Discrete Comput Geom 24:1–34 (2000)
DOI: 10.1007/s004540010017

Discrete & Computational
Geometry

Morphing Simple Polygons*

L. Guibas,[1] J. Hershberger,[2] and S. Suri[3]

[1]Department of Computer Science, Stanford University,
Stanford, CA 94305, USA
guibas@cs.stanford.edu

[2]Mentor Graphics, 8005 S.W. Boeckman Road,
Wilsonville, OR 97070, USA
john_hershberger@mentor.com

[3]Department of Computer Science, Washington University,
St. Louis, MO 63130, USA
suri@cs.wustl.edu

Abstract. In this paper we investigate the problem of morphing (i.e., continuously deforming) one simple polygon into another. We assume that our two initial polygons have the same number of sides n, and that corresponding sides are parallel. We show that a morph is always possible through an interpolating simple polygon whose n sides vary but stay parallel to those of the two original ones. If we consider a uniform scaling or translation of part of the polygon as an atomic morphing step, then we show that $O(n \log n)$ such steps are sufficient for the morph. Furthermore, the sequence of steps can be computed in $O(n \log n)$ time.

1. Introduction

In computer graphics a recent area of interest has been *morphing*, i.e., continuously transforming one shape into another. Morphing algorithms have numerous uses in shape interpolation, animation, video compression, as well as obvious entertainment value. Some of the relevant graphics papers are [6], [8], and [7]. In general there are numerous ways to interpolate between two shapes and little has been said about criteria for comparing the quality of different morphs and notions of optimality. A common problem with many morphing algorithms is that although locally the geometry of the interpolated shape remains faithful to the original shapes, global properties such as symmetry,

* This research was initiated while the second author was visiting at Stanford University.

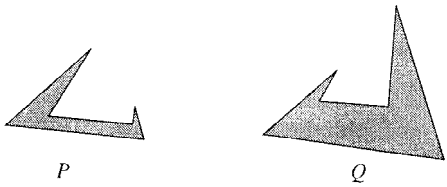

Fig. 1. Two parallel polygons.

simplicity (i.e., freedom from self-intersection), etc., can be easily violated. In fact, the very existence of morphs that can maintain high-level global properties of the interpolated shapes is often far from obvious.

In this paper we address a two-dimensional morphing problem posed by John F. Hughes. We are given two simple polygons P and Q, which are assumed to be oriented counterclockwise. The polygons P and Q are *parallel*, where this is taken to mean that P and Q have the same number of sides and their sides can be put in one to one correspondence in cyclic order so that corresponding sides are parallel and oriented the same way. This is equivalent to saying that the two polygons have the same sequence of angles. The morphing problem is to interpolate between P and Q via a continuously changing polygon $R(t)$, where t denotes time. We set $R(0) = P$ and $R(1) = Q$. We require that $R(t)$ remains parallel to P and Q at all times, and that it also remains simple. See Fig. 1.

We provide a discrete solution to Hughes's problem. We take as our basic operation the uniform scaling and/or translation of a contiguous part of a polygon parallel to itself. We call such a step a *parallel move* of the corresponding polygonal chain. The affected edges in a parallel move expand or contract so as to keep the two endpoints of the chain on the lines supporting the two adjacent edges to the chain. Also, no parallel move is allowed that would cause any moving edge or vertex to pass over another vertex or edge of the polygon. It is clear that any number of parallel moves keep a simple polygon parallel to itself and simple. Our main result is that if n denotes the number of sides of P and Q, then $O(n \log n)$ parallel moves always suffice to morph P to Q. We present an algorithm of the same complexity that computes such a sequence of moves.

The problem of constructing a simple polygon given a sequence of angles has a substantial history in computational geometry. Culberson and Rawlins [1] first raised this question and gave an algorithm that constructs such a polygon in time $O(nD)$, where the assumption is that all angles are rational multiples of π by a fraction whose denominator is bounded by D. Vijayan and Wigderson improved this result by constructing such a polygon in linear time with no restriction on the angles being rational (reported in [11]). The result of the present paper paper shows that all simple polygons realizing the same sequence of angles are homotopic to each other within this class of polygons. This in itself is an interesting result which, according to Hughes, had been in the folklore but never formally proven. Some of the trickiness involved can be seen in the two polygons of Fig. 2. These are oppositely spiraling polygons, but they are still parallel and therefore morphable.

The main idea of this paper is to use parallel moves to transform each polygon to a reduced form in which the side lengths are of widely different scales. Once both polygons

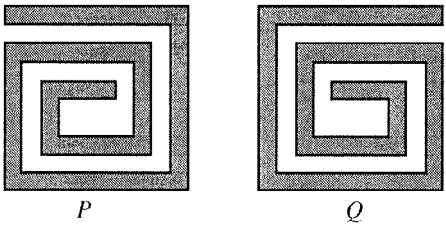

Fig. 2. These oppositely spiraling polygons are still morphable.

are in reduced form, we can use a sequence of elementary local transformations to morph one reduced form to the other. In a reduced form, small-scale structures do not interfere with larger-scale parallel moves—when a large-scale edge translates, any smaller-scale structures adjacent to its endpoints translate along as if they were part of the endpoints. We establish a correspondence between these transformations in reduced polygons and (classical) rotations in binary trees. The catch, however, is that our trees have weights on the leaves and induced weights on the internal nodes. These weights correspond roughly to the polygon angles. Our mapping from polygons to trees is such that all weights of leaves or internal nodes must be in the range $(-1, +1)$. Although the problem of using single rotations to transform one binary tree to another is well studied and is known to be solvable in a linear number of rotations, these methods are not applicable to our situation, as not all rotations maintain the weight condition on the trees. The combinatorial essence of our morphing algorithm becomes then a theorem about rotating one weighted binary tree into another while satisfying the weight conditions. This in itself is an interesting combinatorial problem.

In our setting, the leaves of a binary tree have arbitrary real weights in the open interval $(-1, +1)$. The weight of an internal node is the sum of the leaf weights in its subtree. An internal node is *valid* if its weight is in the interval $(-1, +1)$. A tree is valid if all its nodes are valid. A rotation is valid if it transforms one valid tree into another.

Given two valid n-leaf trees with the *same* weight sequence at the leaves, but different internal structure, we show how to transform one tree into the other using only valid rotations. Our algorithm requires only $O(n \log n)$ rotations; we can also compute these rotations in the same time bound.

We note that for rectilinear polygons the morphing problem is easier, and a solution is already known [10]. We also remark that our transformation to a reduced form is unattractive from a practical point of view, as it may change wildly the scale of different features of the original polygons. We discuss at the end of the paper some concrete and interesting algorithmic issues that arise in keeping small the overall distortion of the original polygons during the morph process.

2. Reduced Forms of Polygons

Our morphing algorithm works by transforming each of the two polygons P and Q into a reduced form. We first transform P to its reduced form, then morph the reduced form

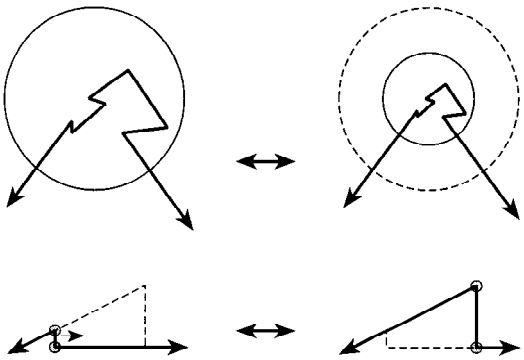

Fig. 3. The two kinds of parallel moves.

of P to that of Q, and then undo the transformations that reduced Q. We will be able to map the morphing of the reduced forms to the purely combinatorial tree rotating problem mentioned above.

A *reduced form* of polygon P is a polygon parallel to P with a particular hierarchical structure. A reduced form of P can be obtained from P by $O(n)$ elementary parallel moves. Reduced forms are not unique, but instead depend on the order in which the parallel moves are performed. In what follows, when we refer to "the reduced form," we mean the reduced form produced by a particular reduction process.

We allow two types of parallel moves, defined as follows: (1) Suppose there exists a circle that intersects exactly two edges of the polygon. Then we may shrink or expand the structure inside the circle uniformly, as long as the intersections of the polygon with the circle do not change. (2) Suppose there exist two disjoint circles, each intersecting an edge e of the polygon and one other edge. Then we may translate e and move the two circles and their contents uniformly along with e, such that each circle's intersections with the polygon are stationary in the reference frame of the circle. In both cases we enforce the condition that the polygon always intersects each circle in exactly two fixed places. See Fig. 3.

A reduced form of P is based on a process of eliminating P's edges one by one. Consider an edge e of polygon P whose endpoints u and v are shared with adjacent edges e_u and e_v. If we translate e parallel to itself while keeping the rest of the polygon fixed, u and v move along the lines supporting e_u and e_v, lengthening or shortening those edges as well as e itself. Suppose that at least one of e, e_u, and e_v gets shorter as e translates. Then one of the edges will eventually be reduced to zero length, if e does not hit another part of P first. See Fig. 4.

We show in Lemma 2.1 below that every polygon has at least one edge translation that reduces an edge to zero length, while keeping the polygon simple. Hence we can iteratively reduce the polygon to a triangle: at each of $n-3$ steps, we eliminate one edge of P by translation. For certain degenerate polygons, it is possible that the elimination process will terminate with a parallelogram instead of a triangle. This case does not pose any additional difficulties for our algorithm, and so we ignore this possibility from here on.

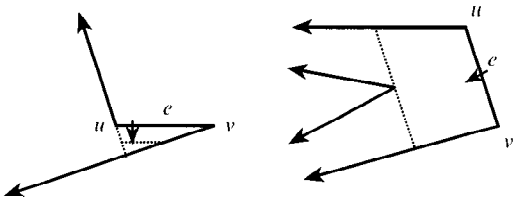

Fig. 4. Translating an edge.

The reduced form of a polygon models the edge elimination process in its structure. To compute it, we run the edge elimination process as above, except that instead of reducing edges to zero length, we reduce them to infinitesimal length. Suppose that the edge elimination process eliminates an edge e, coalescing its vertices u and v to become the common endpoint of adjacent edges e_u and e_v. In the reduced form of the polygon, vertices u and v are kept separate, joined by an infinitesimal edge e. We imagine drawing an infinitesimal disk that contains u and v, centered on the intersection of the lines supporting e_u and e_v. (If the supporting lines are parallel, we drop the centering requirement.) In the remainder of the elimination process, the disk translates along with the intersection of the supporting lines, acting like a finite-size vertex. Because the disk and its contents are infinitesimal, we are sure that no translating edge touches the disk unless it also touches the corresponding vertex in the original edge elimination process. The infinitesimal edge is hidden inside the disk and does not participate directly in the remainder of the elimination process. See Fig. 5.

When the edge elimination process terminates, the reduced form of the polygon consists of a triangle with some *microstructure* at its vertices. Each vertex is contained in a disk, and the three disks are disjoint. If we look inside a disk, we see two *infinite* edges—the ones that cross the disk boundary—and a single *finite* edge joining them. The vertices where the finite edge joins the infinite edges have microstructure themselves: they are contained inside smaller disks that are disjoint from each other and completely inside the surrounding disk; each smaller disk contains recursively similar structures.

