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

David Eppstein† Jeff Erickson‡

Submitted to *Discrete & Computational Geometry*: July 1, 1998
Revised and resubmitted: March 28, 1999

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 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 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.

---

*An extended abstract of this paper was presented at the 14th Annual ACM Symposium on Computational Geometry [35]. See http://www.uiuc.edu/ph/www/jeffe/pubs/cycles.html for the most recent version of this paper.

†Department of Information and Computer Science, University of California, Irvine; 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, and Department of Computer Science, University of Illinois, Urbana-Champaign; jeffe@cs.uiuc.edu; http://www.uiuc.edu/ph/www/jeffe. Research partially supported by NSF grant DMS-9627683 and by U. S. Army Research Office MURI grant DAAH04-96-1-0013.

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 [16, 18, 64], mesh generation [40, 58, 61, 62], motion planning [55, 60], and computer-aided manufacturing [41, 42, 61]. 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 [61], rectilinear Voronoi diagrams [61], or other techniques [17, 51]. 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 either closest points or Voronoi diagrams, there is a third definition that 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 locus traced out by a subset of these breakpoints [41]. 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 is 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, 56], 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). Straight skeletons and related structures have also been used in origami constructions [14, 28, 46].

Several theoretically and practically efficient algorithms are known for constructing medial axes [25, 29, 41, 47]. Unlike the medial axis, however, the straight skeleton cannot be defined by a distance measure or as an abstract Voronoi diagram [45] (except in a few special cases, which we describe in Section 4); consequently, its construction is considerably more difficult. The fastest

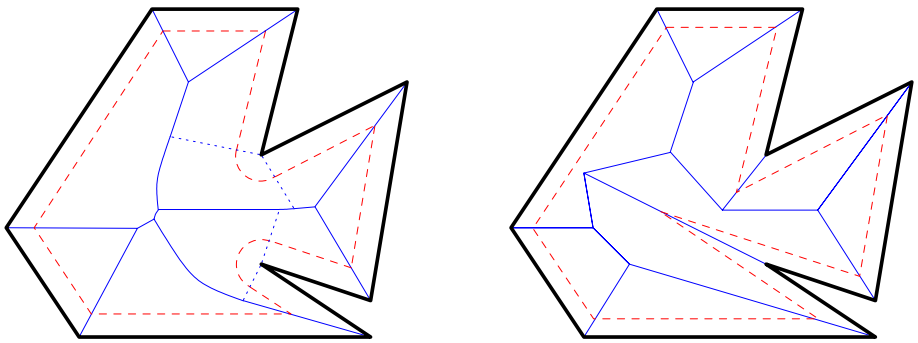

Figure 1. Skeletons and offset curves of a simple polygon. (a) The medial axis (solid), the Voronoi diagram (solid and dotted), and a rounded offset curve. (b) The straight skeleton and a mitered offset polygon.

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 size $O(nr)$, which we describe in Section 2. 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 [32, 33] 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 is first hit by a query ray.

(2) Maintain a changing set of rays in $\mathbb{R}^3$ and answer queries asking for the lowest (minimum z-coordinate) intersection of any ray with a query triangle.

Standard range-searching techniques let us 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 $\mathbb{R}^3$. Despite the fact that we cannot completely determine these triangles without constructing the straight skeleton, we can exploit the overall structure of the set of triangles to obtain a faster "reflex-sensitive" algorithm that runs in time and space $O(n^{1+\varepsilon} + n^{8/11+\varepsilon} r^{9/11+\varepsilon}) = O(n^{17/11+\varepsilon})$. 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 sketch 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 algorithms are based on maintaining a set of features of a shrinking mitered offset polygon. As the polygon shrinks, its vertices and edges collide, and these interactions change the polygon's structure. We describe these interactions in detail in Section 4. Our algorithms maintain the set of polygon features in a data structure that allows us to find each successive interaction quickly.

We formalize this approach as follows. Let R and B be dynamic sets of objects, and let $d\colon R \times B \to \mathbb{R}$ be an arbitrary "distance" function that can be computed in constant time. Our algorithms require a data structure that efficiently maintains the "closest" pair of objects $r \in R$ and $b \in B$ minimizing $d(r, b)$ as objects are inserted and deleted. In our straight skeleton application, R and B are the vertices and edges of the shrinking offset polygon, respectively, and the function d describes the time at which some pair of features collides. In this setting, the closest pair gives us the earliest interaction.

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 most efficient algorithms use a data structure of Eppstein [32] that maintains the closest pair using a dynamic data structure for minimization queries as a black box.

Theorem 2.1 (Eppstein [32, 33]). *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.*

Eppstein previously used Theorem 2.1 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 [32]. This technique generalizes and improves previous results of Bentley and Saxe [15] (for insertions only), Vaidya [63] (where only one set permits deletions), Eppstein [34] (for offline insertions and deletions), and Dobkin and Suri [30] (where each object's deletion time is given when it is inserted). The original statement of the theorem [32] required $T(n)$ to be a worst-case bound on the query and update times for the minimization query structure, but the proof only requires $T(n)$ to be an amortized time bound. Conversely, the amortized bounds given by the theorem can almost certainly be made worst-case, but this is unnecessary for our results.

