CS174 Project: Automatic Interpolation of 2D Images

Connor DeFanti

June 12, 2013

1 Executive Summary

In animation, artists typically handle the animation by creating a series of keyframes, or frames that capture important moments in the animation. The keyframes are then interpolated to create frames in between the two keyframes, which displays a smooth animation. In computer-generated interpolations, the artist usually does not control every point manually, but rather, creates a structure like a skeleton or deformation graph that tells the components of the keyframe to move. However, creating this deformation structure at each keyframe can be time consuming.

In this project, we implement the beginnings of an automatic interpolation system that takes in two sets of points, along with an initial graph structure, and attempts to interpolate the points from one set to another. Through this method, we do not have to manually move the deformation graph to fit the next frame. Instead, it will be automatically moved to a fitting position.

This project includes a simple renderer, which renders our multiple sets of points and deformation graphs, along with several features which allow modification of the points and nodes. Mouse interface allows the user to manipulate the nodes, which in turn modifies the point set. Several keystrokes allow the user to show and hide different parts of the frame, automatically generate a deformation graph for subsequent frames, and the rendering of the animation which deforms the graph from one frame to the next. The details of the implementation of these features will be discussed below.

2 Abstraction

2.1 The Data

As mentioned in the summary, this project seeks to do automatic animation. We start with an initial frame $a$, which has a set of points $p_a$ and a deformation cage $q_a$. The deformation cage is a special graph of nodes. The nodes in $q_a$ form a loop, so each node $q_a[i]$ has two neighbors. Each point has a set of weights, $w_i$, that is a vector that tells us how much influence each node has on the point. For example, $w_{ij}$ tells us how much of node $j$’s transformation affects point $i$. Furthermore, each frame has a set of correspondence points, $c_a \subseteq p_a$. These correspondence points help the algorithm best fit one frame to the next. For this project, the correspondence points are chosen manually at each frame.

We then have subsequent frames, frame $b$, $c$, etc., but without loss of generality, we will only discuss frames $a$ and $b$. We can ignore other frames, as our method for interpolation is strictly linear, and therefore depends only on two points in time. At frames beyond the first, we only start out with a point set $p_b$ and correspondence set $c_b$, but no deformation graph. The goal is to find the positions of the nodes of $q_b$.

2.2 The Renderer

We use a fairly simple renderer for displaying the output and accepting user input. Currently, no interface like Qt is used, but instead all interaction is done with keystrokes and mouse control. This is something that will change with future revisions of the project, but was not a priority in the time given.

The mouse controls are fairly simple. A user can click and drag nodes using the left click of the mouse.
Chapter 3 Mathematical Representation

3.1 Weighting Points

First, we will discuss how we assign the weight vector \( w_i \) to each point. Without loss of generality, let us assume the points are from frame \( a \). Let us examine a point \( p_a[i] \in p_a \). We know that the weight vector must be normalized, or \( \sum_j w_{ij} = 1 \), where \( Q \) is the number of nodes. There exist many different possible weighting methods that work in different scenarios. Here we will cover some of the ones we tried, along with their benefits and drawbacks.

- First, we looked at a simple algorithm. Given point \( i \), we choose the \( k \) closest nodes to \( i \) and assign weights based on an inverse distance squared proportionality. In other words, let \( q_k^a[i] \) be the \( k \) closest nodes to \( p_a[i] \). Then, the un-normalized weight \( w'_{ij} \) is given as:

  \[
  w'_{ij} = \begin{cases} 
  1 / \| p_a[i] - q_a[j] \|_2^2 & : q_a[j] \in q_k^a[i] \\
  0 & : q_a[j] \notin q_k^a[i]
  \end{cases}
  \]

  We then calculate \( w_{ij} \) as \( w_{ij}' / \| w_{ij}' \|_2 \).

  This algorithm was a good first pass, because it was simple to implement, and reasonably distorted the points to match the nodes.

