Raising Roofs, Crashing Cycles, and Playing Pool:
Applications of a Data Structure for Finding Pairwise Interactions

David Eppstein* Jeff Erickson†

Abstract

The straight skeleton of a polygon is a variant of the medial axis, introduced by Aichholzer *et al.*, defined by a shrinking process in which each edge of the polygon moves inward at a fixed rate. We construct the straight skeleton of an n-gon with r reflex vertices in time $O(n^{1+\varepsilon} + n^{8/11+\varepsilon} r^{9/11+\varepsilon})$, for any fixed $\varepsilon > 0$, improving the previous best upper bound of $O(nr \log n)$. Our algorithm simulates the sequence of collisions between edges and vertices during the shrinking process, using a technique of Eppstein for maintaining extrema of binary functions to reduce the problem of finding successive interactions to two dynamic range query problems: (1) maintain a changing set of triangles in $\mathbb{R}^3$ and answer queries asking which triangle would be first hit by a query ray, and (2) maintain a changing set of rays in $\mathbb{R}^3$ and answer queries asking for the lowest intersection of any ray with a query triangle. We also exploit a novel characterization of the straight skeleton as a lower envelope of triangles in $\mathbb{R}^3$. The same time bounds apply to constructing non-self-intersecting offset curves with mitered or beveled corners, and similar methods extend to other problems of simulating collisions and other pairwise interactions among sets of moving objects.

1 Introduction

Suppose we are given the floor plan of a building, and a specification for the slope of its roof planes. How can we design a roof, meeting all the walls at a consistent height, with no dips or flat spots where rainwater can accumulate? Aichholzer *et al.* [8, 7] determined an answer: the *straight skeleton*.

The straight skeleton of a polygon P is, as its name implies, a *skeleton* (one-dimensional topological retract) of the polygon formed from straight line segments. It is closely related to the *medial axis*, another type of skeleton commonly defined as the set of points in P with more than one closest point on the boundary of P. The medial axis is a subset of the Voronoi diagram of the vertices and edges of the polygon and consists of line segments and parabolic arcs. (Voronoi edges that meet reflex vertices of the polygon are technically not part of the medial axis.) Medial axes are used in several applications, including shape recognition and reconstruction [14, 16, 52], mesh generation [31, 47, 50, 51], motion planning [45, 49], and computer-aided manufacturing [32, 33, 50]. Despite these many uses of medial axes, their curved arcs have been considered a shortcoming and have led several researchers to form piecewise-linear approximations by sampling the input [50], rectilinear Voronoi diagrams [50], or other techniques [15, 42]. The straight skeleton provides an alternate piecewise linear construction which unlike these other approaches is not sensitive to sampling rates or to the polygon's orientation.

Although, as described above, the medial axis can be defined in terms of closest points or Voronoi diagrams, there is a third definition which Aichholzer *et al.* generalized in their definition of the straight skeleton. An *offset curve* of P is defined as the set of points having some fixed distance d to P; this curve consists of straight line segments parallel to P's sides and circular arcs centered at P's reflex vertices. As d grows, the straight segments of the offset curve move at a constant speed away from the corresponding edges of P, and the circular arcs grow radially away from their centers. The breakpoints between consecutive line segments and circular arcs in the offset curve trace out edges of the Voronoi diagram of the polygon, and the medial axis can be defined as the

*Department of Information and Computer Science, University of California, Irvine, CA 92697, USA, eppstein@ics.uci.edu, http://www.ics.uci.edu/~eppstein. Research partially supported by NSF grant CCR-9258355 and by matching funds from Xerox Corporation.

†Center for Geometric Computing, Department of Computer Science, Duke University, Box 90129, Durham, NC 27708–0129, USA, jeffe@cs.duke.edu, http://www.cs.duke.edu/~jeffe. Research supported by NSF grant DMS-9627683 and by U. S. Army Research Office MURI grant DAAH04-96-1-0013.

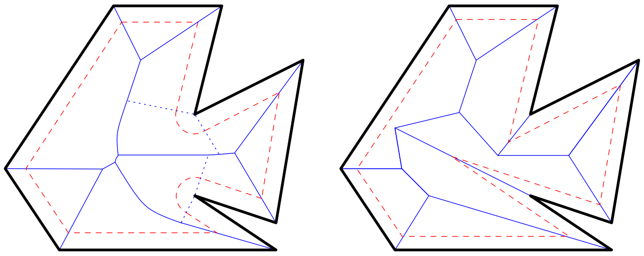

Figure 1. Skeletons and offset curves of a simple polygon. (a) The medial axis (solid), the Voronoi diagram (solid and dotted), and a rounded offset curve. (b) The straight skeleton and a mitered offset polygon.

locus traced out by a subset of these breakpoints [32]. See Figure 1(a). A generalization of offset curves (available for instance in drawing programs such as Adobe Illustrator) again has straight segments paralleling the polygon edges at a fixed distance, but allows the circular arcs connecting these segments to be replaced by other types of *caps*. The form of offset curve that concerns us has *mitered caps* in which the straight segments parallel to P's edges are extended until they meet each other. Similarly to the way the medial axis can swept out by offset curves with round caps, the straight skeleton is swept out by offset polygons with mitered caps. See Figure 1(b).

The straight skeleton has several applications, including architecture, where it describes the shape of a fixed-slope roof rising over a given set of walls [8, 46], and geographic information systems, where it can be used to reconstruct terrains from a given set of rivers and coastlines [7]. The straight skeleton can be used to reconstruct the offset polygons from which it was defined, and form a consistent family of non-self-intersecting mitered-corner offset polygons (unlike those formed by, *e.g.*, Adobe Illustrator, in which crossings must be removed manually). Techniques related to straight skeletons have also been used in origami constructions [37].

Several efficient algorithms are known for constructing medial axes [38, 24, 21, 32]. Unlike the medial axis, however, the straight skeleton cannot be defined by a distance measure or as an abstract Voronoi diagram [36]; consequently, its construction is considerably more difficult. The fastest previously published algorithms both use $O(n^2 \log n)$ time in the worst case [7, 8]; a more careful analysis shows that the running time of one of these algorithms [8] is actually $O(nr \log n)$, where r is the number of reflex vertices. This can be further improved to $O(nr + n \log n) = O(n^2)$ using a quadtree-like data structure of Eppstein of size $O(nr)$ [26]. This paper describes an algorithm that constructs the straight skeleton of a polygon in time and space $O(n^{8/5+\varepsilon})$ for any fixed $\varepsilon > 0$. In fact, our algorithm can construct weighted straight skeletons (where each edge moves at a different rate) of arbitrary planar straight line graphs, within the same time and space bounds. As an immediate application, we also obtain the first subquadratic algorithm for computing mitered offset polygons.

