Refolding Planar Polygons

Hayley N. Iben* James F. O'Brien* Erik D. Demaine**

*University of California, Berkeley **Massachusetts Institute of Technology

Abstract

This paper describes a guaranteed technique for generating intersection-free interpolation sequences between arbitrary planar polygons. The computational machinery that ensures against self intersection guides a user-supplied distance heuristic that determines the overall character of the interpolation sequence. This approach provides the user with a powerful control mechanism for determining how the interpolation should appear, while still being assured against intersection. Our framework provides additional user control by accommodating algebraic constraints that can be enforced throughout the interpolation sequence.

Keywords: Polygon interpolation, morphing, shape transformation, refolding.

CR Categories: I.3.5 [Computer Graphics]: Computational Geometry and Object Modeling—Shape Interpolation

1 Introduction

In this paper we describe a technique for interpolating between two planar, non-self-intersecting polygons. Our technique is guaranteed to always find an interpolation path for any two input polygons, and the polygons along the interpolation path will never self intersect. The computational machinery that ensures against self intersection guides a user-supplied distance heuristic that determines the overall character of the interpolation sequence. This approach provides the user with a powerful control mechanism for determining how the interpolation should appear. Our framework provides additional user control by accommodating algebraic constraints that can be enforced throughout the interpolation sequence.

Our technique is based on recent results from [Connelly et al., 2003] and [Streinu, 2000] showing that any planar collection of polygons and polylines can be *unfolded* to an *outer-convex* configuration. In the case of a single polygon, these results imply that any arbitrary polygon can be continuously deformed into a convex polygon without changing any of its edge lengths. The motions implied by [Connelly et al., 2003] and [Streinu, 2000] are difficult to compute directly, but based on the existence of these motions we have shown in [Cantarella et al., 2004] that a much simpler class of motions can also unfold any collection of polygons and polylines to an outer-convex configuration. The simpler motions are easy to compute, and they correspond to the downward gradient of a "repulsive" energy function based on the vertex-to-edge distances within the polygon.

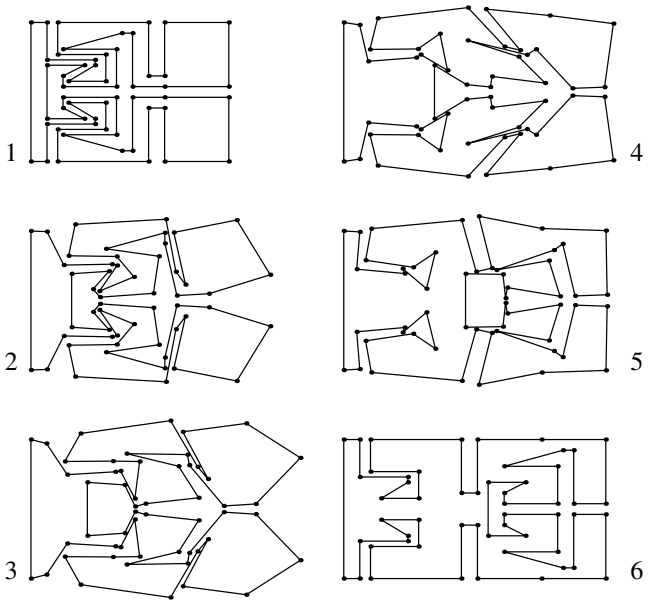

Figure 1: An intersection-free interpolation sequence generated using our algorithm. The first and last frames are the two polygons being interpolated between. For this example, all edge lengths were held constant, vertex positions were used for the direction metric, and the total computation time was 3.5 minutes.

Because one can trivially interpolate between any two convex polygons, these unfolding results provide an obvious way to build a path from one polygon to another. However, interpolating between two similar polygons by ballooning the first polygon into a convex shape and then folding it back down to the shape of the second polygon is probably not useful for most applications.

Our technique uses a heuristic supplied by the user to determine how one polygon should change its shape to match another. This heuristic does not have to be particularly sophisticated: even linear interpolation of the vertex positions will suffice. As the heuristic attempts to build a path interpolating between the two polygons, our algorithm uses the repulsive energy function to steer the heuristic around self intersections. As demonstrated by Figure 1, the appearance of the resulting motion is predominantly governed by the user's heuristic yet still avoids self intersection.

E-mail: {iben|job}@eecs.berkeley.edu edemaine@mit.edu

Please see http://www.cs.berkeley.edu/b-cam/ for further information.

2 Background

The task of interpolating between polygons is often divided into two subproblems: establishing vertex correspondences and computing vertex paths. In some cases, for example [Sederberg and Greenwood, 1992] and [Carmel and Cohen-Or, 1997], researchers have focused primarily on establishing vertex correspondences while using a simple method, such as linear interpolation of the vertex positions, to create the intermediate polygons. In this paper, we do not address algorithms for finding vertex correspondences. We assume that some other algorithm, or the user, will supply suitable correspondences. So long as the correspondences order the vertices consistently our interpolation algorithm is guaranteed to succeed.

