Data Structures for Mobile Data

Julien Basch *Leonidas J. Guibas*
Computer Science Department
Stanford University
Stanford, CA 94305, USA
{jbasch,guibas}@cs.stanford.edu

John Hershberger
Mentor Graphics Corp.
8005 SW Boeckman Road
Wilsonville, OR 97070-7777, USA
john_hershberger@mentorg.com

Abstract

A *kinetic data structure* (KDS) maintains an attribute of interest in a system of geometric objects undergoing continuous motion. In this paper we develop a conceptual framework for kinetic data structures, propose a number of criteria for the quality of such structures, and describe a number of fundamental techniques for their design. We illustrate these general concepts by presenting kinetic data structures for maintaining the convex hull and the closest pair of moving points in the plane; these structures behave well according to the proposed quality criteria for KDSs.

1 Introduction

We present a set of novel data structures for the efficient maintenance of various continuous and discrete attributes of mobile data. For example, given n points moving continuously in the plane, we give methods for maintaining their convex hull or the separation of their closest pair. We call the combinatorial description of these attributes a *configuration function* of the mobile data[1]. Since motion is common with objects in the physical world, the examples we discuss in this paper come primarily from computational geometry and are motivated by problems

[1] By combinatorial description of the convex hull we mean the circular list of points forming the hull; the combinatorial description of the closest pair is just the identity of the two closest points.

like collision detection in robotics and animation, visibility determination in computer graphics, etc. Our techniques, however, are more generally applicable to the processing of discrete functions associated with any kind of continuously changing data. We call our data structures *kinetic*, to distinguish them from their more classical static or dynamic (in the other sense, as we explain below) counterparts, and we abbreviate the term "kinetic data structure" to KDS for short. We call *kinetization* the process of transforming an algorithm on static data into a data structure that is valid for continuously changing (moving) data.

The problems of convex hull and closest pair maintenance have been exhaustively studied in computational geometry [14, 16, 22, 25, 26, 29, 31], but almost exclusively in the context of static objects with operations like insertion and deletion. Our emphasis instead is on the maintenance of such configuration functions under *continuous motion* of the given objects. Though in principle the continuous motion of a single object can be approximated, after a discrete sampling of time, by deleting it and reinserting it at a new position at each time step, this method is clearly ill-adapted to our purposes and wasteful of computation. In particular, any fixed rate sampling of the evolving system will either oversample or undersample the system, as the events of interest to the configuration function typically occur in irregular patterns. The aim of our technique is to take full advantage of the coherence present in continuous motion so as to process a minimal number of combinatorial events, yet still maintain the configuration function correctly. In this respect, the way of analyzing our data structures is akin to the *dynamic computational geometry* framework introduced by Atallah [7] in order to study the number of combinatorially distinct configurations of a given kind (e.g., convex hull or closest pair) that arise during the continuous motion of geometric objects. Unlike Atallah's scheme, however, our data structures do not require us to know the full motion of the objects in advance. Thus they are better suited to real-world situations in which objects can change their motion on-line because of interactions with each other, external impulses, etc.

We assume that each moving object has a posted *flight plan* that gives full or partial information about its current motion. As mentioned above, flight plans can change. A flight plan *update* can occur because of interactions between our object and other moving objects, the environment, etc. For example, a collision between two moving objects will in general result in updates to the flight plans of both objects. The interface between our kinetic data structures and the object motions is through a global event queue. Thus our techniques most closely resemble plane sweep methods in computational geometry, except that in our case the dimension being swept over is time. A key aspect of our data structures is that we have a "narrow interface" to the motion. What we mean by this is that the kinds of events we have in our event queue correspond to possible combinatorial changes involving a constant (and typically small) number of objects each. For example, in the case of 2-D convex hull maintenance, one type of event we will use is "the points A, B, C become collinear" or, equivalently, "the triangle ABC reverses sign (orientation)". Indeed, it will turn out that the correctness of whatever configuration function we maintain can be guaranteed with a conjunction of such low-degree algebraic conditions involving a bounded number of objects each—we call these conditions the *certificates* of the KDS.

At any one time, our event queue will contain several KDS events corresponding to times when certificates might change sign. The times for these events are calculated using the posted flight plans of the objects involved. If, because of other events, the flight plan of an object is updated, then all certificates involving that object must be located and have their "sign change" time recalculated according to the new plan. In this way the event queue adapts to the evolving motions of the objects. We can deal in the same way with objects whose flight plan is only partially known. In our "sign of the triangle ABC" example above, given some partial bounds on the positions and velocities of the points A, B, C, we can easily calculate a time interval Δt during which we can be sure that the sign of ABC does not change. Thus we can schedule an event to occur after Δt time units, and at that point we can recheck the sign of ABC and proceed similarly (after updating our knowledge of the motions of the participating points). In general our philosophy will be that each moving object needs to be aware of all the events in the event queue that involve it and the validity assumptions about its motion on which these events are based. If the motion of the object changes so that any of these assumptions is no longer valid, then it is the responsibility of the object to take the steps necessary to have these events rescheduled at the times appropriate for its new motion. A general issue that arises here is how best to spend "sensing dollars" in order to acquire the information about the moving objects that is necessary to detect the events of interest for the KDS. We will not address this issue in this paper.

A key property of a kinetic data structure is that the certificates of the KDS form, at any time, a proof of correctness of the current configuration function. Their failure times are the only times at which the configuration function can possibly change. It is in this sense that the proof being maintained by the KDS guides the algorithm to examine the evolving system only at the set of relevant times—i.e., at those times at which the current proof may become invalid. When a certificate fails and the proof becomes invalid, the KDS needs to update the proof and possibly also the value of the configuration function.

We will analyze and evaluate a kinetic data structure in a number of different ways. For the analyses of KDSs we will assume that the instantaneous motions of the objects are known and are parameterizable by what we call *pseudo-algebraic* functions of time. These are functions with the property that each of the certificates involved in the kinetization changes sign at most a bounded number of times—very much in the spirit of situations in which Davenport-Schinzel sequences have been used [34] in computational geometry. Most obviously, a KDS is good if the cost of processing a certificate failure is small. In order to quantify this and the following measures, let n describe the complexity of the evolving system, for example the number of points moving in the plane in the convex hull and closest pair examples of the later sections. We will speak of a cost as being small if it is asymptotically of the order of $O(\text{Polylog}(n))$, or $O(n^\epsilon)$, for some small $\epsilon > 0$.