Like earlier algorithms, our algorithm simulates the sequence of interactions between edges and vertices in the shrinking process described above. A technique of Eppstein [25] for maintaining closest pairs reduces the problem of finding the next interaction to two dynamic range query problems: (1) maintain a changing set of triangles in $\mathbb{R}^3$ and answer queries asking which triangle would be first hit by a query ray, and (2) maintain a changing set of rays in $\mathbb{R}^3$ and answer queries asking for the lowest intersection of any ray with a query triangle. Standard range-searching techniques solve these problems in sublinear time per operation, hence our subquadratic time bounds. The same method applies to other problems of simulating pairwise interactions among a set of moving objects. For example, we can trace the paths of a collection of moving billiard balls in sublinear time per collision.

We also present a novel characterization of the unweighted straight skeleton as the projection of the lower envelope of $n + O(r)$ triangles in $\mathbb{R}^3$. These triangles are not (and cannot be [8]) locally defined; nevertheless, we can exploit the structure of these triangles to obtain a "reflex-sensitive" algorithm that runs in time $O(n^{1+\varepsilon} + n^{8/11+\varepsilon} r^{9/11+\varepsilon}) = O(n^{17/11+\varepsilon})$. Practical variants of our algorithm run in time $O(n \log n + nr)$ using space $O(n + r^2)$ and in time $O(n \log n + nr + r^2 \log r)$ using space $O(n)$.

The rest of the paper is organized as follows. In Section 2, we describe Eppstein's technique for maintaining closest pairs. We describe data structures for lowest intersection queries in Section 3. Section 4 contains a description of our straight skeleton algorithms. In Section 5, we describe and solve a similar problem, constructing the *motorcycle graph* of a set of moving points. We briefly describe our algorithm for dynamically simulating moving balls in Section 6. Finally, in Section 7, we list a few open problems.

2 Maintaining Closest Pairs

Our straight skeleton algorithm is based on maintaining a set of moving offset curve features and using a data structure to find each successive interaction between these features. We formalize this approach as follows. Let R and B be sets of objects, and let $d : R \times B \to \mathbb{R}$ be an arbitrary "distance" function that can be computed in constant time. A data structure supports *minimization queries* if, for any object $r \in R$, an object $b \in B$ minimizing $d(r, b)$ can be determined quickly, and vice versa. Our algorithm uses a data structure that efficiently maintains a "closest" pair of objects $r \in R$ and

$b \in B$ minimizing $d(r, b)$ as objects are inserted and deleted, using a dynamic data structure for minimization queries as a black box.

Theorem 2.1 (Eppstein [25, 26]). *Suppose that after $P(n)$ preprocessing time, we can maintain a data structure of size $S(n)$ that supports insertions, deletions, and minimization queries, each in amortized time $T(n)$. Then after $O(P(n) + nT(n))$ preprocessing time, we can maintain the closest pair between R and B in $O(S(n))$ space, $O(T(n) \log n)$ amortized insertion time, and $O(T(n) \log^2 n)$ amortized deletion time.*

The original statement of the theorem [25] required $T(n)$ to be a worst-case bound on the query and update times for the minimization query structure, but the proof still works if $T(n)$ is only an amortized time bound.

Eppstein previously used this technique to maintain closest and furthest bichromatic pairs among a changing set of red and blue points, and for maintaining the Euclidean minimum spanning tree of a changing set of points in the plane [25]. In our application, R and B will be sets of offset curve features, and d will describe the time at which some pair of features interacts. The closest pair in this setting will give us the time of the earliest interaction.

A simplified version of this data structure for maintaining closest pairs from non-geometric point sets (for which $T(n)$ is the trivial $O(n)$ bound for sequential search), is described along with other non-geometric closest pair algorithms, applications, and experiments, in a companion paper [26]. That paper also contains the following result which we use later as part of a more practical simplification of our algorithm.

Theorem 2.2 (Eppstein [26]). *We can maintain the closest pair between two sets of objects in $O(n)$ time per insertion or deletion, using $O(n^2)$ space and preprocessing time.*

The technique consists of forming a quadtree-like structure on the distance matrix of the objects, constructed by forming pairs of members of R and B and defining the distance between pairs to be the minimum of the four distances between members of the pairs, then by continuing recursively in the resulting smaller distance matrix. The closest pair can be found as the single distance at the root of this hierarchical structure; each update causes $O(n)$ distances to be recomputed.

We can obtain sharper bounds if one of the sets is much smaller than the other. Let $r = |R|$ and $n = |B|$ and suppose $r < n$. A simple modification of Eppstein's quadtree structure maintains the closest pair between R and B in $O(n)$ time per change in R, and in $O(r + \log(n/r)) = O(r + \log n)$ time per change in B, using $O(nr)$ space and preprocessing time. We omit further details.

3 Lowest Intersection Queries

We now consider a dynamic range searching problem that arises as a subproblem in our straight skeleton algorithm: maintaining a set of rays in $\mathbb{R}^3$ so that we can quickly determine the lowest intersection between a ray and a query triangle. These *lowest intersection queries* are, in a sense, the inverse of ray shooting queries; instead of the first triangle hit by a query ray, we want the "first" ray hit by a query triangle.

Throughout this section, we describe several dynamic range searching data structures that allow continuous tradeoffs between space and query time. In the interest of brevity, we explicitly state only the query time as a function of the size s of the data structure, omitting additive terms of the form $O(\operatorname{polylog} n)$ or $O(n^\varepsilon)$. All the data structures we describe can be constructed in time $O(s^{1+\varepsilon})$ and can be modified to support insertions and deletions in time $O(s^{1+\varepsilon}/n)$, at the expense of at most an $O(n^\varepsilon)$ factor in both the preprocessing and query times [2, 5, 40].

Theorem 3.1. *Given n rays in $\mathbb{R}^3$, we can answer lowest intersection queries for triangles in time $O(n^{1+\varepsilon}/s^{1/4})$.*

Proof: We follow the same pattern used to construct ray shooting data structures for halfplanes and triangles in $\mathbb{R}^3$ [3, 4, 5]. It suffices to construct a data structure that supports queries asking whether any of the rays intersects a query quadrilateral. To answer a lowest intersection query using this data structure, we apply parametric search [43] or Chan's recent randomized reduction technique [17] to find the largest value z^* so that no ray crosses the intersection of the query triangle and the halfspace $z \le z^*$. The query algorithm we describe below can easily be executed in parallel in $O(\log n)$ time using $O(n^{1+\varepsilon}/s^{1/4})$ processors, so the additional cost of the parametric search is negligible. Chan's reduction is significantly simpler than parametric search, but the resulting randomized query algorithm is only efficient in expectation; in the worst case, the algorithm is actually slower than brute force.

