A Physically Based Approach to 2–D Shape Blending

Thomas W. Sederberg
Eugene Greenwood
Brigham Young University[1]

Abstract

This paper presents a new algorithm for smoothly blending between two 2–D polygonal shapes. The algorithm is based on a physical model wherein one of the shapes is considered to be constructed of wire, and a solution is found whereby the first shape can be bent and/or stretched into the second shape with a minimum amount of work. The resulting solution tends to associate regions on the two shapes which look alike. If the two polygons have m and n vertices respectively, the algorithm is $O(mn)$. The algorithm avoids local shape inversions in which intermediate polygons self-intersect, if such a solution exists.

Categories and Subject Descriptors: I.3.3 [**Computer Graphics**]: Picture/Image Generation; I.3.5 [**Computer Graphics**]: Computational Geometry and Object Modeling.

General Terms: Algorithms

Additional Key Words and Phrases: Computer graphics, shape blending, animation, physically based algorithms.

1   Introduction
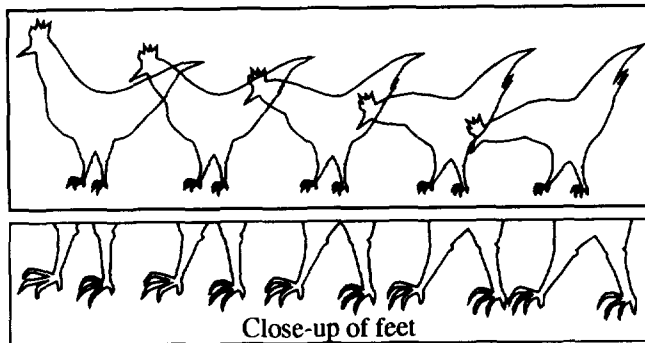
The topic of this paper is illustrated in Figures 1–3. Given

Figure 1: Shape blend example

Figure 2: Shape blend example

two polygonal shapes, the problem is to compute a continuous shape transformation from one to the other. For example, in Figure 1, the far left and far right sketches of a chicken are given, and the three intermediate shapes are automatically computed with no user interaction. This operation is known variously as shape averaging, shape interpolation, metamorphosis, shape evolving, and shape blending. It has widespread application in illustration, animation, and industrial design. 2–D shape blending is an increasingly popular feature in many commercial illustration software packages (such as [1], [6], [7], [17], [18]).
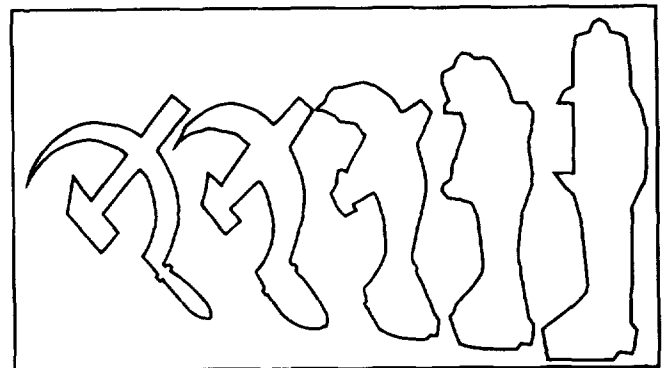

Figure 3: Shape blend example

Solutions to the 3–D shape interpolation problem have also been proposed ([4], [10], [14]). Indeed, the research effort reported in this paper initially focused on the 3–D problem. However, the authors soon realized that even the 2–D problem had many open questions, such as how can a shape blend algorithm avoid chaotic intermediate shapes and how can an

[1] Engineering Computer Graphics Laboratory
368 Clyde Building
Brigham Young University
Provo, UT 84602
(801)378-6330
(801)378-2478 FAX
tom@tws.ce.byu.edu

algorithm recognize similar, though not identical, features on the two terminal shapes (such as the feet and head of the chicken in Figure 1) and maintain those features throughout the blend.

We tested several commercial shape blending software packages on some of our shape examples. The best of any results for the chicken outline is shown in Figure 4 and the best E to F blend is shown in Figure 5. Notice how the chicken feet in Figure 4 degenerate to a self-intersecting scribble.
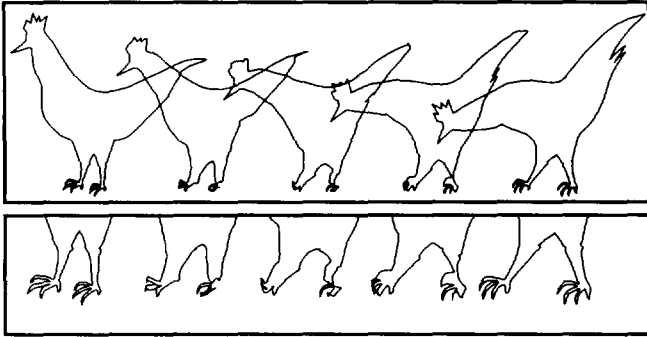

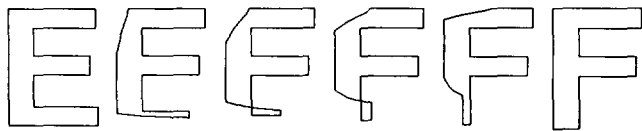Figure 4: Shape blend of chicken using commercial software

Figure 5: Shape blend of E to F using commercial software

The algorithm presented in this paper is based on a physical model. Imagining that each shape is made of a piece of wire, the blend is determined by computing the minimum work required to bend and stretch one wire shape into the other. The user can specify some physical properties of the wire, which control the relative difficulty with which it can be bent or stretched. A severe penalty is charged for blends which experience a local self intersection due to the wire bending through an angle of zero degrees. This penalty nearly always prevents the self intersection problem in Figure 4. All of the blends in this paper were generated automatically with no user intervention (Figures 4 and 5 by commercial packages, the rest by our algorithm) except for initially specifying the physical attributes of the wire.

1.1 Related work

Shape blending is a problem which has been motivated by several different applications and attacked in several different ways. For example, if we envision the family of blend polygons as forming a ruled surface in (x, y, t) space, as shown in Figure 6, the shape blending problem bears strong similarity to the contour triangulation problem [5], [8], [9], [15]. This is the background from which we approached the problem, and our solution borrows graph theory concepts from [15] and [8]. The algorithms used in the commercial illustration software cited above probably resemble the triangulation algorithms in [5] and [9] since these are $O(n)$ in time and memory, thus more suitable for PC applications than ones based on graph theory.

The first paper on 3-D shape interpolation [4] was motivated by industrial design. It tackles the problem by slicing the two 3-D shapes into contours, blending corresponding contours, then reconstructing the 3-D blended surface. More recent solutions, [10] and [14], are based on Minkowski sums. These approaches give impressive results, but leave some room for further investigation. For example, two non-convex objects
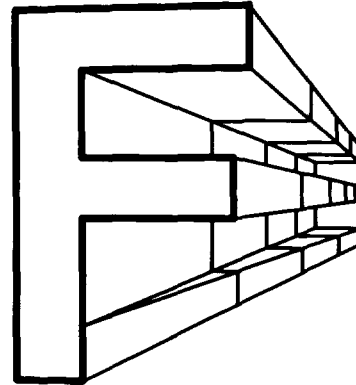

Figure 6: Family of blend polygons as a function of t

with similar features (such as a dog and a horse) will lose protruding details such as legs during intermediate shapes. In fact, the blend of a non-convex object with itself is not a constant shape.

Problems related to 2-D shape blending arise in shape recognition [2], [19] and curve matching for graphical search and replace [16]. In these applications, the primary concern is determining how similar two complete objects are. Shape blending also resembles the computer vision problem of contour identification, for which one solution is based on energy minimization [13], as is the shape blending algorithm described herein.

When the two shapes to be blended are taken to be keyframes in a character animation (such as in Figure 1) shape blending is similar to inbetweening — an important component of the general problem of computer-assisted animation [3]. The problem addressed in this paper, inbetweening of polygonal shape outlines, is simpler than the more general problem of inbetweening complete drawings.