If we ignore the disks of the reduced form and look only at the edges, we see that the reduced form of a polygon P is a special hierarchical morph of P: the reduced form is parallel to P, and it is obtained from P by parallel moves that preserve simplicity.

The following lemma shows that at least one edge reduction is always possible.

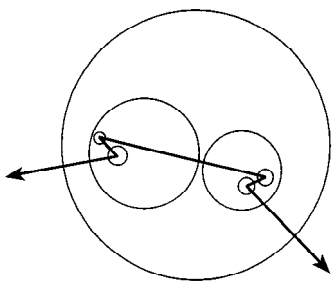

Fig. 5. A reduced form.

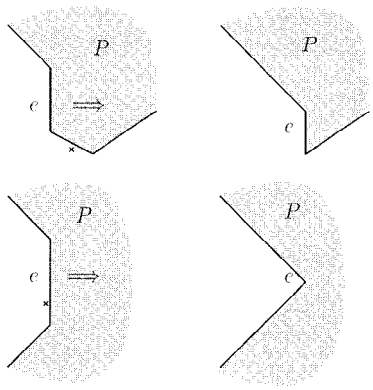

Fig. 6. Parallel translation for a good edge e. The x indicates the side that is reduced to zero length.

Lemma 2.1. *Let P be a simple polygon of n sides. Then there always exists a side e of P so that a parallel move of e toward the interior of P will reduce to zero length either e itself or one of its adjoining two sides.*

Proof. We call an edge e of P *good* if its parallel move eliminates an edge, and *bad* otherwise. In Figs. 6–10 the translating edge e will always be drawn vertical and the interior of P will be to its right. Thus, if e is a good edge, its parallel move terminates in one of two cases, as depicted in Fig. 6.

If edge e is bad, its parallel move will be obstructed by a nonneighboring vertex v or side s of P. These cases are depicted in Fig. 7. In either case, let v denote the first point of contact between the translating edge e and the rest of the boundary of P. Note that, in both cases, the initial position of e and the point v define a triangle Δ_e into which the boundary of the polygon P does not enter.

We want to argue that P always has a good edge. We will show this by associating with a bad edge e a positive real number $\xi(e)$, which we shall term the *cost* of e. We define $\xi(e) = 0$ for a good edge. We will show that if e is a bad edge, then there is another edge f in P with $\xi(e) > \xi(f) \geq 0$. This will imply that if e is the (an) edge of minimum cost in P, then $\xi(e) = 0$ and e must be good—for otherwise there would have to be another edge of strictly lower cost.

For a bad edge e we define its cost $\xi(e)$ as follows: let d be the line segment of shortest length between the stopping point v and e. It is clear that d is contained in Δ_e and that d divides P into two subpolygons, P_L (the one in our figures lying below d, and in general

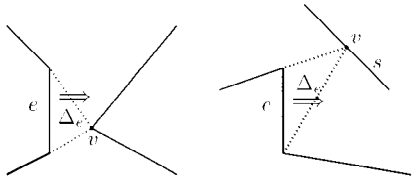

Fig. 7. The translation of a bad edge e is stopped at v.

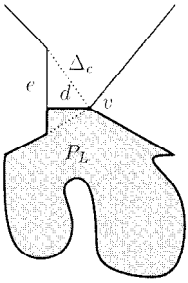

Fig. 8. The definition of the cost $\xi(e)$ of an edge e.

the one lying to the left of d viewed as oriented from v to e) and P_U. We define $\xi(e) > 0$ to be the length of the perimeter of P_L, including d itself. See Fig. 8.

To prove our claim we consider the successor edge f to e as we proceed around P (and P_L) counterclockwise. Let g be the successor edge to f. We distinguish three cases, according to the angles between e, f, and g:

Case A. Suppose e and f make a reflex angle, as in Fig. 9. Now we translate f parallel to itself toward the interior of P. Either f will prove to be good, or it will get stopped at another point w, which must be a point of P_L, as the whole area swept out by the parallel move of f is in P_L. The shortest segment d' joining w to f partitions P_L and generates another subpolygon P'_L, which clearly has a strictly smaller perimeter than P_L. Thus the edge f has the property that $\xi(e) > \xi(f) \geq 0$.

*Case B*1. Suppose e and f make a convex angle, but the angle from f to g is reflex. Then we translate g toward the interior of P. Again, either g will prove to be good, or it will hit an obstructing point w which must be in P_L, because again the entire area swept out by the parallel translation is contained in P_L—and the argument continues exactly as above to show that $\xi(e) > \xi(g) \geq 0$. See Fig. 10.

*Case B*2. If both the angles from e to f and from f to g are convex, we choose to translate f toward the interior of P. Let u be the upper endpoint of e. The crucial observation is that as f translates, it will either prove to be good, or it will get stopped by a point of P_L, *before it has a chance to enter* P_U. See Fig. 10.

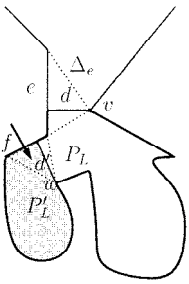

Fig. 9. When the e to f angle is reflex.

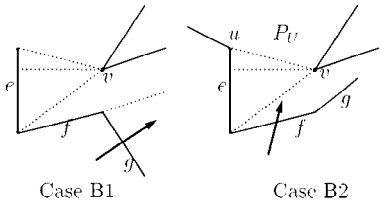

Fig. 10. When the e to f angle is convex.

Note that in cases A and B1, it was clearly impossible for the translating edge to enter P_U. In the current case, because the interior of Δ_e is free of any portions of the polygon boundary, vertices u and v will stop the translating edge f before it can hit any vertex or edge of P_U. If f reaches u, then it is good; if it is stopped by v, then we are in the usual case where the parallel move is stopped by a part of the boundary of P_L. In all cases we can conclude that $\xi(e) > \xi(f) \geq 0$. □

The preceding reducibility lemma can be extended to an algorithm that computes a reduced form of a polygon in $O(n \log^2 n)$ time.

Lemma 2.2. *Let P be a simple polygon with n sides. A reduced form of P can be computed in $O(n \log^2 n)$ time. The reduction uses at most $n - 3$ edge translations.*

Proof. The algorithm uses a query data structure that maintains the current polygon and supports queries of the form "Does the translation of a query edge e toward the polygon interior eliminate some edge?" When a *good* edge is found (one whose translation does eliminate an edge), the translation is performed and the query data structure is updated. Each query or update takes $O(\log^2 n)$ time. There are $n - 3$ edge translations, and we show that there are $O(n)$ queries, so the total time of the algorithm is $O(n \log^2 n)$.

As the algorithm runs, it maintains a set of contiguous edges $B = \{i, i+1, \ldots, j-1, j\}$ of the current polygon that are known to be bad (i.e., their inward translation hits the polygon boundary before any edge is eliminated). The algorithm executes a single basic step $O(n)$ times. Each step tests edge $j+1$ (the next edge after B) for reducibility. If edge $j+1$ is bad, the algorithm sets $B = B \cup \{j+1\}$ and goes on to the next step. If edge $j+1$ is good, the algorithm translates it to eliminate an edge (either j, $j+1$, or $j+2$). The algorithm resets $B = B \backslash \{j-1, \ldots, j+3\}$, removing all edges that are modified or whose neighbors are modified by the translation of edge $j+1$, and goes on to the next step. (Note that with a wraparound of polygon indices, i and $i+1$ may be removed from B, as well as $j-1$ and j.) At most four edges are removed from B, and the remaining edges are all bad: for each $k \in B$, edges $k-1$, k, and $k+1$ are unmodified, and because the translation of edge $j+1$ shrinks the interior of P, edge k's translation is still stopped by some point of P not on edges $k-1$, k, or $k+1$.

Because Lemma 2.1 guarantees the existence of a good edge, this algorithm always succeeds in reducing P to a triangle. The final size of B is at least ⟨number of bad edges found⟩ $-$ 4⟨number of good edges found⟩; the final size must be zero, so the number of bad edges found is at most $4n$. Hence the algorithm performs at most $5n$ queries.

Queries are implemented using the dynamic geodesic triangulation of Goodrich and Tamassia [3]. A geodesic triangulation is a partition of the polygon interior into $O(n)$ "geodesic triangles" whose vertices are also polygon vertices. A geodesic triangle is a simple polygon with exactly three convex vertices and three reflex chains joining them. For any polygon edge e, let A_e be the trapezoidal region that e sweeps over when it translates until an edge is eliminated or e contacts a nonneighboring part of the polygon boundary. Because any line segment inside P intersects $O(\log n)$ geodesic triangles [3], A_e also intersects $O(\log n)$ geodesic triangles.

Using the data structure of Goodrich and Tamassia, we can detect in $O(\log^2 n)$ time whether an edge e is good or bad, spending $O(\log n)$ time per geodesic triangle intersected. For each geodesic triangle τ adjacent to e, we find the first contact between the translating edge e and the boundary of τ in $O(\log n)$ time by binary search. Similarly, we find $O(1)$ adjacent geodesic triangles into which e can translate without hitting any vertex or polygon edge on the boundary of τ, and explore them recursively. Trapezoid A_e is determined either by the endpoints of e's adjacent edges (when e is good) or by one of the $O(\log n)$ geodesic triangle contact points (when e is bad).

The dynamic data structure of Goodrich and Tamassia can maintain geodesic triangulations in a set of simple polygons as the polygons are modified by slicing along straight cuts and gluing adjacent polygons along a common edge. If the total number of polygon vertices is n, each slice or gluing takes $O(\log^2 n)$ time. These operations suffice to maintain a geodesic triangulation as P is reduced: at each reduction step we slice P along the cuts supporting the edges of A_e, then glue together the fragments that constitute the reduced polygon $P \setminus A_e$. □

The following lemma, due to Welzl, gives an alternative proof that $O(n)$ edge translations suffice to reduce a polygon to a triangle. Although the number of edge translations required is not optimal, as in Lemma 2.1, the reduction algorithm can be implemented to run in $O(n \log n)$ time.

Lemma 2.3 [12]. *Given a simple polygon P with n vertices, we can reduce it to a triangle by a sequence of $O(n)$ edge translations, each of which preserves simplicity and $n-3$ of which shorten some edge to zero length.*

Proof. The reduction algorithm repeatedly translates edges toward the interior of the polygon. It maintains a set of *frozen* edges that have already been translated and may not be translated again. It works in a subpolygon of P, and maintains the invariant that at most two (adjacent) edges of the current subpolygon are frozen.

The generic step of the algorithm picks an arbitrary unfrozen edge e of the current polygon (initially P) and translates it toward the interior of P. As the edge translates parallel to itself, its endpoints move along the lines supporting the two edges adjacent to e, and those two edges shorten or lengthen. The translation stops when one of two things happens: (1) e or one of its two neighboring edges is shortened to zero length, or (2) e collides with some vertex v of P, or a translating endpoint v of e collides with some other edge of P. See Fig. 11(a).