To detect intersections, we construct a multi-level data structure. (We assume the reader is familiar with this standard technique for composing geometric range searching data structures. Otherwise, see [41] or [1].) The first level is a halfspace range searching data structure of Matoušek [41], which lets us (implicitly) find the rays intersecting the plane containing a query quadrilateral q, in time $O(n^{1+\varepsilon}/s^{1/3})$. This reduces the problem to detecting intersections between q and a set of *lines*.

Let $\ell_1, \ell_2, \ell_3, \ell_4$ be the lines containing the edges of q, oriented so that q lies on the same "side" of each line. A line λ intersects q if and only if it has the same relative orientation with respect to all the ℓ_i. The relative orientation of two oriented lines λ, ℓ is defined to be the orientation of any simplex $abcd$, where λ passes through the points a and b, and ℓ through c and d, in that order. Equivalently, the relative orientation is given by the inner product of the Plücker coordinates of the two lines [48].

To determine whether any line intersects q, we use a four-level data structure. Each of the first three levels is used to find the lines oriented correctly with respect to one of the ℓ_i. We use a data structure of Agarwal and Matoušek [4], which supports such queries in time $O(n^{1+\varepsilon}/s^{1/4})$. Finally, for the last level, we only need to know whether a query line ℓ_4 lies entirely above a set of lines. Chazelle *et al.* [19] describe a data structure that supports such queries in time $O(n^{1+\varepsilon}/s^{1/2})$. $\square$

3.1 Fixed-Slope Lines and Nice Triangles

The most difficult part of answering a lowest-intersection query is finding the lines oriented positively with respect to a query line; in the remainder of this section, we refer to these as *line queries*. This is also the bottleneck in answering ray shooting queries among triangles [4, 5]. Let the *slope* of a line in $\mathbb{R}^3$ denote the tangent of its angle from the xy-plane. In some of our applications of ray shooting and lowest intersection queries, either the rays, the edges of the triangles, or both, have fixed slope. In such cases, we can speed up the relevant line queries slightly, and thus improve the time to answer ray shooting and lowest intersection queries.

A line with fixed slope has only three degrees of freedom: its intersection with the xy-plane and the slope of its projection onto the xy-plane, for example. Except for extreme cases, which we can handle with a lower-dimensional data structure, we can represent any fixed-slope line as a point in $\mathbb{R}^3$. For an *arbitrary* line ℓ, the set of fixed-slope lines that intersect ℓ forms a bounded-degree algebraic surface in $\mathbb{R}^3$. One halfspace of this surface contains the fixed-slope lines oriented positively with respect to ℓ.

Thus, preprocessing a set of fixed-slope lines for arbitrary line queries is equivalent to preprocessing a set of points in $\mathbb{R}^3$ for semialgebraic range queries. Agarwal and Matoušek [4] describe a linear-space data structure that supports such queries in time $O(n^{2/3+\varepsilon})$. Combining this with a data structure of Agarwal and Sharir [5] that supports line queries for an *arbitrary* set of lines in $O(\log n)$ time, using $O(n^{4+\varepsilon})$ space and preprocessing, we obtain the following tradeoff between space and query time. (For further details on space-time tradeoffs, see [1, 41].)

Lemma 3.2. *Given a set of n lines in $\mathbb{R}^3$ with fixed slope, we can answer arbitrary line queries in time $O(n^{8/9+\varepsilon}/s^{2/9})$.*

Conversely, preprocessing a set of arbitrary lines for fixed-slope line queries is equivalent to preprocessing a set of algebraic surfaces in $\mathbb{R}^3$ for point location queries. Chazelle *et al.* [18] describe a data structure of size $O(n^{3+\varepsilon})$ that supports such queries in $O(\log n)$ time. Agarwal and Matoušek [4] describe a linear-size data structure that supports *arbitrary* line queries in $O(n^{3/4+\varepsilon})$ time. Combining these two data structures, we achieve the following space-time tradeoff.

Lemma 3.3. *Given a set of n arbitrary lines in $\mathbb{R}^3$, we can answer fixed-slope line queries in time $O(n^{9/8+\varepsilon}/s^{3/8})$.*

Finally, by combining our two three-dimensional structures, we can support fixed-slope line queries for a set of fixed-slope lines in time $O(n^{1+\varepsilon}/s^{1/3})$. (The slope of the given lines and the slope of the query lines could be different.)

Say that a triangle is *nice* if two of its edges have fixed (but possibly different) slopes. By replacing the general line query data structure used in Theorem 3.1 with our fixed-slope data structure, we obtain the following result.

Theorem 3.4. *Given a set of n rays in $\mathbb{R}^3$, we can support lowest intersection queries for nice triangles in time $O(n^{9/8+\varepsilon}/s^{3/8})$. If all the rays have fixed slope, we can support lowest intersection queries for nice triangles in time $O(n^{1+\varepsilon}/s^{1/3})$ and for arbitrary triangles in time $O(n^{8/9+\varepsilon}/s^{2/9})$.*

We can make a similar improvement for the standard ray shooting problem. Agarwal and Matoušek [4] and Agarwal and Sharir [5] describe data structures for ray shooting among triangles. The bottleneck in both of their data structures is answering line queries. By substituting our data structure into theirs, we obtain a faster algorithm for ray shooting among nice triangles, or with fixed-slope rays, or both.

Theorem 3.5. *Given a set of n triangles in $\mathbb{R}^3$, we can support fixed-slope ray shooting queries in time $O(n^{9/8+\varepsilon}/s^{3/8})$. If all the triangles are nice, we can support arbitrary ray shooting queries in time $O(n^{8/9+\varepsilon}/s^{2/9})$ and fixed-slope ray shooting queries in time $O(n^{1+\varepsilon}/s^{1/3})$.*

4 Straight Skeletons (Raising Roofs)

We now describe our subquadratic straight skeleton algorithms.

The definition of straight skeleton generalizes easily to disconnected polygons, polygons with holes, and even arbitrary planar straight-line graphs [7]. Although for simplicity we explicitly consider only simple polygons, all of the results in this section apply to these more general cases as well.