1.2 Overview

Section 2 discusses geometric aspects of the shape blending problem. The physical work model is discussed in section 3. The minimum work solution is found by means of a directed graph, as discussed in section 4. Section 5 presents several examples and discusses the relative influence of stretching and bending work.

2 Geometric preliminaries

Given two polygons $\mathbf{P}_0$ and $\mathbf{P}_1$ with the same number of vertices, shape blending is accomplished by performing a linear interpolation between the corresponding vertices of the two polygons. If

$$\mathbf{P}^0 = [\mathbf{P}_0^0, \mathbf{P}_1^0, \ldots, \mathbf{P}_n^0]; \quad \mathbf{P}^1 = [\mathbf{P}_0^1, \mathbf{P}_1^1, \ldots, \mathbf{P}_n^1] \quad (1)$$

where $\mathbf{P}_i^k$ denote vertices, intermediate polygons in the blend can be defined

$$
\begin{aligned}
\mathbf{P}(t) &= u\mathbf{P}^0 + t\mathbf{P}^1 \\
&= [u\mathbf{P}_0^0 + t\mathbf{P}_0^1, u\mathbf{P}_1^0 + t\mathbf{P}_1^1, \ldots, u\mathbf{P}_n^0 + t\mathbf{P}_n^1] \\
&= [\mathbf{P}_0(t), \mathbf{P}_1(t), \ldots, \mathbf{P}_n(t)]
\end{aligned}
\quad (2)
$$

where $u = 1 - t$. The motion of three adjacent vertices undergoing a shape blend is shown in Figure 7.

Consider the simple example in Figure 8, where the vertex numbers are labeled. Each intermediate shape is determined by linearly interpolating each node as shown. The paths for vertices 1 and 3 are shown in dotted lines.
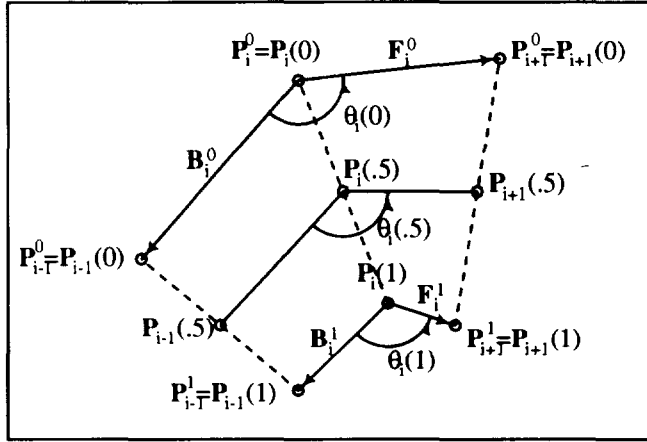
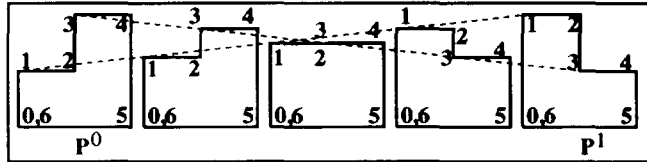Figure 7: Blending of three adjacent vertices

Figure 8: Simple example, solution 1

If the vertices are renumbered, a shape blend such as in Figure 9 can be obtained.
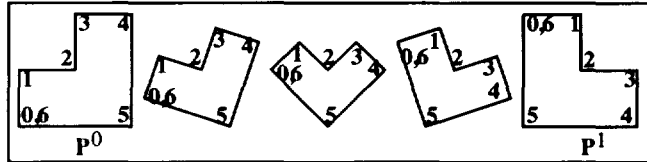

Figure 9: Simple example, solution 2

Different blends can be achieved if we insert new vertices in each polygon. In Figure 10, a vertex labeled "5" is inserted in $\mathbf{P}^0$ and a vertex labeled "1" is inserted in $\mathbf{P}^1$ as shown. Another variation is obtained by adding two new vertices at the same point. In Figure 11, vertices labeled "5" and "6" are inserted in $\mathbf{P}_0$ and vertices labeled "1" and "2" are inserted in $\mathbf{P}_1$ as shown.

Typically, two polygons to be blended do not initially have the same number of vertices, and even if they do, the correspondence will not generally produce a pleasing blend. The examples in Figures 8–11 suggest that the principle task in shape blending is that of adding vertices to each polygon such that each polygon ends up with the same number of vertices, and the resulting vertex correspondences produce the desired blend.

So, how can an algorithm automatically decide, with little or no human intervention, where to add the vertices? Section 2.1 shows two geometric conditions that an algorithm can identify and try to avoid, and section 3 discusses a physical model which can further guide an algorithm.

2.1 Angles

Consider the solution to the step problem shown in Figure 12. This blend serves little useful purpose (except to illustrate shape blending gone awry), because the angle at the circled vertex goes to zero, so that the two edges meeting at that
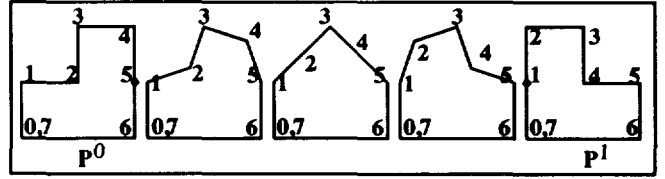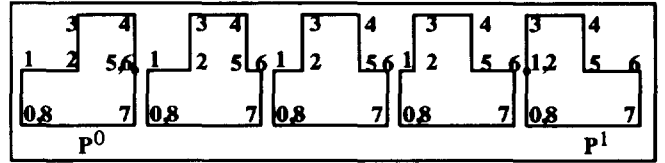

Figure 10: Simple example, solution 3

Figure 11: Simple example, solution 4

vertex pass over one another. When this happens, at least part of the shape is turning itself "inside out".

Another example of ill-behaved intermediate angles is shown in Figure 13. Here, the terminal angles at vertices 4 and 6 are both 90°, yet those angles in the intermediate blends exceed 130°. Also, vertex 5 begins and ends on a straight line, yet that line becomes noticeably bent during the blend operation.

Figures 12 and 13 suggest two angle constraints that should be imposed on blend solutions. First, if at all possible we should avoid

$$\theta_i(t) = 0 \qquad 0 \le t \le 1 \tag{3}$$

at each vertex, since that implies that an intermediate shape is self-intersecting. Second, it seems preferable, when possible, for each intermediate angle to be bounded by its terminal angles. That is, $\theta_i(t)$ should change monotonically from $\theta_i(0)$ to $\theta_i(1)$.

It happens that there is an unexpectedly simple representation for the angle $\theta_i(t)$ which greatly aids the understanding and analysis of these two conditions. In the following,

$$\mathbf{P}_i \times \mathbf{P}_j \equiv (x_i, y_i) \times (x_j, y_j) \equiv x_i y_j - x_j y_i,$$

$$\mathbf{P}_i \cdot \mathbf{P}_j \equiv (x_i, y_i) \cdot (x_j, y_j) \equiv x_i x_j + y_i y_j,$$

and

$$\|\mathbf{P}_i\| = \sqrt{x_i^2 + y_i^2}.$$

Letting $u = 1 - t$, the angle $\theta(t)$ can be computed

$$\theta_i(t) = \angle[(\mathbf{P}_{i-1}^0 u + \mathbf{P}_{i-1}^1 t), (\mathbf{P}_i^0 u + \mathbf{P}_i^1 t), (\mathbf{P}_{i+1}^0 u + \mathbf{P}_{i+1}^1 t)]$$

$$= \angle[(\mathbf{B}_i^0 u + \mathbf{B}_i^1 t), 0, (\mathbf{F}_i^0 u + \mathbf{F}_i^1 t)] \tag{4}$$

where $\mathbf{B}_i^k = \mathbf{P}_{i-1}^k - \mathbf{P}_i^k$ and $\mathbf{F}_i^k = \mathbf{P}_{i+1}^k - \mathbf{P}_i^k$ as shown in Figure 7. Recalling that

$$\sin(\angle \mathbf{P}_1, 0, \mathbf{P}_2) = \frac{\mathbf{P}_1 \times \mathbf{P}_2}{\|\mathbf{P}_1\| \|\mathbf{P}_2\|};$$
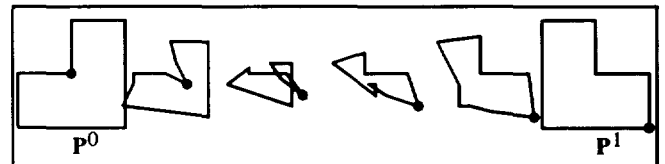