We call a KDS *responsive* if the worst-case cost of processing a certificate failure is small—this is the cost of discovering how to update the proof and (possibly) the configuration function; this in general will involve descheduling certain events (as some old certificates leave the proof) and scheduling some new events (as new certificates enter the proof).

A second key performance measure for a KDS is the worst-case number of events processed. We make a distinction between *external events*, i.e., those affecting the configuration function we are maintaining (e.g., convex hull or closest pair), and *internal events*, i.e., those processed by our structure because of its internal needs, but not affecting the desired configuration function. Our aim will be to develop kinetic data structures for which the total number of events processed by the structure in the worst case is asymptotically of the same order as, or only slightly larger than, the number of external events in the worst case (technically, we require that the ratio of total events to external events is small). This is reasonable, as the number of external events is a lower bound on the cost of any algorithm for maintaining the desired configuration. A KDS meeting this condition will be called *efficient*.

We define the *size* of a KDS to be the maximum number of events it needs to schedule in the event queue at any one time. We call a structure *compact* if its size is roughly linear in the number of moving objects.

Finally, we call a KDS *local* if, at any one time, the maximum number of events in the event queue that depend on a single object is small. This property is crucial for fast handling of flight plan updates[2].

To summarize, our kinetic data structures are different from classical dynamic data structures: though we can (and often want to) accommodate insertions and deletions, our focus is on continuous motion and not discrete modifications. We can use Atallah's framework of dynamic computational geometry to get lower bounds on the amount of work we have to do. But our structures are on-line and can be used to implement correct simulations even when the object flight plans change because of interactions between the objects themselves or the objects and the environment, or even when only partial information about the motions is available. Furthermore, we provide some general tools for the kinetization of static algorithms that lead to KDSs that are easy to analyze and perform well.

[2]We remark that locality implies compactness, but all other quality measures are independent.

1.1 An illustrative example

To make the issues above more concrete, let us consider the following simple 1-D situation. Given a set of points moving continuously along the y-axis, we are interested in knowing at all times which is the topmost point (the largest, if we think of the points as numbers). If two points meet, we allow them to pass each other without interaction. Suppose further that we know that the points are moving with constant velocities (but possibly a different one each), starting from an arbitrary initial configuration.

If we draw the trajectories of the points in the ty-plane (where the t axis is horizontal and denotes time), then our problem is nothing but computing the upper envelope of a set of straight lines in the plane (or at least the part of it that is after the initial time t_0). This upper envelope computation can be trivially done in $O(n \log n)$ time with a divide and conquer algorithm (this bound holds even if points can appear and disappear at arbitrary times, but then it is not trivial [24]). In the worst case, the number of times during the motion that the topmost point changes is $\Theta(n)$. Thus we have a method for computing the configuration function of interest in time that is only a logarithmic factor higher than the maximum number of changes in the configuration function itself.

For our purposes, however, this solution is unsatisfactory, because it is really based on knowing in advance the full motions of the points. What we seek is a strategy that works on-line and can accommodate flight plan updates. So suppose instead that we try to maintain the sorted order of our points along the y-axis, on-line. For every pair of points that are currently consecutive along the y-axis we schedule an event that is the first time when these points cross (or if, as above, our knowledge of the motions is incomplete, we schedule an event based on our estimate of how long we can be sure that the relative order of the points does not change). When two adjacent points meet, this destroys two old adjacencies and creates two new ones along the sorted list, as in the plane sweep algorithm of Bentley and Ottmann [13] modified by Brown [15]. Thus we deschedule (up to) two events and schedule (up to) two new events. In this process we always maintain the sorted list of points, and in particular we always know the topmost one as well. Unfortunately, although the kinetic data structure obtained is responsive, local, and compact, it may have to process $\Theta(n^2)$ events even when the points have the simple motion described above (imagine that half the points are stationary, and the other half pass over them). Thus the number of internal events here is an order of magnitude greater than those affecting the configuration function we are interested in—this solution is not efficient.

A third structure, and one that lets us meet all our objectives, is to maintain the moving points in a heap, with the root being the topmost (maximal) element. The kinetization of the heap is as follows. For each link in the heap, we have a certificate that guarantees that the child point is below the parent point, and an associated event at the time these points meet. To process an event and maintain a valid heap, it is enough to interchange the parent and child of the link associated with the event (Figure 1), as all the other heap inequalities are still valid at that time (here, we are making strong use of the continuity of the motions and of the hypothesis of non-degeneracy). When a swap of two elements happens in the heap, up to four adjacency (parent/child) relationships can change in the heap, so we may have to deschedule four events and reschedule four more. This describes our *kinetic heap*, which maintains the topmost element at all times. Again, responsiveness, locality, and compactness are immediate.

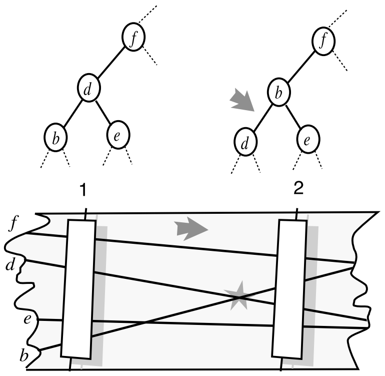

Figure 1: Maintenance of a kinetic heap: on the left, the link bd has an associated event. To process this event, it is enough to interchange b and d in the heap, and adjust the certificates and events that depend on them. When b and e meet, there is no event as there is no change in the heap structure.

But how many events does the kinetic heap have to process in the worst case, when the points move with constant velocities? This question turns out to be surprisingly non-trivial; we can show by a potential argument that the kinetic heap under linear point motions processes $O(n \log^2 n)$ events, and thus is a data structure meeting our requirements (the proof appears in [10]).

To prepare ourselves for the solutions to the other problems we will present below, let us also consider the following fourth solution to the kinetic maximum maintenance problem. Consider first an algorithm that computes the maximum of n (static) numbers. The algorithm computes the maximum recursively, by partitioning the numbers into two approximately equal-sized groups (arbitrarily), computing the maximum of each subgroup, and then comparing the two winners to select the final true maximum. If viewed from the bottom up, this is exactly a tournament for computing the global leader. In the end this algorithm has performed $O(n)$ comparisons that altogether prove that the maximum it computed is indeed the true maximum. Now imagine that our numbers start varying—our points can move. As long as each of the comparisons the algorithm made stays valid, the identity of the maximum element cannot change.