A simplified version of this data structure for maintaining closest pairs from non-geometric 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 [33]. The companion paper also contains a special case (with $r = n$) of the following result, which we use as part of a theoretically slower but more practical straight skeleton algorithm.

Theorem 2.2. *Suppose* $|R| \leq r$ *and* $|B| \leq n$ *for some* $r \leq n$*. We can maintain the closest pair between* R *and* B *using* $O(rn)$ *space and preprocessing time, in* $O(n)$ *time per insertion or deletion in* R*, or* $O(r + \log n)$ *time per insertion or deletion in* B*.*

Proof: In a nutshell, we construct a quadtree-like decomposition of the distance matrix of the objects, by recursively subdividing it into four equal-sized blocks, and maintain the minimum distance within each cell of the "quadtree". We update the distances within each cell by looking at each of its four children.

In more detail, assume without loss of generality that r and b are powers of two. At all times, we maintain the $r \times n$ matrix D of distances between objects in R and objects in B. If necessary, we pad the matrix with $r - |R|$ "dummy" rows and $n - |B|$ "dummy" columns, all of whose entries are infinite. These dummy rows and columns can appear anywhere in the matrix. To delete an object, we replace all the entries in the corresponding row or column by ∞. Similarly, to insert a new object, we fill in the entries in an arbitrary dummy row or column. Thus, each insertion into or deletion from R changes an entire row of D, and each insertion into or deletion from B changes an entire column. It remains to show how to maintain the minimum element in D after every entry in some row or column has been replaced.

First suppose $r > 1$. We partition R and B each into two equal-sized subsets $R_1 \cup R_2$ and $B_1 \cup B_2$. Let D_{ij} denote the submatrix of distances between R_i and B_j. After a row or column of D changes, we recursively update the minima of exactly two of the four submatrices D_{ij}, and then recompute the global minimum in constant time by comparing the minima of all four submatrices.

On the other hand, if $r = 1$, we can only split B into equal sized subsets $B_1 \cup B_2$. Let D_i denote the submatrix of distances between R and B_i. After a column of D (*i.e.*, a single entry) changes, we recursively update the minimum of either D_1 or D_2. After a row of D (*i.e.*, the entire matrix) changes, we recursively recompute the minima of both D_1 and D_2. Finally, in either case, we compare the two sub-minima to compute the new minimum of the entire matrix in constant time.

Let $S(r, n)$ denote the space and preprocessing time for the overall data structure, $T_R(r, n)$ the update time after a change to R, and $T_B(r, n)$ the update time after a change to B. We have the following recurrences.

$S(1,1) = 1$	$T_R(1,1) = 1$	$T_B(1,1) = 1$	
$S(1,n) = 1 + 2S(1,n/2)$	$T_R(1,n) = 1 + 2T_R(1,n/2)$	$T_B(1,n) = 1 + T_B(1,n/2)$	if $n > 1$
$S(r,n) = 1 + 4S(r/2,n/2)$	$T_R(r,n) = 1 + 2T_R(r/2,n/2)$	$T_B(r,n) = 1 + 2T_B(r/2,n/2)$	if $r, n > 1$

The solutions $S(r, n) = O(rn)$, $T_R(r, n) = O(n)$, and $T_B(r, n) = O(r + \log(n/r)) = O(r + \log n)$ follow easily by induction. □

3 Lowest Intersection Queries

We now consider a dynamic range searching problem that arises as a subproblem in our straight skeleton algorithm: maintain a set of rays in $\mathbb{R}^3$ so that given a query triangle, we can quickly find the ray-triangle intersection point with smallest z-coordinate. 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.

3.1 Range Searching Techniques

Throughout this section, we use a number of standard techniques to combine and modify geometric range searching data structures. For example, we describe several data structures that allow continuous tradeoffs between space and query time. These space-time tradeoffs are obtained by merging portions of two data structures, one with linear size but large query time, the other with much larger size but only polylogarithmic query time. In the interest of brevity, we explicitly state only the query time of the combined data structure in the form $O(n^{a+\varepsilon}/s^b)$, where s is the size of the data structure and a and b will be explicit constants. The stated query time implies the following bounds on space, preprocessing time, and update time.

- The size s can be chosen anywhere between $\Omega(n)$ and $O(n^{a/b})$. Thus, the query time can range from $O(n^{\varepsilon})$ to $O(n^{a-b+\varepsilon})$.
- The data structure can be constructed in time $O(s^{1+\varepsilon})$.
- The data structure supports online insertions and deletions, each in amortized time $O(s^{1+\varepsilon}/n)$. Although in most cases we will cite papers that describe only static range searching data structures, these can be made dynamic using standard techniques [2, 5, 15, 49].

We also use the standard technique of composing several data structures into a single *multi-level* data structure. This technique allows us to decompose complicated query ranges into simpler components and devise independent data structures for each component. (For example, we can decompose any two-dimensional rectangular query into a pair of one-dimensional interval queries.) Under some mild technical assumptions [1, 50], which are satisfied by all the data structures used in this paper, we can cascade the component structures into a single data structure that supports the original complex queries. The size (resp. query time) of a multi-level structure is the size (resp. query time) of its largest (resp. slowest) component, times a factor of at most n^{ε}.