- In our second pass, we created a weighting system that was similar to the first, but instead of looking at the \( k \) closest nodes, we looked at nodes in a certain radius \( r \). As before, we had an un-normalized weight \( w_{ij} \) that was given by:

  \[
  \max(0, 1 - \left( \frac{|p_a[i] - q_a[j]|_2}{r} \right)^2)
  \]

  So, as the distance from the point to the node increased, the weight decreased until zero. This was a better refinement at a simple weighting scheme, as it created a circle of influence, so points couldn’t be affected by nodes far away. We had to impose the constraint that \( r > \min_{i,j} \| p_i - q_j \|_2 \), so that every point was influenced by at least one node.

  The problem that we had with these two algorithms, however, was that we wanted an algorithm that enforced linear precision, or that \( p_a[i] = \sum_j w_{ij} q_a[j] \). This was not enforced by these previous two algorithms.

- The algorithm we wound up using was barycentric mapping. This imposed two constraints on our system. First, we must have a deformation cage as opposed to an arbitrary graph, so each node \( q_a[j] \) had two neighbors, \( q_a[j - 1] \) and \( q_a[j + 1] \). For \( j = 1 \), we replace the \( j - 1 \) neighbor with \( q_a[n] \) and for \( j = n \), we replace the \( j + 1 \) neighbor with \( q_a[1] \). Second, all points must reside on the interior of our deformation cage.

  Define \( A \) as the edge between \( p_a[i] \) and \( q_a[j] \), \( B \) as the edge between \( q_a[j] \) and \( q_a[j - 1] \), and \( C \) as the edge between \( q_a[j] \) and \( q_a[j + 1] \). Then, let \( \gamma_j \) be the angle between \( A \) and \( B \), and \( \delta_j \) the angle between \( A \) and \( C \). We then assign un-normalized weights as:

  \[
  w'_{ij} = \frac{\cot(\gamma_j) + \cot(\delta_j)}{||A||^2_2}
  \]

  We then normalize as described above. This has the distinct advantage of linear precision, but the drawback of having the noted constraints.
In Fig. 1, we see a summary of the differences between weighting algorithms. As will be discussed later, redder points weight the red selected node more. We can see that the first algorithm spread the weights out very evenly, the second algorithm localized the weights a bit more, and the final algorithm emphasized that localization.

In Fig 2., we see how these different weighting schemes affect the deformation of the points. The discrepancies are subtle, but they exist.

3.2 Rendering Points

Once we implemented a formula and algorithm to calculate each weight $w_{ij}$, we must use those weights to render the point in the right position. While we mentioned that for the last method, we enforced linear precision, or $p'_a[i] = \sum_j w_{ij} q'_a[j]$ where $p'$ and $q'$ are the updated positions of the points and nodes. However, for the first two, we could not use that method, so our formula for rendering was

$$p'_a[i] = p_a[i] + \sum_j w_{ij}(q'_a[j] - q_a[j])$$

We note that $p'_a[i] - p_a[i]$ is the same in both cases.

However, for barycentric mapping, we did update our algorithm to take into account the transformation of the node edges. We included a matrix $A_j$, which represents the transformation from the initial frame $a$ to the new frame $b$ of each node edge. To calculate $A_j$, we created a vector $u = [e_1.x, e_1.y, e_2.x, e_2.y]$, with
Figure 3: Deformation with and without the $A$ matrix, respectively.

e_1 = q_a[j - 1] - q_a[j]$ and $e_2 = q_a[j + 1] - q_a[j]$, and a matrix $U$ which was defined as:

$$U = \begin{bmatrix}
e_{1,x}' & e_{1,y}' & 0 & 0 \\
e_{2,x}' & e_{2,y}' & 0 & 0 \\
0 & 0 & e_{2,x}' & e_{2,y}'
\end{bmatrix}$$

with $e_1 = q_b'[j - 1] - q_b'[j]$ and $e_2' = q_b'[j + 1] - q_b'[j]$. We then solve the linear equation $Ua = u$ and set $A = \begin{bmatrix} a_1 & a_2 \\ a_3 & a_4 \end{bmatrix}$.

