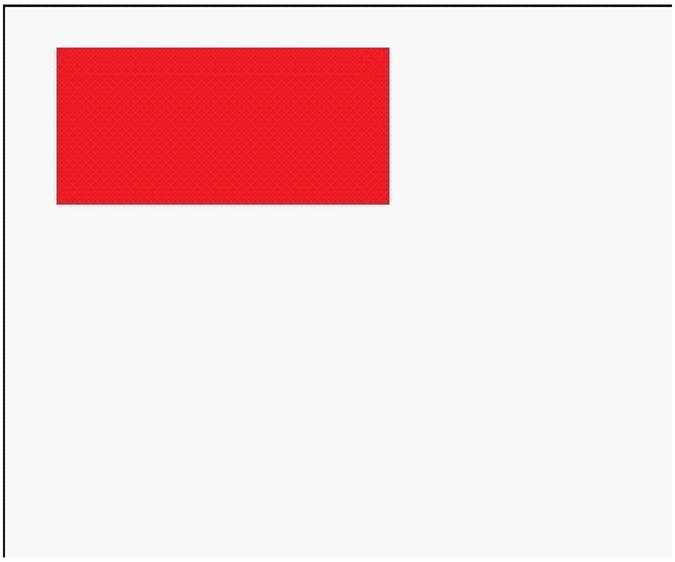
Using the colour detection tool in Microsoft Paint, one can easily use the RGB values to check that this shade of purple is the correct one:

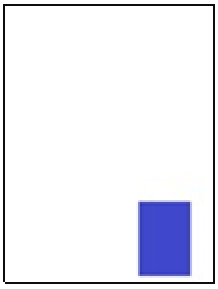


Note that:

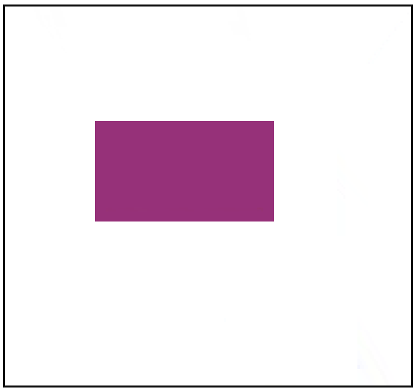
$$150 = \frac{237+63}{2}, \ 49 = \frac{27+71}{2}, \ 120 = \frac{36+204}{2}$$

After modifying the code to accommodate photos of different sizes and dimensions, I inputted the following two photos:

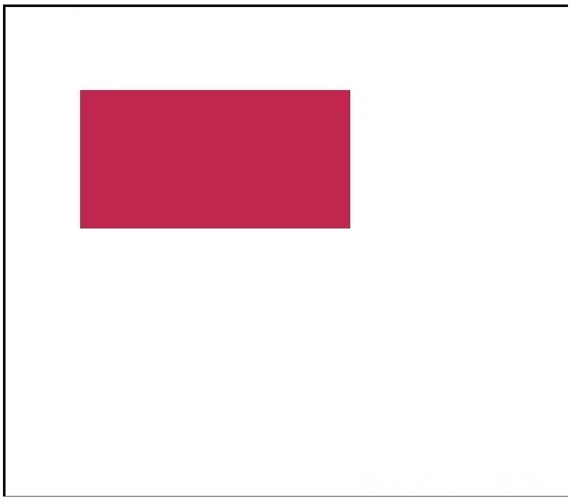


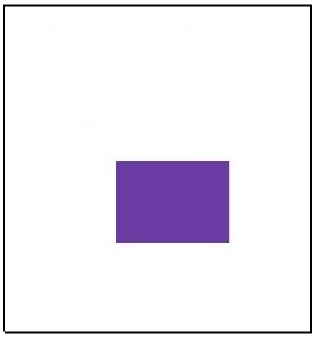


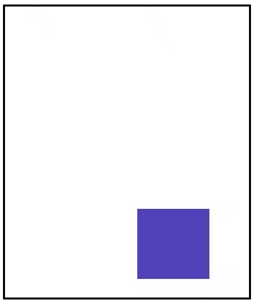
and generated this output:



Finally, I inputted the same two images but changed the ratios to 0.25, 0.75 and 0.9, respectively, getting:







All aspects proved to be correct when checked.

3.4 Image display

At this point there were two options. When designing the program, I could have set it either to draw each pixel straight onto the frame as soon as it is calculated, or to wait until all the pixels have been looped through and then insert the finished image (which is stored in the 2D array) onto the frame. I used the former approach while working on the code, as this made it far easier to detect bugs in the code that caused the program to terminate partway through the looping process. I initially intended to keep it this way, as this created an interesting wiping effect that is not only fun to watch, but also assures the user that the program is working immediately rather than forcing them to wait several seconds until the program has finished compiling to see any results. However, this had one major drawback-whenever the GUI was minimized or enlarged the picture would disappear due to the graphics being repainted. As a result, I was forced to switch tactics, with the image now being displayed as a java object after a waiting time of several seconds.