For further details on space-time tradeoffs, dynamization, multi-level data structures, and other geometric range searching techniques, see the survey by Agarwal and Erickson [1].

3.2 Lowest Intersection Queries

Our data structure for lowest intersection queries follows the same pattern used to construct ray shooting data structures for halfplanes and triangles in $\mathbb{R}^3$ [3, 4, 5, 13].

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: It suffices to construct a data structure that supports queries asking whether any of the rays intersects a query *quadrilateral*—the portion of the original query triangle that lies below some horizontal plane. To answer a lowest intersection query using this data structure, we apply parametric search [52] or one of Chan's recent randomized reduction techniques [19, 20] 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 techniques, although much simpler than parametric search, lead only to expected time bounds.

To detect intersections, we preprocess the rays into a multi-level data structure. The first level is a halfspace range searching data structure of Matoušek [50] that lets us (implicitly) find the rays intersecting the plane containing the query quadrilateral q, in time $O(n^{1+\varepsilon}/s^{1/3})$. This reduces the problem to detecting intersections between q and a set L 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 and 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 [22, 59].

To determine whether any line intersects q, we use a four-level data structure, one for each edge of q. Each of the first three levels is used to find the lines in L oriented correctly with respect to one of the lines ℓ_i. We use a data structure of Agarwal and Matoušek [4] that 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.* [22] describe a data structure that supports such queries in time $O(n^{1+\varepsilon}/s^{1/2})$. □

3.3 Fixed-Slope Lines and Nice Triangles

The most time-consuming part of answering a lowest-intersection query is answering *line queries*—which lines in a set L are oriented positively with respect to a query line ℓ? Line queries are 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 (Section 4.3 and Section 5), every ray has the same slope, or every triangle has edges with the same slope, or both. 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 3-space.[1] 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 lines, all with the same slope, 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 among 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.

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

[1] For any $\alpha > 0$, the set of lines in $\mathbb{R}^3$ with slope α is homeomorphic to $\mathbb{R}^2 \times S^1$. The set of horizontal lines ($\alpha = 0$) is homeomorphic to $\mathbb{RP}^2 \times \mathbb{R}$, and the set of vertical lines ($\alpha = \infty$) is just $\mathbb{R}^2$.

Conversely, preprocessing a set of arbitrary lines for fixed-slope line queries (that is, every query line has the same slope, which is specified in advance) is equivalent to preprocessing a set of algebraic surfaces in $\mathbb{R}^3$ for point location queries. Chazelle *et al.* [21] 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 preprocessed lines and the slope of the query lines could be different.)

Say that a set of triangles is *nice* if two edges of every triangle have fixed (but possibly different) slopes. That is, for some constants α and β, every triangle in the set has one edge with slope α and another with slope β. By replacing the general line query data structure used in Theorem 3.1 with our fixed-slope data structures, we obtain the following result.

Theorem 3.4. *Given a set of n rays in $\mathbb{R}^3$, we can answer lowest intersection queries for nice triangles in $O(n^{9/8+\varepsilon}/s^{3/8})$ time, or $O(n^{1+\varepsilon}/s^{1/3})$ time if every ray has the same slope.*

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 fixed-slope data structures into theirs, we obtain faster algorithms for ray shooting among nice triangles.

Theorem 3.5. *Given a nice set of n triangles in $\mathbb{R}^3$, we can answer ray shooting queries in $O(n^{8/9+\varepsilon}/s^{2/9})$ time, or in $O(n^{1+\varepsilon}/s^{1/3})$ time if the slope of the query rays is fixed.*

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 (although some care must be taken with vertices of degree less than two) [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.

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, maintaining sharp corners at each of the reflex vertices. If 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

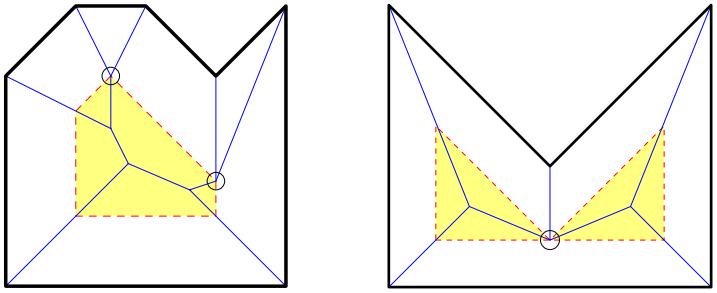

Figure 2. (a) Two simultaneous edge events. (b) A split event.

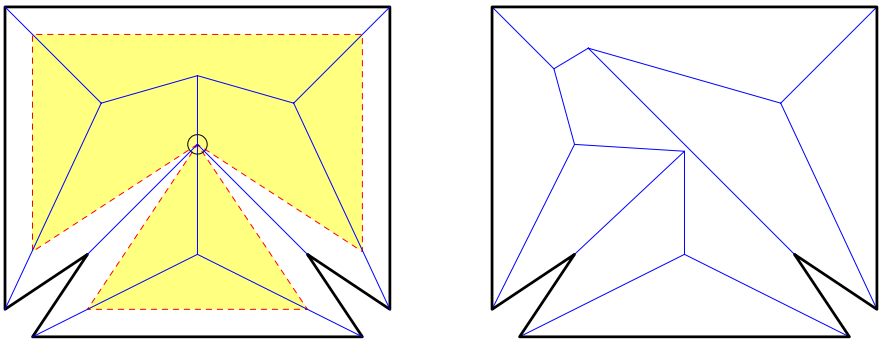

Figure 3. (a) A vertex event. (b) Perturbing a degenerate polygon can radically alter its skeleton.