Other approaches have focused on more sophisticated interpolating schemes for computing vertex paths. In [Sederberg et al., 1993], intermediate frames between two shapes are computed by linearly interpolating the vertex angles and the edge lengths, giving better results for rigid transformations than previous work using vertex positions. The authors of [Goldstein and Gotsman, 1995] create a multiresolution representation for each input polygon. Their algorithm interpolates between these representations to create the intermediate polygons. The method described in [Shapira and Rappoport, 1995] decomposes each input polygon into a planar tree of star-shaped pieces, called a star skeleton. The points of the star skeleton, represented in polar coordinates, are linearly interpolated to create the intermediate shapes. In [Alexa et al., 2000], they decompose the input objects into compatible triangulations. They then compute transformations between the triangulations that minimize local distortion. None of these methods guarantee that the intermediate polygons they generate will be intersection-free.

Both [Guibas and Hershberger, 1994] and [Efrat et al., 2001] generate non-intersecting sequences for limited types of input. The method in [Guibas and Hershberger, 1994] operates on pairs of polygons that have corresponding parallel edges. The method in [Efrat et al., 2001] operates on simple polylines.

A more general method appearing in [Gotsman and Surazhsky, 2001] embeds the polygons inside a convex region, generates a pair of compatible triangulations, and then builds a sequence between them by interpolating the stochastic matrices whose unit eigenvectors encode the triangulations' geometries. Like the method we present here, their method can guarantee that all intermediate polygons will not self intersect. However, their method does not afford the user with a way to control the character of the interpolation sequence.

Our algorithm ensures that interpolation sequences will be intersection-free, and it also decouples vertex correspondence and path computation from intersection avoidance. Intersection avoidance does, of course, affect the vertex paths, but the user is free to supply a direction heuristic that will generate whatever type of path they like. The intersection avoidance machinery only interferes to the extent required. Thus, one could see our method either as an independent interpolation method, or as a wrapper to be used with any of the above methods that generate interesting, but possibly intersecting, vertex paths. For example, [Alexa et al., 2000] produce paths that avoid needless distortion, but that might intersect. If combined with our method, we expect that the resulting algorithm would produce predominantly "rigid-as-possible" motions that distort only as needed to avoid intersection.

In addition to methods that operate directly on explicit polygonal representations, several other methods for interpolating shapes have been described in the literature. Both [Turk and O'Brien, 1999] and [Cohen-Or et al., 1998] interpolate between shapes by interpolating scalar fields that implicitly define the shapes. The authors of [He et al., 1994] and [Hughes, 1992] discuss methods for interpolating volumetric data. A method based on Minkowski sums appears in [Kaul and Rossignac, 1991].

3 Unfolding Groundwork

Our method stems from recent results showing that any planar collection of polygons and polylines can be *unfolded* to an *outer-convex* configuration. In an outer-convex configuration, all polygons or polylines that are not contained inside another polygon are separated from each other, and either convexified (polygons) or straightened (polylines). An unfolding motion preserves edge lengths and avoids self intersection. The existence of these unfolding motions has been proven independently in both [Connelly et al., 2003] and [Streinu, 2000] using two distinct approaches.

While both proofs imply the existence of unfolding motions, actually computing the motions they directly imply can be difficult. However, the motion implied by [Connelly et al., 2003] has the additional property that it is *strictly expansive*, meaning that the motion strictly increases the distances between *all* vertices not sharing an edge. In [Cantarella et al., 2004] we show that given the existence of expansive motions, we can reformulate the unfolding problem as one where we simply seek to minimize a suitable energy function. A suitable energy function is one with the following properties:

Charge — the value of the function is finite for any intersection-free configuration and approaches $+\infty$ as the system approaches self intersection.

Repulsive — the energy function decreases to first order under any expansive motion.

Separable — as distinct connected-components recede from each other, any energy terms relating them should vanish.

$C^{1,1}$ — the function should be C^1 continuous with bounded curvature.

It is easy to show that a simple optimization strategy, such as gradient descent, can be used to generate an intersection-free interpolation path from any polygon to a convex polygon, and that the space of valid configurations contains no local minima to get stuck in. The results also imply that the energy function contains no critical points of any kind at non-outer-convex points in the space of valid configurations and that the valid configuration space is simply connected. A detailed convergence proof with step bounds appears in [Cantarella et al., 2004], but in summary, for a single polygon:

1. By charge, the energy function is finite for any valid initial polygon and the energy function approaches $+\infty$ as the system approaches self intersection, so any path that starts with a non-intersecting polygon and strictly decreases energy cannot lead to a self intersection.

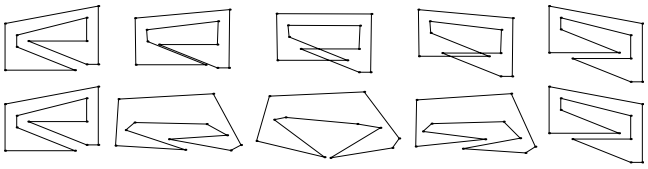

Figure 2: The *top row* demonstrates how using the vertex-position metric alone will, as expected, generate a sequence with self intersections. The *bottom row* illustrates how the collision avoidance machinery alters the vertex motions to avoid self intersection. Computation time was less than two seconds.