In case (1) we have eliminated an edge. In case (2) the collision between an edge and the vertex v has partitioned the polygon in two, and we recursively apply the same

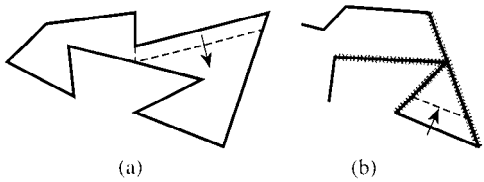

Fig. 11. Eliminating edges to find a reduced form.

procedure on one of the two pieces. Specifically, we translate e away from the polygon interior slightly, so the polygon is not completely partitioned by the collision at v (the amount to back away is upper-bounded by a constant that can be computed from the geometry of the polygon), and freeze all three edges participating in the collision: the two edges incident to v, and the edge v hit. Because at most two adjacent edges of the current polygon were frozen, the same is true of at least one of the two subpolygons created by the near-partition at v. We recurse on the subpolygon with fewer frozen edges. Each of the subpolygons is at least a triangle, so each also has fewer edges than the current polygon.

The recursion bottoms out, in the worst case, when the current polygon is a triangle with two frozen edges (Fig. 11(b)). Translating the third edge inward eliminates one of the frozen edges. When a frozen edge is eliminated, it contracts into the vertex v that caused its freezing. We unfreeze the other two edges that were frozen because of v. This restores the invariant: the collision at v originally created two subpolygons P_1 and P_2, each with at least three edges, with one edge belonging to both P_1 and P_2. The edge elimination reduces the total number of edges in P_1 and P_2 by one or two, to a minimum of three. The new subpolygon is the union of P_1 and P_2; it has at least three edges, at most two of which are frozen.

The size of the current subpolygon decreases at each recursive call, and the algorithm eliminates an edge when the current subpolygon is a triangle. Thus the algorithm successfully eliminates all the edges of P (or reduces P to a triangle, if we stop just before the last elimination). Edges are unfrozen only when we move up one recursive level. Each unfreezing can be charged to the edge elimination that occurred just before it. The total number of edge translations is bounded by n plus the number of unfreezings (at most $2n$) and hence is $O(n)$. □

Lemma 2.4. *Let P be a simple polygon with n sides. In $O(n \log n)$ time we can compute a sequence of $O(n)$ simplicity-preserving edge translations that reduce P to a triangle.*

Proof. The procedure of Lemma 2.3 does not say how to determine whether an edge translation eliminates an edge, so no running time can be associated with it. To establish a running time for the procedure, we specify the query data structure and the queries to be performed on it.

We maintain the additional invariant that, in the generic algorithm step, at most three edges of the current polygon are not in their original positions; the other edges have exactly the same position and length as in the original polygon. This means that we can check whether an edge translation hits the polygon boundary before eliminating an edge

by performing two queries, one on the original polygon P and one on a collection of at most three modified edges. The latter query can be done by brute force; the former query requires more sophistication.

To check whether an edge e is good, we compute the translation trapezoid A_e introduced in the proof of Lemma 2.2. That lemma uses a dynamic data structure with $O(\log^2 n)$ query time. In the present case, we query only the original polygon P, so we can use a static data structure with better performance.

We use a Steiner triangulation of P with the property that any trapezoid contained in the interior of P intersects $O(\log n)$ triangles [4]. This triangulation can be computed in linear time. By looking at $O(\log n)$ triangles in the neighborhood of an edge e, we can determine whether the translation of e eliminates an edge, and, if not, find the first collision of the edge with the boundary. The search is similar to the one performed on geodesic triangles in Lemma 2.2, but simpler, because no binary search is required.

To maintain the invariant that at most three edges of the algorithm's current polygon are not in their original positions, whenever possible we translate edges that have already been modified, preferably ones whose neighbors have been modified. We argue that there are always at most three contiguous modified edges, and the middle edge is not frozen. To begin, we pick one edge and translate it until it is eliminated or frozen (several neighbors may be eliminated before the edge is eliminated itself). If the edge is eliminated, we pick one of the modified neighbors and continue. During this process, only the current translating edge and its neighbors are not as they were in the original polygon. If the translating edge collides with the polygon boundary and becomes frozen, we pick the neighbor of the modified, frozen edge in the new current polygon and translate it. In this case there are three modified edges: the frozen edge, the translating edge, and its unfrozen neighbor. A short case analysis shows that whenever a subpolygon is eliminated and some edges are unfrozen, the new current polygon has at most three contiguous modified edges, with at least the middle one not frozen.

Using the Steiner triangulation for translation queries on the original polygon, and brute force on the modified edges, we can perform each translation query in $O(\log n)$ time. The procedure of Lemma 2.3 performs $O(n)$ queries, and all the other algorithmic bookkeeping adds only $O(n)$ work. Hence the complete algorithm takes only $O(n \log n)$ time. □

The key feature of a reduced form is the hierarchy of edge eliminations, as the following lemma shows.

Lemma 2.5. *If two reduced forms are parallel polygons and have the same hierarchical structure, then one can be morphed to the other using a linear number of edge translations.*

Proof. Let P and Q be the two reduced forms. The algorithm follows the hierarchy of the reduced form. We repeatedly translate edges of P to be collinear with the corresponding edges of Q. When we are done, the two polygons are identical. The first step translates the top-level triangle edges of P to be collinear with those of Q, moving the microstructure disks along with the intersections of the edges. Within each microstructure disk, we translate the finite edge of P to be collinear with the corresponding edge

of Q. (The subdisks translate along with the intersection points of the finite and infinite edges.) We then recursively make the two polygons identical inside each subdisk. □

3. The Tree Representation of a Reduced Form

The hierarchical nature of a reduced form can be represented by a tree structure. The infinitesimal disks of the reduced form correspond to nodes in a binary tree; a disk and the two subdisks it contains correspond to a node and its children. Each of the three top-level disks of the reduced form corresponds to a tree. Logically these three trees should be joined in a ring; however, to maintain the convenience of a consistent binary tree representation, we omit the ring edges and work with the trees separately. Each tree represents the reduced form of a simple bi-infinite chain (one whose initial and final edges are rays).

In this section through to Section 8 we work only with chains and their corresponding trees; we show how to morph two parallel chains to each other. In Section 9 we address the original problem of morphing polygons.

Each node of a tree has an angular weight associated with it, namely, the turn angle at the corresponding disk, i.e., the difference between the direction angles of the incoming and outgoing edges of the disk (we assume the polygon is oriented counterclockwise). For convenience we normalize the weight of each vertex of P by dividing it by π, so the weight is in the range $(-1, +1)$; the sum of the weights over all the vertices is therefore 2. The weight at each disk of the reduced form is the sum of the weights of all the polygon vertices inside it.

Because the edges incident to each vertex of P and each disk of the reduced form make an angle strictly between $-\pi$ and π, the weight of each node in the binary tree is in the open range $(-1, +1)$. We say that a node whose weight is in this range is *valid*. If all the weights in a tree are valid, we say the whole tree is valid. The binary tree associated with the reduced form of a bi-infinite chain is always valid. The following lemma shows that the converse is also true.

Lemma 3.1. *Any valid tree corresponds to a reduced form of some bi-infinite chain.*

Proof. The proof is by recursive expansion. Let r be the root of the tree. The weight of r corresponds to the total turn of the chain. We draw the infinite chain edges and draw a circle $\mathcal{C}$ centered on the intersection of their supporting lines. Let α and β be the weights of the children of r. Thus $(\alpha + \beta)\pi$ is the angle between the two infinite edges. Within the circle we draw an edge that makes angles of $\alpha\pi$ and $\beta\pi$ with the two infinite edges, and two smaller circles inside $\mathcal{C}$ centered on the endpoints of the new edge. Applying this process recursively with each subtree and its corresponding circle produces a reduced form for the tree. (At each step the drawing inside the innermost circle is provisional, to be replaced when the angle inside the circle is expanded.)

There is one complication that arises when a node has weight zero. In this case the two infinite edges are collinear, and the process just described does not give us the flexibility to separate the disks corresponding to the two child nodes. To overcome this difficulty, we build in flexibility at the parent node: whenever one child of a node has weight zero,

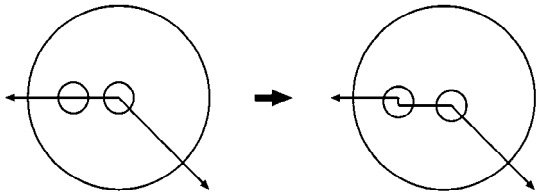

Fig. 12. Introducing a jog inside a zero-weight disk.

we make the two subdisks contained in $\mathcal{C}$ small enough that we can introduce a jog inside the zero-weight disk and compensate for it by moving the finite edge inside $\mathcal{C}$. See Fig. 12. □

In the remainder of this section we reduce the problem of morphing between two parallel bi-infinite chains to a problem of transforming one valid tree into another by rotations. Because we want the tree transformation to correspond to polygon morphing, we insist that the tree be valid at all times. In particular, the transformation may use only *valid rotations*, ones whose initial and final states are both valid. For example, in Fig. 13, if A, B, and C represent node weights, it must be the case that each of A, B, C, $A + B$, $B + C$, and $A + B + C$ lies in the range $(-1, 1)$.

Let T_1 and T_2 be two valid trees with the same sequence of weights at the leaves. Sections 4–8 show how to transform T_1 to T_2 using only valid rotations. We now show how to interpret a tree rotation as a morphing operation that transforms one reduced form into another. Let A, B, and C be the tree nodes involved in the rotation, grouped $((A\,B)\,C)$ before and $(A\,(B\,C))$ after the rotation. The tree structure $((A\,B)\,C)$ corresponds to a disk of the reduced form in which the edges before A and after C are infinite, the edge $\overline{BC}$ is finite, and the edge $\overline{AB}$ is infinitesimal, contained in a subdisk. Of course, A, B, and C are vertices only at this level of the reduced form; they may have microstructure of their own. See Fig. 13. We want to morph $\overline{AB}$, $\overline{BC}$, and the two infinite edges such that $\overline{AB}$ becomes finite and $\overline{BC}$ infinitesimal.

Lemma 3.2. *Let A, B, and C be the turn angles of a four-edge bi-infinite chain, and suppose the angles $A + B$, $B + C$, and $A + B + C$ are all valid. Then either of the*

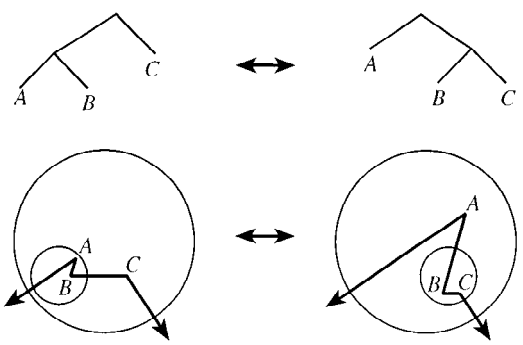

Fig. 13. A tree rotation and the corresponding morph of a reduced form.