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. In most cases, we can handle these events one at a time using standard perturbation techniques[2], replacing the high-degree node with several nodes of degree three, connected by zero-length edges. 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 3(a). 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, our algorithms must handle vertex events directly. See Figure 3(b).

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. (Aichholzer *et al.* call this process "flooding" [8].) The key observation is that although we do not

[2]In fact, we can usually process simultaneous events in arbitrary order.

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 the sweep plane hits the intersection of a ray and 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 z-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 insertions or deletions 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 support queries, insertions, and deletions, each 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 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 planar 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

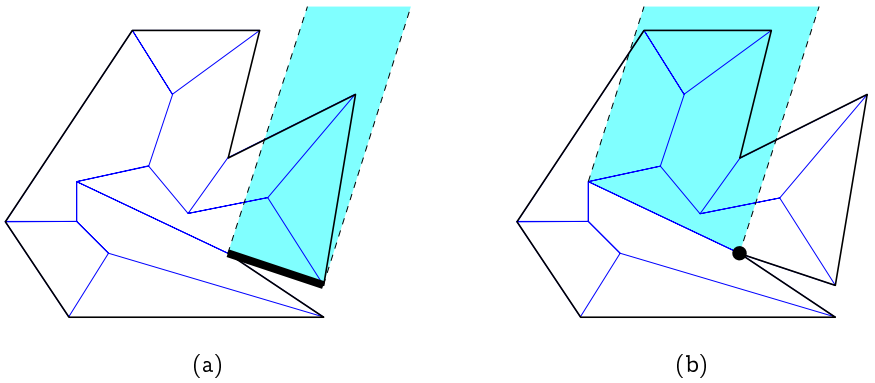

Figure 4. (a) Top view of an edge slab. (b) Top view of a reflex slab.

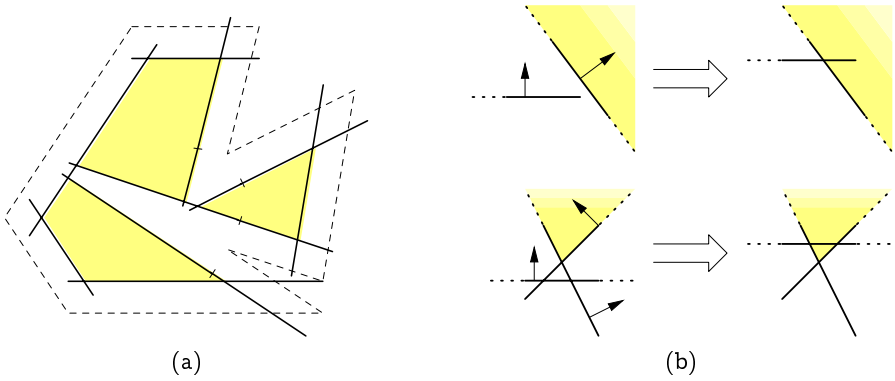

Figure 5. (a) A cross section of the slabs, after two split events. (b) Two impossible transitions; see Lemma 4.2.

the edge of the roof induced by v, and bounded on the sides by rays perpendicular to either e or e'. See Figure 4.

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 5(a). 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 5(b). Since all the segments and their endpoints are moving at the same speed, neither of these transitions can occur. □

Aichholzer *et al.* [8] show that the roof cannot be expressed as the lower envelope of partial linear functions, where the domain of each function is locally defined by a small neighborhood of

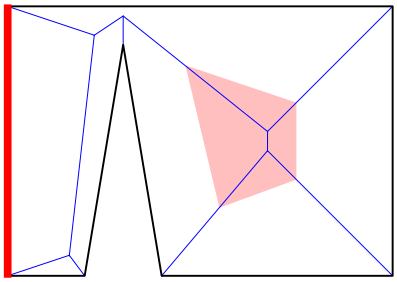

Figure 6. The roof of a weighted polygon is not the lower envelope of its slabs. The leftmost edge has higher weight than the other edges, and its edge slab cuts through the roof in the shaded region.

an edge. Our reflex slabs are not locally defined, since the positions of their innermost edges may depend on the entire sequence of vertex-edge collisions. We emphasize that Lemma 4.2 holds only for unweighted straight skeletons. Counterexamples for the weighted case are easy to construct; see Figure 6.

The advantage of this lower envelope formulation is that unlike the triangles used in our earlier algorithm, the slabs form a nice set of triangles—two of their edges have slope 1—so we can use the more efficient data structures in Theorems 3.4 and 3.5 to answer lowest intersection and ray shooting queries.