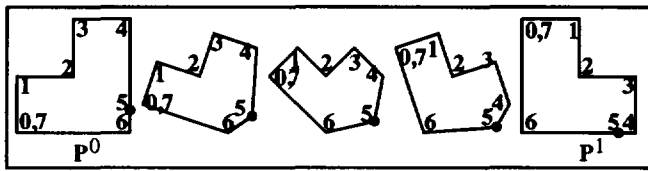Figure 12: Simple example, "solution" 5

Figure 13: Simple example, "solution" 6

$$\cos(\angle P_1, 0, P_2) = \frac{P_1 \cdot P_2}{\|P_1\| \|P_2\|};$$

$$\tan(\angle P_1, 0, P_2) = \frac{P_1 \times P_2}{P_1 \cdot P_2}, \tag{5}$$

$$\tan(\theta_i(t)) = \frac{(F_i^0(1-t) + F_i^1 t) \times (B_i^0(1-t) + B_i^1 t)}{(F_i^0(1-t) + F_i^1 t) \cdot (B_i^0(1-t) + B_i^1 t)}$$

$$= \frac{y_0(1-t)^2 + y_1 2t(1-t) + y_2 t^2}{x_0(1-t)^2 + x_1 2t(1-t) + x_2 t^2} \tag{6}$$

where

$$x_0 = F_i^0 \cdot B_i^0; \quad x_1 = \frac{F_i^1 \cdot B_i^0 + F_i^0 \cdot B_i^1}{2}; \quad x_2 = F_i^1 \cdot B_i^1; \tag{7}$$

$$y_0 = F_i^0 \times B_i^0; \quad y_1 = \frac{F_i^1 \times B_i^0 + F_i^0 \times B_i^1}{2}; \quad y_2 = F_i^1 \times B_i^1. \tag{8}$$

Equation 6 can be interpreted as a degree two Bézier curve

$$Q(t) = (x_0, y_0)(1-t)^2 + (x_1, y_1)2t(1-t) + (x_2, y_2)t^2$$

$$= Q_0(1-t)^2 + Q_1 2t(1-t) + Q_2 t^2. \tag{9}$$

As illustrated in Figure 14, $Q(t)$ has the important property that $\theta_i(t) = \angle((1,0),(0,0),Q(t))$. Thus, $\theta_i(t) = 0$ only if $Q(t)$
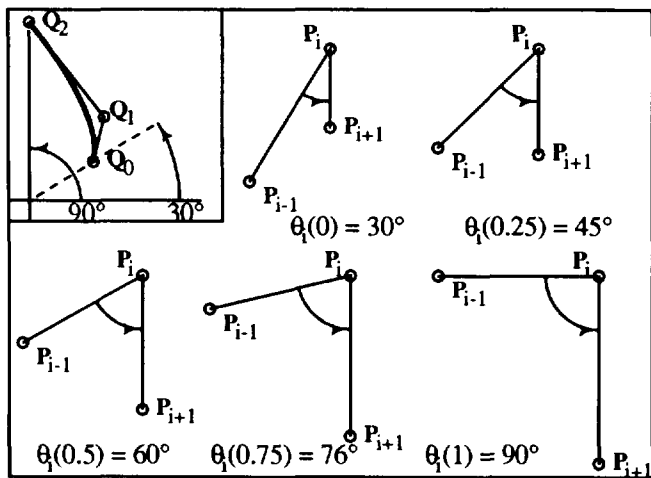

Figure 14: Relationship between $\theta_i(t)$ and $Q(t)$

intersects the positive x axis (as shown in Figure 15).

Angle monotonicity is assured if no line through the origin intersects $Q(t)$ more than once (as shown in Figure 16). The angle function $\theta_i(t) = \angle[(1,0),(0,0),Q(t)]$ has four possible extrema: $\theta_i(0)$, $\theta_i(1)$, $\theta_i(t_1)$, or $\theta_i(t_2)$ where t_1 and t_2 satisfy the equation

$$(Q(t) - 0) \times Q'(t) = 0.$$

This produces a cubic polynomial which always degree reduces to a quadratic polynomial in Bernstein form:

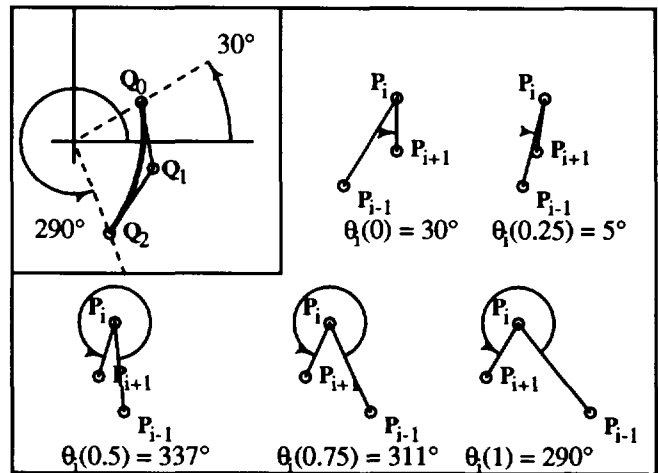$$d(t) = d_0(1-t)^2 + d_1 2t(1-t) + d_2 t^2 = 0; \tag{10}$$

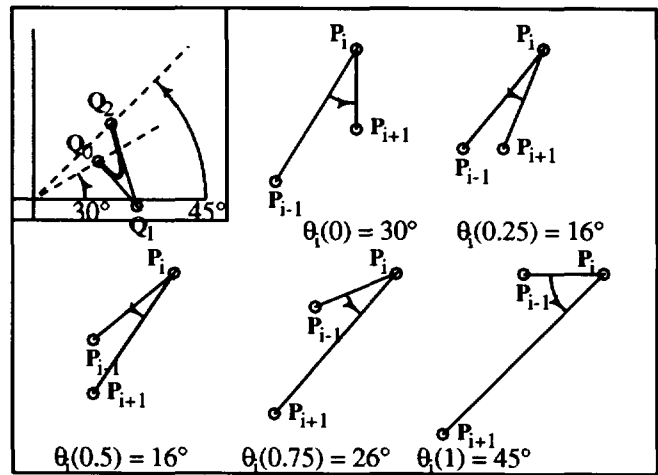Figure 15: $\theta_i(t)$ goes to zero

Figure 16: $\theta_i(t)$ is not monotonic

where

$$d_0 = Q_0 \times Q_1;$$

$$d_1 = \frac{Q_0 \times Q_2}{2};$$

$$d_2 = Q_1 \times Q_2.$$

$\theta_i(t)$ changes monotonically if and only if equation 10 has no real roots in the unit interval.

The work model in section 3.2 requires us to compute the angle change $\Delta\theta_i$. If triangle $\triangle Q_0 Q_1 Q_2$ does not contain the origin, then $\Delta\theta_i = |\theta_i(1) - \theta_i(0)| \mod 180°$. If triangle $\triangle Q_0 Q_1 Q_2$ does contain the origin, it's possible for $\Delta\theta_i$ to exceed $180°$ as illustrated in Figure 17. Necessary and sufficient conditions for $\Delta\theta_i$ to exceed $180°$ is for $\triangle Q_0 Q_1 Q_2$ to contain the origin, and $d_1^2 - d_0 d_2 < 0$ (the discriminant of the quadratic formula in equation 10). Thus,

$$\Delta\theta_i = \begin{cases} 360° - |\angle(Q_0,(0,0),Q_2)| & \text{if } d_1^2 - d_0 d_2 < 0 \text{ and} \\ & \triangle Q_0 Q_1 Q_2 \supset (0,0) \\ |\angle(Q_0,(0,0),Q_2)| & \text{otherwise} \end{cases} \tag{11}$$

where $\angle(Q_0,(0,0),Q_2) \le 180°$.