2. By repulsiveness, an expansive direction in configuration space is a direction that decreases the energy, and from [Connelly et al., 2003] we know that such a direction always exists unless the polygon is already convex. Therefore, the gradient can never vanish except for convex polygons, and there can be no local minima that do not correspond to a convex configuration.

Together these two observations guarantee that any continuous gradient descent path starting from any valid polygon will converge to a convexified polygon, and that at no point along the path will the polygon intersect itself.

4 Energy and Parameterization

In [Cantarella et al., 2004], we used an energy function based on the elliptic distance between edges and vertices because a C^2 energy function facilitates placing an actual bound on the worst-case number of Euler steps that might be required to convexify a given collection of polygons and polylines. We also used an angle-based parameterization because it allows us to guarantee that all edge lengths are preserved exactly.

Here, however, we prefer to use an energy based on Euclidean distances because it converges faster in practice. Additionally, we choose to parameterize using the vertex positions directly and enforce any desired edge-length preservation using algebraic constraints. This decision simplifies interpolation between polygons with different edge lengths, and it also preserves any symmetries by treating all edges equivalently.

For a polygon with N vertices, let $\boldsymbol{v}_i$ with $i \in [1 \dots N]$ denote the positions of the vertices, let e_i be the edge between $\boldsymbol{v}_i$ and $\boldsymbol{v}_{i+1}$, and let l_i be the edge's length.[1] The energy corresponding to the polygon's configuration is given by

$$E = \sum_{i=1}^{N} \sum_{j=1}^{N} \frac{1}{\mathrm{dist}(\boldsymbol{v}_i, e_j)^2} \qquad (1)$$

where $\mathrm{dist}(\boldsymbol{v}_i, e_j)$ is the Euclidean distance between edge j and vertex i. It is easy to verify that this energy function is charge, separable, $C^{1,1}$, and, except for a few degenerate examples, repulsive.

5 Refolding

Our interpolation method relies on the energy-based unfolding framework to guarantee that it can always construct an intersection-free sequence between any two polygons. In the worst case, the algorithm will convexify both polygons, trivially interpolate between the two convex polygons, and produce the sequence begin-polygon → convexified-begin-polygon → convexified-end-polygon → end-polygon.

For many applications, this worst-case result is not particularly useful, so the algorithm uses a user-supplied direction heuristic to generate a more desirable path. Because the energy function provides a guiding framework, this heuristic can be quite simplistic and still produce good results. In fact, many of the examples shown in this paper were produced using the trivial heuristic that would simply move the vertices on a straight line to their target location. As shown in Figure 2, this heuristic alone produces intersecting sequences, but it can be guided around intersections by an appropriate energy function.

We can also include algebraic constraints that should be enforced throughout the interpolation. These constraints could be simply bundled into the direction heuristic, but then the intersection-avoidance machinery would tend to violate them needlessly. Instead, we combine the projection step that prevents self intersection with the projections that preserve the user constraints. In the special case where the user constraints seek to make edge lengths constant (or change them monotonically) we can guarantee, based on the previously described unfolding results, that they will not conflict with intersection avoidance. However, arbitrary constraints may conflict with intersection avoidance, so they will only be enforced to the extent that they do not cause the algorithm to fail.

5.1 The Algorithm

The following pseudo-code describes our algorithm for generating an interpolation sequence between two polygons, A and B:

1. Establish compatibility and correspondence:

   The user, or some heuristic, indicates the desired correspondence between A and B and renumbers vertices accordingly. If one of the polygons contains fewer vertices than the other, then additional vertices are inserted.

2. While A and B are different:

   a. Compute the energy for A and B.
   b. Use the direction heuristic to determine a direction, D, that would move the higher energy polygon, H, toward the lower energy one, L.
   c. Optional: Project D to enforce edge-length or other constraints.
   d. If D would move H to a higher energy configuration:
      i. Project D so that it is perpendicular to the energy gradient. (Attempt to honor any constraints if they are in use.)
   e. If D is null:
      i. Set D to the direction of the downward energy gradient at H. (Attempt to honor any constraints if they are in use.)
   f. Move H in the direction D.

3. Output the path taken by A to the common configuration and the reverse of the path taken by B.

[1] Addition and subtraction on indices should be understood to wrap around appropriately, so that $\boldsymbol{v}_{N+1}$ is equivalent to $\boldsymbol{v}_1$.