A general kinetization strategy we will use consists of taking the certificates of correctness in the computation performed by our static algorithm—the comparisons in this case—and associating with each of them an event in the global queue that describes when that certificate will (may) be violated in the future. When a violation happens, we hope that there will be a simple and efficient way to update the output of the algorithm and the set of certificates to be maintained. In our example, suppose that a particular comparison involved in the maximum computation flips. This comparison is between the leaders of two subgroups at a certain level of the tournament tree. If the winner changes, then this winner has to be percolated up the tournament tree, till it is either defeated or declared the overall maximum. But because a tournament tree is balanced, this computation takes only $O(\log n)$ time and can affect at most $O(\log n)$ existing certificates (a constant number of deschedulings and reschedulings per level). We call this fourth structure a *kinetic tournament*.

If our points move with constant velocities, how many events will our kinetic tournament have to process? The key insight to answering this question is to realize that the kinetic tournament is implementing a divide-and-conquer algorithm for the computation of the upper envelope of n straight lines in the ty-plane (the point trajectories). For example, the comparisons performed over time at the top level for declaring the final leader are exactly those needed to merge the upper envelopes of the two subgroups of the lines. The overall cost of the merge is easily seen to be $O(n)$. Thus this divide-and-conquer way of implementing the upper envelope computation has a worst-case cost satisfying the recurrence $C(n) = 2C(n/2) + \Theta(n)$, which solves to $C(n) = O(n \log n)$. The number of kinetic tournament events (reschedulings, etc.) is proportional to the number of times the identity of one of the contestants at a node of the tournament tree changes. Each such identity change corresponds to an intersection in one of the sub-envelopes computed by the divide-and-conquer algorithm, and hence is counted by the $O(n \log n)$ bound on $C(n)$. Therefore the kinetic tournament accomplishes our goal of maintaining on-line the maximum of a set of moving points, and it is a responsive, efficient, compact, and local KDS. If we use a priority queue to store the relevant events and perform a discrete-time simulation, then the event counts for all the structures described here can be made into run-times with an extra $O(\log n)$ factor (the priority queue cost).

1.2 Previous results and summary of the work

A number of works in the early eighties [7, 19, 28] considered the problem of *computing* a configuration function of moving points. In all cases, the motion was considered fully known, and the problem was typically cast and solved in one dimension higher. The method of Edelsbrunner and Welzl [19] for computing the k-th order statistic of a set of points moving at constant speed along the x-axis (introduced as a motivation for computing the k-level of an arrangement of lines) is most similar to a KDS.

More recently, questions concerning the maintenance of the Voronoi diagram of moving points (or its dual, the Delaunay triangulation) have received extensive attention [18, 21, 23, 32]. The significance of our work is best understood in comparison. The Delaunay triangulation contains a proof of its correctness involving only four-point certificates for each of the edges of the triangulation. In that sense, it is what we might call a *self-certifying* structure. As such, its kinetization is immediate: we need only maintain a certificate for each of the edges. Whenever any certificate changes sign, we know that we can update the triangulation (and the corresponding certificate structure) by an edge-flip on the failing edge. The structure has no internal events, hence the issue of efficiency does not arise. It is also well known that the Delaunay triangulation can be used to compute both the convex hull and the closest pair, so that we readily have a common kinetic data structure to maintain these configuration functions (closest pair maintenance requires in addition a kinetic tournament on the edge lengths), but this solution has two drawbacks: it is not local (a point can be a vertex of linearly many triangles), nor known to be efficient (the tightest upper bound known on the number of changes to the Delaunay triangulation of points in algebraic motion is roughly cubic in the number of points [23], whereas the convex hull and the closest pair can change roughly a quadratic number of times in the worst case). In general, one can view the process of kinetization as "sufficiently augmenting a configuration function to make it self-certifying."

Algorithms for collision detection in robotics by Lin and Canny [27] and Ponamgi et al. [30] exploit temporal coherence to maintain the minimum distance between all pairs of moving objects, but their approach re-tests the validity of separating planes at every step, and recalculates these separators from scratch when the old ones fail.

We have applied the methodology described above to a number of problems in 2-D computational geometry. In this paper, we present responsive, efficient, compact, and local kinetic data structures for two important and common configuration functions, giving representative examples of the kinetization process. Convex hull maintenance (Section 2) calls upon some deep theorems of combinatorial geometry to prove the efficiency of the structures we develop. Closest pair maintenance (Section 3) requires the development of a novel static algorithm, and specialized data structures to handle events efficiently. In Section 4, we take up some further issues generated by this framework for mobile data and present plans for further work.

Following the publication of the conference version of this paper [9], several kinetic data structures have been developed for the maintenance of a variety of structures: binary space partitions [1, 3], closest pair and minimum spanning trees in arbitrary dimensions [12], and diameter and width [2]. The framework has also been applied to the problem of collision detection between polygons in two dimensions [8, 20].

2 2-D convex hull

In this section, we present an efficient kinetic data structure to maintain the convex hull of a set of moving points in the plane. Following our general strategy for kinetization, we first describe a static algorithm and its certificate structure, simplify these certificates to attain certain desirable properties, and then show how to maintain the certificate structure once the points start moving.

Before we proceed, we dualize the problem, as the algorithm is more natural to describe in the dual setting. We focus here on computing the upper convex hull, and dualize each point (p, q) to the line $y = px + q$. In the dual, the goal is to maintain the upper envelope of a family of lines whose parameters change in a continuous, predictable fashion. We will perform the kinetization in the style of the $O(n \log n)$ divide and conquer algorithm mentioned in Section 1.1 for the analysis of the kinetic tournament: we divide the set of n lines into two subsets of roughly equal size, compute their upper envelopes recursively, and then merge the two envelopes. To focus on the merge step, we first study how to maintain the upper envelope of two convex piecewise linear univariate functions.

2.1 Upper envelope of two chains

We represent a piecewise linear function by a doubly linked list of edges and vertices ordered from left to right, and we call this representation a *chain*. In this section, we consider two chains—a red and a blue—and present a KDS to maintain the purple chain that represents the upper envelope of the two input chains.

As the supporting lines are the primary objects in our problem, we denote by a lowercase letter an edge or its supporting line, and by ab the vertex between edges a and b. For a vertex ab, the edge from the other chain that is above or below ab is called the *contender edge* of ab and denoted $\texttt{ce}(ab)$; we add to each vertex a pointer to its contender edge. We denote by $\chi(\cdots)$ the color (red or blue) of an input vertex or edge. Finally, we denote by $ab.\texttt{prev}$ (resp. $ab.\texttt{next}$) the red or blue vertex closest to ab on its left (resp. right). This is easily found by comparing the x-coordinate of the neighbor vertex in the chain to which ab belongs with that of one of the endpoints of the contender edge of ab.