If $\theta_i(t)$ is not monotonic, the development in section 3.2, needs to know how far $\theta_i(t)$ deviates from monotonicity. This deviation is a non-negative angle denoted by $\Delta\theta_i^*$ as shown
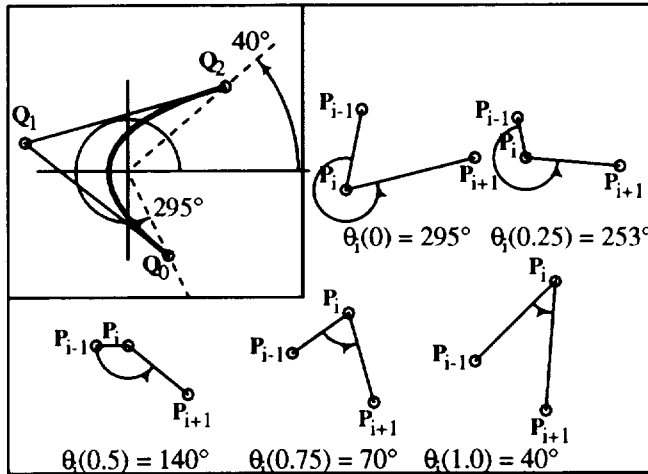
$\theta_i(0) = 295°$ $\theta_i(0.25) = 253°$

$\theta_i(0.5) = 140°$ $\theta_i(0.75) = 70°$ $\theta_i(1.0) = 40°$

Figure 17: $\Delta\theta_i$ exceeds $180°$

in Figure 18(a) for a single deviation, and Figure 18(b) for a double deviation.

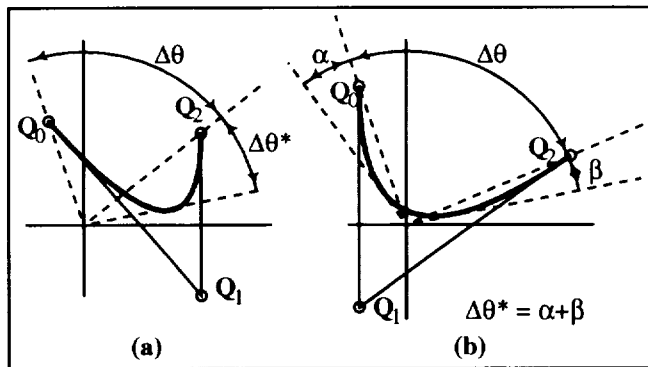$\Delta\theta^* = \alpha + \beta$

(a)   **(b)**

Figure 18: Measurement of $\Delta\theta_i^*$ for non-monotonic $\theta_i(t)$

2.2   Coincident vertices

Coincident vertices are a common occurrence which invite special attention. When n adjacent vertices on one polygon lie at the same point, $n - 1$ edges on the other polygon collapse to that point. Since $\angle(P_{i-1}, P_i, P_{i+1})$ is undefined if P_i is coincident with either of its neighbors, the angle *change* when such a case is involved in a shape blend is also undefined, as is the bending work discussed in section 3.2. Our tests verify that the following heuristic for assessing angle change when vertices are coincident gives good results.

We imagine that coincident vertices actually lie evenly spaced along the base of an infinitesimal isosceles triangle, as shown in Figure 19. In this case, $\theta_2 = \theta_4 = 90° + \frac{\alpha}{2}$ and $\theta_3 = 180°$ in radians. In general, if vertices $P_i, \ldots, P_j$ are coincident, $\theta_i = \theta_j = 90° + \frac{\alpha}{2}$ and $\theta_{i+1} = \theta_{i+2} = \cdots = \theta_{j-1} = 180°$.

In Figure 20, all three vertices of one polygon are coincident. However, as portrayed in Figure 19, those vertices are treated as though they are infinitesimally spaced along a line segment. Thus, in such cases, control point Q_2 (or Q_0) of the Q curve will always be located an infinitesimal distance from the origin along the $-x$ axis. Figure 21 shows the Q curve for vertex $i = 2$ in Figure 20. In this case, Q_2 lies an infinitesimal distance from the origin along a ray $110°$ from the $+x$ axis as shown, and $\Delta\theta = 38°$. Figure 22 shows an example of coincident vertices in which an intermediate angle goes to zero.
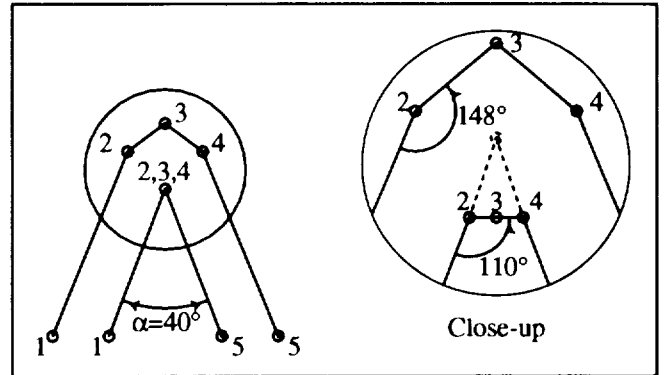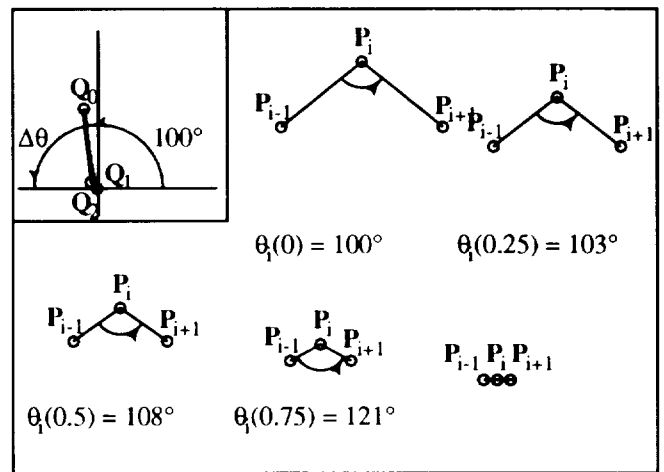

Figure 19: Treatment of coincident vertices

$\theta_i(0) = 100°$ $\theta_i(0.25) = 103°$

$\theta_i(0.5) = 108°$ $\theta_i(0.75) = 121°$

Figure 20: Q curve for three coincident vertices

3   Physically based model

Section 2 defined the shape blending problem to be one of deciding where to add vertices to two polygons so that intermediate polygons in the blend could be defined by interpolating corresponding vertices on the given polygons. The decision on where to add vertices must be guided by some heuristic. The heuristic we propose is to model polygon P^0 as a piece of wire made of some idealized metal. The "best" shape blend is the one which requires the least work to deform P^0 into P^1 through bending and stretching.

This section discusses a simplified model for assessing the work involved in moving each vertex and line segment through the shape blend. Section 4 shows how to compute a globally optimal least work solution for all possible vertex correspondences.

We distinguish between work which causes bending, and work which causes stretching. Stretching work is computed for each line segment (that is, each adjacent pair of vertices) whereas bending work is computed for each adjacent pair of line segments (that is, for each set of three adjacent vertices).