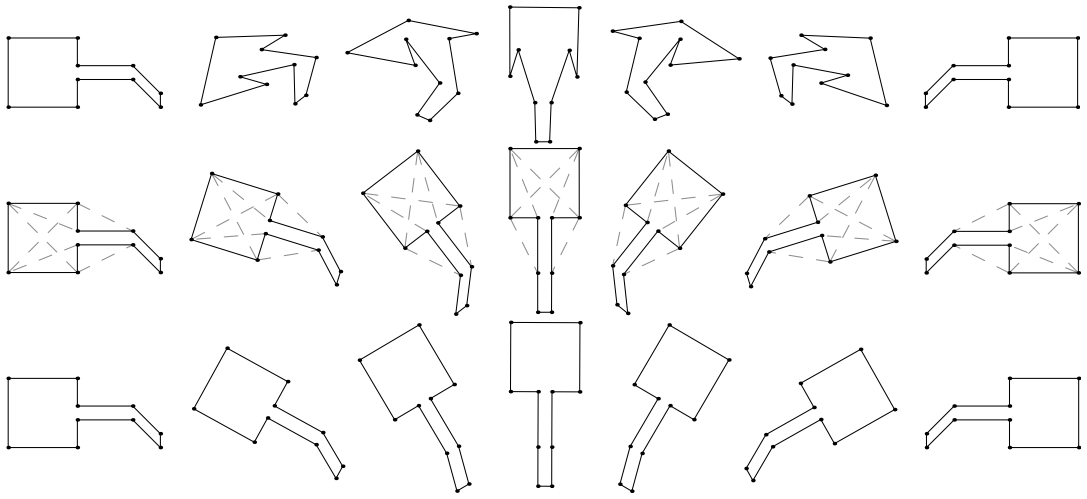

Figure 3: These images show interpolation between a box with an arm-like protrusion and a rotated version of the box with the arm bent. These simple examples demonstrate how the direction metric and constraints can affect the computed sequence. The *first row* shows the result computed using a vertex position metric. The *second row* shows the result for the vertex position metric after several distance constraints have been added. The *bottom row* uses a metric based on joint angles with no constraints. For each row, the edge lengths were held fixed and less than three seconds of computation was required.

At each iteration of the while loop, the higher-energy polygon, H, attempts to move closer to the other, lower-energy one, L. The projection step in (2.d.i) ensures that H does not move up in energy and therefore protects against self intersection. If the direction from H toward L is the same as the upward energy gradient at H,[2] the projection would take D to the null vector. In that case the algorithm simply moves H downward in energy, which we know is always possible from [Cantarella et al., 2004].

If we assume that the heuristic used in step (2.b) to compute D is a reasonable one, we can guarantee that the above algorithm will always converge. By reasonable, we mean that it actually computes a direction that will move H closer (by some measure) to L. Informally, we note that each iteration of the while loop makes either an "approach" move or a "descent" move. The descent moves may undo some of the progress made by approach moves, but the approach moves cannot undo progress made by the descent moves. The algorithm cannot fail to converge by taking an infinite number of descent moves because each decreases the energy toward a minimal value and no moves ever increase the energy. Similarly, the algorithm should not be able to take an infinite number of approach moves unless the user's direction heuristic is broken and does not actually generate moves that bring the polygons closer together. A sequence of an infinite number of interleaved approach and descent moves continually undoing each other cannot occur because the approach moves cannot undo descent progress.

A more rigorous proof that the algorithm converges is too bulky to include here. In addition to guaranteeing that the step directions exist and lead to convergence, one must also deal with issues such as avoiding step sizes that converge toward zero. We refer the reader to [Cantarella et al., 2004] where we present a rigorous proof that the descent steps do converge in bounded time. We also suggest [Dennis and Schnabel, 1996] for a discussion of the conditions under which descent methods generally converge, and to [Atkinson, 1989] or [Press et al., 1994] for a general introduction to relevant numerical methods. For our current implementation we have found it sufficient to use a fixed step size that has been selected conservatively by the user.

[2]Recall that because the gradient of the nonlinear energy function varies over configuration space this situation will occur occasionally.

5.2 A Direction Heuristic

As described above, the interpolation algorithm is designed to work with a user-supplied direction heuristic. Given an initial configuration, S, and a target configuration T, the heuristic should compute a direction, D, that moves S closer to T where closer is defined relative to some distance metric on the space of polygon configurations.

In our implementation, each polygon configuration is represented as a vector of length $2N$ that contains the interleaved x and y coordinates of each vertex. The most obvious direction heuristic is simply $D = T - S$. If we were to use this naïve heuristic alone, the resulting motion would most often include self intersections. However, when embedded in our energy guided algorithm it generates an interpolation sequence free of self intersection.

In Section 6 we show results generated using this simple direction heuristic and with others. The ability to specify an arbitrary direction heuristic affords the user with some aesthetic control over the resulting interpolation sequence, but it could also cause the algorithm to fail. If given the opportunity, the heuristic must cause the two polygons to converge in a finite number of steps. Further, to guarantee convergence, the directions generated by the heuristic should not include any extraneous components or else the energy projection could potentially cancel the useful portion leaving a non-zero vector that might then fail to converge. One way to ensure that the direction will not break is to define a distance metric on the space of configurations and then compute D from the metric's gradient. For example, the $D = T - S$ direction heuristic is the gradient of $\|T - S\|^2$. Alternatively, the direction heuristic could include additional spurious components that do not correspond to the gradient of any distance metric, but the condition in step (2.e) should

then test that the projected vector is not orthogonal to the distance metric's gradient, rather than just testing if it is null.