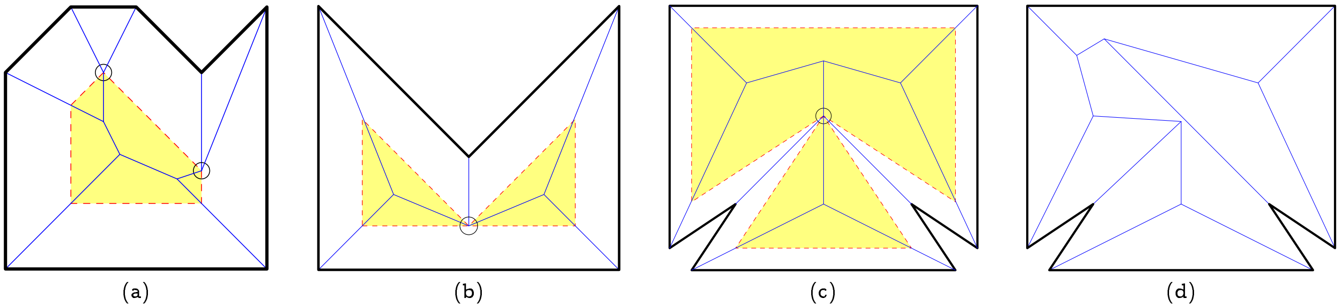

Figure 2. (a) Two simultaneous edge events. (b) A split event. (c) A vertex event. (d) Perturbing the polygon to eliminate the vertex event produces a radically different skeleton.

4.1 Events

The straight skeleton of a polygon is constructed by a shrinking process in which each edge moves inwards at a fixed rate. Assuming the polygon is in general position, it undergoes only two types of combinatorial changes as it shrinks. An *edge event* occurs when an edge collapses down to a point; if its neighboring edges still have nonzero length, they become adjacent. See Figure 2(a). Note that one of the endpoints of the disappearing edge can be a reflex vertex. A *split event* occurs when a reflex vertex collides with and splits an edge; the edges adjacent to the reflex vertex are now adjacent to the two parts of the split edge. Each split event divides a component of the shrinking polygon into two smaller components. See Figure 2(b). Each event introduces a node of degree three into the evolving straight skeleton.

In degenerate cases, the straight skeleton can have vertices of degree higher than three, introduced by simultaneous events at the same location. Most of the time, we can handle these events one at a time using standard perturbation techniques, replacing the high-degree node with several nodes of degree three. The only exception occurs when two or more reflex vertices (and nothing else) reach the same point simultaneously. We call this a *vertex event*. See Figure 2(c). Unlike edge or split events, a vertex event can introduce a new reflex vertex into the shrinking polygon, although the total number of reflex vertices always decreases. Any perturbation of the polygon that removes a vertex event radically changes the structure of the polygon's straight skeleton; consequently, we must handle vertex events directly. See Figure 2(d).

4.2 A Subquadratic Algorithm

Like earlier algorithms [8, 7], our basic approach is to simulate the sequence of edge, split, and vertex events that define the skeleton. We view time as a third spatial dimension, so that the shrinking process becomes an upward sweep of the *roof* of the polygon with a horizontal plane. The key observation is that although we do not know the entire sequence of events until we are finished, at any time we can efficiently compute the *next* event using only local information.

Each edge of the polygon defines a (possibly unbounded) triangle in $\mathbb{R}^3$, whose other two edges are defined by the initial paths of the two endpoints; similarly, each reflex vertex defines a ray in $\mathbb{R}^3$. If the first event is an edge event, it happens when the sweep plane reaches the top of a triangle. If the first event is a split or vertex event, it happens when a ray hits a triangle.

We store the triangles and rays in a data structure that maintains the first event of each type. To update the data structure at each event, we perform a constant number of insertions and deletions. At each edge event, we delete three triangles (defined by the collapsed edge and its two neighbors), possibly delete one ray (defined by a reflex endpoint of the collapsed edge), and insert two triangles (defined by the newly adjacent edges). At each split event, we delete one ray (the reflex vertex), delete three triangles (the edge being split and the two edges adjacent to the reflex vertex), and insert four triangles (the two pairs of adjacent edges defined by the split). At a vertex event involving k reflex vertices, we delete $2k$ triangles and k rays, and insert $2k$ triangles and possibly one ray. Over the entire event sequence, we perform $O(n)$ insertions and deletions.

We maintain potential edge events by storing the triangles in a simple priority queue, where the priority of a triangle is the t-coordinate of its top vertex. The overall time spent maintaining the priority queue is $O(n \log n)$.

To find the next split or vertex event, we use a data structure that that maintains the lowest intersection between a set of rays and a set of triangles in $\mathbb{R}^3$. Theorem 2.1 reduces this to two range-searching problems: (1) maintain a set of triangles and answer ray shooting queries, and (2) maintain a set of rays and answer lowest-intersection queries. To answer ray-shooting queries, we use a data structure of Agarwal and Matoušek [4], and to answer lowest-intersection queries, we use Theorem 3.1. Both data structures support queries in time $O(n^{1+\varepsilon}/s^{1/4})$ and updates in time $O(s^{1+\varepsilon}/n)$; balancing the query and update times, we obtain data structures of size $s = O(n^{8/5+\varepsilon})$ that sup-

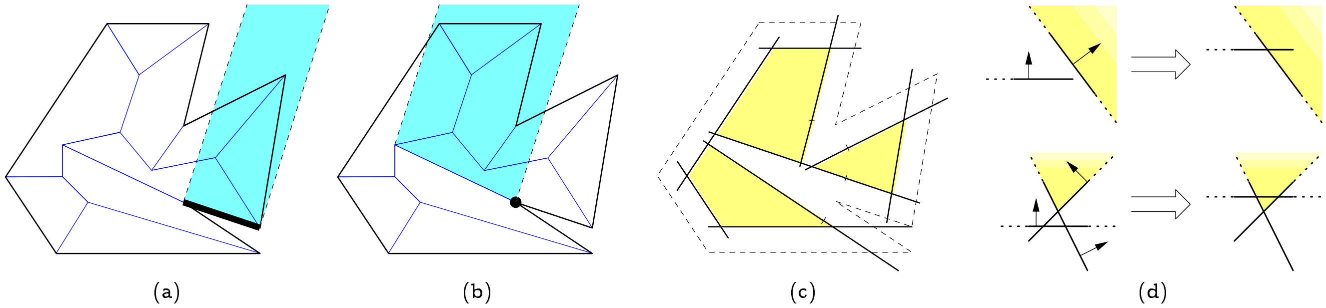

Figure 3. (a) Top view of an edge slab. (b) Top view of a reflex slab. (c) A cross section of the slabs, after two split events. (d) Two impossible transitions; see Lemma 4.2.

ports queries or updates in time $O(n^{3/5+\varepsilon})$. By Theorem 2.1, the total time to initialize the combined data structure and perform $O(n)$ insertions and deletions is $O(n^{8/5+\varepsilon})$. Since this dominates the time to handle the edge events, we obtain the following theorem.