Then, with $A$ computed per node, we render points at the position

$$p_a'[i] = \sum_j w_{ij}(q_a'[j] + A(p_a[i] - q_a[j]))$$

Note that at the first frame, if the nodes have not been transformed, then $A = I$, and so $p_a'[i] = \sum_j w_{ij}(p_a[i]) = p_a[i]$ as expected. This change helped us place emphasis on the rotation and scaling of the deformation graph, instead of just the translation.

In Fig. 3, we can see the effect of using the matrix $A$ as opposed to not using it. For coloring the points, we use white as a baseline. However, to reflect the weight vectors, we also color the points with a redder hue if nodes that affect it are selected. In order to precisely define this color, we let the red channel be 1, and then let the green and blue channels be given by

$$1 - \sum_{\text{for each selected node } j} w_{ij}$$

For correspondence points, we arbitrarily chose different colors to always color the correspondence points with to make them always clearly visible. We can see an example of this in Fig. 4.

3.3 Generating the Subsequent Graphs

Now that we have discussed the framework of how we deform points, we can talk about how we deform the original graph structure from one frame to the next. This was the most critical part of the project. To state the method briefly, we represented each component of our data as a matrix or vector, and then solved a linear system. We will now discuss what each component was and the system we solved. Let $Q$ be the number of nodes, $P$ be the number of correspondence points, and $E$ be the number of edges in our graph. Then, we have the following definitions:
• $p^a$: a vector of size $2P$ where $p^a[2i] = c_a[i].x$ and $p^a[2i + 1] = c_a[i].y$

• $q^a$: a vector of size $2Q$ where $q^a[2i] = q_a[i].x$ and $q^a[2i + 1] = q_a[i].y$

• $D$: a matrix of size $2E \times 2Q$. $D_{2i,2j} = D_{2i+1,2j+1}$ if node $j$ is the second node of edge $i$, $-1$ if node $j$ is the first node of edge $i$, and 0 otherwise.

• $L^a$: a matrix of size $2E \times 2E$ which denotes the importance of each edge. Here we use the inverse of the edge length at frame $a$ as the importance of the edge.

• $W$: a matrix of size $2P \times 2Q$ which holds the weights. $W_{2i,2j} = w_{ij}$.

• $R$: a matrix of size $2E \times 2E$ that represents the rotation of each edge from frame $a$ to frame $b$. For a first iteration, this is calculated using the covariance, which will be described below.

• $v$: a vector of size $2P$ that tells us how much each point is transformed by the transformation of the nodes. We calculate $v^b_i = \sum_j w_{ij} A_j(c_a[i].x - q_a[j].x)$ and $v^b_{i+1} = \sum_j w_{ij} A_j(c_a[i].y - q_a[j].y)$

With these variables defined, we now define a fitting energy $F$ and a rotational energy $R$:

$$F[q^b] = \frac{1}{2} \sum_i ||p^b_i - v_i - (Wq^b)_i||^2$$

$$= \frac{1}{2}(p^b - v - Wq^b)^T(p^b - v - W^b)$$

$$R[q^b] = \frac{1}{2} \sum_{\text{edge } (i,j) \in E} L^a_{ij}||q^b_j - q^b_i - R_{ij}(q^a_j - q^a_i)||^2$$

$$= \frac{1}{2}(Dq^b - RDq^a)^T L^a(Dq^b - RDq^a)$$

We then define the total energy $E$ as $E[q^b] = \alpha R[q^b] + F[q^b]$ for some constant $\alpha$. We then try to minimize $E$. Since $E$ is quadratic with respect to $q^b$, we have one unique critical point of $E$, where $\nabla E = 0$. We can find where this critical point with respect to $q^b$ is by solving the following linear equation:

$$(\alpha D^T L^a D + W^T W)q^b = (\alpha D^T L^a D)q^a + W(p^b - v)$$

We solve for $q^b$ to get the updated nodes’ positions for frame $b$. 