When the 'save' button is clicked, the program will convert the 2D array into an image and save it in the computer.

3.5 Film strips

Since one of the intended uses for the images produced by my program was to create film strips, I decided that it would be nice if there was an inbuilt function that could create the film strips for the user- even if they would be limited to a select number of options in terms of length. As for what these lengths would be, I selected 6 (the ratio increases by 20% with each image), 11 (10%) and 21 (5%). These would both be relatively straightforward to implement and would also likely be the most prominently created types of strips anyway (the ratio increase would have to be a divisor of 100 for the strip to depict a progression that is 'honest', and it is highly unlikely many people would be interested in a strip with more than 21 images- even that is bordering on excessive).

In the film strip function, when a user has selected how many images they want displayed (6, 11 or 21), the program will need to perform several calculations before any interpolation can begin. This process is to ensure that the images in the strip are shrunken down just enough so that they all fit on the screen without being squashed or stretched, but not more than that. First, the program will allocate a number roughly equal to the length of the screen. This exact number will vary between the three options, as the value needs to be a divisor of the value selected by the user (called X for now). This number is then divided by X, with the resultant value being the average length of each image in the strip.

$$X = \text{number of images in strip}$$

$$T = \text{length of strip}$$

$$l_{average} = \frac{T}{X}$$

The length and height of the output photo with a ratio of 0.5 are then measured. The height of this photo is divided by the length and multiplied by the average length calculated previously. The resultant value will be the height of all the images in the strip.

$$H_{average} = \text{average height of output images before being shrunk}$$

$$L_{average} = \text{average length of output images before being shrunk}$$

$$h = \frac{H_{average} * l_{average}}{L_{average}}$$

Now, the interpolation process is carried out for each of the images pretty much as before, only this time, the image is shrunken down to scale with the calculated height before being displayed on the panel in the appropriate positions.

$$\text{for } i = 1, 2, ..., X$$

$$l_i = \frac{L_i * h}{H_i}$$

3.6 User interface

Since most of the program's intended users will not be familiar with the face morphing process, it was important to me to make the user interface for the program as easy to use and accessible as possible.

While having larger images obviously makes the point selection easier, there had to be a limit to the size of the image displayed on the GUI. There were several reasons for this. The first was purely aesthetic- controlling the size of the images allowed me to control the overall layout when they have been entered. More importantly however, allowing images of limitless size to be inputted would mean they could potentially overlap with each other or even stretch outside the limits of the screen. Thus, I created two boarders on either side of the GUI and set my program to shrink down any images that do not fit inside of them.

One of the most important decisions regarding the user interface was to only allow the user to select one pair of points at a time. This serves two main functions. The first of these is ensuring that each coordinate in a pair is stored in the same location on each list (which, as already shown, is integral to the stability of the morphing process). Secondly, it gives the user a straightforward, structured procedure to go through. While there exist several programs that begin by dropping all the points at once in approximate positions for the user to drag to the correct one, given the sheer number of coordinates that are being dealt with here, this would simply not have been practical, with the user likely to become lost in the sea of coordinates.

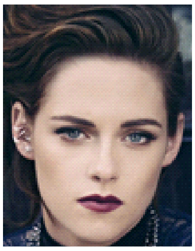
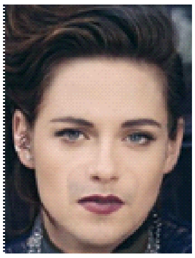
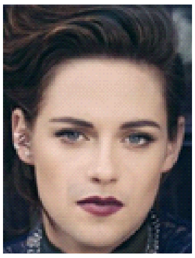
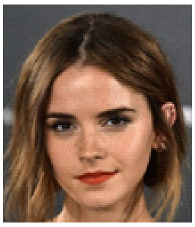
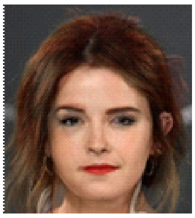
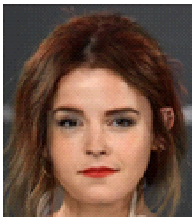
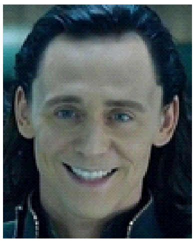
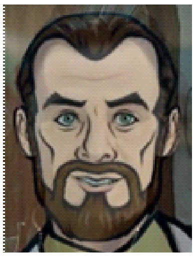
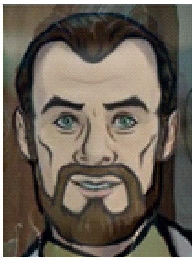
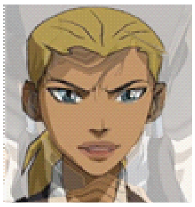
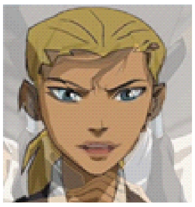
As one would expect, the locations of these coordinates are not obvious, and it was therefore essential that the user be given a guide as to where on each image to click during every step of the point selection process. While I initially only had a single instruction strip at the top of the GUI giving the user writing directions (select mid-left chin, etc.), these often proved to be too vague when tested on users. To compensate for this, I added an image above it of a face sketch, with the location to be selected highlighted for each stage.