Theorem 4.1. *The straight skeleton of an n-gon can be constructed in time and space $O(n^{8/5+\varepsilon})$.*

Our algorithm can also be used to construct *weighted* straight skeletons, where each edge moves at a different different speed, in the same time and space. The only difference from the unweighted case is that both endpoints of a collapsing edge can be reflex vertices, in which case the edge event introduces a new reflex vertex.

4.3 A Faster Reflex-Sensitive Algorithm

We can improve our algorithm in the unweighted case. For any polygon in the xy-plane, we define two families of *slabs* in $\mathbb{R}^3$. Each slab is an semi-infinite strip, bounded on two sides by parallel rays pointing upwards from the xy-plane at a 45-degree angle and perpendicular to some edge of the polygon. Each edge e defines an *edge slab*, bounded below by e and on the sides by rays perpendicular to e. Each reflex vertex $v = e \cap e'$ defines two *reflex slabs*, bounded below by the edge of the roof induced by v, and bounded on the sides by rays perpendicular to either e or e'. See Figure 3(a) and (b).

Lemma 4.2. *The straight skeleton of a polygon is the intersection of the polygon and the vertical projection of the lower envelope of its edge slabs and reflex slabs.*

Proof: Consider the shrinking process for generating the straight skeleton, or equivalently, sweep a horizontal plane upwards through the roof. Each point in a cross-section of a slab corresponds to exactly one point on the base edge of the slab, so we can think of these base edge points as all traveling upwards along the slabs.

Each of these points starts on the boundary of the shrinking polygon; points on edge slabs appear at the beginning, and points on reflex slabs appear later. As the polygon shrinks, slab points leave the polygon boundary. See Figure 3(c). As long as every slab point stays outside the polygon after it leaves the boundary, only points on the boundary of the polygon can contribute to the lower envelope of the slabs.

There are only two ways that a point can reenter the polygon after it leaves: either a slab endpoint overtakes an edge of the polygon, or a slab overtakes a convex vertex of the polygon. See Figure 3(d). Since all the segments and their endpoints are moving at the same speed, neither of these transitions can occur. $\square$

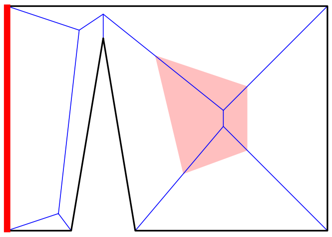

Figure 4. The roof of a weighted polygon is not the lower envelope of its slabs. The leftmost edge has higher weight than the other edges, and its edge slab cuts through the roof in the shaded region.

Aichholzer *et al.* [8] show that the roof cannot be expressed as the lower envelope of partial linear functions, each function locally defined by a small neighborhood of an edge. Our reflex slabs are not locally defined. We emphasize that Lemma 4.2 holds only for unweighted straight skeletons. Counterexamples for the weighted case are easy to construct; see Figure 4.

To exploit Lemma 4.2, we classify split events into two classes, *edge splits* and *reflex splits*, depending on which type of slab is hit by the ray. For the moment, let us assume that the polygon is in general position, so there are no vertex events. We preprocess the edge slabs into a ray shooting data structure, and for each ray, we find the first edge slab that it hits. Since two sides of each slab have slope 1, the total time to preprocess the n slabs and answer r ray shooting queries is $O(n^{1+\varepsilon} + n^{8/11+\varepsilon} r^{9/11+\varepsilon})$ by Theorem 3.5. This gives us a superset of the possible edge split events, which we then sort chronologically.

We simultaneously maintain a priority queue for the next edge event, scan the sorted list of potential edge split events, and maintain the next reflex split event using the following variant of our earlier algorithm. We maintain a data structure storing only the r rays and $2r$ reflex slabs defined by reflex vertices. Initially, we store *unbounded* reflex slabs, each with only two edges, since we do not yet know where the third edge is. At each split event of either type, we replace two unbounded slabs with the correct reflex slabs and delete one ray. If an edge event involves a reflex vertex, we update its slab and delete its ray. Theorem 2.1 reduces maintaining the next reflex split event to maintaining data structures for ray shooting and lowest intersection queries. Since one side of every unbounded slab and two sides of every reflex slab have slope 1, we can maintain the next reflex split in time $O(r^{6/11+\varepsilon})$ per split event, by Theorems 3.4 and 3.5. Since the data structure is updated r times, the total time spent maintaining this data structure is $O(r^{17/11+\varepsilon})$.

Theorem 4.3. *The straight skeleton of an n-gon with r reflex vertices can be constructed in time $O(n^{1+\varepsilon} + n^{8/11+\varepsilon} r^{9/11+\varepsilon})$.*

In degenerate cases, we must also handle vertex events. At each vertex event involving k reflex vertices, we delete k rays and replace $2k$ unbounded slabs with the correct reflex slabs. If the event creates a new reflex vertex, we perform another ray shooting query among the edge slabs, update the sorted sequence of potential edge splits, and insert a new ray into the next-reflex-split data structure. Even with vertex events, there are at most $2r - 1$ reflex vertices over the entire history of the polygon, so the asymptotic running time of our algorithm is unchanged.

4.4 Final Remarks

Although our algorithms are a significant theoretical improvement over earlier results, because of the complexity of the range-searching algorithms involved, they would be less efficient in practice. The running time of Aichholzer and Aurenhammer's algorithm derives from the number of topological changes in a triangulation (or trapezoidal decomposition) of the shrinking polygon [7]. Although there can be $\Theta(n^2)$ such changes in the worst case, "typical" inputs require much less work. Even for polygons specially constructed to make their algorithm run slowly, our algorithm would be slower for reasonable values of n.

However, if we use our modification of Eppstein's quadtree-based data structure (Theorem 2.2) to maintain the next split event, instead of Theorem 2.1, we obtain a practical algorithm for weighted skeletons that runs in time $O(n \log n + nr)$ using $O(nr)$ space. In the unweighted case, we can improve the space bound to $O(n + r^2)$ by using brute force ray-shooting to find the potential edge splits and Eppstein's quadtree to maintain reflex splits. Alternately, if we use Aichholzer and Aurenhammer's triangulation-based algorithm [7] to find reflex splits, we obtain a practical linear-space algorithm that runs in time $O(n \log n + nr + r^2 \log r)$. All of these algorithms improve the previously best time bound $O(nr \log n)$ [8].

If the input polygon is c-oriented, we can compute its straight skeleton in time $O(c^4 n \operatorname{polylog} n)$. Each of the relevant ray shooting and inverse ray shooting queries can be answered using one of $O(c^4)$ orthogonal range query data structures, since each ray has one of $O(c^2)$ orientations, and there are $O(c^2)$ parallel families of slabs. Finally, if the input polygon is convex, we can construct its straight skeleton in linear time [6, 20], since the straight skeleton is identical to the medial axis in that case.