5.3 Energy Projection

To avoid self intersection, each step must move H to a equal- or lower-energy configuration. This requires that $D \cdot G \leq 0$ where G is the normalized gradient of the repulsive energy function evaluated at H. The algorithm accomplishes this by testing a candidate direction against the gradient direction. If the dot product is less than or equal to zero, then the direction is left alone. Otherwise, the direction is replaced with

$$D := \left(I - G\,G^{\mathsf{T}}\right) D \tag{2}$$

where I is the identity matrix. Because the gradient is not constant, a finite sized step following D may still yield an increase in energy even if $D \cdot G \leq 0$. When this condition occurs, we bias D downward by subtracting γG from the direction where γ is a small positive number.

5.4 Constraints

In addition to specifying vertex correspondences and a direction heuristic, the user can also control the interpolation by specifying constraints that should be satisfied by each polygon in the sequence. One could choose to incorporate user constraints into the direction heuristic, but the energy gradient projection done by Equation (2) would tend to violate the constraints needlessly. Instead, when the user desires constraints we can attempt to satisfy both them and the energy constraint simultaneously. If they cannot all be satisfied simultaneously, then the energy constraint will be satisfied and the user constraints as much as possible. We treat the energy constraint with higher priority because it is what assures convergence.

We assume that each constraint applies to an individual polygon, P, is differentiable, and can be expressed in the form

$$\Omega\left(P\right) = 0 \quad . \tag{3}$$

For example, we could constrain the edge lengths of a polygon to be constant with

$$\|\boldsymbol{v}_i - \boldsymbol{v}_{i+1}\|^2 - l_i^2 = 0 \quad \forall i \in [1 \ldots N] \tag{4}$$

where the $\boldsymbol{v}_i$ and l_i are the vertex positions and edge lengths of P.

If there are M constraints, let $\boldsymbol{J}$ be the $M \times N$ matrix whose rows are the gradient vectors for each of the constraints. If the initial polygons honor the constraints, then in step (2.c) we can project D to a direction that will not violate them with

$$D := D - \boldsymbol{J}^{\mathsf{T}}\,\boldsymbol{l} \tag{5}$$

where $\boldsymbol{l}$ is solved for using

$$\boldsymbol{J}\,\boldsymbol{J}^{\mathsf{T}}\,\boldsymbol{l} = \boldsymbol{J}\,D \quad . \tag{6}$$

In general, a finite step in this direction would still allow any nonlinear constraints to be violated by a small amount, and this error could accumulate to unacceptable levels if not dealt with. If $\boldsymbol{e}$ is the length M vector whose entries are each of the Ω evaluated at H, then we can prevent error accumulation by instead solving for $\boldsymbol{l}$ using

$$\boldsymbol{J}\,\boldsymbol{J}^{\mathsf{T}}\,\boldsymbol{l} = \boldsymbol{J}\,D + \alpha\,\boldsymbol{e} \tag{7}$$

where α is a small constant. (See for example [Baumgarte, 1972] for a discussion of constraint stabilization.)

If the adjusted direction would move upward in energy, it must be adjusted. However, using Equation (2) could break the projection done by Equation (7) because, in general, G will not be orthogonal to all of the constraints (rows of $\boldsymbol{J}$). To avoid violating the constraints needlessly, let $\boldsymbol{K}$ be the matrix formed by appending G as an extra row to $\boldsymbol{J}$ and let $\boldsymbol{f}$ be the vector formed by appending $-\gamma/\alpha$ to $\boldsymbol{e}$. Step (2.d) sets

$$D := D - \boldsymbol{K}^{\mathsf{T}}\,\boldsymbol{l} \tag{8}$$

where $\boldsymbol{l}$ was solved for using

$$\boldsymbol{K}\,\boldsymbol{K}^{\mathsf{T}}\,\boldsymbol{l} = \boldsymbol{K}\,D + \alpha\,\boldsymbol{f} \tag{9}$$

with some small value used for γ. This value is iteratively increased until a downward energy step results.

Both Equations (7) and (9) can be solved efficiently using the conjugate-gradient method. The matrices $\boldsymbol{J}\,\boldsymbol{J}^{\mathsf{T}}$ and $\boldsymbol{K}\,\boldsymbol{K}^{\mathsf{T}}$ may be under-constrained, over-constrained, or both. When the matrix is over-constrained, not all of the constraints can be satisfied and the conjugate-gradient method will produce a solution that satisfies them all equally in a least-squares sense. Increasing γ causes the energy constraint to have greater importance until it is satisfied. Figure 3 shows a simple example computed with and without additional constraints.