A split event occurs when the ray induced by some reflex vertex of the polygon hits one of the polygon's slabs. 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. 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 the ray shooting data structure described by Theorem 3.5, and for each ray, we find the first edge slab that it hits. If we build a data structure of size $s = O(\max\{n, n^{8/11}r^{9/11}\})$, then the total time to preprocess the n edge slabs and answer the r ray shooting queries is $O(n^{1+\varepsilon} + n^{8/11+\varepsilon}r^{9/11+\varepsilon})$. This gives us a superset of the possible edge split events of size r, which we then sort chronologically in $O(r \log r)$ time.

To construct the straight skeleton, we simultaneously maintain three different data structures, one for each type of event. We use a simple priority queue to maintain the next edge event as in our earlier algorithm. To maintain the next edge split event, we scan the sorted list of potential events described in the previous paragraph. To maintain the next reflex split event, we use the following variant of our earlier algorithm. We build 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 innermost edge of each reflex slab is located. 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 the unbounded slabs also form a nice set of triangles, we can use the data structures described by Theorems 3.4 and 3.5. In both cases, by building a data structure of size $s = O(r^{17/11+\varepsilon})$, we can both answer queries and perform updates in $O(r^{6/11+\varepsilon})$ time. Thus, by Theorem 2.1 we can maintain the next reflex split in time $O(r^{6/11+\varepsilon})$ per split event. Since there are at most r split events, the total time spent maintaining the next reflex split is $O(r^{17/11+\varepsilon})$. This is dominated by the time spent finding edge splits, since $r \leq n$.

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

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.

In nondegenerate cases, we can reduce the space requirement to $O(n + r^{17/11+\varepsilon})$ using the *streaming* technique of Edelsbrunner and Overmars [31]. Instead of constructing an online data structure and performing individual ray-shooting queries to find the potential edge splits, we can perform an implicit depth-first search of the data structure, simultaneously preprocessing the n edge slabs and performing the r queries. However, if the polygon is degenerate, we may need to answer new ray-shooting queries among the edge slabs online, so we cannot use streaming in that case.

4.4 Practical Variants and Extensions

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 linear-space 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 split events, instead of Theorem 2.1, we obtain practical algorithms for constructing straight skeletons that still improve on previous results. For example, using the algorithm in Section 4.2, we obtain a practical algorithm for *weighted* skeletons that runs in $O(n \log n + nr)$ time using $O(nr)$ space. the algorithm spends $O(n \log n)$ time maintaining the priority queue for edge events. Maintaining the next split event requires $O(nr)$ time to initialize the "distance" matrix, $O(r + \log n)$ time for each of the n edge events, plus $O(n)$ time for each of the r split events.

In the unweighted case, we can improve either the space or the time bounds by using the reflex-sensitive algorithm of Section 4.3. If we find potential edge splits by brute force in $O(nr + r \log r)$ time and use the matrix decomposition to maintain reflex splits, our algorithm runs in $O(n \log n + nr)$ time and uses only $O(n + r^2)$ space. Alternately, if we use Aichholzer and Aurenhammer's triangulation-based algorithm [7] to find reflex splits, we obtain a practical algorithm that runs in $O(n \log n + nr + r^2 \log r)$ time using only $O(n)$ space. All of these algorithms improve the previously best time bound $O(nr \log n)$, obtained by sorting the list of all $O(nr)$ potential split events [8].

We can also improve our algorithms for certain special classes of polygons. 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

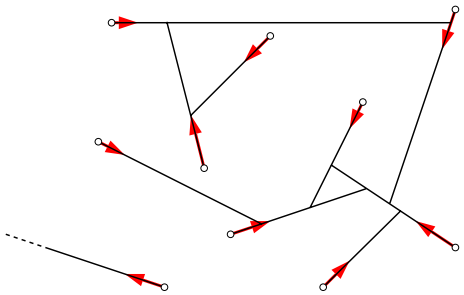

Figure 7. A motorcycle graph.

families of slabs. If $c = 2$, the straight skeleton is actually the medial axis of the polygon in a metric whose unit circle is a rhombus; in particular, the straight skeleton of an orthogonal polygon is its L_∞ medial axis [24]. In this case, we can compute the skeleton in $O(n \log^* n)$ expected time using a randomized incremental algorithm [29] or in $O(n \log n)$ worst-case time using a divide-and-conquer algorithm [47]. Finally, if the input polygon is convex, the straight skeleton is identical to the (Euclidean) medial axis; consequently, we can construct it in linear time [6, 23].

Once we compute the straight skeleton, we can compute any desired mitered offset polygon from it in linear time. Alternately, we can compute offset polygons directly by halting the inward sweep when we reach the desired offset; although this approach is likely to be more efficient in practice, it does not reduce the worst-case time bound, since we cannot predict in advance how many events will occur before 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

Oh man... when you're on the other side of the screen, it all looks so easy!
— Jeff Bridges as Kevin Flynn, "Tron" (1982)

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 7. The problem is to construct this graph as quickly as possible. A similar problem was previously considered by Lisberger *et al.* [48].