Once we compute the straight skeleton, we can compute any desired mitered offset polygon from it in linear time. Alternatively, we can compute offset polygons directly, with the same worst-case time bound, by halting the inward sweep when we reach the desired offset. We can also compute offset polygons with *beveled* caps by adding a zero-length edge at each reflex vertex, perpendicular to the vertex's bisector, and applying the same algorithm.

5 Crashing Cycles

In this section, we consider a problem that captures the most difficult part of constructing straight skeletons: determining how the reflex vertices interact. Imagine several people riding motorcycles out in the desert. Each motorcycle is specified by an initial location and a velocity. All the bikes start moving simultaneously, and thereafter, no turning, braking, or acceleration is allowed. The bikes are extremely fragile; if any motorcycle runs over the track previously left by another bike, it immediately crashes. If two motorcycles collide, they both crash. After all the bikes either crash or escape to infinity, their tracks form a planar directed graph, which we call the *motorcycle graph*. See Figure 5. The problem is to construct this graph as quickly as possible. A similar problem was previously considered by Lisberger *et al.* [39]

If we form a non-simple polygon in which each motorcycle is replaced by a small hole in the form of a sharp isosceles triangle, the straight skeleton edges traced out by the sharp reflex vertices of these triangles will approximate the motorcycle graph. Motorcycle graphs are also a generalization of the *weighted planar partitions*

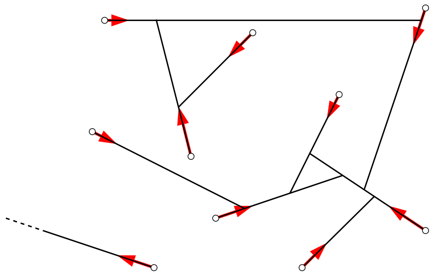

Figure 5. A motorcycle graph.

introduced by Czyzowicz *et al.* as a tool to solve certain art gallery problems [23]. Given a sequence of n non-intersecting line segments in the plane, the weighted planar partition is obtained by extending each segment, one at a time in the order presented, until each endpoint reaches another (possibly extended) segment.

Every motorcycle graph is a pseudo-forest with at most $2n$ vertices and $2n$ edges. Despite the simplicity of this graph, no near-linear algorithm is known for its construction, except in a few special cases, described briefly below. Aichholzer and Aurenhammer's straight skeleton algorithm can easily be adapted to construct motorcycle graphs in $O(n^2 \log n)$ time; the time bound can be improved to $O(n^2)$ using Eppstein's quadtree data structure [26]. Our methods yield a subquadratic algorithm.

Theorem 5.1. *The motorcycle graph of n moving points can be constructed in time and space $O(n^{17/11+\varepsilon})$.*

Proof: Our algorithm simulates the interactions between bikes and tracks as the bikes move and the tracks grow. As in our previous straight skeleton algorithms, we treat time as a third spatial dimension. A bike with initial position (x, y) and velocity (u, v) induces a ray, based at $(x, y, 0)$ and pointing in the direction $(u, v, 1)$. We classify each track as either *live* or *dead*, depending on whether the corresponding bike has crashed or not. Each track induces a vertical *curtain*, consisting of all the points directly above the corresponding ray (if the track is live) or segment (if the track is dead) in $\mathbb{R}^3$. Initially we have a set of n rays and n live curtains. Whenever a bike crashes, we delete the bike's ray and add a third side to its curtain. The next collision is always given by the lowest intersection of a bike with a curtain.

Theorem 2.1 reduces maintaining the next collision to two range searching problems: (1) maintain a changing set of curtains and answer queries asking which curtain is first hit by a query ray, and (2) maintain a changing set of rays in $\mathbb{R}^3$ and answer queries asking for the lowest intersection of any ray with a query curtain. Since all but one side of every curtain is vertical, curtains are nice triangles. We can apply Theorems 3.4 and 3.5 to solve both of these subproblems in time $O(n^{6/11+\varepsilon})$ per query or update, using a data structure of size $O(n^{17/11+\varepsilon})$. □

We can improve the performance of some of the data structures. De Berg *et al.* [13] describe a data structure that supports ray shooting queries among curtains with query and update time $O(n^{1/3+\varepsilon})$. For each dead track, it suffices to store the vertical strip consisting of points directly above *or below* each dead track, instead of three-sided curtains; this allows us to reduce the lowest-intersection query time for dead tracks to $O(n^{1/3+\varepsilon})$. Unfortunately, these improved time bounds are still dominated by the time to find the lowest intersection with a live curtain, so we do not obtain a faster algorithm overall.

There are several special cases of motorcycle graphs that we can compute more quickly. If all the bikes move at the same speed (or in a constant number of different speeds), we can build the motorcycle graph in time and space $O(n^{3/2+\varepsilon})$. If the bikes move in only a constant number of directions, we need only $O(n^{4/3+\varepsilon})$ time and space. If there are only a constant number of different velocity vectors, we can compute the graph in $O(n \operatorname{polylog} n)$ time and space using orthogonal range searching data structures. Finally, if all the velocity vectors have, for example, non-negative x-coordinates, we can use a simple sweep-line algorithm to compute the motorcycle graph in $O(n \log n)$ time. Further details will appear in the full version of the paper.

6 Playing Pool

Our techniques can be applied to any problem that calls for the efficient detection of collisions among moving objects. For example, consider the following *billiard problem*. We are given a set of unit disks in the plane representing billiard balls, each with an initial position and velocity, and are asked to compute the sequence of collisions that occur as the balls move and bounce off each other, following the standard laws of classical physics.

Basch *et al.* [12] observe that only the closest pair of balls can collide, and suggest simulating collisions using a kinetic data structure to efficiently maintain the closest pair. (See also [11, 30].) In the worst case, however, the closest pair changes $\Theta(n^2)$ times without a single collision, which makes this approach less efficient than even the most naïve algorithm that simply checks every pair of balls. Kim *et al.* [35] describe an event-driven algorithm that divides space into a uniform grid of cells and uses $O(\log n)$ time whenever a ball enters a cell, leaves a cell, or collides with another ball in the same cell; a similar approach (but for more general objects) is described by Mirtich and Canny [44]. These algorithms perform quite badly if the balls are very far apart or if there are few collisions. Several other collision-detection algorithms are known that are efficient, at least in practice, for many types of objects; see, for example, [10, 22, 27, 34]. Almost none of these have theoretical guarantees on their performance. In fact, like the algorithms in [35, 44], most perform well only in dense environments.

Our methods yield the first known collision-detection algorithm with a guaranteed sublinear time bound per collision.