3.1   Stretching work

A force P will stretch an actual wire of length L_0 [12] an amount

$$\delta = \frac{P L_0}{A E} \tag{12}$$

where A is the cross sectional area and E is the *modulus of elasticity*, a constant of the material (for example, E for steel
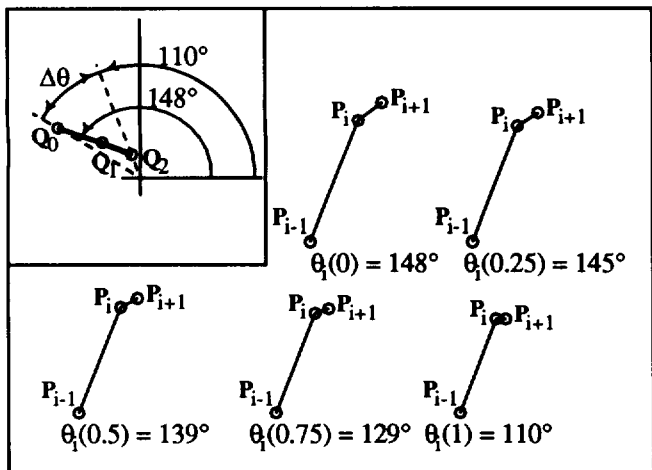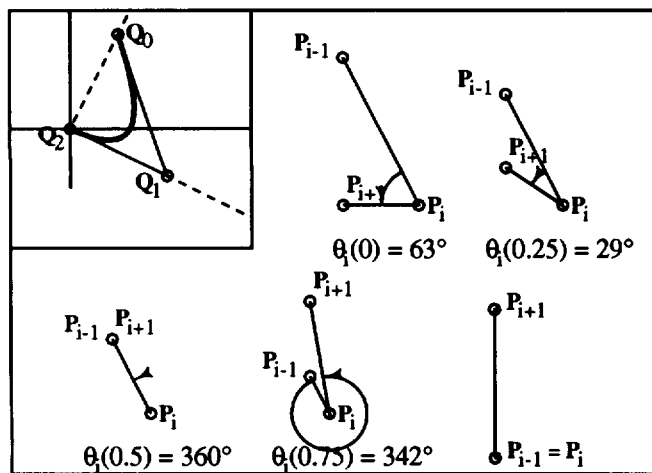
Figure 21: Q curve for two coincident vertices

Figure 22: Coincident vertices with $\theta(.5) = 0$

is $29,000,000$ psi). The work expended in stretching a real piece of wire an amount δ is [12]

$$W = \frac{\delta^2 AE}{2L_0}. \tag{13}$$

Since AE is a constant for the wire, for our purposes we replace it with a single user-definable "stretching stiffness constant" k_s. If L_0 is the initial length of a section of wire, and if L_1 is its final length, equation 13 will compute different values if the initial and final shapes are swapped ($\frac{\delta^2 AE}{2L_0}$ in one case and $\frac{\delta^2 AE}{2L_1}$ in the other). Furthermore, if an edge collapses to a single vertex (i.e., $L_0 = 0$), equation 13 requires infinite work. These two considerations motivate the following modification to equation 13:

$$W_s = k_s \frac{(L_1 - L_0)^2}{(1 - c_s)\min(L_0, L_1) + c_s \max(L_0, L_1)} \tag{14}$$

where $\delta = L_1 - L_0$ and c_s is a user definable constant which controls the penalty for edges collapsing to points.

The exponent 2 in equations 13 and 14 assumes the wire is *linearly elastic*, which is the case if the wire has not stretched very much. If excessive stretching occurs, less work is required to elongate the wire because it undergoes *plastic deformation*

[12]. In this case, an exponent of 1 more closely expresses the work expended. Thus, we make one final modification to our stretching work equation:

$$W_s = k_s \frac{|L_1 - L_0|^{e_s}}{(1 - c_s)\min(L_0, L_1) + c_s \max(L_0, L_1)}, \tag{15}$$

where k_s, c_s, and e_s are user definable constants.

In physical reality, these work equations only make sense if the wire is getting longer ($L_1 > L_0$), not if it is getting shorter ($L_1 < L_0$). For our purposes, we compute both stretching and compressing work using equation 15.

Section 4 calls for notation which expresses which segment of wire is being stretched. Letting $L_0 = \|P_{i_1} - P_{i_0}\|$ and $L_1 = \|P_{j_1} - P_{j_0}\|$, we denote by

$$W_s([i_0, j_0], [i_1, j_1]) = \frac{k_s |L_1 - L_0|^{e_s}}{(1 - c_s)\min(L_0, L_1) + c_s \max(L_0, L_1)} \tag{16}$$

the stretching work required to map $P_{i_0}-P_{i_1}$ to $P_{j_0}-P_{j_1}$, where $i_1 = i_0$ or $i_1 = i_0 + 1$ and $j_1 = j_0$ or $j_1 = j_0 + 1$.

3.2 Bending work

Analogous to the equation for stretching work developed in section 3.1, work which causes bending is defined in equation 18 for angle $\angle(P_{i_0}^0, P_{i_1}^0, P_{i_2}^0)$ bending into angle $\angle(P_{j_0}^1, P_{j_1}^1, P_{j_2}^1)$:

$$W_b([i_0, j_0], [i_1, j_1], [i_2, j_2]) = \tag{17}$$
$$\begin{cases} k_b(\Delta\theta + m_b \Delta\theta^*)^{e_b} & \text{if } \theta(t) \text{ never goes to zero} \\ k_b(\Delta\theta + m_b \Delta\theta^*)^{e_b} + p_b & \text{if } \theta(t) \text{ does go to zero} \end{cases}$$

where $\Delta\theta$ and $\Delta\theta^*$ are measured in radians and are defined in sections 2.1 and 2.2. k_b, m_b, e_b, and p_b are user definable constants. The constant k_b indicates bending stiffness, m_b penalizes angles which are not monotonic, e_b is an exponent which plays a role similar to e_s, and p_b penalizes angles from going to zero.

3.3 Normalization

Notice that the work due to bending is independent of the size of the shapes. Thus, if the two shapes are scaled uniformly, the bending work computation does not change. However, the stretching work varies with the scale of the shapes. To make the constants k_s and e_s independent of scale, it is a good idea to map each shape to a unit *rectangle*, scaling the same amount in x and y, so that the largest dimension of the bounding box is one. It is important to scale uniformly, or else the angles will change, along with the bending work computation.

It is noteworthy that uniform scaling of the shapes *does* affect the $Q(t)$ curves, but not the bending work computation. If P^1 is scaled by a constant c, then Q_0 is unchanged, Q_1 is scaled by c, and Q_2 is scaled by c^2. This creates a different $Q(t)$ curve, but the angle function $\angle((0,1), (0,0), Q(t)) = \theta(t)$ does not change.

3.4 Numerical examples

This section provides two numerical examples of the work required to transform a unit isosceles right triangle into a unit square. In Figure 23, there is no stretching work in line segments 0-1 and 3-4 and no bending work in angle 0. The stretching work in legs 1-2 and 2-3 is each $k_s \frac{|1 - \frac{\sqrt{2}}{2}|^{e_s}}{\frac{\sqrt{2}}{2}(1 - c_s) + 1 \cdot c_s}$.