The comparisons done by a standard sweep for merging the red and blue chains lead to certificates of two types: x-certificates proving the horizontal ordering of vertices, denoted by $<_x$, and y-certificates proving the vertical position of a vertex with respect to an edge, denoted by $<_y$. Unfortunately, if we were to keep all these comparisons as certificates, the kinetic data structure thus obtained would not be local, as a given edge could be the contender of linearly many vertices from the other envelope. We thus build an alternative list of certificates that also involves comparisons between line slopes, denoted by $<_s$ (Figure 2).

The following table gives this modified list of certificates. The first column contains the name of a certificate, the second column contains the comparison that this certificate guarantees, and the third column contains additional conditions for this certificate to be present in the KDS. For instance, the first line in the table says that there is a certificate called $\texttt{x}[ab]$ in the KDS only when ab and its right neighbor are of different colors (the *condition*). In this case, the *comparison* certifies the local x-ordering. The equation associated with this comparison has to be solved for t in order to find the first time at which the certificate fails.

Cert.	*Comparison*	*Condition(s)*
$\texttt{x}[ab]$	$ab <_x ab.\texttt{next}$	$\chi(ab) \neq \chi(ab.\texttt{next})$
$\texttt{yli}[ab]$	$ab <_y$ or $>_y \texttt{ce}(ab)$	$b \cap \texttt{ce}(ab) \neq \emptyset$
$\texttt{yri}[ab]$	$ab <_y$ or $>_y \texttt{ce}(ab)$	$a \cap \texttt{ce}(ab) \neq \emptyset$
$\texttt{yt}[ab]$	$\texttt{ce}(ab) <_y ab$	$a <_s \texttt{ce}(ab) <_s b$ $\texttt{ce}(ab) <_y ab$
$\texttt{slt}[ab]$	$a <_s \texttt{ce}(ab)$	
$\texttt{srt}[ab]$	$\texttt{ce}(ab) <_s b$	
$\texttt{sl}[ab]$	$b <_s \texttt{ce}(ab)$	$b <_s \texttt{ce}(ab)$ $ab <_y \texttt{ce}(ab)$ $\chi(ab) \neq \chi(ab.\texttt{next})$
$\texttt{sr}[ab]$	$\texttt{ce}(ab) <_s a$	$\texttt{ce}(ab) <_s a$ $ab <_y \texttt{ce}(ab)$ $\chi(ab) \neq \chi(ab.\texttt{prev})$

The certificates have the following meaning: (1) The exact x-ordering of vertices is recorded with $\texttt{x}[\cdots]$ certificates. (2) Each intersection is surrounded by $\texttt{yli}[\cdots]$ and $\texttt{yri}[\cdots]$ certificates ("y left/right intersection"). (3) If an edge is not part of the upper envelope, the certificates place its slope in the sequence of slopes of the edges covering it: either three "tangent" certificates ($\texttt{yt}[\cdots], \texttt{slt}[\cdots], \texttt{srt}[\cdots]$), or one certificate proving there is no tangent ($\texttt{sl}[\cdots]$ or its symmetric $\texttt{sr}[\cdots]$). Illustrations of the certificates appear in Figure 2. To be complete, we need to add slope certificates between the two leftmost edges and between the two rightmost edges.

Lemma 2.1 *Consider a configuration $\mathcal{C}$ of two convex piecewise linear functions and the certificate list $\mathcal{L}$ for their upper envelope as defined above. Let*

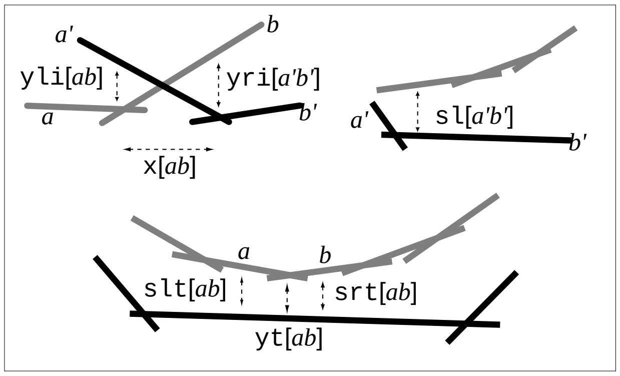

Figure 2: Depending on the relative positions of the red and blue convex chains, different certificates are used to certify the intersection structure (top left case) or the absence of intersection (top right and bottom cases). Arrows point to the elements being compared (vertices or edges).

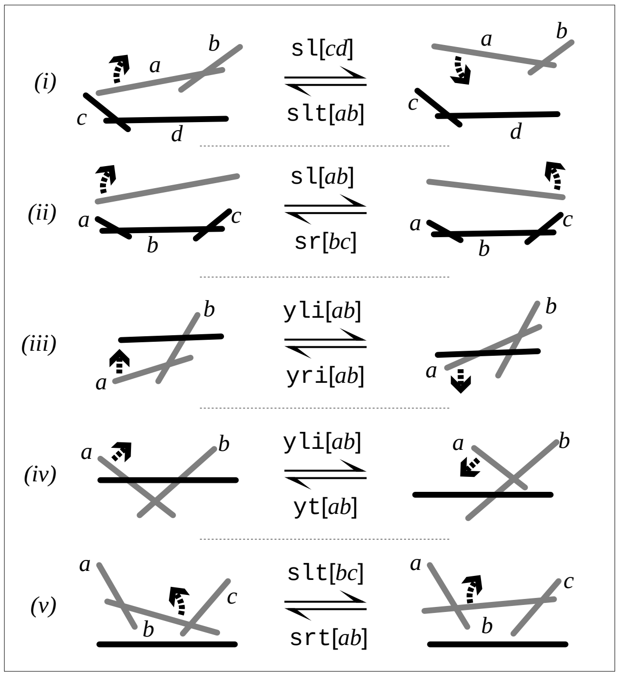

Figure 3: A partial list of events. The certificate that changes sign is indicated for each transition. There are three additional cases not shown: *(i)* and *(iii)* in mirror image, and the event corresponding to the x-certificate.

$\mathcal{C}'$ be a configuration for which all the certificates of $\mathcal{L}$ hold. Then the upper envelope of $\mathcal{C}'$ has the same combinatorial description (i.e., the same sequence of vertices and edges) as that of $\mathcal{C}$.