Human error also needed to be taken into account when designing the GUI. More specifically, if the user clicks on the wrong location on an image, they should be able to fix that decision. This required inserting two capabilities. First, I designed the code so that when a user clicks on an image and then clicks it again, the point they previously selected will be nullified and replaced by the new one. Second, I inserted code allowing the user to navigate backwards to a previous stage in the selection process to change one or more of the coordinates from a previous pair.

Finally, I selected several different colours and had the colours of the coordinate marking dots cycle through them. This made it easier for the user to differentiate between the coordinates they have marked, something especially useful when a large number of points are clustered near each other.

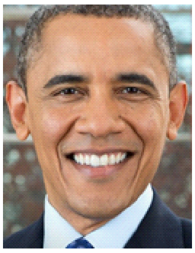
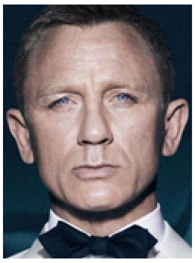
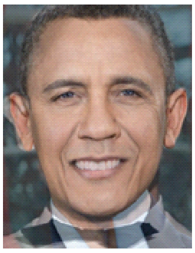
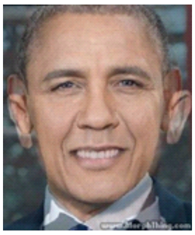
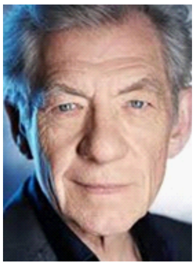
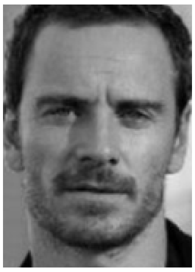
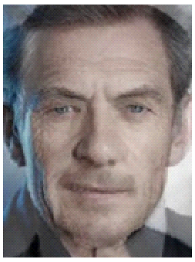
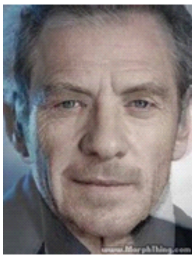
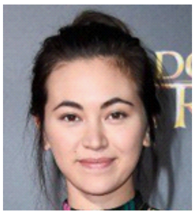
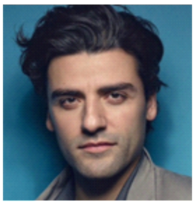
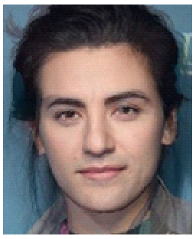
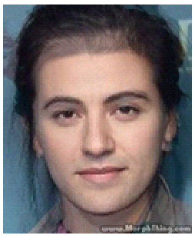
4 Results

The following chart demonstrates the results of several morphs, in addition to those carried out using the same process on Morpheus:

Input image 1	Input image 2	Result using my program	Result using Morpheus	Ratio (percentage of image 2 in output)
				**0.2**
				**0.4**
				**0.6**
				**0.6**

As can be seen, my program produces nearly identical results to that of Morpheus, with any variation likely stemming from natural fluctuations in the locations the control points were placed in. This confirms that the interpolation aspect of the program has been wholly successful.

This does not, however, mean the process itself is perfect, and that there do not exist programs using different algorithms and procedures that do a better job. Therefore, despite the drawbacks of automatic point detection, it was still worth investigating the difference between the results when applied to regular human faces. Here are several comparisons to output images using MorphThing (these would have been included in the previous table, were it not that MorphThing is limited to finding the exact halfway point, while oddly enough, Morpheus cannot do this):

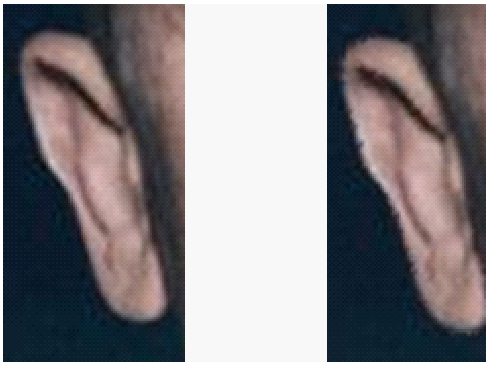
Input image 1	Input image 2	Result using my program	Result using MorphThing	Ratio (percentage of image 2 in output)
				**0.5**
				**0.5**
				**0.5**

What's most interesting here is the variety in the results. While the morph between Obama and Craig certainly looks better in my version (even discounting the disaster with the ears) , the same is arguably not true for the others. Looking at the bottom morph, while my version was able to align the ears more effectively and account for the different hair levels by tweaking the locations of the control points, the version created by MorphThing has done significantly better with respect to the eyes and eyebrows. Clearly, while inputting the control points manually has the potential to create better images when done correctly, human error means that this potential will not always be reached.