Figure 4: Coloring and rendering the correspondence points
3.4 Calculating the Covariance

As mentioned above, since we initially have no nodes for frame \( b \), if we want to make a rotation, we need some estimate that will tell us how the edges will best rotate to fit the new point set. We do this by calculating the covariance of each point set. Let \( p_{\text{avg}} = \frac{1}{n} \sum_i p[i] \). Then, the covariance matrix is given as a 2x2 matrix of the following form:

\[
\text{Cov}(p) = \begin{bmatrix}
\sum_i^n (p[i].x - p_{\text{avg}}.x)^2 & \sum_i^n (p[i].x - p_{\text{avg}}.x)(p[i].y - p_{\text{avg}}.y) \\
\sum_i^n (p[i].x - p_{\text{avg}}.x)(p[i].y - p_{\text{avg}}.y) & \sum_i^n (p[i].y - p_{\text{avg}}.y)^2
\end{bmatrix}
\]

We note that this matrix is symmetric.

To find the rotation associated with this covariance, we simply must find the eigenvectors of \( \text{Cov}(p) \). If we let the eigenvectors of \( \text{Cov}(p_a) \) be \( u_1, u_2 \) and the eigenvectors of \( \text{Cov}(p_b) \) be \( v_1, v_2 \), then the associated rotations, \( R_a \) and \( R_b \) would be given by \( R_a = \begin{bmatrix} u_{11} & u_{12} \\ u_{21} & u_{22} \end{bmatrix} \), \( R_b = \begin{bmatrix} v_{11} & v_{12} \\ v_{21} & v_{22} \end{bmatrix} \).

Finally, the rotation from the points in frame \( a \) to the points in frame \( b \) would be given by \( R = R_b R_a^{-1} \).

4 Algorithms/Computational Implementation

This project could be heavily summarized by the mathematical descriptions given above. However, there are some nuances that are important to note. First we shall cover our simple data structures, then the algorithms we used to carry out the mathematical phenomena described above, and then a description of the renderer.

4.1 Data Structures

4.1.1 Points

One of the most fundamental data structures in this project was the \texttt{Point} class. This class contained position and node weight data. In order to get and set this data, simple accessors and mutators were created. Additionally, in order to calculate the weights based on the algorithms described above, the \texttt{Point} class has a function named \texttt{setWeights}. This function originally took in another function as a parameter, but this feature was removed for cross-platform functionality. This function calls a weight function, which after removing the function as a parameter, was added as a private member to the \texttt{Point} class.

In addition to the point class, we had a \texttt{Stroke} class which simply contained a vector of the points it contained, and a \texttt{Group} class which contained a vector of strokes it contained. Both of these had accessors and mutators to modify the data. These classes did little else than provide a hierarchy for the system.

4.1.2 Nodes

Similar to the \texttt{Point} class, the \texttt{Node} class is very simple. It is just a container for some data along with accessors and mutators. A Node knows its position as well as its initial position. This distinction allows for deformation of the points when moving the nodes around. A Node also knows its neighbors. In our special case of a deformation cage, this list of neighbors has precisely two elements. Finally, a Node stores its weight matrix, which we described in the literature above in section 3.2 as \( A_j \).

4.1.3 Frame

The last data structure for this project is the \texttt{Frame} class. This class contains all of the data for one keyframe: a list of groups, a list of nodes, and a list of correspondence points. As usual, this class has accessors and mutators to modify and view the data.
4.2 Algorithms

Most of the information behind the algorithms are covered above in section 3. Here we will discuss some of the nuances of the code, how the algorithms flowed and worked together, and discussion for the choices made in some techniques for running the algorithms.