Proof:

First, the x-certificates prove the correctness of the contender edge pointers. Any vertex that has a y-certificate in $\mathcal{L}$ is also guaranteed to be placed in $\mathcal{C}'$ as in $\mathcal{C}$. It remains to show that those vertices without a y-certificate cannot be placed differently in $\mathcal{C}$ and $\mathcal{C}'$. For this, we consider a maximal contiguous sequence S of vertices without y-certificates in $\mathcal{L}$, and we assume without loss of generality that the red function is above the blue function in this stretch in $\mathcal{C}$. Let $\Delta(x)$ be the difference between the red slope and the blue slope at x. This function can only increase at red vertices and decrease at blue vertices. It cannot change sign at a red vertex of S, as there would be a $\texttt{yt}[\cdots]$ certificate in this case. Hence, on the interval defined by S, $\Delta(x)$ increases up to a certain blue vertex, then decreases. If the contender edge of this vertex is a, there will be $\texttt{sl}[\cdots]$ certificates to the left of the left endpoint of a and $\texttt{sr}[\cdots]$ certificates to the right of the right endpoint of a. These certificates guarantee that the red function remains above the blue function on the interval defined by S. $\square$

Our certificate list certifies the correctness of the upper envelope. As in the case of any kinetic data structure, all these certificates are placed in a global event queue, where each certificate is stamped with its failure time. When it is time to process the first event in the queue, we need to update the certificate list. Below is the list of changes that need to be performed for each type of event. A pictorial description is shown in Figure 3.

```
Event: failure of $yli[ab]$
    Delete $\texttt{yri}[ab.\texttt{next}], \texttt{yli}[ab]$;
    if $\exists\texttt{yri}[ab]$
      then begin
        Delete $\texttt{yli}[ab.\texttt{prev}], \texttt{yri}[ab]$
        Create $\texttt{slt}[ab], \texttt{srt}[ab], \texttt{yt}[ab]$
        Remove $\texttt{ce}(ab)$ from output
      end;
      else begin
        Create $\texttt{yri}[ab], \texttt{yli}[ab.\texttt{prev}]$
        Add $a$ or remove $b$ in output
      end;
```

```
**Event:** failure of $yt[ab]$
    **Delete** yt[$ab$], slt[$ab$], srt[$ab$];
    **Create** yli[$ab$], yri[$ab$.next];
    **Create** yri[$ab$], yli[$ab$.prev];
    Add ce($ab$) to output
```

```
**Event:** failure of slt[$ab$]
    **Delete** slt[$ab$], srt[$ab$], yt[$ab$];
    $cd \leftarrow ab$.prev;
    **if** $d = a$ /* i.e., same color */
        **then Create** slt[$cd$], srt[$cd$], yt[$cd$];
        **else Create** sl[$cd$];
```

```
**Event:** failure of sl[$ab$]
    **Delete** sl[$ab$];
    $cd \leftarrow ab$.next;
    **if** $\chi(cd) \neq \chi(ab)$
        **then Create** slt[$cd$], srt[$cd$], yt[$cd$];
        **else Create** sr[$cd$];
```

```
**Event:** failure of x[$ab$]
    **Delete** x[ab];
    $cd \leftarrow ab$.next;
    ce($ab$) $\leftarrow d$;
    ce($cd$) $\leftarrow a$;
    /* $cd$.prev and $ab$.next now have new values */
    **if** $\chi(cd$.prev$) = \chi(cd)$
        **then Delete** x[$cd$.prev];
        **else Create** x[$cd$.prev];
    **if** $\chi(ab$.next$) = \chi(ab)$
        **then Delete** x[$cd$];
        **else Create** x[$ab$];
    /* Now update intersection certificates */
    **if** $\exists$yri[$ab$]
        **then Delete** yri[$ab$]; **Create** yri[$cd$];
    **if** $\exists$yli[$cd$]
        **then Delete** yli[$cd$]; **Create** yli[$ab$];
    /* Update slope certificates if $ab$ is below $cd$ */
    **if** $\exists$sl[$ab$] **then Update** sl[$ab$] to point to new ce($ab$);
    **elseif** $\exists$yt[$cd$]
    then Delete $\texttt{slt}[cd], \texttt{srt}[cd], \texttt{yt}[cd]$; Create $\texttt{sl}[ab]$;
  elseif $\exists\texttt{sr}[ab]$ then if $a <_s d$
                        then Delete $\texttt{sr}[ab]$; Create $\texttt{slt}[cd], \texttt{srt}[cd], \texttt{yt}[cd]$;
                        else Update $\texttt{sr}[ab]$ to point to new $\texttt{ce}(ab)$;
  /* Symmetric treatment if $cd$ is below $ab$ (not shown) */
```

As an example, consider the event $\texttt{yt}[ab]$, which corresponds to case (iv), right to left, of Figure 3: a red edge moves above a blue vertex. In this case, we remove the three certificates proving that the red edge was below the blue chain (bottom case of Figure 2), and add certificates to bracket the two newly formed intersections. Finally, the new edge on the upper envelope is added to the output.

Events corresponding to certificates $\texttt{yri}[ab]$, $\texttt{sr}[ab]$, and $\texttt{srt}[ab]$ are exactly symmetric to $\texttt{yli}[ab]$, $\texttt{sl}[ab]$, and $\texttt{slt}[ab]$.

Lemma 2.2 *The preceding procedures correctly update the certificate list when the corresponding events happen.*

Proof: By examination of each case, omitted in this paper. □

In general, when a y-certificate changes sign, this modifies the output: either two neighbor vertices merge into one, or the reverse. Hence, for the purpose of the recursive construction, it is necessary to be able to handle such local structural changes in the input, and it can be checked that this also changes $O(1)$ certificates.

2.2 Divide and conquer upper envelope

To kinetize the divide and conquer algorithm, we keep a record of the entire computation in a balanced binary tree. A node in this tree is in charge of maintaining the upper envelope of the two upper subenvelopes computed by its children. If an event triggers a change in the output of a node, this node passes on the event to its parent, as a local structural change in the input, and so on to upper levels of the computation tree while this change remains visible.

As in the case of the one-dimensional kinetic tournament data structure for known motions, we analyze efficiency by considering time as an additional static dimension and charging each event to a feature of a three-dimensional structure with known worst-case complexity. The primal version of the problem is ill-suited for such an analysis, as the static structure described by the convex hull over time is not the convex hull of the trajectories of the underlying points. On the other hand, in the dual, the structure described by the upper envelope over time is exactly the upper envelope of the surfaces described by the underlying lines. We can thus use results proving near-quadratic complexity for the upper envelope of algebraic surfaces [33]. We also make use of the recent result of Agarwal, Schwarzkopf, and Sharir [4] about the near-quadratic complexity of the overlay of the projections of two upper-envelopes to obtain sharp bounds on the number of events due to x-certificates.