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* introduced by Czyzowicz *et al.* as a tool to solve certain art gallery problems [27]; see also [9]. 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 (every directed path either goes off to infinity or ends in a directed cycle) 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, which we describe at the end of this section. Aichholzer and Aurenhammer's straight skeleton algorithms [7, 8] can easily be adapted to construct motorcycle graphs in $O(n^2 \log n)$ time and linear space. The time bound can be improved to $O(n^2)$, at the expense of quadratic space, using Theorem 2.2. Our methods yield the first 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 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, the curtains comprise a nice set of triangles. Thus, 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 used in this algorithm. De Berg *et al.* [13] describe a data structure that supports ray shooting queries among curtains with query and update time $O(n^{1/3+\varepsilon})$. It suffices to store the vertical strip consisting of points directly above *or below* each dead track, instead of three-sided curtains; this observation 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 are moving at the same speed (or at only a constant number of different speeds), each bike's ray lies on a line of fixed slope, so we can answer lowest-intersection queries in time $O(n^{1/2+\varepsilon})$ using the second half of Theorem 3.4. It follows that we can build the motorcycle graph in $O(n^{3/2+\varepsilon})$ time and space in this case. If the bikes move in only a constant number of directions, similar techniques allow us to compute the motorcycle graph in 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 to answer the ray-shooting and lowest-intersection queries.

Finally, if all the velocity vectors have positive x-coordinates, we can use a simple sweep-line algorithm to compute the motorcycle graph in $O(n \log n)$ time. Note that in this case, the motorcycle graph is acyclic. Sweep a vertical line across the motorcycle graph from left to right,

maintaining the current cross-section of the motorcycle graph in a balanced binary tree. This cross-section can change combinatorially in only two ways: *birth*, when the sweepline passes the initial location of a motorcycle, and *death*, when the sweepline passes the crossing point of two tracks that are adjacent along the sweepline and one of the two motorcycles crashes. We keep these events in a priority queue, sorted by their x-coordinates. Initially, the event queue contains only the n births. To handle a birth, we insert the new track into the binary tree, delete the death event (if any) caused by its two neighbors, and insert two new death events. To handle a death, we determine which of the two motorcycles dies, delete it from the binary tree, delete its other death event, and insert a new death event between its two neighbors. There are exactly n births and at most n deaths, and each event is handled in $O(\log n)$ time. Extending this algorithm to deal with head-on collisions and bikes that move directly up or down is straightforward. Even though it may take $O(n \log n)$ time to move the sweep line past a single vertical bike, this time can be amortized across the other bikes that it kills.

Unlike all the other algorithms we describe, this sweepline algorithm does not necessarily process events in chronological order; the leftmost crash may not be the earliest crash. Even in the general case, it is not necessary to process events chronologically, as long as any two events that involve the same motorcycle (or in the case of straight skeletons, the same edge or vertex) are processed in the correct order. Whether this observation can be used to improve our other algorithms is an interesting open question.

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. To simplify the problem, we assume that the balls lie on an infinite plane rather than a finite table, and we ignore the effects of friction and spin. Several variations of this problem have been surveyed by Shamos [57].

The trivial solution is to repeatedly check every pair of balls to find each successive collision; this requires $O(n^2)$ time per collision. Very few other algorithms are known with theoretical guarantees of any kind. 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, 39].) In the worst case, however, the closest pair changes $\Theta(n^2)$ times without a single collision, which makes the kinetic approach less efficient than the trivial algorithm in the worst case. Kim *et al.* [44] 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 [53]. 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, 26, 36, 43]. Almost none of these have theoretical guarantees on their performance. In fact, like the algorithms in [44, 53], most perform well only in dense environments.

Using geometric range searching techniques, we can find the first collision in subquadratic time.

Our method allows us to save almost a linear factor in the time to find the second and succeeding collisions, giving us 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 sketch: 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. Using a technique similar to Mohaban and Sharir's algorithm for ray shooting among spheres [54], we express each cylinder ray shooting query as a composition of several four- and five-dimensional semialgebraic range queries, which we can answer 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 unit balls and stationary line segments undergoes a finite number of collisions is PSPACE-complete, since a polynomial-space Turing machine can be simulated by such a system [37].)

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 [38]. A simple modification of either proof shows that construction of motorcycle graphs is also P-complete, even when the motorcycles move only south, east, or southeast; see the Appendix for details. (Recall that 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 seems to more closely resembles a Voronoi diagram. Can we construct the straight skeleton more quickly if every reflex angle is obtuse, or more generally, if every reflex angle is bigger then some constant? If the smallest reflex angle is bounded, then so is the ratio between the speed of any reflex vertex and the speed of the edges. 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. *Advances in Discrete and Computational Geometry*, B. Chazelle, J. E. Goodman, and R. Pollack, editors, pp. 1–58. Contemporary Mathematics 223, Amer. Math. Soc., 1999.

[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-Verlag, 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, 1995.

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

[12] J. Basch, L. J. 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] M. Bern, E. Demaine, D. Eppstein, and B. Hayes. A disk-packing algorithm for an origami magic trick. To appear in *Proc. Int. Conf. Fun with Algorithms*, 1998. ⟨http://daisy.uwaterloo.ca/~eddemain/papers/FUN98/⟩.

[15] J. Bentley and J. Saxe. Decomposiable searching problems I: Static-to-dynamic transformations. *J. Algorithms* 1:301–358, 1980.

[16] 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.