For the special case where all of the user constraints correspond to edge-length preservation, we know from [Connelly et al., 2003] that Equation (9) will never be over-constrained because expansive unfolding motions will always exist. Thus, for two polygons with the same edge lengths we can always interpolate between them while holding the edge lengths constant. When the polygons have different edge lengths, we can force them to change monotonically by only including the appropriate row of $\boldsymbol{J}$ or $\boldsymbol{K}$ if omitting that row would result in the an edge getting further in length from its target rather than closer. This type of linear-programming approach could also be used to include other semi-algebraic constraints.

6 Results and Discussion

We have implemented our algorithm and used it to create the examples shown in this paper. The accompanying video contains animations corresponding to these examples. Information about the running times and the methods used to create the examples can be found in their respective figure captions. The running times for our C^{++} implementation were measured in CPU seconds on a 3.06 GHz Pentium IV computer with 1 GB of memory.

The rows of images in Figure 3 illustrate the use of different direction heuristics. As can be seen in the top and bottom rows, direction heuristics based on the Cartesian coordinates of the vertices and on joint angle coordinates produce very different results. The middle row shows how the motion can be modified by adding additional constraints.

In Figure 1, the input polygons are symmetric about a central horizontal axis. It is evident that the animation preserves this symmetry throughout the interpolation. Similarly, the input for Figure 7 is symmetric about a central vertical axis. Figures 3 and 4, both demonstrate animations where the input polygons mirror each other and the method creates symmetric sequences.

Our method also enables the user to choose the behavior of the edge lengths during the animation. The video

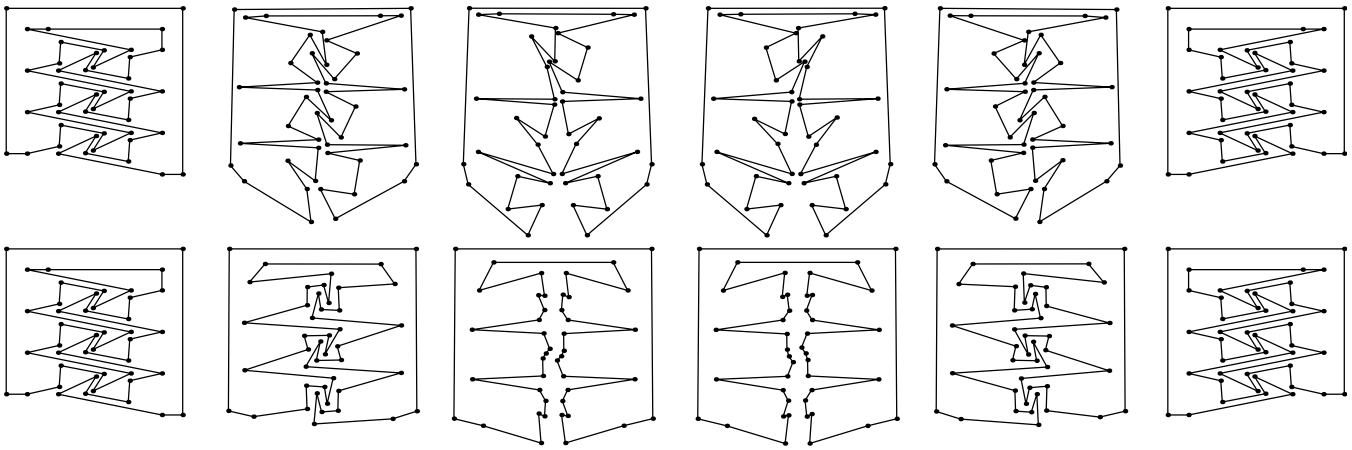

Figure 4: This example interpolates between two configurations of interlocked teeth. The *top row* shows the result computed with the edge lengths constrained to change monotonically and required 12.5 minutes of computation. The *bottom row* shows the result computed with unconstrained edge lengths and required 5.4 minutes of computation.