Also worth investigating is the resolution of the output images compared with that of the input images. When both input images are the same and marked with the same control coordinates, this is the result:

Left: Inputs, Right: Outputs.

While it may not initially appear that there is any difference at all, a closer examination does reveal a few areas where the resolution has decreased. Take for example the right ear:

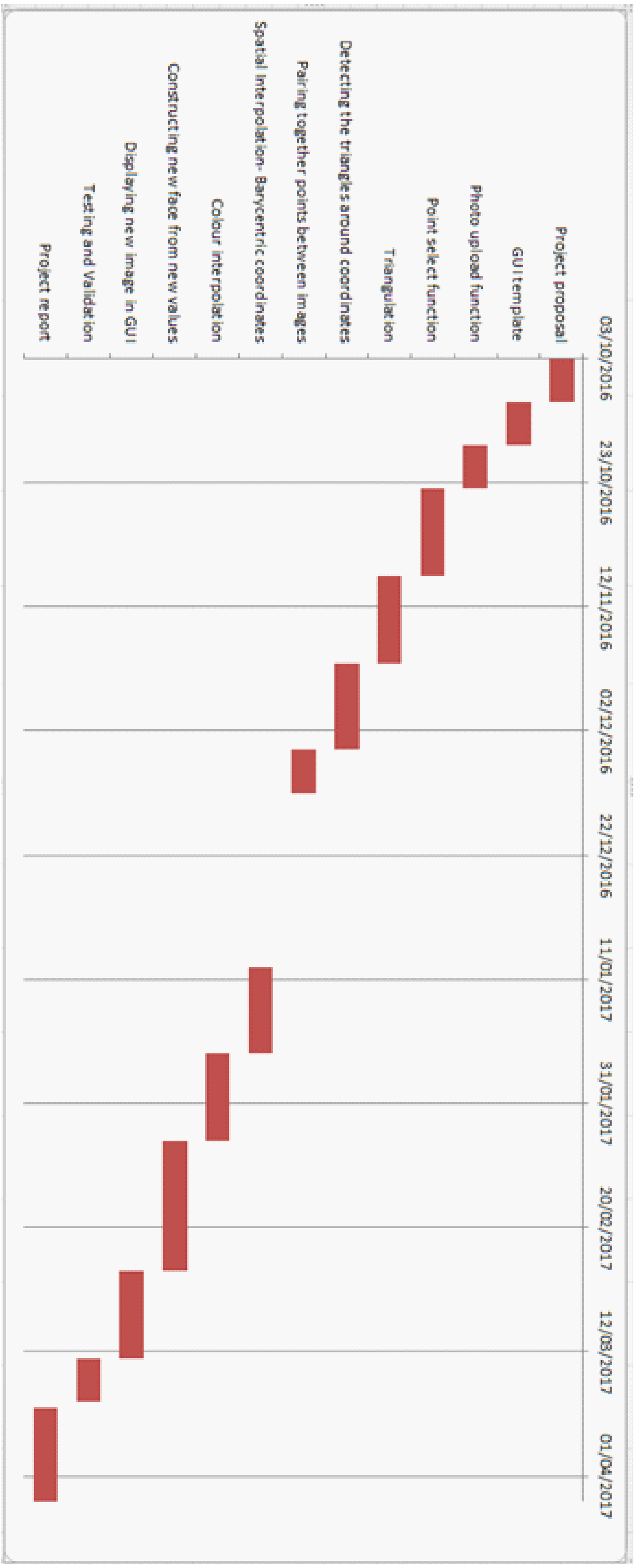


Left: Inputs, Right: Outputs.

These imperfections are so minor, however, that for almost all practical purposes they might as well not exist.

Beyond the quality of the output images, there is the general performance of the application. When it comes to producing a morphed image, the program will take between four and six seconds depending on the size of the image. For the film strip function, the time is longer, ranging from approximately forty seconds (6 images) to eighty seconds (21 images) when no other programs are running. The time for the single image is roughly in line with that taken by MorphThing, suggesting that there are no obvious alternative methods that will increase the speed of the process considerably.