The bending work in angles 1 and 3 is each $k_b(\frac{\pi}{4})^{e_b}$ and in vertex 2 is $k_b(\frac{\pi}{2})^{e_b}$. Thus, the total work is

$$W_1 = 2k_s \frac{.293^{e_s}}{.707 + .293c_s} + 2k_b.785^{e_b} + k_b 1.571^{e_b}. \quad (18)$$
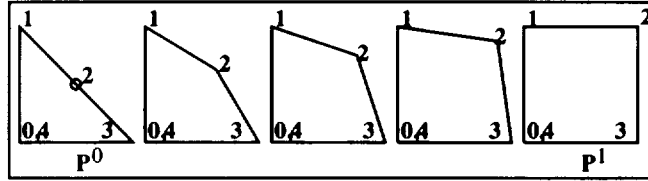

Figure 23: Work computation example, blend 1

In Figure 24, there is also no stretching work in line segments 0-1 and 3-4 and no bending work in angle 0. The stretching work in line segment 1-2 is $k_s \frac{|1-0|^{e_s}}{0\cdot(1-c_s)+1\cdot c_s} = \frac{k_s}{c_s}$ and in line 2-3 is $k_s \frac{|1-\sqrt{2}|^{e_s}}{1(1-c_s)+\sqrt{2}c_s}$. The bending work in angles 1 and 2, based on section 2.2, is $k_b \left[\left(\frac{1}{2}\frac{\pi}{4} + \frac{\pi}{2}\right) - \frac{\pi}{2}\right]^{e_b} = k_b \left(\frac{\pi}{8}\right)^{e_b}$. Angle 3 has a $\Delta\theta$ of $\pi/4$ and hence a bending work of $k_b \frac{\pi}{4}^{e_b}$. Thus, the total work to perform the transformation in Figure 24 is

$$W_2 = k_s \left(\frac{1}{c_s} + \frac{.414^{e_s}}{1 + .414c_s}\right) + k_b[2(.393)^{e_b} + .785^{e_b}]. \quad (19)$$
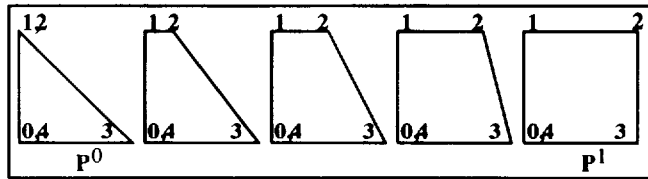

Figure 24: Work computation example, blend 2

By selecting different coefficients k_s, k_b, e_s, e_b, and c_s, we can coerce either blend to have a smaller work requirement. For example if $k_s = k_b = .5$, $e_s = e_b = 1$ and $c_s = 0.5$, $W_1 = 1.91$ and $W_2 = 1.96$. However, if we change $k_s = 0.4$ and $k_b = 0.6$, then $W_1 = 2.16$ and $W_2 = 1.88$. So, if the wire stretches more easily than it bends, blend 2 uses less work.

4 Least work solution

This section presents a method for determining where to insert vertices so that the shape transformation is accomplished with the least work.

This method determines a globally optimal least work solution for all possible correspondences of *existing* vertices, so vertices can only be inserted at existing vertices. As a preprocessing step, additional vertices can be added to break up long line segments. The reason the optimization search is restricted to existing vertices is that otherwise it becomes a non-linear constrained optimization problem whose solution is very expensive and whose global optimality is difficult to verify. By contrast, the discrete solution presented here can be solved in $O(mn)$ time in the number of respective vertices, and global optimality is assured.

One of the first papers written on contour triangulation, [15], employs a directed graph to compute an optimal triangulation between a pair of contour lines. [8] further refined the use of the directed graph for that problem. Our least work solution is primarily based on those two excellent papers. This section briefly reviews the use of directed graphs, giving only enough detail to explain how the ideas in [8] are adapted.

Given two polygons $\mathbf{P}^0 = [\mathbf{P}_0^0, \mathbf{P}_1^0, \ldots, \mathbf{P}_m^0]$ and $\mathbf{P}^1 = [\mathbf{P}_0^1, \mathbf{P}_1^1, \ldots, \mathbf{P}_n^1]$, all vertex correspondences can be represented in an $m \times n$ rectangular matrix, or "graph". The columns of the graph represent vertices on $\mathbf{P}^0$ and the rows of the graph represent vertices on $\mathbf{P}^1$. The point at which column i meets row j signifies a correspondence between $\mathbf{P}_i^0$ and $\mathbf{P}_j^1$.

Denote by $[i, j]$ a correspondence between $\mathbf{P}_i^0$ and $\mathbf{P}_j^1$, which can be represented on the graph as a dot at the junction of column i and row j. A complete shape transformation requires every vertex in $\mathbf{P}^0$ to correspond to at least one vertex in $\mathbf{P}^1$ and vice versa. Furthermore, we only allow $[i, j]$ to be a correspondence if $[i-1, j]$, $[i, j-1]$, or $[i-1, j-1]$ is also a correspondence — else intermediate polygons in the shape transformation would split apart. Given that $[0, 0] = [m, n]$ is a correspondence, a complete solution can be represented on the graph as a string of dots starting at $[0, 0]$ and ending at $[m, n]$, with each subsequent dot positioned one step East, South, or Southeast from the preceding dot. This is illustrated in Figure 25, where the dots are connected by arrows. We will refer to such a sequence of dots as a *path*, denoted by
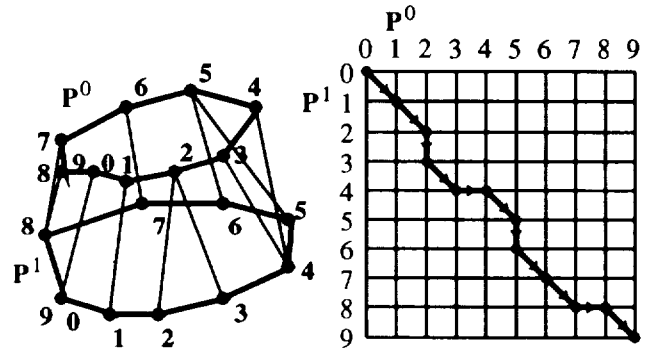

Figure 25: Graph representation of a shape transformation

$\{c_0, c_1, \ldots, c_k\}$. In Figure 25, $c_0 = [0, 0]$, $c_3 = [2, 3]$, etc. Note that k is the number of vertices in each intermediate polygon in the blend, with $\max(m, n) \leq k \leq m + n$.

We do not allow a South step to be immediately followed by an East step, or an East step to be followed by a South step, because such a combination is more expensive than a single Southeast step, except possibly under unusual work coefficients. The main reason for this rule is to save some computation. Thus, we may say that a path must travel in a Southeasterly direction, and make no 90° turns.

Consider how to evaluate the bending and stretching work for the example in Figure 25. Stretching work is computed for each pair of neighboring dots on the path (c_{l-1}, c_l), since each such pair of dots on the path represents two points on $\mathbf{P}^0$ transforming to two points on $\mathbf{P}^1$. Bending work must be computed using *three* neighboring dots on the path (c_{l-1}, c_l, c_{l+1}), since an angle change involves three points on $\mathbf{P}^0$ moving to three points on $\mathbf{P}^1$.

For large $m = n$, there are $O\left(\frac{m^m}{m!}\right) = O\left(\frac{e^m}{\sqrt{m}}\right)$ legal paths. However, using a graph, the least work solution can be determined by visiting each junction only once ($O(m^2)$

31

computation expense). The basic strategy proceeds by considering polygon fragments consisting of vertices $0, \ldots, i$ of $\mathbf{P}^0$ and vertices $0, \ldots, j$ of $\mathbf{P}_1$. Denote these polygon fragments $\mathbf{P}^0(i)$ and $\mathbf{P}^1(j)$. We wish to compute the minimum work required to transform $\mathbf{P}^0(i)$ into $\mathbf{P}^1(j)$, according to the work equations in section 3. In graph theory language, we want to find the path which connects $[0, 0]$ to $[i, j]$ using the minimum amount of work, denoted $W(i, j)$. This is easily accomplished using the simple observation that if we know the minimum work values $W(i-1, j)$, $W(i, j-1)$, and $W(i-1, j-1)$, then $W(i, j)$ must equal one of those three predecessors plus the incremental work involved in connecting that predecessor with $[i, j]$.