Theorem 6.1. *The billiard problem described above can be solved in time* $O(n^{20/29+\varepsilon}) = O(n^{0.6897})$ *per collision using space* $O(n^{49/29+\varepsilon}) = O(n^{1.6897})$.

Proof: Theorem 2.1 reduces the billiard problem to maintaining a ray shooting data structure for a changing set of elliptical cylinders in $\mathbb{R}^3$ with circular horizontal cross-sections. We express each ray shooting query as a composition of several four- and five-dimensional semi-algebraic range queries, which we can solve using techniques of Agarwal and Matoušek [4]. We omit further details. □

Using trivial ray-shooting data structures, we also immediately obtain practical algorithms for the billiard problem that use $O(n \log^2 n)$ time per collision using linear space, or $O(n)$ time per collision using quadratic space. We can also maintain collisions between the balls and any collection of stationary line segments—the sides of a pool table, for example—with the same asymptotic space and time bounds. (Deciding whether a system of moving balls and stationary line segments ever reaches a steady state is PSPACE-complete, since a polynomial-space Turing machine can be simulated by such a system [28].)

7 Open Problems

The most obvious open problem is to improve the running times of our algorithms. The best lower bound for constructing either skeletons of planar straight-line graphs or motorcycle graphs is $\Omega(n \log n)$, by an easy reduction from sorting; no nontrivial lower bound is known for constructing straight skeletons of simple polygons. It seems especially unlikely that $O(n^{17/11})$ is the best possible time bound for constructing a motorcycle graph or a single offset polygon.

Atallah *et al.* [9] show that the construction of weighted planar partitions is P-complete. A similar technique was used by Griffeath and Moore to prove the P-completeness of certain two-dimensional cellular automata [29]. A simple modification of either proof shows that construction of motorcycle graphs is also P-complete, even when the motorcycles move only north, east, or north-east. (There is a simple $O(n \log n)$-time sequential algorithm for this special case.) Similar arguments show that constructing the straight skeleton of an arbitrary planar straight-line graph is a P-complete problem, but the proof does not extend to simple polygons. Can straight skeletons of simple polygons be constructed efficiently in parallel?

The main difficulty in constructing straight skeletons is computing the nonlocal effects of the reflex vertices. This non-locality may also prevent straight skeletons from being as useful as medial axes in applications such as mesh generation and motion planning. Aichholzer and Aurenhammer [7] observe that if the polygon contains no acute angles, then the resulting skeleton shares several properties with Voronoi diagrams. Can we construct the straight skeleton more quickly if we have a constant bound on the smallest external angle in the input polygon? If the minimum external angle is bounded, then so is the ratio between the speed of any reflex vertex and the speed of any edge. Although the straight skeleton is still not an abstract Voronoi diagram in this case, it seems likely that this property restores enough locality to the skeleton that faster divide-and-conquer or incremental techniques can be applied.

Acknowledgments The second author would like to thank Pankaj Agarwal, Lars Arge, and T. M. Murali for helpful discussions on shooting rays, crashing motorcycles, and raising roofs.

References

[1] P. K. Agarwal and J. Erickson. Geometric range searching and its relatives. Technical report CS-1997-11, Duke University, May 1997. To appear in *Discrete & Computational Geometry: Ten Years Later*, B. Chazelle, J. E. Goodman, and R. Pollack, editors, American Mathematical Society Press, 1998. ⟨http://www.cs.duke.edu/~jeffe/pubs/survey.html⟩.

[2] P. K. Agarwal and J. Matoušek. Dynamic half-space range reporting and its applications. *Algorithmica* 13:325–345, 1995.

[3] P. K. Agarwal and J. Matoušek. Ray shooting and parametric search. *SIAM J. Comput.* 22(4):794–806, 1993.

[4] P. K. Agarwal and J. Matoušek. On range searching with semialgebraic sets. *Discrete Comput. Geom.* 11:393–418, 1994.

[5] P. K. Agarwal and M. Sharir. Applications of a new space-partitioning technique. *Discrete Comput. Geom.* 9:11–38, 1993.

[6] A. Aggarwal, L. J. Guibas, J. Saxe, and P. W. Shor. A linear-time algorithm for computing the Voronoi diagram of a convex polygon. *Discrete Comput. Geom.* 4(6):591–604, 1989.

[7] O. Aichholzer and F. Aurenhammer. Straight skeletons for general polygonal figures in the plane. *Proc. 2nd Annu. Internat. Conf. Computing and Combinatorics*, pp. 117–126. Lecture Notes in Computer Science 1090, Springer, 1996.

[8] O. Aichholzer, F. Aurenhammer, D. Alberts, and B. Gärtner. A novel type of skeleton for polygons. *J. Universal Comput. Sci.* 1(12):752–761, 1995. ⟨http://www.iicm.edu/jucs_1_12/a_novel_type_of⟩.

[9] M. J. Atallah, P. Callahan, and M. T. Goodrich. P-complete geometric problems. *Internat. J. Comput. Geom. Appl.* 3:443–462, 1993.

[10] D. Baraff. Interactive simulation of solid rigid bodies. *IEEE Comput. Graph. Appl.* 15(3):63–75, May 1995.

[11] J. Basch, L. Guibas, and J. Hershberger. Data structures for mobile data. *Proc. 8th ACM-SIAM Sympos. Discrete Algorithms*, pp. 747–756, 1997.

[12] J. Basch, L. Guibas, and L. Zhang. Proximity problems on moving points. *Proc. 13th Annu. ACM Sympos. Comput. Geom.*, pp. 344–351, 1997.

[13] M. de Berg, D. Halperin, M. Overmars, J. Snoeyink, and M. van Kreveld. Efficient ray shooting and hidden surface removal. *Algorithmica* 12:30–53, 1994.

[14] H. Blum. A transformation for extracting new descriptors of shape. *Models for the Perception of Speech and Visual Form*, pp. 362–380. MIT Press, 1967.

[15] F. L. Bookstein. The line-skeleton. *Comput. Graph. Image Process.* 11:123–137, 1979.

[16] L. Calabi and W. E. Hartnett. Shape recognition, prairie fires, convex deficiencies and skeletons. *Amer. Math. Monthly* 75:335–342, 1968.

[17] T. M. Chan. Geometric applications of a randomized optimization technique. To appear in *Proc. 14th Annu. ACM Sympos. Comput. Geom.*, 1998.

[18] B. Chazelle, H. Edelsbrunner, L. Guibas, and M. Sharir. A singly-exponential stratification scheme for real semi-algebraic varieties and its applications. *Theoret. Comput. Sci.* 84:77–105, 1991.