5 Project Management



Above is the schedule I drew up for myself at the beginning of the year. While I certainly did not reach every milestone on time, it functioned as a very useful guide. While I initially intended not to do any work over the winter holiday, common sense ultimately got the better of me and I did the first several weeks' worth of work from semester 2 during this time. This gave me some much needed wiggle room during this semester, as the process of interpolation proved to be far more difficult and time consuming that I had anticipated. It also gave me some extra time which I used to create the filmstrip function- which had not been part of the original plan.

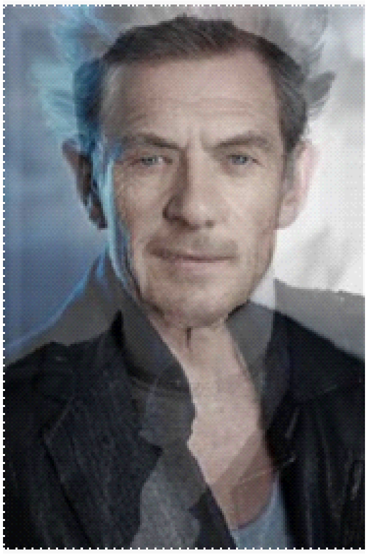
One thing that perhaps stands out as odd about this schedule is that the GUI was done first. The motivation behind this was simply that it would be the least enjoyable part of the project, and was therefore worth finishing first. In hindsight, this was probably a mistake, as I was somewhat limited in my design choices for the morphing process by what I had already implement into the GUI. While there was no specific moment where I recall having to consciously sacrifice quality for this, it is entirely possible that, given the opportunity to tackle various problems from different angles, I would have come up with something that would have been ultimately superior to what I had.

I made sure to keep my code well commented and neat while working on it. This was essential to keeping track of where I was at all times in the process and stopping it from becoming too overwhelming.

6 Discussion

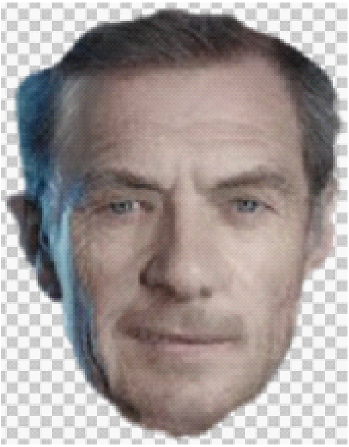
While I am pleased with my progress on this project so far, there is still much room for improvement – and given more time there are definitely a few features I would add and changes I would make.

The program's biggest flaw, in my opinion, is the way in which it blends the backgrounds of the images together. As explained before, this is certainly better than the alternative of having no background at all, and most people I have spoken to were not bothered about the area behind the head looking the way it does in the output photos. With the necks, shoulders, and hair blending together on the other hand, the results can often look downright bizarre. See, for example, the two upper bodies and unflattering head spikes attached to the face in this old photograph:



But how could this be fixed? One possibility might be to set all images to have a specific angle, upper and lower body level and hair style, and require the user to input the corresponding control points. This, however, would be highly limiting for the user. Alternatively, users could be given a list of options for the approximate lower body cut-off point and hairstyle to select from, and this would determine what coordinates they are told to select later on. In cases where these values for each image do not match, one would need to somehow re-triangulate the hair/upper body area so that organic interpolation could occur. That being said, however, this would likely lead to a very crude approximation in the output image at best. After all, how could one possibly list every possible hairstyle that might occur?

A more sophisticated method would be to use image extraction software (such as that available in Photoshop) to cut out the 'outline' of the face after the image has been created. If this is done correctly, one could then just eliminate the background altogether without having to deal with the jagged edges. But I think we could go one step further, and simply have the program trace out the shape of one of the upper bodies, to which the rest of the image could be attached. It wouldn't solve the hair problem, but it would be a significant step in the right direction.



Another flaw with the program is the lowered resolution of the output images. Again, this was unavoidable when it came to the interpolation itself, but antialiasing software and techniques do exist that could do a ‘pass’ on the image once it has been created.

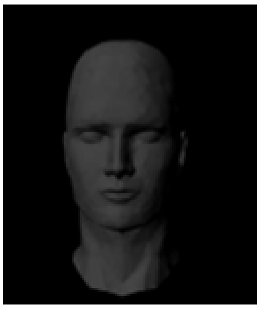
There would also absolutely be room for a function that would automatically select all the correct points. While I do maintain that there is value to letting the user select the points themselves, this is obviously not what everyone would want to do, especially those using the program for more casual purposes. The software to do this could either be written specifically for the program, or simply taken from an existing package.

One other feature present in a lot of similar software (including MorphThing and Moonjee) is the ability to morph more than two images. Since incorporating a slider into this function would not make sense and it would be impossible to create a filmstrip in these cases anyway, it would make most sense to simply add a function that interpolates more than two images to a set ratio (one third each for three images, one quarter each for four images, etc.). My code can be expanded to incorporate this fairly easily.