The results mentioned above are currently proven only in the context of *algebraic* surfaces of bounded degree, since a general theory of two-dimensional Davenport-Schinzel sequences is still lacking. Hence, in this section, we assume that the position of a point is given, as a function of time, by two algebraic functions of bounded degree.

Theorem 2.3 *The KDS for maintaining the convex hull is efficient, responsive. compact, and local.*

Proof: The KDS is clearly responsive, local, and compact. We turn to the proof of efficiency.

We first focus on the events attached to a specific node of the computation tree that involves a total of n red and blue lines. Consider time as a static third dimension: a line whose parameters are polynomial functions of time describes an algebraic surface in three dimensions. The blue (red) family of lines is now a family of bivariate algebraic functions. Looking at the upper envelopes of the blue and red families, and at their joint upper envelope in turn, we observe that a purple vertex on this upper envelope corresponds to a change of sign of a y-certificate (a "y-event") in the kinetic interpretation. A monochromatic vertex corresponds to the appearance/disappearance of an edge triggered by some descendant in the computation tree. As our surfaces are algebraic of bounded degree, their upper envelope has complexity $O(n^{2+\epsilon})$ for any $\epsilon > 0$ [33], and therefore the number of events due to y-certificate sign changes is bounded by this quantity[3].

Consider now the events corresponding to the x reordering of two vertices of different colors (called "x-events"). In the 3-dimensional setting, a blue envelope vertex becomes an edge of the blue surface upper envelope. Hence, an x-event corresponds to a point (x, t) above which there is an edge in both the blue and the red upper envelopes. In other words, each x-event is associated with a bichromatic vertex in the overlay of the projections of the red and blue upper envelopes on the xt-plane ($y = 0$). If there are n bivariate algebraic surfaces of bounded degree in total, the complexity of this overlay is also $O(n^{2+\epsilon})$ for any $\epsilon > 0$ [4]. Hence, there are at most that many x-events.

Finally, each pair of lines becomes parallel a constant number of times, so there are $O(n^2)$ slope events attached to the node we have been focusing on up to now.

[3]The best known bound for this specific problem is tighter [2], but this bound is sufficient for our purposes.

Getting back to the full computation tree, we conclude that the total number of events $C(n)$ satisfies the recurrence $C(n) = 2C(n/2) + O(n^{2+\epsilon})$, and therefore $C(n) = O(n^{2+\epsilon})$. In the worst case, the convex hull of n points in linear or higher order motion changes $\Omega(n^2)$ times [2]. Hence our KDS is efficient. $\square$

3 Closest pair in 2-D

Not all static algorithms lend themselves to an efficient kinetization. For instance, consider the following classic algorithm of Shamos [31] for finding the closest pair within a set of points in the plane: divide the points into the left half and the right half and recursively compute the closest distances δ_L and δ_R within each half. Then check all pairs that are within distance $\delta = \min(\delta_L, \delta_R)$ of a vertical median line $y = y_0$. A kinetic version of this algorithm would require, for each point p on the left side, a certificate of the form $x_p < y_0 - \delta$ or the converse. The resulting KDS would not even be responsive: when the identity of the pair that realizes δ changes, all certificates of the form above need to be updated, and there might be a linear number of them. Surprisingly, we were not able to find a good kinetization of any known closest pair algorithm.

In this section, we describe a new static algorithm for the closest pair based on the plane sweep paradigm. We then add data structures to record the history of the algorithm, and show that these data structures can be maintained as the points move. The data structures always reflect the history that would result if the plane sweep algorithm were applied to the current configuration of points. This resulting kinetic data structure has the qualities described in Section 1.

3.1 The static plane sweep algorithm

The static closest-pair algorithm is based on the idea of dividing the space around each point into six 60° wedges. It is a trivial observation that the nearest neighbor of each point is the closest of the nearest neighbors in the six wedges. We show that an approximate definition of nearest neighbor in each wedge (using the L_∞ norm) is still sufficient to find the closest pair. The relaxed definition lets us compute neighbors efficiently, and aids in the kinetization of the algorithm.

We define the *dominance wedge* of a point p, call it $Dom(p)$, to be the right-extending wedge bounded by the lines through p that make $\pm 30°$ angles with the x-axis. The dominance wedge is defined to be open on the bottom and closed on top (it includes its upper boundary, but not its lower boundary). We define $Circ(p, r)$ to be the circle with radius r centered on point p. In this section of the paper, the distance between two points p and q is denoted by pq.

Our algorithm uses all three right-extending wedges bounded by the vertical line through p and by the $\pm 30°$ lines, but we frame our arguments in terms of the single dominance wedge that contains the point $(\infty, 0)$. The same arguments apply to the other wedges by rotation.

Let the closest pair of points in S be (a, b), with a to the left of b (or below b, if their x-coordinates are equal). For notational convenience, we write this as $a \prec b$. Without loss of generality, assume that $b \in Dom(a)$; if this is not the case, then consider the $\pm 60°$ rotated plane that puts b in $Dom(a)$.

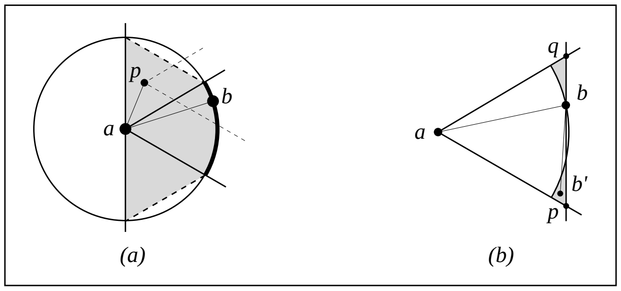

Figure 4: If (a, b) is the closest pair and $b \in Dom(a)$, then: (a) there is no point p to the right of a that dominates b, as such p would lie in the shaded region and be closer to a than b is; (b) point b is also the leftmost point in $Dom(a)$—any point b' left of b would be closer to b than a is.