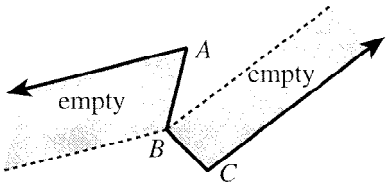

Fig. 14. Each finite edge may be eliminated by translating an infinite edge.

edges $\overline{AB}$ and $\overline{BC}$ can be reduced to zero length by translating the infinite edge adjacent to it.

Proof. Let $\overrightarrow{e_A}$ be the infinite edge incident to A, and define $\overrightarrow{e_B}$ to be the ray parallel to $\overrightarrow{e_A}$ whose origin is B. Let $\overrightarrow{e_C}$ be the infinite edge incident to C. Edge $\overline{AB}$ can be shortened to zero length by translating $\overrightarrow{e_A}$ if and only if $\overline{BC}$ and $\overrightarrow{e_C}$ do not enter the infinite trapezoid bounded by $\overrightarrow{e_A}$, $\overline{AB}$, and $\overrightarrow{e_B}$. (See Fig. 14.) If edge $\overline{BC}$ enters the trapezoid, then angle $A + B$ is invalid; if the infinite edge $\overrightarrow{e_C}$ enters the trapezoid, then angle $A + B + C$ is invalid. By assumption, both angles are valid, so $\overline{AB}$ can be shortened. A similar argument for $\overline{BC}$ finishes the proof. □

The following lemma shows that a reduced form allows us enough flexibility to apply Lemma 3.2.

Lemma 3.3. *In a reduced form, the disk sizes can be chosen so that the infinite edges incident to any disk are free to translate a distance at least equal to the diameter of the disk.*

Proof. The proof is top-down. The base case is easily established: the top-level infinite edges of a chain are unconstrained. (In the top-level triangle of a polygonal reduced form, keeping the disk radius at most one-fourth of the minimum edge length of the triangle establishes the base case.) Within each disk $\mathcal{D}$ of a reduced form, there are two infinite edges f and g, joined by a finite edge e. The endpoints of e are contained in two subdisks concealing microstructure. In the inductive step we assume that f and g are able to translate at least the diameter of $\mathcal{D}$, so they can certainly translate at least $\delta =$ the maximum diameter of a subdisk. Edge e appears infinite from the vantage point of each of its disks. When we translate e relative to the disk defined by (e, f), the other disk translates uniformly along with the intersection of the lines supporting e and g. Therefore, we choose δ small enough that translating e by δ relative to one subdisk does not move the other subdisk out of $\mathcal{D}$. □

To morph $((A\,B)\,C)$ to $(A\,(B\,C))$, we erase the disk around $\overline{AB}$, then translate the infinite edges to lengthen $\overline{AB}$ and shorten $\overline{BC}$, leaving vertex B (and its microstructure) fixed. Once $\overline{AB}$ is of finite length and $\overline{BC}$ is infinitesimally short, we draw a new disk around $\overline{BC}$, leaving A at the top level. During these translations and regroupings, we

may scale the microstructure at A, B, and C uniformly if necessary. (Unless the infinite edges are parallel, we can actually do the rotation without translating the infinite edges at all. If they are parallel and we do have to move them, we make sure the movement is small enough that the condition of Lemma 3.3 is maintained at the other ends of those edges.)

Because a valid rotation on trees can be mapped directly to a constant-complexity morph of one reduced form to another, any algorithm for transforming trees via valid rotations corresponds directly to an algorithm for morphing the reduced forms of two parallel bi-infinite chains.

Theorem 3.4. *Let P and Q be two parallel bi-infinite chains in reduced form, and let T_P and T_Q be the two weighted binary trees representing the reduced forms. If there exists a sequence of k valid rotations that transforms T_P into T_Q, then there is a corresponding sequence of $O(k)$ parallel moves that maintains simplicity and continuously deforms P until it has the same reduced form as Q.*

4. Tree Reconciliation via Rotations—Preliminaries

In this section through to Section 8 we address the purely combinatorial problem of transforming one binary tree into another via a sequence of (single) rotations. If there are no additional constraints, it is well known that $\Theta(n)$ rotations are necessary and sufficient to rotate one tree into another in the worst case [2], [9] (the best-known upper bound is $2n - 6$). However, in our setting there is additional structure to cope with. Our trees have *weights* at their leaves. The internal nodes also have induced weights, namely, the sum of the leaf weights in all leaves belonging to the subtree rooted at that internal node. These weights restrict the possible trees we can have, as well as the allowed rotations between them.

As mentioned earlier, a weighted tree is called *valid* if the weights of all its nodes (leaves, as well as internal nodes) are reals in the range $(-1, +1)$ (open interval). A rotation is called valid if it transforms a valid tree into another valid tree. It is clear that rotations preserve the sequence of leaf weights, but change the grouping structure on these weights imposed by the tree.

Let T_1 and T_2 be two valid n-leaf binary trees whose leaves have the same weight sequence $w_1, \ldots, w_n$. The structures of T_1 and T_2 above the leaves are in general not the same. In particular, the two trees may have vastly different heights, ranging from $\Theta(\log n)$ to $\Theta(n)$. See Fig. 15.

The result mentioned above asserts that one can always go from one tree to the other via a sequence of rotations. Arbitrary rotations, however, can destroy validity by generating internal nodes whose weights are outside the range $(-1, +1)$. So the natural question that arises is whether it is always possible to go from T_1 to T_2 via a sequence of valid rotations and, if so, what is the maximum required length of such a sequence.

Theorem 7.5 shows that the answer to this question is affirmative, and gives an $O(n \log n)$ upper bound on the number of rotations in the sequence. The proof of the theorem is the subject of the next few sections.

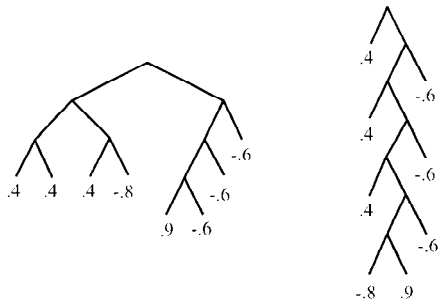

Fig. 15. Two valid trees with the same weight sequence at their leaves.

We refer to tree T_1 as the *source tree* and T_2 as the *target tree*. The following high-level algorithm produces a sequence of valid rotations that transforms T_1 into T_2:

ALGORITHM BASICMORPH

while $|T_1| > 1$ **do**

1. Let w' and y' be a pair of sibling leaves in the target tree T_2, and let w and y be the corresponding leaves in the source tree T_1.
2. Perform valid rotations on T_1 to make w and y into siblings in T_1.
3. Collapse w and y into their parent in T_1, and assign the parent node (a new leaf) the sum of the weights of w and y. Similarly collapse w' and y' into their parent in T_2. This produces two smaller valid trees with the same weight sequence at the leaves.

endwhile

4.1. *Spines and Sibling Operations*

The core of Algorithm BasicMorph is Step 2. We call one execution of this step a *sibling operation*, because it makes two consecutive leaves into siblings. The two consecutive leaves of T_1 that are made into siblings are joined by a path whose structure, shown in Fig. 16, is best described in terms of *spines*.

A spine is a maximal sequence of nodes and edges that form a path in the tree all of whose links go in the same direction (all left or all right). The *length* of a spine is the number of edges on it. The *internal nodes* of a spine are all except its top and bottom nodes. For example, in Fig. 16, the path $[v..w]$ is a *right spine*, and $[z..y]$ is a *left spine*. All the nodes of these spines except v, w, y, and z are internal.

The top of a right spine is either the root of the tree or the left child of its parent. A left spine is symmetric. Each leaf is the bottom of exactly one spine, either left or right. The total length of all the spines in an n-leaf tree is $2n - 2$, since every edge belongs to exactly one spine. We denote by $lca(a, b)$ the lowest common ancestor of two leaves a and b. In Fig. 16, $x = lca(w, y)$.

We base our procedure for making w and y into siblings on a sequence of *spine-contraction* steps, as in the lemma and corollary below.

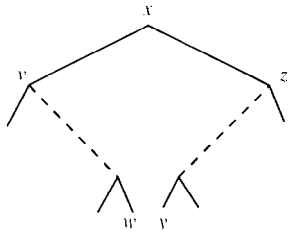

Fig. 16. Two adjacent nonsibling leaves.

Lemma 4.1. *Among any four consecutive internal nodes along a spine in a valid tree, at least one can be rotated out of the spine by a valid single rotation.*

Proof. Without loss of generality assume that the spine is a left spine, like $[z..y]$ in Fig. 16. Let A, B, C, and D be the right children of four consecutive internal nodes on the spine, and let X be A's sibling, as in Fig. 17(left). By abuse of notation let A, B, C, D, and X also denote the weights of the nodes. If any consecutive pair of right children, say (A, B), has a valid sum ($-1 < A + B < 1$), then a valid rotation can be applied at the parent of B. This makes A and B into siblings whose grandparent lies on the spine, and removes one edge from the spine. See Fig. 17(right).

If no consecutive pair has a valid sum, then $|A+B| \geq 1$ and $|C+D| \geq 1$. All four of A, B, C, and D must have the same sign, so $|A+B+C+D| = |A+B|+|C+D| \geq 2$. The weight at the parent of D is $w = X + A + B + C + D$. Because $|X| < 1$, we have $|w| > 1$, so the initial tree was invalid, a contradiction. □

If we partition a spine into consecutive groups of four nodes each and apply the valid rotation guaranteed by the previous lemma to each group independently, we can reduce the length of our spine by one-quarter. By iterating this process until the previous lemma is no longer applicable, we obtain the following spine-contraction corollary. We note that when no rotation can be applied along a spine, all the weights of the spine nodes must be of the same sign.

Corollary 4.2. *Given a spine of length k, we can reduce it to length at most four by $k - 4$ valid rotations at internal nodes. Furthermore, when this procedure terminates, the original children of the spine will have been regrouped into trees attached to the contracted spine, each of height at most $O(\log k)$.*

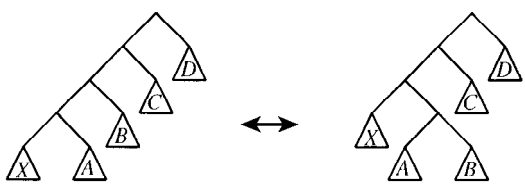

Fig. 17. A rotation may be performed on any spine with more than four nodes.

If we apply a spine contraction to the spines $[v..w]$ and $[z..y]$, we can make the least common ancestor x of leaves w and y be at most five levels above the two leaves. We can then finish as follows.