To accomplish this, we must actually concern ourselves with *three* values of $W(i, j)$, denoted by $W_\uparrow(i, j)$, $W_\nwarrow(i, j)$, and $W_\leftarrow(i, j)$, which indicate the minimum work required to transform $\mathbf{P}^0(i)$ into $\mathbf{P}^1(j)$ if $[i, j-1]$, $[i-1, j-1]$, or $[i-1, j]$ respectively is the preceding dot on the path. These three values are required because each bending work computation relies on three dots on the path. We thus proceed by assigning $W_\uparrow(0, 0) = W_\leftarrow(0, 0) = W_\nwarrow(0, 0) = 0$ and computing for $j = 0, \ldots, n$ and for $i = 0, \ldots, m$ (i and j not both $= 0$):

$$W_\leftarrow(i, j) = W_s([i-1, j], [i, j]) + \qquad (20)$$
$$\min\{W_\leftarrow(i-1, j) + W_b([i-2, j], [i-1, j], [i, j]),$$
$$W_\nwarrow(i-1, j) + W_b([i-2, j-1], [i-1, j], [i, j])\}$$

$$W_\uparrow(i, j) = W_s([i, j-1], [i, j]) + \qquad (21)$$
$$\min\{W_\uparrow(i, j-1) + W_b([i, j-2], [i, j-1], [i, j]),$$
$$W_\nwarrow(i, j-1) + W_b([i-1, j-2], [i, j-1], [i, j])\}$$

$$W_\nwarrow(i, j) = W_s([i-1, j-1], [i, j]) + \qquad (22)$$
$$\min\{W_\uparrow(i-1, j-1) + W_b([i-1, j-2], [i-1, j-1], [i, j]),$$
$$W_\nwarrow(i-1, j-1) + W_b([i-2, j-2], [i-1, j-1], [i, j]),$$
$$W_\leftarrow(i-1, j-1) + W_b([i-2, j-1], [i-1, j-1], [i, j])\}$$

where $W_b([i_1, j_1], [i_2, j_2], [i_3, j_3]) = \infty$ for $i_1 < 0$ or $j_1 < 0$.

The value $\min(W_\leftarrow(m, n), W_\nwarrow(m, n), W_\uparrow(m, n))$ is the global least work. This is of secondary interest; what we *really* want to know is what path results in this minimum work. The path is determined by backtracking through the graph, a process discussed in [8]. Actually, our backtracking is slightly more complicated because each graph node must keep track of *three* backpointers, one for each direction from which the node can be approached in the backtrack.

4.1 Implementation

There is no need to store more than two rows of $W_\leftarrow(i, j)$, $W_\nwarrow(i, j)$, $W_\uparrow(i, j)$ information. Once a complete row of work values has been computed, the previous row can be discarded.

The algorithm as stated goes to a lot of effort to assure that it has found a path of globally minimal work. In particular, the bending work costs much more to compute than does the stretching work, since each node in the graph can represent the middle vertex of seven different angle changes. A much more economical (three times faster!) implementation is possible, which no longer assures a global minimum work solution, but which provides virtually identical results to the rigorous algorithm dicussed above. The simplified algorithm uses only one value of $W(i, j)$ (instead of calculating $W_\uparrow(i, j)$, $W_\nwarrow(i, j)$, and $W_\leftarrow(i, j)$) as follows.

For purposes of discussion, if point $[i, j]$ lies on a path, the preceding point on the path is indicated using the functions $west(i, j)$ and $north(i, j)$. If the preceding point is directly West of $[i, j]$, then $west(i, j) = 1$ and $north(i, j) = 0$. If the preceding point is straight up from $[i, j]$, then $west(i, j) = 0$

and $north(i, j) = 1$. If the preceding point is North-West of $[i, j]$, then $west(i, j) = 1$ and $north(i, j) = 1$. Define

$$w_0 = W(i-1, j) + W_s([i-1, j], [i, j]) + \qquad (23)$$
$$W_b([i-1-west(i-1, j), j-north(i-1, j)],$$
$$[i-1, j], [i, j])$$

$$w_1 = W(i, j-1) + W_s([i, j-1], [i, j]) + \qquad (24)$$
$$W_b([i-west(i, j-1), j-1-north(i, j-1)],$$
$$[i, j-1], [i, j])$$

$$w_2 = W(i-1, j-1) + W_s([i-1, j-1], [i, j]) + \qquad (25)$$
$$W_b([i-1-west(i-1, j-1), j-1-north(i-1, j-1)],$$
$$[i-1, j-1], [i, j])$$

where w_0 is undefined for $i = 0$, w_1 is undefined for $j = 0$, and w_2 is undefined for $i = 0$ or $j = 0$. Then

$$\text{if } w_0 \leq w_1, w_2 \quad : \quad W(i, j) = w_0; \quad west(i, j) = 1;$$
$$north(i, j) = 0 \qquad (26)$$

$$\text{if } w_1 \leq w_0, w_2 \quad : \quad W(i, j) = w_1; \quad west(i, j) = 0;$$
$$north(i, j) = 1 \qquad (27)$$

$$\text{if } w_2 \leq w_0, w_1 \quad : \quad W(i, j) = w_2; \quad west(i, j) = 1;$$
$$north(i, j) = 1 \qquad (28)$$

The algorithm for computing the least work solution amounts to setting $W(0, 0) = 0$ and from equations 23 — 28 computing $W(i, j)$; $i = 0, \ldots, m$; $j = 0, \ldots, n$. $W(m, n)$ is then the optimal total least work, and the *north* and *west* information can be used to backtrace the path which leads to this least work solution.

We recommend using this simplified algorithm, because it is easier to implement, it runs three times faster than the theoretically precise algorithm, and the results are visually similar.

4.2 Starting points

The above discussion assumes that point $\mathbf{P}_0^0$ corresponds to point $\mathbf{P}_0^1$. A globally minimum work solution for *any* initial correspondence can be computed in $O(mn \ln n)$ time (see [8]). The example in Figure 9 shows a zero work solution which was found by considering all possible starting correspondences. All the other Figures in the paper had the initial correspondence specified.

5 Examples and discussion

The work equations in section 3 contain seven user definable constants: k_s, k_b, e_s, e_b, p_b, m_b, and c_s. k_s and k_b can be restricted to the unit interval. Table 1 shows the coefficients used by our algorithm to blend the figures in this paper. The meaning of diagonal deviation dd is discussed in section 5.2.

Figures 8-11 underscore the inherent ambiguity of the shape blending problem. Without human guidance, no algorithm could discern which of these four solutions is appropriate, since one can think of specific instances in which each of them might be preferred. Note that for these shape pairs, parameter adjustment can achieve the different desired results. As mentioned, the only user intervention for the examples in this paper is the specification of starting points and of the seven constants. However, it is easy to contrive examples where no set of seven constants will produce a prescribed blend (such as in Figure 26, which consists of a combination of Figures 8 and 11). In some cases, it may be needful for the user to specify a few other correspondences as well.

Figure	dd	k_s	k_b	e_s	e_b	p_b	m_b	c_s
1	.06	1	.2	2	2	1000	10	.5
2	.09	0	1	2	2	1000	10	.5
3	.09	1	1	2	.5	1000	10	.5
8	0	0	1	2	.8	1000	10	.5
10	0	1	0	2	1	1000	10	.5
11	.125	0	1	2	1	1000	10	.5
27	.245	.3	1	2	.3	1000	10	.5
28	.04	1	.1	2	.1	1000	10	.5

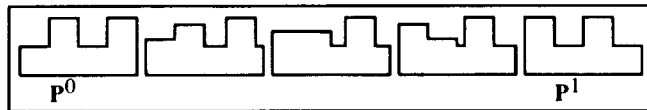Table 1: Coefficients for various example figures

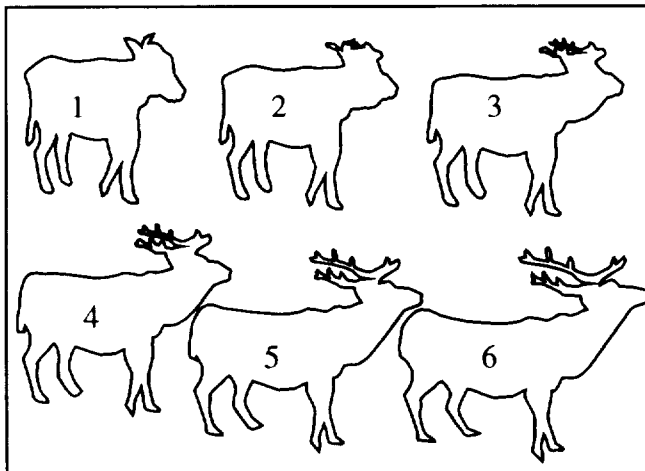

Figure 26: Unattainable example

Figure 27: Cow to deer

Figure 27 shows a blend from a cow to a deer. Notice that some of the antlers cross each other in the intermediate shapes. A high value of p_b prevents *local* self-intersections (angles going to zero) but not global self intersections.