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

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

[19] T. M. Chan. Fixed-dimensional linear programming queries made easy. *Proc. 12th Annu. ACM Sympos. Comput. Geom.*, pp. 284–290. 1996.

[20] T. M. Chan. Geometric applications of a randomized optimization technique. *Proc. 14th Annu. ACM Sympos. Comput. Geom.*, pp. 269–278. 1998.

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

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

[23] 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.

[24] P. Chew. personal communication, 1998.

[25] 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.

[26] 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.

[27] 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.

[28] E. D. Demaine, M. L. Demaine, and A. Lubiw. Folding and cutting paper. To appear in *Proc. Japan Conference Discrete Comput. Geom.* Lecture Notes Comput. Sci., Springer-Verlag, 1999. ⟨http://daisy.uwaterloo.ca/~eddemain/papers/JCDCG98/⟩.

[29] 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.

[30] D. Dobkin and S. Suri. Maintenance of geometric extrema. *J. ACM* 38:275–298, 1991.

[31] H. Edelsbrunner and M. H. Overmars. Batched dynamic solutions to decomposable searching problems. *J. Algorithms* 6:515–542, 1985.

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

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

[34] D. Eppstein. Offline algorithms for dynamic minimum spanning tree problems. *J. Algorithms* 13:111-122, 1995.

[35] D. Eppstein and J. Erickson. Raising roofs, crashing cycles, and playing pool: Applications of a data structure for finding pairwise interactions. *Proc. 14th Annu. ACM. Sympos. Comput. Geom.*, pp. 58–67. 1998.

[36] 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.

[37] 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.

[38] D. Griffeath and C. Moore. Life Without Death is P-complete. *Complex Systems* 10(6):437–447, 1997.

[39] L. J. Guibas. Kinetic data structures: A state of the art report. *Robotics: The Algorithmic Perspective (Proc. 1998 WAFR)*, pp. 191–210. A. K. Peters, 1998.

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

[41] M. Held. Voronoi diagrams and offset curves of curvilinear polygons. *Comput. Aided Design* 30(4):287–300, 1998.

[42] 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, 1994.

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

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

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

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

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

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

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

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

[51] 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.

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

[53] B. Mirtich and J. Canny. Impulse-based simulation of rigid bodies. *Symposium on Interactive 3D Graphics*, pp. 181–188. ACM Press, 1995.

[54] S. Mohaban and M. Sharir. Ray shooting amidst spheres in three dimensions and related problems. *SIAM J. Comput.* 26:654–674, 1997.

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

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

[57] M. I. Shamos. *The Illustrated Encyclopedia of Billiards.* Lyons and Burford Publishers, 1993.

[58] 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, 1992.

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

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

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

[62] 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.

[63] P. Vaidya. Geometry helps in matching. *SIAM J. Comput.* 18:1201–1225, 1989.

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

Appendix: Motorcycle Graphs Are P-Complete

In this appendix, we prove the following theorem.

Theorem A.1. *Constructing motorcycle graphs is P-complete under LOGSPACE reductions.*

Proof: We use a reduction from the CIRCUIT VALUE problem: Given a boolean circuit and an input vector, compute the output of the circuit, a single boolean value. In fact, we can restrict our attention to *gate arrays*, in which the gates are arranged in an $n \times n$ grid and each gate can only receive signals from its northwest, north, and northeast neighbors. Signals proceed through a gate array level by level from top to bottom. Since we can encode n computation steps of an arbitrary n-cell Turing machine by a gate array, this special case of CIRCUIT VALUE is still P-complete.

We closely follow the reductions of Atallah *et al.* [9] from circuits to weighted planar partitions and of Griffeath and Moore [38] from circuits to initial configurations of a certain two-dimensional cellular automaton. We represent each wire in the circuit by a collection of *tracks*. The presence or absence of a motorcycle on a track represents a signal of 1 or 0, respectively. Since a bike can only be at one point on its track at any moment, we can think of the signals themselves as moving along the tracks.

Our transformation uses two basic primitives. The first is a *stopper*, which is a fixed barrier used to stop a signal. We can construct a stopper using four closely spaced motorcycles that collide cyclically, as in Figure 8(a). The second primitive is the *blocking collision*, based on the observation that if two tracks intersect, only one bike can pass through the intersection point. Thus, if signal x arrives at an intersection point after signal y, then it is transformed into the signal $x \wedge \bar{y}$; see Figure 8(b). In all of our figures, the track that appears to be "in front" of each intersection point carries the earlier signal. Using these primitives, we can easily construct NOT, AND, OR, and fanout gates, as well as gadgets to turn a signal, delay a signal, and to allow two signals to cross. These are illustrated in Figure 8(c)–(e).

To perform the reduction, we replace each gate, turn, and wire crossing in the gate array with the corresponding gadget. We also introduce delays between levels so that all signals enter each level of the gate array at the same time. Each input is represented by the presence or absence of a motorcycle heading downwards into the array from above. (Alternately, we can encode each input by the direction or speed of a motorcycle, so that only "true" motorcycles enter the circuit proper.) A single output motorcycle leaves the array downwards if and only if the output of the circuit is 1; otherwise, every bike crashes.