Lemma 4.3. *If the weights of two adjacent leaves w and y have a valid sum, then w and y can be made siblings by at most eight valid single rotations once the spines $[v..w]$ and $[z..y]$ have been contracted.*

Proof. After the spine-contraction process, we can assume that no rotation involving only internal nodes of spine $[v..w]$ or spine $[z..y]$ is valid. Let $l_1, \ldots, l_k$ denote the weights of the left children of spine $[v..w]$, in left-to-right order, and similarly let $r_1, \ldots, r_j$ denote the weights of the right children of $[z..y]$. Observe that the l's all have the same sign; the same is true for the r's. By abuse of notation, let w and y also denote the weights of the corresponding nodes. Denote the weight at x by $x = l_1 + \cdots + l_k + w + y + r_1 + \cdots + r_j$. We know $|x| < 1$.

If l_1 has the same sign as x, say positive, then $x > x - l_1 > x - 1 \geq -1$, so $x - l_1$ is valid; if x and l_1 are negative, a symmetric argument shows that $x - l_1$ is valid. Similarly, if $\text{sign}(r_j) = \text{sign}(x)$, then $x - r_j$ is valid. If $\text{sign}(l_1) = \text{sign}(r_j) \neq \text{sign}(x)$, then $\text{sign}(x) = \text{sign}(w + y)$, say positive. Then $x < x - l_1 < w + y < 1$, and $x - l_1$ is valid (the middle inequality holds because the l's and r's all have the same sign, and $x - (l_1 + \cdots + l_k + r_1 + \cdots + r_j) = w + y$); symmetric inequalities apply if x is negative. The same methods show that $x - r_j$ is also valid.

Without loss of generality assume $x - l_1$ is valid. Perform a rotation at edge (v, x), moving the right child of v to be the left sibling of z. The new parent of z and the moved node has weight $x - l_1$, which is valid. All the other modified nodes have weights that were present before the rotation, so the rotation is valid. It shortens by one the length of the path between w and y. After the rotation, we can still apply the argument above to show that at least one of the edges incident to the lowest common ancestor of w and y can be rotated. Performing at most eight rotations, we shorten the path length to two, making w and y siblings. □

The spine contracting rotations of Corollary 4.2 affect only internal nodes of the spine, and not the nodes at the tops. However, the rotations of Lemma 4.3 involve the tops of the spines.

The overall rotation cost of making leaves w and y siblings is proportional to the sum of the original lengths of the two spines $[v..w]$ and $[z..y]$. This is two less than the tree distance of w and y. We call the latter quantity simply the *distance* in T_1 between w and y.

Our discussion so far, coupled with our overall tree morphing strategy mentioned earlier, provides a proof that T_1 can be rotated into T_2. In fact, we also get an $O(n^2)$ upper bound on the number of rotations needed, as the distance between any two adjacent leaves is $O(n)$, and the number of sibling operations is at most $n - 1$.

4.2. *Spine Normalization*

In order to improve the $O(n^2)$ tree morphing bound, we must control the distances between adjacent leaves that are being made into siblings. The chief difficulty is that

rotations that reduce one spine length can increase other spine lengths—actually it is very easy to check that any single rotation will decrease the distance between two pairs of leaves by one, and it will increase the distance between two other pairs by one. We can, however, argue that there is a sequence of spine contractions over a whole tree that reduces the maximum spine length.

Lemma 4.4. *Any valid tree on n leaves with maximum spine length k can be rotated using $O(n)$ rotations into another valid tree whose maximum spine length is $O(\log k)$.*

Proof. Apply Corollary 4.2 on the left and right spines of the original tree, reducing them to constant length. Each subtree that was originally a child of the left or right spine is now at depth $O(\log k)$. Let T be such a subtree. Without loss of generality, assume that the parent of T is to its left. Apply Corollary 4.2 on the right spine of T, reducing it to length at most four. (This process does not modify any ancestors of T.) Now the length of the right spine that includes T's right spine (including ancestors) is $O(\log k)$. Recursively apply this procedure to the original child trees of T's right spine, until every node in the tree has participated in one spine-reduction operation.

Spine-reduction does not affect the internal structure of the children of the spine; the cost of spine-reduction is proportional to the length of the spine. Each spine-reduction operation is applied to a spine that has not previously been operated upon. Thus the total work is $O(n)$. By construction, each spine in the final tree consists of at most four nodes that result from the reduction of that spine, plus $O(\log k)$ ancestors that were rotation nodes in a spine reduction higher up in the tree. □

Corollary 4.5. *Any valid tree on n leaves can be rotated using $O(n \log^* n)$ valid rotations into another valid tree whose maximum spine length is eight.*

Proof. Recall that $\log^* n = \min(k \mid \log^{(k)} n \leq 1)$, where $\log^{(k)} n$ is the k-fold composition $\log(\log \cdots (n))$. The function $\log^* n$ grows very slowly: $\log^* n = o(\log^{(k)} n)$ for any constant k. By applying Lemma 4.4 $O(\log^* n)$ times, we reduce the maximum spine length to $O(1)$.

We apply Lemma 4.4 until the maximum spine length does not decrease. To prove that the maximum spine length is at most eight, note that the number of valid rotations at internal nodes that can be applied in parallel to a spine of length k is at least $\lfloor (k-1)/4 \rfloor$. The maximum lengthening of child spines is equal to the number of phases of parallel rotations needed to reduce the spine to length four. We denote this quantity by a function $h(k)$, which satisfies the following recurrence:

$$\begin{aligned} h(4) &= 0, \\ h(k) &= h(k - \lfloor (k-1)/4 \rfloor) + 1, \qquad \text{for} \quad k > 4. \end{aligned}$$

Simple calculation shows that $h(k) \leq k - 4$ for all $k \geq 4$, with strict inequality for $k > 8$. If we apply Lemma 4.4 to a tree with maximum spine length k, the resulting tree has maximum spine length at most $h(k) + 4$. This is less than k if $k > 8$. If the maximum spine length is originally more than eight, applying Lemma 4.4 shortens it;

applying Lemma 4.4 repeatedly shortens it to at most eight. Because $h(k) = O(\log k)$, the number of times that Lemma 4.4 must be applied is $O(\log^* n)$. □

We call the operation of transforming a tree into one whose maximum spine length is eight a *normalization.*

In the next three sections we describe an algorithm that uses only $O(n \log n)$ rotations to morph one tree into another. The number of rotations needed to perform a sibling operation in Step 2 of BasicMorph is the sum of the spine lengths above the two consecutive leaves. To achieve the rotation bound, we focus on the spines that participate in sibling operations. We identify those spines a priori and maintain them at constant length. The cost of a single sibling operation is therefore constant; however, we need $O(\log n)$ rotations to restore the spine length bound after each sibling operation.

5. Node Inclinations and the Inclination Invariant

The pattern of sibling operations (though not the exact sequence) is completely determined by the target tree T_2. Each sibling operation collapses a pair of sibling leaves of T_2 into their parent node, while keeping the rest of the target tree unchanged.

At some time during the run of the algorithm, each leaf of the target tree collapses into its parent. Internal nodes of the target tree become leaves when their children collapse into them, and then they collapse into their parents. For each node, the direction of the edge to its parent is fixed. If a target tree node is a left child, then it will be paired with a leaf to its right when it eventually collapses into its parent. We say that a leaf that is a left child in the target tree is *right-inclined*, because it will merge with a leaf to its right.

Although leaf inclinations are defined on the target tree, they are useful because of what they tell us about the source tree. If a' is a right-inclined leaf in the target tree, then the corresponding leaf a in the source tree *is also right-inclined*: it will merge with a leaf to its right when it participates in a sibling operation.

5.1. *Merge Centers*

Consider the sequence formed by taking the leaf inclinations in left-to-right order. The sequence is composed of alternating runs of right and left inclination. Each alternation from right inclination to left inclination is associated with a pair of sibling leaves in the target tree. Each such pair is a possible site for a sibling operation, and we call the gap between the two such leaves a *merge center*. Because the leaf inclinations are the same in the two trees, the merge centers are also the same.

The number of merge centers is nonincreasing as the algorithm runs: Each sibling operation removes the two leaves adjacent to a merge center, replacing them by a single leaf whose inclination is determined by the target tree. The new leaf may or may not belong to a merge center. No other leaves change inclination. See Fig. 18.

The inclination of nodes in the target tree is determined by that tree's structure, but the inclination of nodes in the source tree is less obvious, since the structure of the source

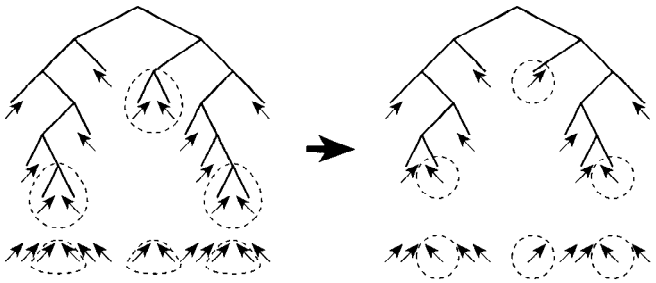

Fig. 18. The target tree: leaf inclinations, merge centers, and the results of performing sibling operations at all the merge centers.

tree differs from that of the target tree. We define the inclination of a source tree node to be the union of the inclinations of its leaf descendants. Thus every node has some inclination, and some are inclined both left and right (such nodes are *both-inclined*; nodes inclined only one way are *singly inclined*). The labels in Fig. 19 show the inclinations of a, b, and $lca(a, b)$.

5.2. *The Invariant*

If leaves a and b participate in a sibling operation, the cost of the operation depends on the lengths of the right spine above a and the left spine above b. Any *left spine* involving a or *right spine* involving b has no effect on the cost of this sibling operation. For this reason, we maintain the following invariant:

Inclination Invariant. If a node in the source tree is right inclined (left inclined), then the portion of the right (left) spine above it to which it belongs (if any) has constant length.

The Inclination Invariant is established initially by the normalization operation (Corollary 4.5), which ensures that all spines have constant length (≤ 8).

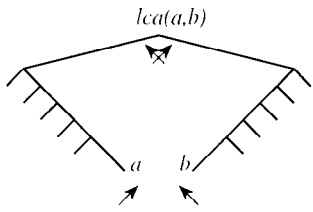

Fig. 19. Leaves a and b in the source tree are ready for a sibling operation.

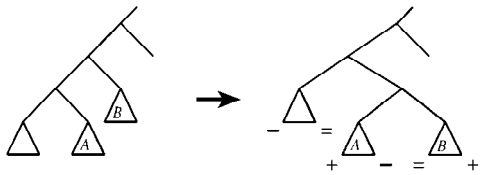

Fig. 20. $+/-/=$ indicate changes in spine lengths due to the rotation.

Lemma 5.1. *If the Inclination Invariant holds, then the two leaves at any merge center are joined by a constant length path in the source tree.*

Proof. The path consists of two spines, which have constant length by the Inclination Invariant, plus two edges linking the spines to the lowest common ancestor of the two leaves. □

Before describing how to restore the Inclination Invariant after a sibling operation, we must understand the effect of rotations on spine lengths. The following lemma summarizes those effects.