4.2.1 Updating the Nodes

In order to generate nodes for frame \( b \) from frame \( a \), we must be careful in what order we calculate and set our data. First, we had to set the weights of the correspondence points to be the same throughout all frames. Then, once these weights are set, we create the nodes in frame \( b \) using the method detailed in section 3.3. Finally, we set the weights of all the points in frame \( b \) according to one of our weighting algorithms detailed in section 3.1. We can outline this procedure with the following pseudocode:

```plaintext
for each frame in keyframes beyond the first:
    for each correspondence in frame:
        correspondence.weights = initial_correspondence.weights
        generateNodes(frame)
    for each point in frame:
        point->setWeights(frame->nodes)

Then, in the generateNodes function, we take the following procedure. First, we calculate the covariance of the two input frames to get the rotation. Then, we set up the matrices and vectors described in section 3.3. Then, we use a linear solver to solve for \( q_b \). We will next talk about our choice of linear solvers.

4.2.2 Linear Solvers

There exist many libraries available for linear algebra, including SuiteSparse, Lapack, Eigen, and BLAS, just to name a few. For solving for eigenvectors when calculating covariance, we use the Eigen library. However, for the sake of familiarity and simplicity, for solving linear systems, we used a SparseMatrix library developed by Fernando de Goes, a contributor to this project. This library borrows from SuiteSparse to solve linear systems via QR decomposition. QR decomposition decomposes a matrix \( A \) such that \( A = QR \), where \( Q \) is an orthogonal matrix and \( R \) is an upper triangular matrix. This decomposition makes it easy to invert when \( A \) is nonsingular, and easy to solve when \( A \) is singular. This solver was applied when computing the node transformation matrix \( A \) and the updated nodes, \( q_b \), both described in previous sections.

The choice of using the SparseMatrix library also provided the benefit of saving space, as the matrices could be potentially very large, depending on the number of nodes and correspondence points. Many of our matrices were in fact sparse, so using dense matrices would have wasted computational space and time.

4.2.3 Interpolating Between Frames

We chose to do linear interpolation between frames. However, we did not linearly interpolate the positions of the nodes. Instead, we linearly interpolated the rotations and scalings between edges of nodes, and rebuilt the graph from these nodes. The only node we did linearly interpolate the position was the first indexed node, but only to get the position of the whole graph correct. If we let \( t \in [0, 1] \) be the time step between two frames, then we can summarize this procedure with the following pseudocode:

```plaintext
new_nodes = t*nodes_b[0] + (1-t)*nodes_a[0]
for i < frame[a]->nodes.size() - 1:
    e0 = nodes_a[i+1] - nodes_a[i]
    e1 = nodes_b[i+1] - nodes_b[i]
    s = length(e1) / length(e0)
    r = acos(dot(e0, e1) / (length(e0) * length(e1)))
    Matrix S = [[s, 0], [0, s]]
    Matrix R = [[cos(r), -sin(r)], [sin(r), cos(r)]]
    new_nodes[i+1] = t * R * S * e0 + (1 - t) * e0
```
for i < frame[a]->nodes.size():
    nodes_a[i] = new_nodes[i]

4.2.4 The Renderer

For our renderer, we used OpenGL, along with several utility libraries, including GLU, GLUT, and GLM. These libraries provided simple tools for interacting with the renderer. Since this project was in 2D, we used orthonormal projection via GLU’s `glOrtho` function. All lighting and other shading was disabled. To render points, we simply used `glBegin(GL_POINTS)`. Additionally, correspondence points could be further emphasized by adding two crossed lines intersecting at the point. This was rendered using `glBegin(GL_LINE_STRIP)`. Nodes were rendered as boxes via `GL_POLYGON`, and then defining the four corners to be at some constant plus or minus the node position. Lines were drawn between nodes to denote neighboring nodes. Aside from the coloring mentioned in section 3.4, colors were mostly arbitrary. Points were usually white, nodes and connections between nodes were gray, and selected nodes were red. The last rendering feature we implemented was the ability to render the points from frame a in frame b using the position of the nodes in frame b.

For interaction, we relied mostly on the functions of GLUT. As we mentioned in the abstraction, clicking and dragging nodes allowed for manipulation of the nodes, and clicking on points toggled them as correspondence points for that frame. In order to decide what was selected, we first prioritize nodes. We find the closest node to the point where we click, and if the click is within that node’s bounding box, we select it. If no nodes have been selected using this process, we find the closest point, prioritizing correspondence points, in a fixed radius. This point will be toggled as a correspondence point. Finding the closest object was done by naively searching through all points and nodes.

5 Results

In its current state, most tests proved to be ineffective. Our algorithms can handle very simple transformations, such as transforming a circle to a rotated ellipse, but other cases that cannot be summarized as a set of global rotations and scalings are not handled well. Currently for many cases, some manual tweaking is required, although the algorithm oftentimes provides a first good pass.

In figure 5, we can see a clean example that worked. Note that the white outline in frame b is occluded by the interpolated frame a points. This means our algorithm matched perfectly.

In figure 6, we can see the interpolation from frame a to frame b. This interpolation is handled by the algorithm described in section 4.2.3. Next, we tried a more realistic example of what we might expect.
as input. Figure 7 shows the same circle to ellipse transformation, but with some noise added to the result. This would resemble a fatter stroke.

![Figure 7: Example case: a more realistic example](image)

While we can notice that we lose some of the perfection, it is still very close. The quality of this changed per render due to the randomness of the noise, but the quality remained relatively similar.

Finally, we tried a more complex example. In this example, we tried to deform a circle shape to an egg shape. This is a transformation that could not be expressed only by global rotations and scalings. We can see in Figure 8 that the result is not very close, and required a bit of manual tweaking to get it right.

5.1 Choosing $\alpha$

As mentioned before in section 3.3, we have a free parameter $\alpha$ which tells us the proportion of $\mathcal{R}$ to $\mathcal{F}$ we use. We found that $\alpha = \epsilon$ for some small $\epsilon$ worked best. When $\alpha = 0$, the solution would try to pass through as many correspondence points as it could, but the shape would hold little resemblance to the desired shape. When $\alpha$ was some small value, we got the results seen above. When $\alpha$ was noticeable, i.e. greater than 0.1, we saw that the rotations were captured very well, however the scaling was not. We needed the fitting energy to capture scalings and translations. The rotation energy alone could only capture the rotation of edges. In figure 9, we see these results.
Figure 8: Example case: a more complicated deformation. On the left we see our algorithm's guess, on the right, manual tweaking shows a solution exists, but our algorithm could not recover this solution.

Figure 9: Results using $\alpha = \{0, 0.001, 1.0\}$, respectively.

6 Conclusions

As we can see from the results, there is much refining to do with the algorithm. We so far have a good baseline, but looking into different weighting schemes and modifications our node generation algorithm will be a topic for future investigation. We believe that with further work on the algorithm, we can make it much more robust and able to handle many general cases, and minimize the need for manual tweaking to a large degree. What we have seen so far is a system that for simple cases, does precisely what we want: Given only a few manual correspondences and an initial set of nodes, we can create a new set of nodes that allows interpolation from one frame to the next.

Furthermore, more work could be done with the interface. Right now, it is very simple. Attaching an interface like Qt would help its usability. Additionally, a parser for input data would help us obtain more realistic results. Early on in the project, we were looking into an XML reader, but decided to forgo this in order to focus on the main portion of the project. We needed to solve our simple cases first before thinking about more complex, real cases.

Overall, I believe this project was a success. While we did not solve our problem completely, we will continue to improve the work done here in order to hopefully create a robust engine that can solve this interpolation problem for animation.

7 Appendix

Below are a list of links for further reading.

- More papers covering shape-matching algorithms: Temporally Coherent Completion of Dynamic Shapes.
Li, Lua, Vlasic, Peers, Popovic, Pauly, Rusinkiewicz.
Global Correspondence Optimizations for Non-Rigid Registration of Depth Scans. Li, Summer, Pauly.