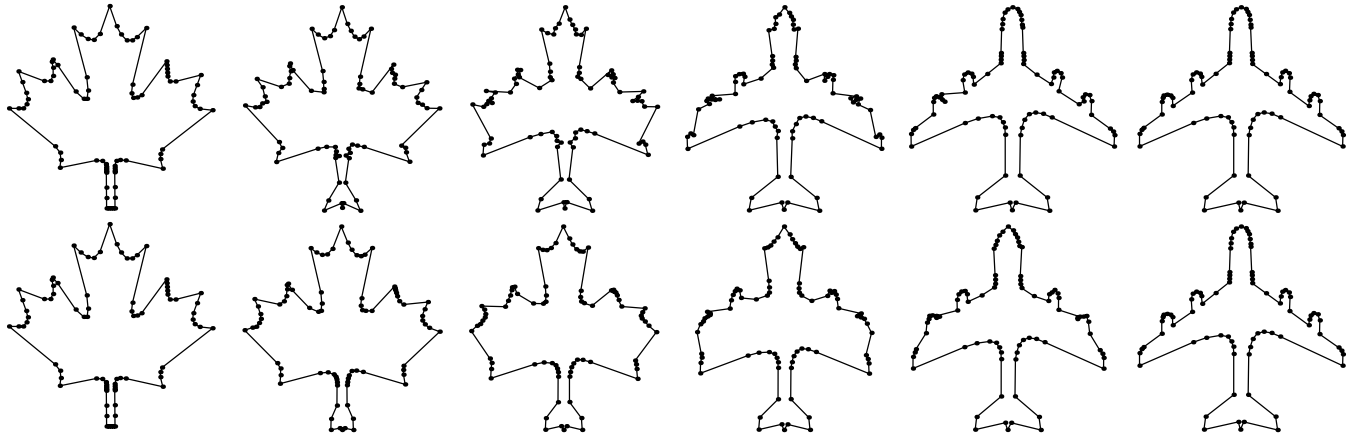

Figure 5: Examples with relaxed energy constraint, see text. In the *top row* constrained edge lengths, 10.1 minutes computation. In the *bottom row* unconstrained edge lengths, 5.7 minutes computation. The leaf and plane outlines were provided by Marc Alexa.

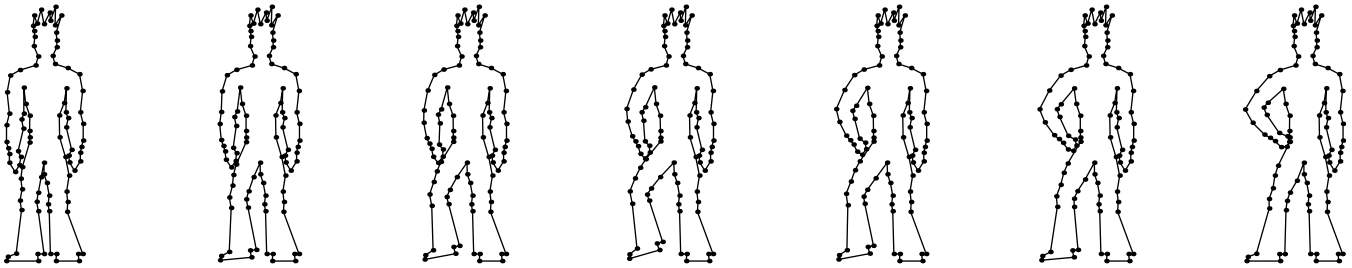

Figure 6: This example was created by generating two successive sequences using three key frames. The keys are shown in the first, center, and last positions. Total computation time was forty-seven seconds.

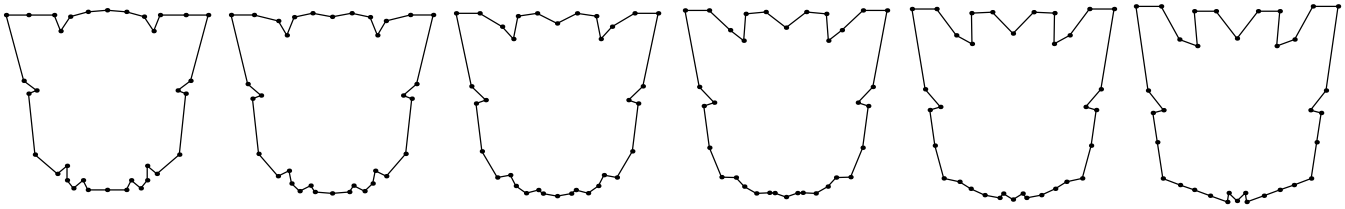

Figure 7: Transforming between two polygons. Unrestricted edge lengths, is less than eight seconds computation.

sequence in Figure 1 shows our method with the constraint that edge lengths are held constant. The examples in Figures 4 and 5 illustrate the difference between constraining the edge lengths to change monotonically (top row) or allowing them to change freely (bottom row). For some examples, constraining the edge lengths generated pleasing results. However, in the leaf-plane example the constraint causes an ugly pinch to form in the leaf-stem/plane-tail. Because a "good" sequence depends on the subjective criteria applied by the user, we feel the flexibility afforded by our approach is highly desirable. Other examples using unconstrained edge lengths are pictured in Figures 6 and 7. In Figure 8, we decided based on aesthetic considerations to morph from T to E with constrained edge lengths while the other letters' animations are unconstrained.

We can also relax the requirement that steps never increase the energy. As an experiment, we allowed the leaf-plane examples in Figure 5 to take steps that increase the energy up to a threshold. This modified algorithm still avoids self-intersection, but it could potentially fail to converge.

One possible problem with our method is that it only uses local information, the energy function gradient, to avoid collisions. As a result, we cannot guarantee that the path generated is optimal in any sense. We can only guarantee that we can find a path. In practice, however the algorithm appears to do a good job finding paths that do not detour needlessly. We have experimented with applying relatively expensive optimization procedures to, for example, shorten a computed path. So far, we have not observed that this effort produces any significant improvements.