Lemma 5.2. *Let subtrees A and B be two children of a spine, consecutive in left-to-right order. If a rotation is applied, making A and B into sibling grandchildren of the spine, then the left spine of A and the right spine of B are lengthened by one. No other spine gets longer.*

Proof. See Fig. 20. □

6. Restoring the Invariant

In this section we describe how to restore the Inclination Invariant after it has been invalidated by rotations. Although this may in general require many rotations, by restricting the invalidating rotations to subtrees containing only a few merge centers, we can apply some of the ideas from balanced tree algorithms to restore the invariant with only $O(\log n)$ rotations.

We choose the constant c implied by the Inclination Invariant to be nine (required by Lemma 6.2 below). We first consider restoring the invariant for a subtree in which all nodes have the same inclination.

Lemma 6.1. *Let $T' \subseteq T_1$ be a subtree of the source tree, and suppose that all the left (right) spines in T' have length at most c, for some constant $c \geq 8$, except for one spine longer than c that is incident to the root of T'. Then, with $O(\log|T'|)$ rotations, we can shorten the long spine by one edge, while keeping all other left (right) spines of T' no longer than c. No spine in $T_1 - T'$ becomes longer.*

Proof. We prove the lemma for left spines; the other case is symmetric. Lemma 4.1 implies that there are at least two possible rotation sites on the spine that do not involve

the top node of the spine. (Because the top node is not involved, neither of the possible rotations lengthens the single spine that extends from T' up into the rest of the tree.) Each rotation takes two children of the spine and makes them into sibling grandchildren of the spine (Lemma 5.2). The two rotations can be chosen to involve two disjoint pairs of children. Let the two pairs be (A, B) and (C, D), each pair ordered left-to-right, and denote by AB and CD the two possible subtrees, children of the spine, produced by the two possible rotations.

We shorten the long spine of T' by performing the rotation that involves the smaller of the subtrees AB and CD. Without loss of generality suppose that $|AB| \leq |CD|$, and so AB is chosen. Because AB and CD are disjoint, it follows that $|AB| \leq \frac{1}{2}|T'|$. The rotation lengthens exactly one left spine: the left spine of AB is one edge longer than the left spine of A before the rotation. The length of the left spine of AB may increase to $c+1$, in which case we recursively apply the same algorithm to shorten it. Because $|AB| \leq \frac{1}{2}|T'|$, the number of rotations needed to reduce the maximum left spine length to c obeys the recurrence $t(m) \leq t(m/2)+1$, where m is the subtree size. This well-known recurrence solves to $t(m) = O(\log m)$, completing the proof. □

We subdivide the leaf-sequence of the source tree into blocks of leaves with the same inclination. The breakpoints between consecutive blocks are called *inclination reversals*. Each inclination reversal where the inclination changes from right to left is a merge center. Thus the total number of inclination reversals in any subsequence of the leaves containing k merge centers is at most $2k+1$.

Lemma 6.1 is not directly applicable to the general case of sibling operations, because we cannot limit invariant violations to subtrees with only a single inclination. (In such singly inclined subtrees, the Inclination Invariant requires a length bound on spines of only one orientation, and Lemma 6.1 applies.) To remedy this problem, we prove in the next section that we can always restrict the invariant violations to subtrees containing only a constant number of inclination reversals. Therefore, the following lemma lets us restore the invariant with only $O(\log n)$ rotations in the general case as well.

Lemma 6.2. *Let $T' \subseteq T_1$ be a subtree of the source tree in which the Inclination Invariant holds (with constant $c \geq 9$), except for one left (right) spine of length greater than c incident to the root of T'. Suppose that there are at most k inclination reversals among the leaves in the left (right) subtree of T'. Then, using $O(\sqrt{k}\log|T'|)$ valid rotations, we can shorten the long spine by one edge, and ensure that the Inclination Invariant holds for all other spines in T'. No spine in $T_1 - T'$ becomes longer.*

Proof. As in the proof of Lemma 6.1 above, the long spine has at least two disjoint rotation sites not involving the top node of the spine. (Because the top node is not involved, neither of the possible rotations lengthens the single spine that extends from T' up into the rest of the tree.) Let the two pairs of subtrees involved in these rotations be (A, B) and (C, D), and denote by AB and CD the subtrees that result from the rotations. Let k_{AB} and k_{CD}, respectively, denote the number of inclination reversals among the leaves of the subtrees AB and CD.

We shorten the long spine of T' by performing the rotation that involves the subtree with the smaller number of inclination reversals; suppose that $k_{AB} \leq k_{CD}$, and so the first rotation is chosen. Because AB and CD are disjoint, it follows that $k_{AB} \leq k/2$. The rotation lengthens two spines: the left spine of A and the right spine of B are lengthened by one. This increase in length may result in a violation of the Inclination Invariant. We restore the invariant recursively on the left and right spines of AB separately, choosing rotations strictly below the root of AB. (One of the two spines includes the edge from the root of AB to its parent. However, because $c \geq 9$, the spine length below the root of AB is at least nine, and two independent rotations are always available below the root.) Since the rotations are below the root of AB, the two recursive invariant restorations are independent. If either A or B is singly inclined, then we apply Lemma 6.1 to restore the Inclination Invariant in that subtree.

The inclination reversals in AB are divided between the subtrees A and B, meaning that $k_A + k_B \leq k_{AB}$. (Equality holds unless an inclination reversal falls between A and B.) The number of rotations needed to restore the invariant is bounded from above by a function $g(k, m)$ satisfying the following recurrence:

$$\begin{aligned} g(k, m) &\leq \max\{g(k_1, m_1) + g(k_2, m_2) + 1 \mid k_1 + k_2 \leq \lfloor k/2 \rfloor \text{ and } m_1 + m_2 \leq m\}, \\ g(0, m) &= \lfloor \log_2 m \rfloor, \end{aligned}$$

where k is the number of inclination reversals in a subtree, and m is the subtree size. To remove the dependence on two variables, we upper bound the first inequality by

$$g(k, m) \leq g(k_1, m) + g(k_2, m) + 1.$$

The resulting system of inequalities is satisfied by

$$g(k, m) = (\sqrt{k} + 1)\lfloor \log_2 m \rfloor,$$

as can easily be verified. This completes the proof. □

7. Choosing a Sibling Operation

Lemma 6.2 shows that correcting violations of the Inclination Invariant is relatively inexpensive, as long as the number of inclination reversals in the subtree below the violating spine is small. This section shows how to choose sibling operations that are not too destructive: the invariant-violating spines they induce have only $O(1)$ inclination reversals below them.

Let x be a merge center, and let v_x be the lowest common ancestor (LCA) of x in the source tree; a merge center corresponds to two consecutive leaves, and the LCA of these leaves is defined as the LCA of the merge center. Node v_x is both-inclined, since its subtree includes two leaves that are inclined toward each other. By the Inclination Invariant, the top of the spine that extends up from v_x is a constant number of edges away. Let this spine top, which is a strict ancestor of v_x unless the latter is the root, be called the *ancestral-spine top* of x, and denoted w_x.

At each execution of Step 2 of the main algorithm, we select a merge center at which to perform a sibling operation using the following criterion:

Merging Criterion.

1. Among all ancestral-spine tops defined by merge centers, choose a spine-top w that is the ancestor of no other ancestral-spine top.
2. Among the merge centers with ancestral-spine top w, choose one whose LCA v is the ancestor of no other LCA.

Algorithmically, one easy way to accomplish this selection is to sort all merge centers lexicographically on the key $(\mathit{height}(w), \mathit{height}(v))$ and select a minimal element from the sorted list.

This selection criterion is well defined: The LCA function is a one-to-one mapping between pairs of consecutive leaves along the bottom of the source tree and internal nodes of the tree. Thus part 2 of the Merging Criterion selects a single merge center, since there is only one merge center associated with each LCA. Furthermore, since any internal node w (except the root) is the top of only one spine, all the LCAs with w as their spine top lie along one spine (or at most two spines). There is exactly one LCA (or at most two, if w is the root) that is the ancestor of no other—it is the lowest on its spine.

The following lemmas show that a merge center selected according to the Merging Criterion creates violations of the invariant that are easy to repair.

Lemma 7.1. *The rotations needed to process a sibling operation selected according to the Merging Criterion affect only the spine from the LCA v to its spine top w and the spines below v.*

Proof. The rotations needed to process the sibling operation involve no nodes above v (Lemmas 4.1 and 4.3). The only spine above v that may be affected is the spine from v to its spine top w. □

Lemma 7.2. *Let v be the LCA of a merge center selected according to the Merging Criterion. Then there is at most one inclination reversal in each of the left and right subtrees of v.*

Proof. There is only one merge center below v, namely, the one with v as its LCA: if there were another merge center, then either the spine top of that merge center would lie below w, contradicting part 1 of the Merging Criterion, or the LCA would lie below v, contradicting part 2. A subtree containing no merge centers, such as the left or right subtree of v, can have at most one inclination reversal among its leaves. □

Lemma 7.3. *Let v be the LCA and let w be the ancestral-spine top of a merge center selected according to the Merging Criterion. Let z be the child of w that is an ancestor of v. (If v is the root, then let $w = z = v$.) Then there are $O(1)$ inclination reversals among the leaf descendants of z.*

Proof. We assume for convenience that w is not the root of the source tree; the proof is similar when w is the root.

Claim 1. *$O(1)$ merge centers have w as their spine top.*

Each such merge center must have its LCA on the unique spine from v to w. This spine has constant length, by the Inclination Invariant. Since each LCA corresponds to a unique merge center, the claim follows.

Claim 2. *All merge centers in the subtree rooted at z have w as their spine top.*

The spine of such a merge center cannot end above w, because z and the parent of w are on the same side (left or right) of w. The spine of such a merge center cannot end below w, or else w would not have been selected by part 1 of the Merging Criterion. The two claims together prove the lemma. □

Lemma 7.4. *Suppose that a merge center is chosen by the Merging Criterion. If the Inclination Invariant holds before the corresponding sibling operation is performed, then the invariant can be restored after the sibling operation using only $O(\log n)$ rotations.*

Proof. A sibling operation performs k rotations in the subtree below the LCA v, for some constant k, and affects only spines below the spine top w (Lemmas 5.1 and 7.1). These rotations add a total of at most $2k$ to the lengths of spines below v and/or extending up from v to w (Lemma 5.2).

We restore the Inclination Invariant from the bottom up. We repeatedly pick a spine that violates the invariant and has no violations below it. Suppose the length of the spine is $c + j$, where c is the constant in the invariant. If the spine lies below v, then it has at most three inclination reversals below it, by Lemma 7.2. The invariant can be restored on and below the spine with $O(j \log n)$ rotations, by Lemma 6.2. If the spine has w as its top, then it has $O(1)$ inclination reversals below it, by Lemma 7.3. Its length can also be reduced from $c + j$ to c by $O(j \log n)$ rotations. (The constant of proportionality is worse for the spine that reaches up to w, but the asymptotic rotation bound is the same as for the spines below v.) □

Combining the preceding lemmas yields a proof that efficient tree morphing is possible.