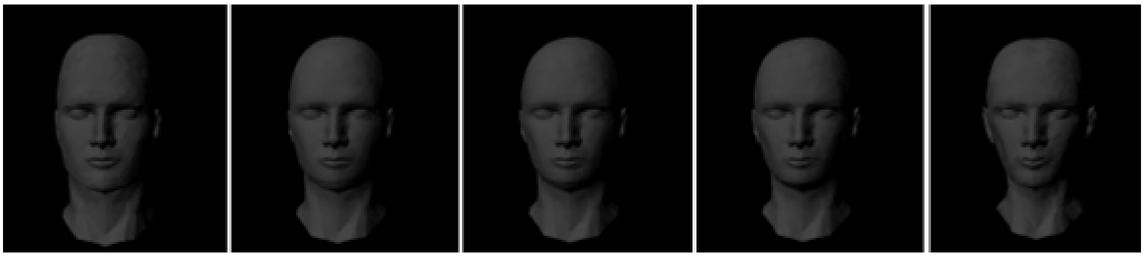
Finally, one feature that many similar programs have but this one lacks is an animation function. This would be relatively simple to implement – just have the program create a certain number of transition images in the same way it is done in the ‘film strip’ function, and have the program display one after the other on a loop to create a fluid transition effect.

These features all have nothing to do with the actual interpolation process itself, however, and are simply ways to play with the output that the process produces. But what about expanding the core mechanics of the project? Going beyond simple add-ons like these, one could take the basic structure of the code and expand into a whole other dimension – literally.

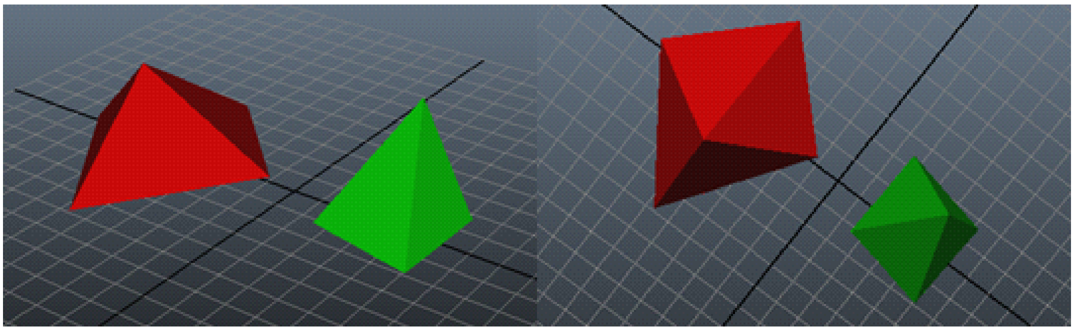
In their paper on 3D morphing, Urtasun, Salzmann and Fua presented the following two models:



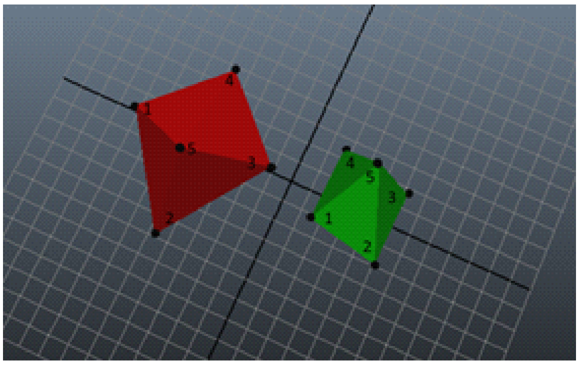
They showed that a smooth animation between them could be created simply by morphing the two together:



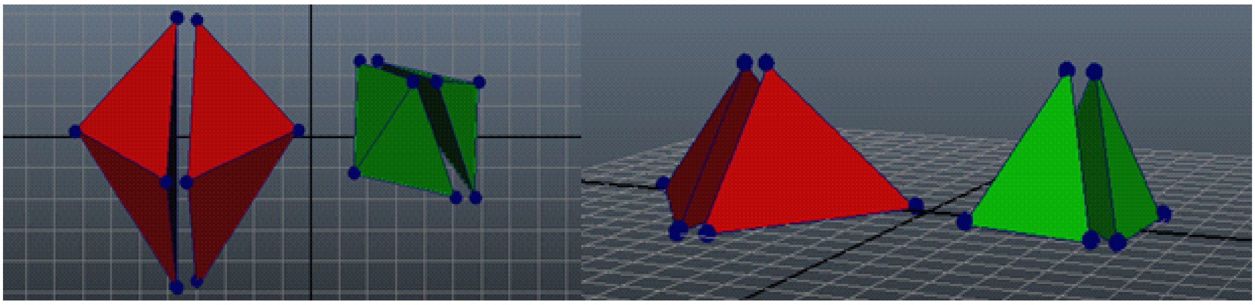
It is easy to imagine how something like this could be done using my template for 2D morphing (demonstrated here with pyramids for the sake of simplicity). One would start once again by uploading the two inputs:



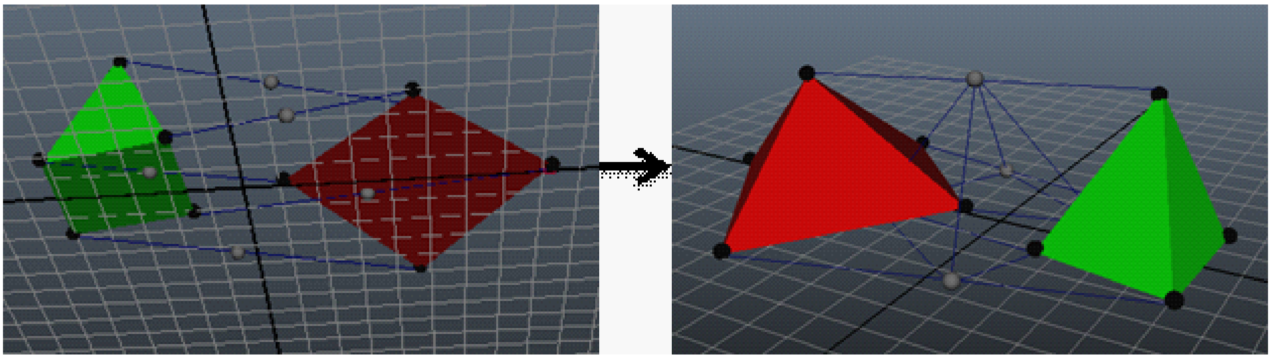
Mark each shape with control points:

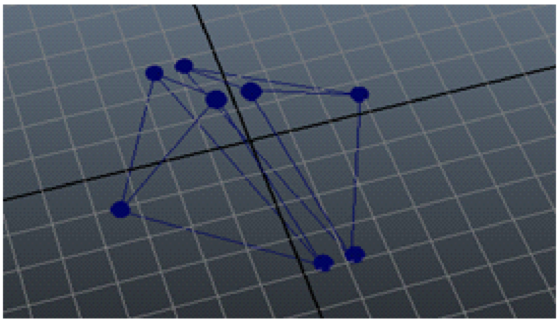


Create lattices – these will be pyramids with four sides:

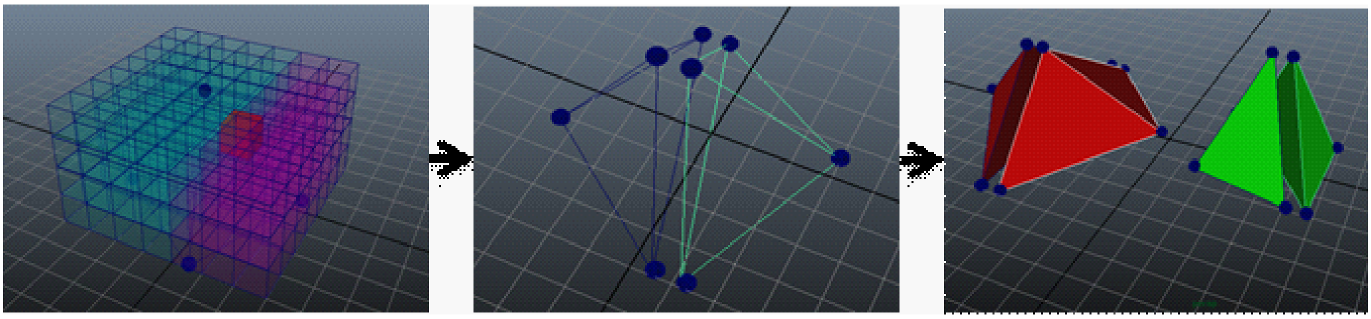


Find the midpoint for each coordinate (for the sake of simplicity the ratio will be 0.5) and use this to create a new lattice:



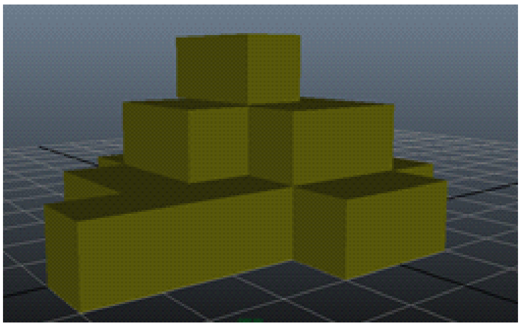


Now loop through each cube (replacing pixels) in the lattice, finding the sub-pyramid it is present in:

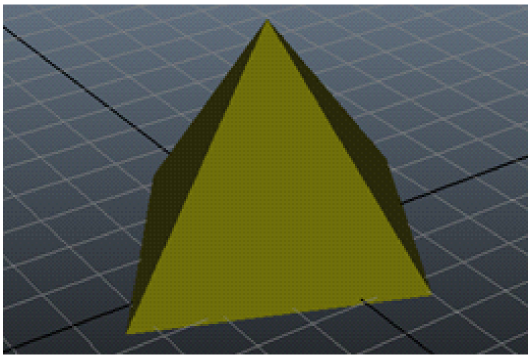


We then convert the coordinates of this cube from Cartesian coordinates to four sets of barycentric coordinates and use these to find the appropriate cube in the pyramids in the input lattices. Interpolate the colours of these two cubes to find the colour for the cube in the output lattice.

Going strictly by this example, the final output would look something like this:



Though of course in reality the size of the cubes would be much smaller in comparison to the object being morphed, and the result would be much closer to this:



7 Conclusion

Overall, I am very pleased with the end result of this project. I believe that I have created a program that produces facial morphs that are just as good, if not better, then programs available online. Through the high number of coordinates used and the slider function with 100 values, this program gives the user a higher level of control over their final image than any existing software I was able to find.

The program takes coordinates inputed by the user and interpolates triangulates them to form lattices over the input images. These lattices are then spatially interpolated by a certain ratio as determined by the user by using dummy values who's values are filled in by the value on the slider. The interpolation produces a third 'output' lattice. Colour interpolation is then carried out on all the pixels in this lattice to produce an image that acts as a visual midpoint between the two inputs.

8 Appendix: Instructions for running the program

To run this program, you will need only the packages 'delaney_triangulation' and 'main folder'. In 'main folder', find the file Morphing.java and run it. A GUI should pop up. Click on the two buttons above the boarders, which will open up selection boxes. When choosing your images, it is vital that you only upload jpeg files, as the program will not recognise any others. Once you have this, there will be a button telling you to begin. Press this, and you will be able to input points onto the two images, navigating forwards and backwards with these buttons. When you have finished, the button to morph the images will pop up. Click it, and wait a few seconds. As well as playing around with the slider (clicking the morph button again after moving it), you will then also have the option of clicking the filmstrip button which will open up a new window with its own set of options.

References

[1] Wikipedia Contributors. "Morphing". Wikipedia, The Free Encyclopaedia. Wikipedia, The Free Encyclopaedia, 28 Mar. 2017. Web. 28 Mar. 2017. https://en.wikipedia.org/wiki/Morphing

[2] Indiana Jones and the Last Crusade. Dir. Steven Spielberg. Perfs. Harrison Ford, Sean Connery, Alison Doody. 1989. DVD. Paramount Pictures. 2003.

[3] MorphThing. 2007. http://www.morphthing.com/

[4] Blend Faces- Make A Face Average. Pleasanton, California, USA, Moonjee Corporation. 2008. https://corporate.moonjee.com/action/average.php

[5] Morpheus Photo Morpher. V3.17. Morpheus Development, LLC, available at- http://www.morpheussoftware.net/

[6] Beier, T. and Neely, S. Feature-Based Image Metamorphosis. ACM SIGGRAPH Volume 26 Issue 2, July 1992, Pages 35-42. Available at- http://dl.acm.org/citation.cfm?id=134003

[7] Wolberg, G. Recent Advances in Image Morphing. Computer Graphics International, June 1996, Pages 24-28, Available at- http://ieeexplore.ieee.org/document/511788/

[8] Ruprecht, D., Muller. H. Image Warping with Scattered Data Interpolation, IEEE Computer Gaphics and Application Voulume 15, Issue 2, March 1995, Pages 37-43, Available at- http://ieeexplore.ieee.org/document/365004/

[9] Rohlfing, T., Maurer, C.R. Intensity-Based Non-rigid Registration Using Adoptive Multilevel Free-Form Deformation with an Incompressibility Constraint, Lecture Notes in Computer Science, Vol. 2208, October 2001, Available at- https://link.springer.com/chapter/10.1007/3-540-45468-3_14

[10] Tal, A., Elber, G. Image Morphing with Feature Preserving Texture, Computer Graphics Forum, Vol 18, Issue 3, 1999, Available at- http://onlinelibrary.wiley.com/doi/10.1111/1467-8659.00354/pdf

[11] Takayama, K., Johan, H., Nishita, T. Face Detection and Face Recognition of Cartoon Characters Using Feature Extraction, IIEEJ Image Electronics and Visual Computing Workshop, 2012, Available at- http://www.iieej.org/trans/IEVC/IEVC2012/PDF/4B-1.pdf