Lemma 3.1 *Point b is not contained in $Dom(p)$ for any third point p with $a \prec p$.*

Proof: Point b lies on the arc $Circ(a, ab) \cap Dom(a)$, shown dark in Figure 4(a). Any point p to the right of a that contains b in $Dom(p)$ must lie in the shaded region bounded by the arc, the vertical line through a, and the $\pm 30°$ open lines through the top and bottom of the arc. But this region is entirely contained in $Circ(a, ab)$, which implies that $ap < ab$, a contradiction. □

Lemma 3.2 *The leftmost point of S in $Dom(a)$ is b.*

Proof: Let $r = ab$, the separation of the closest pair. Consider the triangle formed by the intersection of $Dom(a)$ with the half-plane left of b. If the leftmost point b' is not equal to b, then b' must lie in the portion of this triangle outside $Circ(a, r)$. See Figure 4(b). The points of this region farthest from b are the intersections p and q of the vertical line through b with the boundaries of $Dom(a)$ (because the perpendiculars from b to the boundaries of $Dom(a)$ lie inside $Circ(a, r)$). Without loss of generality let $pb \geq qb$. Consider the triangle $\triangle abp$.

By the law of sines, $pb/ab = pb/r = \sin(\angle pab)/\sin(\angle apb) = \sin(\angle pab)/\sin 60°$. Because $\angle pab \leq 60°$, $pb \leq r$. That is, $bb' \leq r$. Furthermore, the partly open/partly closed definition of $Dom(a)$ means that $bb' < r$, a contradiction. □

For any point p, let $Maxima(p)$ consist of the points of S on the boundary of

$$\bigcup_{\substack{q \in S \\ p \prec q}} Dom(q).$$

In words, this set contains all the points to the right of p that are not in the dominance wedge of any other point to the right of p. We define the set of *candidates* associated with p, $Cands(p)$, to be the set $Maxima(p) \cap Dom(p)$. We denote the leftmost of these by $lcand(p)$. See Figure 5. By Lemmas 3.1 and 3.2, we have $b = lcand(a)$ for the closest pair (a, b).

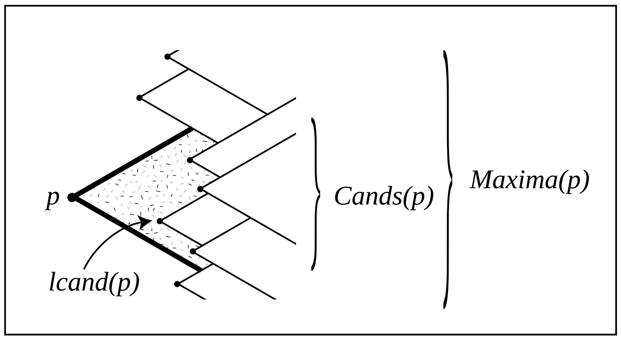

Figure 5: The sets of points $Maxima(p)$ and $Cands(p)$, and the leftmost candidate $lcand(p)$.

The plane sweep algorithm performs the following steps three times, once on the untransformed points of S, once on S rotated around the origin by $+60°$, and once on S rotated by $-60°$. Each of these rotations brings one of the three families of right-extending wedges into the central position, bounded by $\pm 30°$ lines.

1. Initialize a y-ordered list of points $Maxima$ to $\emptyset$. ($Maxima$ contains $Maxima(p)$ at the top of each iteration of the loop below.)
2. For each point $p \in S$ from right to left,
   - (a) Set $Cands(p) = Maxima \cap Dom(p)$.
   - (b) Set $lcand(p)$ to be the leftmost element of $Cands(p)$.
   - (c) Delete the points of $Cands(p)$ from $Maxima$.
   - (d) Insert p into $Maxima$ at its proper place in y-order.

At the end of this procedure, repeated for all three directions, one of the three sets of $(p, \mathit{lcand}(p))$ pairs it produces contains the closest pair (a, b).

It is clear that the plane sweep algorithm can be implemented to run in $O(n \log n)$ time. Sorting the points of S in preparation for sweeping takes $O(n \log n)$ time. We store *Maxima* in a balanced binary tree structure that supports logarithmic-time searches, insertions, deletions, splits, and joins [17]. Computing $\mathit{Cands}(p)$ requires two $O(\log n)$ time searches on *Maxima*, since $\mathit{Cands}(p)$ is a consecutive subsequence of *Maxima*. Finding $\mathit{lcand}(p)$ takes additional time $|\mathit{Cands}(p)|$, but since those points are immediately removed from *Maxima*, the total time spent finding the leftmost points of all the candidate sets is only $O(n)$. Splitting $\mathit{Cands}(p)$ out of *Maxima* and inserting p in its place takes $O(\log n)$ per point p. Thus the total running time is $O(n \log n)$.

3.2 Kinetization

To make the plane sweep algorithm kinetic, we need to transform it into a static data structure that represents the action of the plane sweep algorithm. We also need a set of certificates to show that the data structure is valid for the current set of points.

We define the *maxima diagram* to be the union, over all points p, of the part of the boundary of $\mathit{Dom}(p)$ that lies outside $\cup_{q \in \mathit{Maxima}(p)} \mathit{Dom}(q)$. Each point p of S is the left endpoint of two segments in the maxima diagram that extend from p to the boundaries of $\mathit{Dom}(q)$ and $\mathit{Dom}(q')$, for two points q and q' in $\mathit{Maxima}(p)$. We say that q and q' are the *targets* of p in the maxima diagram.

We use as certificates three sorted orders: the projections of the points in S on the x-axis and on the lines that make an angle of $\pm 60°$ with the x-axis. Each point belongs to up to six certificates, involving its two neighbors in each of the three sorted orders. We also use certificates for a kinetic tournament, described below.

Lemma 3.3 *If two configurations of S have all three orders equivalent, then for each p, $\mathit{Maxima}(p)$, $\mathit{Cands}(p)$ and $\mathit{lcand}(p)$ are the same in the two configurations.*

Proof: By induction on the points of S, from right to left in $\prec$ order. For each point p in turn, we assume that $\mathit{Maxima}(p)$ is the same for both configurations, with the same y-order. We prove that $\mathit{Cands}(p)$ and $\mathit{lcand}(p)$ are the same, and finally show that $\mathit{Maxima}(p')$ is the same for p' the predecessor of p in $\prec$ order.

Point p has the same targets in each version of $\mathit{Maxima}(p)$, since p is in the same $\pm 60°$-orders with respect to $\mathit{Maxima}(p)$ in both configurations. Hence $\mathit{Cands}(p)$ is the same in the two configurations, and because the x-orders are the same, $\mathit{lcand}(p)$ is also the same. The set $\mathit{Maxima}(p')$ is obtained from

Maxima(p) by removing the points of *Cands*(p), then inserting p between its two targets; hence *Maxima*(p') is the same in the two configurations. □

The maxima diagram can undergo a linear number of changes when a pair of points swaps in one of the three linear orders. However, the changes to the maxima diagram can be represented in an implicit data structure that requires only $O(\log n)$ updates per swap. For this purpose, we keep two auxiliary data structures for each $p \in S$, called *Cands*(p) and *Parents*(p), that represent two separate one-to-many relations.