Theorem 7.5. *Let T_1 and T_2 be two valid n-leaf binary trees with the same weight sequence at the leaves. Then there is a sequence of $O(n \log n)$ valid rotations that transforms T_1 into T_2.*

Proof. We establish the Inclination Invariant initially by normalizing the source tree (Corollary 4.5), then perform $n - 1$ sibling operations, each chosen according to the Merging Criterion. Each sibling operation requires a constant number of rotations (Lemmas 4.1, 4.3, and 5.1), but it may introduce violations of the invariant. After each sibling operation, we restore the invariant with an additional $O(\log n)$ rotations, using Lemma 7.4. □

8. The Tree-Morphing Algorithm

The previous section shows that only $O(n \log n)$ valid rotations are needed to transform T_1 to T_2, but does not give an algorithm to identify those rotations in less than quadratic time. In this section we show how to compute the sequence of rotations in $O(n \log n)$ time. To compute the rotations, we maintain a linear amount of auxiliary data at the nodes of the source tree and outside the tree. In all of these structures, we represent positions in the tree as indices into the original list of leaves. Even though the source and target trees change as leaves are merged into their parents, the leaf indices are computed as though the leaves were still present.

We maintain the following auxiliary data, identified below by the notation "(DSx)":

(DS1) An ordered list of inclination reversals, represented as positions in the original list of leaves.
(DS2) For each merge center x:
  (a) Its lowest common ancestor v_x and its ancestral-spine top w_x.
  (b) A pointer to its position in (DS1), the list of inclination reversals.
(DS3) For each node of the source tree:
  (a) Indices of the first and last leaves below the node.
  (b) A list of the merge centers for which the node is an LCA or an ancestral-spine top.
  (c) A boolean flag that is TRUE iff any descendant of this node is the ancestral-spine top of a merge center.
(DS4) A set of merge centers that satisfy the Merging Criterion.

After the initial normalization of the source tree, all of these data can be computed in linear time by traversing the source and target trees.

We use these data to perform three functions critical to the algorithm: (1) identify merge centers that satisfy the Merging Criterion, (2) compute subtree sizes as required by Lemma 6.1, and (3) count inclination reversals in subtrees, as required by Lemma 6.2. (In fact, we do not compute the actual subtree sizes, but only an approximation that is sufficient to give an $O(\log n)$ performance bound in Lemma 6.1.) As the algorithm runs, we must maintain the data as (1) the source tree is modified by rotations, and (2) merge centers move or disappear because of sibling operations.

Lemma 8.1. *The data described above are sufficient to perform each of the following algorithm operations in constant time*: (1) *identifying merge centers that satisfy the Merging Criterion*, (2) *computing subtree sizes*, *and* (3) *counting inclination reversals*.

Proof. Identifying merge centers that satisfy the Merging Criterion is easy, since they are maintained as part of the data (DS4).

Lemma 6.1 requires us to compute subtree sizes, which our data structures do not record. However, we can get an asymptotic bound similar to that of Lemma 6.1 by using an upper bound on the sizes. When Lemma 6.1 requests the size of the subtree at a node v, we return one more than the difference of its first and last leaf indices, using the data (DS3a) stored at v. This counts the "original" size of the subtree, namely, the size it would have if no leaves had collapsed into their parents. This quantity has the crucial

property that the value at a node is at most the sum of the values at the node's children; its maximum value is n. The analysis in the proof of Lemma 6.1 carries through if we use original sizes instead of current sizes for the subtrees, and so we get an $O(\log n)$ bound for restoring the Inclination Invariant when there are no inclination reversals in the subtree.

Lemma 6.2 requires us to count inclination reversals in subtrees, but Lemma 7.3 ensures that there will be only $O(1)$ inclination reversals in the subtrees for which the queries are posed. Once a merge center has been chosen from (DS4), we locate it in the list (DS1) by following the pointer (DS2b). The inclination reversals that will be counted during the application of Lemma 6.2 are all within constant distance of this position in the list. We can answer the counting queries in constant time by examining $O(1)$ list elements in the vicinity of the merge center. □

Lemma 8.2. *The data described above can be maintained in $O(1)$ time per source tree rotation, and in $O(1)$ amortized time per movement or disappearance of a merge center because of a sibling operation.*

Proof. We consider rotations first. The list of inclination reversals (DS1) is unaffected by rotations. Likewise the pointers (DS2b) are unaffected. The leaf indices (DS3a) change only at the rotated nodes. A rotation affects a constant number of nodes in the tree; in Fig. 13 these nodes are B, C, and their parents. Before the rotation, we identify all the merge centers whose LCAs or spine tops are at one of these nodes, using (DS3b). There are only $O(1)$ of them for the rotations performed by the algorithm of Theorem 7.5. After the rotation, some of these LCAs and spine tops may have moved. We compute their new locations by visiting $O(1)$ nodes in the vicinity of the rotation (because the tree satisfies the Inclination Invariant, all spines above LCAs have constant length). We change the affected node variables (DS3b) and likewise change the merge center variables pointing to those locations (DS2a). The flags (DS3c) may change within a constant-sized neighborhood of moved spine tops. Spine tops whose flag value is FALSE correspond to merge centers that satisfy the Merging Criterion. We update the set (DS4) with the merge centers corresponding to any spine tops whose flag value has changed.

When a merge center moves or disappears, there is no change to the structure of the source tree. The leaf indices (DS3a) do not change, and the LCAs and spine tops corresponding to unchanged merge centers likewise remain unchanged. We update the list of inclination reversals (DS1) at the position of the changed merge center. If the merge center moves, it is because two sibling leaves collapse into their parent, and the new merge center is defined by the parent and its sibling in the target tree. The position of the new merge center in the list (DS1) is the same as that of the one from which it is derived, so (DS1) and (DS2b) are trivial to update. Because of the Inclination Invariant, the LCA and spine top for the new merge center can be computed in constant time by walks on the source tree, and variables (DS2a) and (DS3b) are easily updated.

If the merge center simply moves, then flag values (DS3c) and the set (DS4) change only for spine tops in the vicinity of the old spine top. When the merge center disappears, because the parent of the two collapsing leaves has no sibling leaf in the target tree, the flag values may be affected nonlocally. Flag values change from TRUE to FALSE on a path of arbitrary length above the disappearing merge center. We argue that the total

number of flag changes is nevertheless $O(n+m)$, where m is the number of rotations or merge center movements. There are initially $O(n)$ flags set to TRUE. The total number of changes from FALSE to TRUE is $O(m)$, because each rotation or merge center movement affects only $O(1)$ flags. It follows that the number of changes from TRUE to FALSE is $O(n+m)$, which proves the claim. The amortized cost of updating flag values (DS3c) is $O(1)$. Updating the set (DS4) after a flag change takes at most $O(1)$ time per flag change. □

By combining Theorem 7.5 and Lemmas 8.1 and 8.2, we obtain our main theorem on tree morphing.

Theorem 8.3. *Let T_1 and T_2 be two valid n-leaf binary trees with the same weight sequence at the leaves. Then we can transform T_1 into T_2 using $O(n \log n)$ valid rotations. Furthermore, the sequence of rotations can be computed in $O(n \log n)$ time.*

9. Morphing Polygons

In this section we use the tools of the preceding sections to show how to morph one polygon to another parallel polygon. This is more difficult than morphing bi-infinite chains, because we cannot guarantee that the reduced forms of two parallel polygons will have the same top-level triangle. We show below that each morphing operation that involves two of the top-level trees can be transformed with $O(1)$ parallel moves to involve a single tree.

Let the reduced form of polygon P be a triangle ABC; each of the vertices A, B, and C has a corresponding tree representing its microstructure. Let A', B', and C' denote the vertices and the trees for the top-level triangle of Q. In the tree model, our algorithm for morphing P to Q is the same as Algorithm BasicMorph: we pick a pair of leaves that are siblings in one of A', B', and C', rotate them to be siblings in one of A, B, and C, and then collapse the pair into their parent in both the source and target trees. This works fine if the two leaves belong to the same tree A, B, or C: we simply apply the algorithm of Theorem 8.3 to that tree. If the two leaves belong to different trees, however, we must morph the polygon to bring the two leaves into the same tree before we can apply the algorithm of Theorem 8.3.

Lemma 9.1. *Let A and B be two trees representing the microstructure of adjacent angles in the top-level triangle of a reduced form. Let x be the rightmost leaf of A and let y be the leftmost leaf of B, such that x and y represent adjacent turns in the polygon. Let the total length of the spines containing x and y be k. If the sum of the weights of x and y is valid, then x and y can be made into sibling leaves of the same tree (i.e., contained in the same bottom-level microstructure disk) by $O(k)$ parallel moves.*

Proof. We first perform all applicable valid rotations on the spines containing x and y, shortening each spine to length at most four. As a result, all of the subtrees hanging off the spine above x have the same sign for their weights, and the same is true for the subtrees of the spine above y. See Fig. 21.

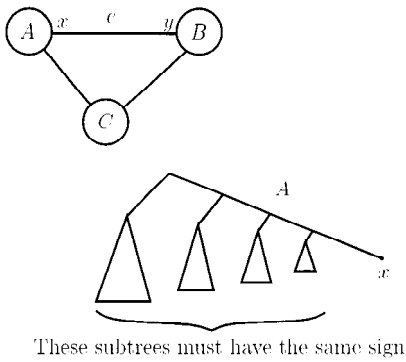

Fig. 21. Bringing x and y together.

We next perform all possible valid rotations at the root of A that shorten the spine above x. When we finish, either x is a child of the root, or the first two subtrees of the spine have weights whose sum is invalid. We process y and B similarly.

Recall that each subtree of A or B has a total weight corresponding to an angle between two consecutive sides in one stage of the reduction process for P. Positive weight corresponds to convex angles and negative to reflex angles. The total weight for each of A and B is of course positive.

For the proof we need to process the tree A according to a number of cases. Because of the assumption that the subtrees hanging off the spine of A have the same sign and that the total weight of A is positive, only three cases are possible. We call A *convex* if x corresponds to a convex angle of P, and so do the other subtrees of the spine of A. We call A *pointy* if x only is convex, while the other subtrees of the spine correspond to concave (reflex) angles. The final case, in which we call A *lumpy*, is when x itself is concave (reflex); then the other subtrees of the spine must all be convex. We make symmetric definitions for y and the spine of B. Figure 22 depicts the geometry of these three cases.

Because of the rotations performed at the root of A, x is a child of the root in the convex and pointy cases (the spine is one edge long). To see this, note that if A is pointy, the subtree of the spine above x has negative weight, and the sum of those weights must be greater than -1, since adding the weight of x gives a positive number. Similarly, if A is convex, the subtrees have positive weight with sum less than 1, since adding the (positive) weight of x produces a valid weight. In both cases, there can be only one subtree of the spine above x—rotations at the root combine all the original subtrees. A symmetric claim holds if B is convex or pointy.

If A is lumpy, we distinguish between two cases, shown in Fig. 23. Let e be the top-level edge joining A to B, and let f be the second-level edge at A. Inside the microstructure disk at A, f is the finite edge; it joins the turns represented by the children of the root of A. Denote the left and right subtrees of A by A_L and A_R.