The collision avoidance technique presented here provides a method for generating intersection-free interpolation sequences between arbitrary, non-intersecting, planar polygons. We can guarantee that such a path can be found when used with any suitable distance metric. The examples illustrate that our method can handle a variety of polygons and produce pleasing results. Although our C^{++} implementation is robust and fast, using an adaptive time step might improve running times. Other areas for future work include exploring interesting direction heuristics and adding other types of constraints to the system.

Acknowledgments

We thank Jonathan Shewchuk, Jason Cantarella and Carlo Séquin for helpful discussions. Iben was supported by an NSF Fellowship. Iben and O'Brien were supported in part by NSF CCR-0204377, State of California MICRO 02-055, and by generous support from Pixar Animation Studios, Intel Corporation, Sony Computer Entertainment America, and the Okawa Foundation.

References

Alexa, M., Cohen-Or, D., and Levin, D. 2000. As-rigid-as-possible shape interpolation. In *Proceedings of ACM SIGGRAPH 2000*, 157–164.

Atkinson, K. E. 1989. *An introduction to numerical analysis*, second ed. John Wiley & Sons Inc., New York.

Baumgarte, J. 1972. Stabilization of constraints and integrals of motion in dynamical systems. *Computer Methods in Applied Mechanics and Engineering 1*, 1–16.

Cantarella, J. H., Demaine, E. D., Iben, H. N., and O'Brien, J. F. 2004. An energy-driven approach to linkage unfolding. In *The 2004 Symposium on Computational Geometry*. To appear.

Carmel, E., and Cohen-Or, D. 1997. Warp-guided object-space morphing. *The Visual Computer 13*, 465–478.

Cohen-Or, D., Solomovic, A., and Levin, D. 1998. Three-dimensional distance field metamorphosis. *ACM Transactions on Graphics 17*, 2, 116–141.

Connelly, R., Demaine, E. D., and Rote, G. 2003. Straightening polygonal arcs and convexifying polygonal cycles. *Discrete & Computational Geometry 30*, 2 (Sept.), 205–239.

Dennis, J. E., and Schnabel, R. B. 1996. *Numerical Methods for Unconstrained Optimization and Nonlinear Equations*. SIAM, Englewood Cliffs, NJ.

Efrat, A., Har-Peled, S., Guibas, L. J., and Murali, T. M. 2001. Morphing between polylines. In *Proceedings of the twelfth annual ACM-SIAM symposium on discrete algorithms*, 680–689.

Goldstein, E., and Gotsman, C. 1995. Polygon morphing using a multiresolution representation. In *Proceedings of Graphics Interface*, 247–254.

Gotsman, C., and Surazhsky, V. 2001. Guaranteed intersection-free polygon morphing. *Computers and Graphics 25*, 1, 67–75.

Guibas, L., and Hershberger, J. 1994. Morphing simple polygons. In *Proceedings of the tenth annual symposium on computational geometry*, 267–276.

He, T., Wang, S., and Kaufman, A. 1994. Wavelet-based volume morphing. In *Proceedings of Visualization '94*, D. Bergeron and A. Kaufman, Eds., 85–92.

Hughes, J. F. 1992. Scheduled fourier volume morphing. In *Proceedings of ACM SIGGRAPH 1992*, 43–46.

Kaul, A., and Rossignac, J. 1991. Solid-interpolating deformations: Construction and animation of PIPs. In *Proceedings of Eurographics '91*, 493–505.

Press, W. H., Flannery, B. P., Teukolsky, S. A., and Vetterling, W. T. 1994. *Numerical Recipes in C*, second ed. Cambridge University Press.

Sederberg, T. W., and Greenwood, E. 1992. A physically based approach to 2-d shape blending. In *Proceedings of ACM SIGGRAPH 1992*, 25–34.

Sederberg, T. W., Gao, P., Wang, G., and Mu, H. 1993. 2-d shape blending: an intrinsic solution to the vertex path problem. In *Proceedings of ACM SIGGRAPH 1993*, 15–18.

Shapira, M., and Rappoport, A. 1995. Shape blending using the star-skeleton representation. *IEEE Computer Graphics and Applications 15* (Mar.), 44–50.

Streinu, I. 2000. A combinatorial approach to planar non-colliding robot arm motion planning. In *Proceedings of the 41st Annual Symposium on Foundations of Computer Science*, 443–453.

Turk, G., and O'Brien, J. F. 1999. Shape transformation using variational implicit functions. In *Proceedings of ACM SIGGRAPH 1999*, 335–342.

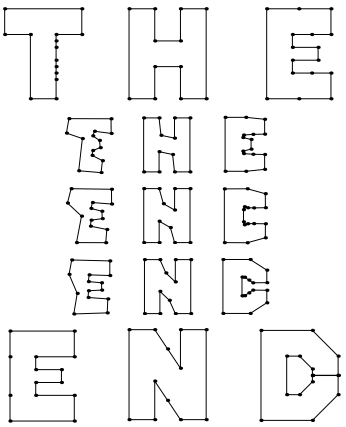

Figure 8: Our final example. Total computation time less than two seconds.