Figure 28 involves excessive movement of the dancer's arm. Due to the linear motion between corresponding vertices, the arm shortens as it moves.
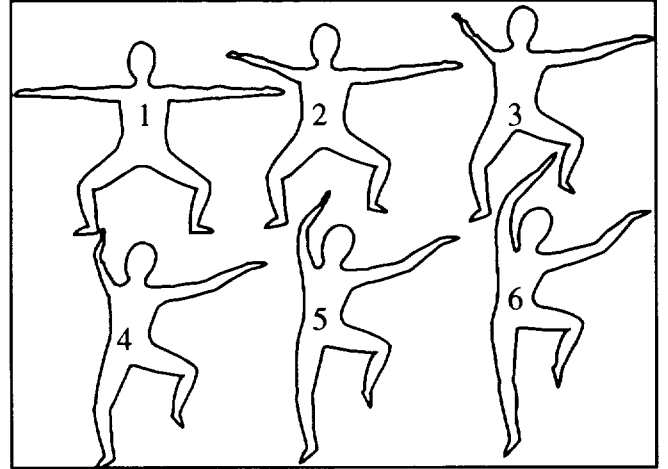

Figure 28: Dancer

5.1 Preprocessing

The heuristic described in this paper relies on a reasonable initial distribution of polygon vertices. For example, Figure 2 requires additional vertices to be inserted along some of the straight segments of the F shape in order to provide a pleasing correspondence with the base of the E.

Since the work equations are more realistic for distinct vertices than for coincident vertices, the algorithm tends to work best if the two polygons have roughly the same number of vertices. This tends to reduce the number of coincident vertices in the final solution, since there must be a minimum of $|m - n|$ coincident vertices.

5.2 Speedups

We timed an example in which each polygon has 100 vertices, and the execution time on an IBM RS6000 Model 530 workstation is 8 seconds using unoptimized code. Using the simplified alorithm in section 4.1, the execution time is 3 seconds.

In most cases, the graph of the least work solution has a path (see Figure 25) which does not deviate very far from the diagonal of the graph. Recall that a polygon $\mathbf{P}^0$ with m vertices and a polygon $\mathbf{P}^1$ with n vertices create a graph with m columns and n rows. The amount which graph point $c[i, j]$ deviates from the graph diagonal is

$$\left| \frac{i}{m} - \frac{j}{n} \right|. \tag{29}$$

The diagonal deviation of an entire path is

$$dd = \max_{[i,j] \in \text{path}} \left| \frac{i}{m} - \frac{j}{n} \right|. \tag{30}$$

The cow-to-deer blend path (see Figure 27) is shown in Figure 29. Its diagonal deviation of .245 is the largest of any of the examples in this paper (see Table 1). Thus, instead of searching the entire rectangular graph, the least work solution can generally be determined by visiting only those elements of the graph within a distance dd from the diagonal.
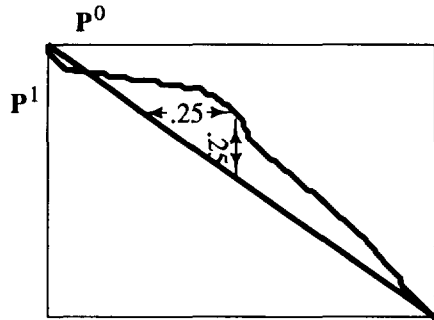
Figure 29: Cow to deer least work path; diagonal deviation

6 Future Work

This research seems to have generated more questions than it has answered. The authors are currently looking at several follow-up problems. For example, to make this process useful for keyframe animation and morphing, an interior preserving map is needed for interpolating raster images which are enclosed by the terminal polygons. We are currently studying how well Schwartz-Christoffel transformations solve this problem.

How do we deal with cases where a scene composed of m number of polygons blends into a scene composed of n polygons, perhaps including holes? Ideas from [5] may help decide.

What about using periodic B-splines instead of polygons? An easy answer is to simply polygonize the curves and apply the current algorithm, but a more satisfying answer is to develop energy minimization methods which work directly on the curves. The calculus of variations may provide help here.

All the blends in this paper involve moving corresponding vertices along linear paths. This can create some undesirable effects, such as the withering arm in blend 4 of Figure 28. Q curves are easily extended to express angle change and segment length when vertices travel along Bézier curves of any degree. Study is underway in identifying curved paths which relieve the withering arm problem.

The work model assumes that each wire has uniform stiffness. There may be merit to specifying that some portions of the wire are more stiff than others, suggesting a relative discouragement towards altering those portions. For example, in the cow-to-deer blend, it may be helpful to assign a smaller stiffness to the antlers than to the rest of the deer.

Of course, extending this algorithm to polygonal surfaces in 3-D is a worthwhile goal.

Applications to other fields such as pattern and signature recognition are also being studied.

More detail on the material in this paper can be found in [11].

References

[1] Adobe Systems, Inc. *Adobe Illustrator 88.*

[2] Bruce G. Baumgart. *Geometric Modeling for Computer Vision.* Phd thesis, Stanford University, Computer Science Department, 1974.

[3] Edwin Catmull. The problems of computer-assisted animation. *Computer Graphics*, 12(3):348-353, 1978.

[4] Shenchang Eric Chen and Richard Parent. Shape averaging and its applications to industrial design. *IEEE CG&A*, 9(1):47-54, 1989.

[5] Henry N. Christiansen and Thomas W. Sederberg. Conversion of complex contour line definitions into polygonal element mosaics. *Computer Graphics (Proc. SIGGRAPH)*, 12(3):187-192, 1978.

[6] Computer Support Corporation. *Arts & Letters 3.01*, 1991.

[7] Corel Systems Corporation, Ottawa, Canada. *CorelDraw! 2.0*, 1990.

[8] Henry Fuchs, Z. M. Kedem, and S. P. Uselton. Optimal surface reconstruction from planar contours. *Comm. ACM*, 20(10):693-702, 1977.

[9] S. Ganapathy and T. G. Dennehy. A new general triangulation method for planar contours. *Computer Graphics (Proc. SIGGRAPH)*, 16(3):69-75, 1982.

[10] Andrew Glassner. Metamorphosis. *preprint*, 1991.

[11] Eugene Greenwood. A physically based approach to 2-d shape interpolation. Master's thesis, Brigham Young University, Department of Mechanical Engineering, 1992.

[12] Archie Higdon, Edward H. Ohlsen, William B. Stiles, John A. Weese, and William F. Riley. *Mechanics of Materials.* John Wiley & Sons, Inc., New York, 1976.

[13] Michael Kass, Andrew Witkin, and Demetri Terzopoulos. Snakes: Active contour models. *International Journal of Computer Vision*, 1(3):321-331, 1988.

[14] Anil Kaul and Jarek Rossignac. Solid-interpolating deformations: Construction and animation of PIPs. In F.H. Post and W. Barth, editors, *Proc. Eurographics '91*, pages 493—505. Elsevier Science Publishers B.V, 1991.

[15] E. Keppel. Approximating complex surfaces by triangulation of contour lines. *IBM Journal of Research and Development*, 19:2-11, 1975.

[16] David Kurlander and Eric A. Bier. Graphical search and replace. *Computer Graphics*, 22(4):113-120, 1988.

[17] Micrografx, Inc., Richardson, TX. *Designer 3.1.*

[18] Software Publishing Corp., Sunnyvale, CA. *Harvard Graphics 3.0*, 1991.

[19] Naonori Ueda and Satoshi Suzuki. Automatic shape model acquisition using multiscale segment matching. In *Proc. 10th ICPR*, pages 897—902. IEEE, 1990.