Each gadget requires a constant number of motorcycles, so there are $O(n^2)$ bikes overall. Each gadget takes up only constant area, so the coordinates of each bike's initial location are integers between 0 and $O(n)$. The motorcycles starting inside each gadget must be slower than the input signals in order for the collisions to happen in the right order; however, since each bike travels only a constant distance, only a constant slowdown is required. Thus, we can arrange for the inputs to gates at level i to arrive at time t^i and travel at speed s^{n-i}, for some constants $s, t > 1$. It follows that if the output signal travels at unit speed, the speed of any motorcycle is singly-exponential in n, so we can represent the bike velocities using a polynomial number of bits.

The entire reduction can clearly be carried out using only logarithmic space, since each gadget can be constructed independently. □

With a little extra work, we can force every motorcycle in the output of our reduction to travel south, east, or southeast. First, replace each delay gadget by a pair of inverters. In order to use

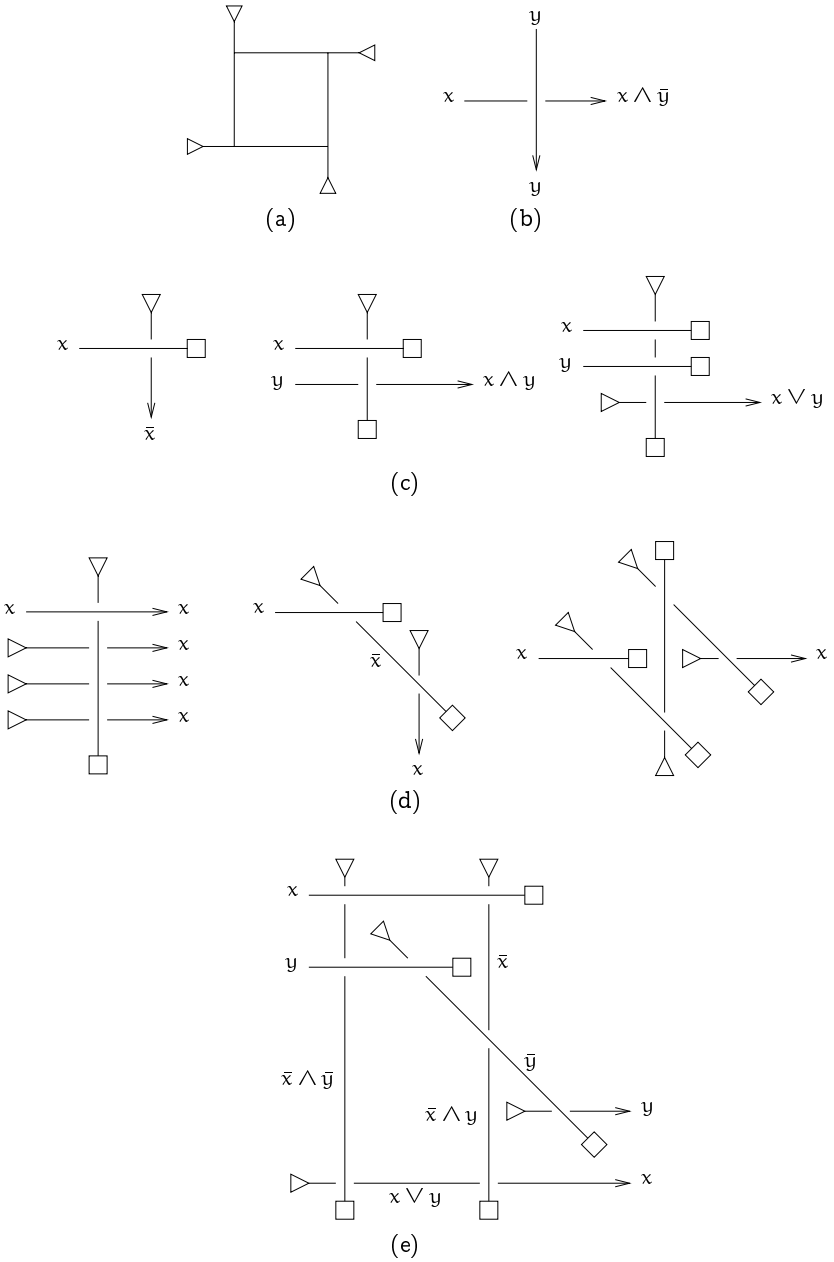

Figure 8. Reducing a boolean circuit to a motorcycle graph. Triangles denote the starting positions and directions of motorcycles. (a) A stopper, hereafter drawn as a square. (b) The effect of a potential collision. (c) NOT, AND, and OR gates. (d) Fanout, bend, and delay gadgets built out of inverters. (e) A crossover gadget.

these revised gadgets, each row of the array must be shifted to the right by a constant distance. Second, replace each stopper by a single motorcycle traveling either south or east, close enough to block the incoming track, but slow enough that that the stopper bikes crash only after everything else. Since the resulting motorcycle graph is acyclic, at least one bike always survives. With some care, we can ensure that *exactly* one bike survives, traveling south or east if the output of the original circuit is 1 or 0, respectively. We conclude:

Theorem A.2. *Constructing motorcycle graphs is P-complete under LOGSPACE reductions, even if the motorcycles move only east, south, or southeast.*