In case (1), f points toward e; in terms of tree weights, the weight at A_R is negative. In this case we translate f away from the polygon interior, so that the microstructure containing x slides toward B. See Fig. 23(1). When the microstructure gets sufficiently close to B, we can enclose it along with B in a higher-level microstructure disk. In tree

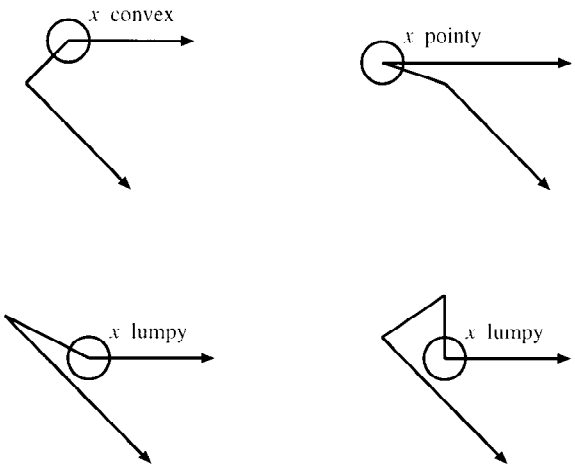

Fig. 22. Classification of x as convex, pointy, or lumpy.

terms, we remove A_R from A and join it to the old tree B as its left sibling. Because A_R has negative weight and B has positive weight, the new tree is valid.

In case (2), f points away from e (the weight of A_R is positive). Let g be the finite edge inside the microstructure disk for A_R, and let A_{RL} and A_{RR} be the left and right subtrees of A_R. By construction, the sum of the weights of A_L and A_{RL} is greater than one, and so the weight of A_{RR} is negative. That is, g points toward e. We translate g away from the polygon interior, so that the microstructure for A_{RR} slides toward B. See Fig. 23(2). As in case (1), we make A_{RR} into the left sibling of B. We are left with a (possibly nonconvex) quadrilateral as the top-level structure, with the four angles represented by

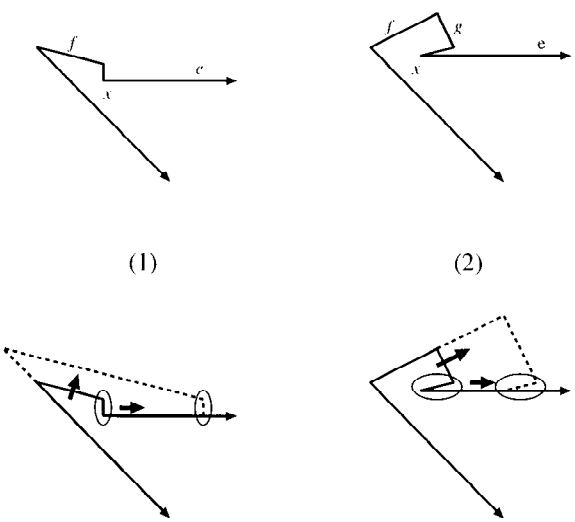

Fig. 23. Moving a leaf in a lumpy tree.

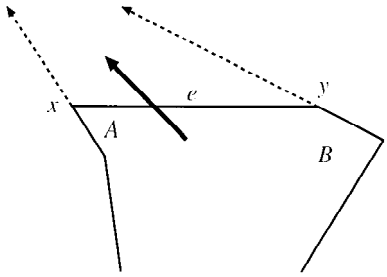

Fig. 24. Moving a leaf in a pointy tree.

A_L, A_{RL}, $A_{RR} + B$, and C (the original third angle of the triangle). One more parallel translation reduces the polygon to a triangle (unless the quadrilateral happens to be a parallelogram).

If B is lumpy, the same construction can be used symmetrically.

Now suppose that A is pointy, while B is convex (or symmetrically B is pointy and A is convex). Note that, since the sum of the weights of x and y is valid, the edge before x in A and the edge after y in B must meet above e (i.e., the other side from ABC). Thus a parallel translation of e away from ABC will eliminate e and cause x and y to come together in the same tree. This also means that at most one of A and B can be pointy.

Because both spines have been shortened as much as possible, x is the right subtree of A and y is the left subtree of B. The edges before x and after y are the finite edges in the top-level microstructure of A and B. If we "unreduce" A and B by one level, those edges belong to the top-level polygon (a 5-gon), and we can translate e to bring x and y together. One more edge elimination reduces the polygon to a triangle again. See Fig. 24.

The final case to consider is when both A and B are convex. In this case we can again bring x and y together by translating edge e away from ABC. All the arguments involved are the same as in the previous case, but simpler. See Fig. 25.

The operations required to bring x and y into the same tree do not increase the lengths of the spines above x and y in the tree representation. Hence $O(1)$ further rotations suffice to make x and y into siblings. □

We apply Lemma 9.1 only when no other sibling operation applies, that is, only when no merge center is contained inside any of the three source trees. This implies that there are at most three merge centers, and the total number of inclination reversals is at most six.

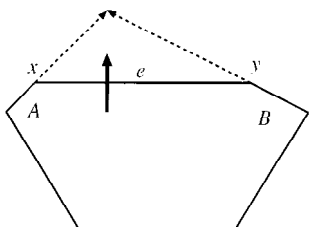

Fig. 25. Moving a leaf in a convex tree.

Lemma 9.2. *The data structures of Section* 8 *can be maintained during the tree rotations and parallel moves of Lemma* 9.1 *at constant cost per operation.*

Proof. The data structures of Section 8 refer to positions by leaf indices in the original tree (assuming there is only one). We extend this to use leaf indices in the cyclic order of all three top-level trees. When the endpoints of an index interval are out of order, it means that the interval wraps around, and operation (2) of Lemma 8.1 (subtree size counting) must take this into account.

With this modification, all the data structures are easy to maintain during the tree surgery of Lemma 9.1. There is at most one merge center in each tree fragment, and so all the data structures that deal with merge centers ((DS2a), (DS2b), (DS3b), (DS3c), and (DS4)) can be maintained by brute force manipulation of $O(1)$ nodes near the tree root. (DS1) has constant size, and is easy to maintain. (DS3a) is also straightforward; it changes only at the $O(1)$ nodes whose descendants change. □

In fact, only some of the data structures of Section 8 need to be maintained. By the time Lemma 9.1 needs to be applied, each tree contains no merge centers. After applying the lemma, one tree contains a single merge center. Thus the Merging Criterion is trivially satisfied. We do need to keep track of subtree sizes and the positions of inclination reversals, but those could be maintained with simpler structures.

We have shown that only $O(1)$ parallel moves are needed to ensure that some sibling operation can be performed on one of the three source trees. Furthermore, these parallel moves can be computed in constant time. Thus the number of parallel moves needed to morph polygons is asymptotically the same as the number needed for bi-infinite chains. Combining this fact with Lemmas 2.4 and 2.5 and Theorems 3.4 and 8.3, we have established our main theorem.

Theorem 9.3. *Given two parallel simple polygons P and Q of n sides, we can continuously deform P to Q by parallel moves while maintaining simplicity. The deformation uses $O(n \log n)$ parallel moves, and it can be computed in $O(n \log n)$ time.*

10. Conclusion

In the few years since its introduction, morphing has become a hugely popular technique in computer graphics. Although its most prominent applications are still in the entertainment industry, there are many serious applications in industrial design [8], medical imaging [5], and data visualization. While morphing is useful in many applications, many of the techniques employed to morph images are not fully understood. In fact, most of the commercially available morphing software packages require a nontrivial amount of human interaction, especially in choosing correspondence points between the source and target images. Furthermore, the intermediate stages of a morphing transformation are often *illegal* images, with no corresponding valid three-dimensional picture.

Our investigation is a first step toward providing theoretical underpinnings for morphing. We have formulated a rigorous model for morphing polygons using "parallel"

moves, and proposed an efficient algorithm to compute the moves required to morph one polygon to another.

It would be interesting to introduce some measure of polygon distortion and look at edge translation algorithms that minimize this measure. Our algorithm is very bad this way, for even if P and Q are very similar-looking polygons, their intermediate morphs can visually look quite different. The tree corresponding to a reduced form of a polygon may have linear height, which implies that the shortest and longest edges in the reduced form have a length ratio that is the product of $\Theta(n)$ infinitesimals. Even if each infinitesimal could be replaced by some constant, say 0.1, the length ratio would still be exponential in n.

Although the weight constraint in our tree morphing problem originated with polygon morphing, it appears to be a rather weak condition, and we suspect our theorem on tree morphing may have other applications outside the polygon morphing context. An open problem suggested by our paper is to determine the true complexity of tree morphing. The number of valid rotations needed to morph between two valid trees is $\Omega(n)$ and $O(n \log n)$; either a tighter lower bound or a better algorithm is needed.

Acknowledgments

The authors wish to acknowledge valuable discussions with John Hughes, Tom Shermer, David Kirkpatrick, G.D. Ramkumar, and Emo Welzl.

References

1. J. Culberson and G. Rawlins. Turtlegons: generating simple polygons from sequences of angles. In *Proceedings of the* 1985 *Computational Geometry Symposium*, pages 305–310, 1985.
2. K. Culik and D. Wood. A note on some tree similarity measures. *IPL*, **15**, 39–42, 1982.
3. M. T. Goodrich and R. Tamassia. Dynamic ray shooting and shortest paths in planar subdivisions via balanced geodesic triangulations. *J. Algorithms*, **23**, 51–73, 1997.
4. J. Hershberger and S. Suri. A pedestrian approach to ray shooting: shoot a ray, take a walk. *J. Algorithms*, **18**, 403–431, 1995.
5. J. F. Hughes. Scheduled Fourier volume morphing. In *Proceedings of the ACM Conference on Computer Graphics*, pages 43–46, 1992.
6. A. Kaul and J. Rossignac. Solid-interpolating deformations. In *Proceedings of Eurographics* 1991, pages 494-505, 1991.
7. T. Sederberg, P. Gao, G. Wang, and H. Mu. 2-D shape blending: an intrinsic solution to the vertex path problem. In *Computer Graphics*, Volume 27, pages 15–18, 1993 (SIGGRAPH '93).
8. T. Sederberg and E. Greenwood. A physically based approach to 2-D shape blending. In *Computer Graphics*, Volume 26, pages 25–34, 1992 (SIGGRAPH '92).
9. D. Sleator, R. Tarjan, and W. Thurston. Rotation distance, triangulations, and hyperbolic geometry. In *Proceedings of the* 1986 *Symposium on Theory of Computing*, pages 122–135, 1986.
10. C. Thomassen. Deformations of planar graphs. *J. Combin. Theory Ser. B*, **34**, 244–257, 1983.
11. G. Vijayan. Geometry of planar graphs with angles. In *Proceedings of the* 1986 *Computational Geometry Symposium*, pages 116–124, 1986.
12. E. Welzl. Personal communication, 1993.

Received May 25, 1999, *and in revised form November* 15, 1999. *Online publication March* 21, 2000.