[19] B. Chazelle, H. Edelsbrunner, L. J. Guibas, M. Sharir, and J. Stolfi. Lines in space: Combinatorics and algorithms. *Algorithmica* 15:428–447, 1996.

[20] L. P. Chew. Building Voronoi diagrams for convex polygons in linear expected time. Technical Report PCS-TR90-147, Dept. Math. Comput. Sci., Dartmouth College, 1986.

[21] F. Chin, J. Snoeyink, and C.-A. Wang. Finding the medial axis of a simple polygon in linear time. *Proc. 6th Annu. Internat. Sympos. Algorithms Comput.*, pp. 382–391. Lecture Notes Comput. Sci. 1004, Springer-Verlag, 1995.

[22] J. D. Cohen, M. C. Lin, D. Manocha, and M. K. Ponamgi. I-collide: An interactive and exact collision detection system for large-scale environments. *Proc. ACM Interactive 3D Graphics Conf.*, pp. 189–196, 1995.

[23] J. Czyzowicz, I. Rival, and J. Urrutia. Galleries, light matchings and visibility graphs. *Proc. 1st Workshop Algorithms Data Struct.*, pp. 316–324. Lecture Notes Comput. Sci. 382, Springer-Verlag, 1989.

[24] O. Devillers. Randomization yields simple $O(n \log^* n)$ algorithms for difficult $\Omega(n)$ problems. *Internat. J. Comput. Geom. Appl.* 2(1):97–111, 1992. ⟨http://www.inria.fr/prisme/biblio/search.html⟩.

[25] D. Eppstein. Dynamic Euclidean minimum spanning trees and extrema of binary functions. *Discrete Comput. Geom.* 13:111–122, 1995.

[26] D. Eppstein. Fast hierarchical clustering and other applications of dynamic closest pairs. *Proc. 9th Annu. ACM-SIAM Sympos. Discrete Algorithms*, pp. 619–628, 1998.

[27] A. Foisy, V. Hayward, and S. Aubry. The use of awareness in collision prediction. *Proc. 1990 IEEE Internat. Conf. Robotics and Automation*, pp. 338–343, 1990.

[28] E. Fredkin and T. Toffoli. Conservative logic. *Internat. J. Theoret. Phys.* 21:219–253, 1981/82. Proceedings of Conference on Physics of Computation, Dedham, Mass., 1981.

[29] D. Griffeath and C. Moore. Life Without Death is P-complete. Working Paper 97-05-044, Santa Fe Institute, 1997. To appear in *Complex Systems.* ⟨http://psoup.math.wisc.edu/java/lwodpc/lwodpc.html⟩.

[30] L. J. Guibas. Kinetic data structures: A state of the art report. To appear in *Proc. 3rd Workshop on Algorithmic Foundations of Robotics*, P. K. Agarwal, L. Kavraki, and M. Mason, editors, A. K. Peters, 1998.

[31] H. N. Gürsoy and N. M. Patrikalakis. An automatic coarse and fine surface mesh generation scheme based on medial axis transform, Part I: Algorithms. *Engineering with Computers* 8:121–137, 1992.

[32] M. Held. Voronoi diagrams and offset curves of curvilinear polygons. To appear in *Comput. Aided Design.* ⟨http://www.cosy.sbg.ac.at/~held/papers/cad96.ps.gz⟩.

[33] M. Held, G. Lukács, and L. Andor. Pocket machining based on contour-parallel tool paths generated by means of proximity maps. *Comput. Aided Design* 26(3):189–203, Mar. 1994.

[34] P. M. Hubbard. Collision detection for interactive graphics applications. *IEEE Trans. Visualization and Computer Graphics* 1(3):218–230, Sept. 1995.

[35] D. Kim, L. Guibas, and S. Shin. Fast collision detection among multiple moving spheres. *Proc. 13th Annu. ACM Sympos. Comput. Geom.*, pp. 373–375. 1997.

[36] R. Klein. *Concrete and Abstract Voronoi Diagrams.* Lecture Notes Comput. Sci. 400. Springer-Verlag, 1989.

[37] R. J. Lang. A computational algorithm for origami design. *Proc. 12th Annu. ACM Sympos. Comput. Geom.*, pp. 98–105, 1996.

[38] D. T. Lee. Medial axis transformation of a planar shape. *IEEE Trans. Pattern Anal. Mach. Intell.* PAMI-4:363–369, 1982.

[39] S. Lisberger, director. *Tron.* Walt Disney Productions, 1982. Motion picture, 96 minutes.

[40] J. Matoušek. Reporting points in halfspaces. *Comput. Geom. Theory Appl.* 2(3):169–186, 1992.

[41] J. Matoušek. Range searching with efficient hierarchical cuttings. *Discrete Comput. Geom.* 10(2):157–182, 1993.

[42] M. McAllister, D. Kirkpatrick, and J. Snoeyink. A compact piecewise-linear Voronoi diagram for convex sites in the plane. *Discrete Comput. Geom.* 15:73–105, 1996.

[43] N. Megiddo. Applying parallel computation algorithms in the design of serial algorithms. *J. ACM* 30:852–865, 1983.

[44] B. Mirtich and J. Canny. Impulse-based simulation of rigid bodies. *Symposium on Interactive 3D Graphics.* ACM Press, 1995.

[45] C. Ó'Dúnlaing and C. K. Yap. A "retraction" method for planning the motion of a disk. *J. Algorithms* 6:104–111, 1985.

[46] A. Recuaero and J. P. Gutiérrez. Sloped roofs for architectural CAD systems. *Microcomputers in Civil Engineering* 8:147–159, 1993.

[47] V. Srinivasan, L. R. Nackman, J.-M. Tang, and S. N. Meshkat. Automatic mesh generation using the symmetric axis transform of polygonal domains. *Proc. IEEE* 80(9):1485–1501, Sept. 1992.

[48] J. Stolfi. *Oriented Projective Geometry: A Framework for Geometric Computations.* Academic Press, New York, NY, 1991.

[49] J. A. Storer and J. H. Reif. Shortest paths in the plane with polygonal obstacles. *J. ACM* 41(5):982–1012, 1994.

[50] A. Sudhalkar, L. Gursoz, and F. Prinz. Box-skeletons of discrete solids. *Comput. Aided Design* 28:507–517, 1996.

[51] T. K. H. Tam and C. G. Armstrong. 2D finite element mesh generation by medial axis subdivision. *Advances in Engineering Software and Workstations* 13(5–6):313–324, Sept. 1991.

[52] P. J. Vermeer. *Medial Axis Transform to Boundary Representation Conversion.* Ph.D. thesis, CS Dept., Purdue University, West Lafayette, Indiana 47907-1398, USA, 1994.