1. *Cands*(p) contains the intersection of *Maxima*(p) with *Dom*(p), as a sequence of points ordered by y-coordinate. (This order is the same as the orders induced along the $\pm 60°$ directions.) This sequence is stored in a balanced binary tree and supports the usual searching and update operations. In addition, each node of the tree has a pointer to its parent in the tree, and the root of the tree for *Cands*(p) points to p. Thus each point of $q \in S$ can find the point $p \in S$ whose candidate it is, $q \in$ *Cands*(p), in $O(\log n)$ time. Each node in a *Cands*() tree also keeps track of the leftmost (in x-order) point in its subtree, and so the root of *Cands*(p) records *lcand*(p). The parent pointers can be maintained as part of the standard tree update operations, within the same asymptotic time bound, as can the "leftmost" fields. As part of our algorithm, we will make sure that the "leftmost" fields are maintained correctly whenever the x-order of points changes.

2. *Parents*(p) records all the points for which p is a target in the maxima diagram. *Parents*(p) is an ordered sequence of (the points corresponding to) the edges that hit *Dom*(p) from the left in the maxima diagram. The sequence is ordered according to the order in which the edges hit *Dom*(p). This order need not correspond to the y-order of the points; however, the sequence can be divided into the points above p, denoted *Parents*$_a(p)$, and the points below p, denoted *Parents*$_b(p)$. *Dom*(p) is the target for the lower edge extending from all elements of *Parents*$_a(p)$, and is the target for the upper edge of all elements of *Parents*$_b(p)$. In each of the two subsequences *Parents*$_a(p)$ and *Parents*$_b(p)$, the order of the points (the order in which their edges hit *Dom*(p)) is the same as their x-order. The sequence *Parents*(p) is stored in a balanced binary tree with parent pointers, so for each of the two edges extending from a point q in the maxima diagram, we can find the point p for which $q \in$ *Parents*(p) in logarithmic time.

These are the only data structures needed for the kinetization. In particular, we don't use the *Maxima* data structure described in the static case. The following algorithmic sketch shows how to update all the affected *Cands*(), *Parents*(), and *lcand*() fields when two points p and q exchange positions in the x-order of S. Without loss of generality, assume that $p \prec q$ (p is left of q) before the exchange. Furthermore, assume that p is below q at the instant of exchange (similar pseudo-code applies if p is above q). See Figure 6.

1. If $p \in \mathit{Parents}(q)$, specifically in $\mathit{Parents}_b(q)$, then
   (a) Split off the portion of $\mathit{Cands}(q)$ inside $\mathit{Dom}(p)$ and join it to the top of $\mathit{Cands}(p)$.
   (b) Let u be the point such that $q \in \mathit{Parents}_a(u)$. Delete point q from $\mathit{Parents}_a(u)$ and insert it into $\mathit{Parents}_a(p)$.
   (c) Let v be the new bottom point of $\mathit{Cands}(q)$, if any, or else the point such that $q \in \mathit{Parents}_b(v)$. Delete p from $\mathit{Parents}_b(q)$ and insert it into $\mathit{Parents}_b(v)$.
2. Let p' and q' be the points such that $p \in \mathit{Cands}(p')$, and $q \in \mathit{Cands}(q')$. If $p' = q'$, then update $\mathit{lcand}(p')$ starting from p and q in the tree for $\mathit{Cands}(p')$.

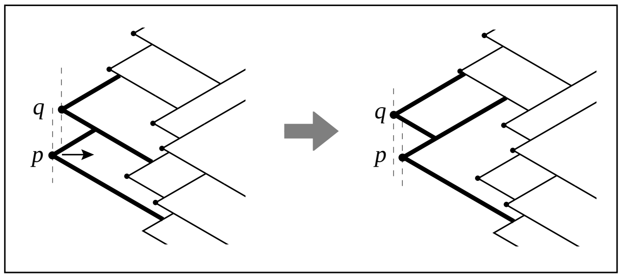

Figure 6: An x event and the change in the *Cands* sets.

Lemma 3.4 *After the preceding procedure for updating the $\mathit{Cands}()$, $\mathit{Parents}()$, and $\mathit{lcand}()$ fields when two points of S exchange in x-order, the data structure correctly represents the maxima diagram for the current configuration of S, and the $\mathit{lcand}()$ fields are correct.*

Proof: If q is a target for p, then their x-exchange changes the maxima diagram. Specifically, the points of $\mathit{Maxima}(q)$ in $\mathit{Dom}(q) \cap \mathit{Dom}(p)$ are transferred from $\mathit{Cands}(q)$ to $\mathit{Cands}(p)$. Point p gets a new target; the new point of contact between the segment from p and its target lies inside $\mathit{Dom}(q)$. Likewise q gets p as a target. Step 1 handles these changes.

The only edges of the maxima diagram that change are those that extend to the right from p and q—there are no target changes for points either right or left of $\{p, q\}$—so the operations of Step 1 suffice to update the maxima diagram.

If neither p nor q is a target for the other, then the maxima diagram does not change—the *Cands*() and *Parents*() fields do not need to be updated.

Whether or not the maxima diagram changes, one *lcand*() field may change. If $p, q \in Cands(u)$ for some point u, we need to ensure that the "leftmost" fields are updated in the binary tree representing $Cands(u)$, so that any comparison of p and q in that tree is re-evaluated; this may cause $lcand(u)$ to change. Step 2 takes care of this. □

The following pseudo-code tells how to update the affected fields when two points p and q exchange positions in the $+60°$-order of S (at the instant of exchange, the line through p and q makes an angle of $-30°$ with the x-axis). Without loss of generality, assume that p is left of q. There are two cases, depending on whether q enters or exits from $Dom(p)$.

In the first case, q enters $Dom(p)$. See Figure 7. Update the data structures thus:

1. If $p \in Parents_a(q)$ then
   (a) Let v be the point such that $q \in Cands(v)$. Delete q from $Cands(v)$ and insert it into $Cands(p)$.
   (b) Let t be the leftmost point in $Parents_b(q)$ that is to the right of p, if any, or else the point such that $q \in Parents_a(t)$. (Recall that x-order in $Parents_b(q)$ is equivalent to the order in which edges hit $Dom(q)$.) Delete p from $Parents_a(q)$ and insert p into $Parents_a(t)$.
   (c) Split off the subsequence of $Parents_b(q)$ whose points are to the left of t (and hence left of p) and join it onto the bottom of $Parents_b(p)$.

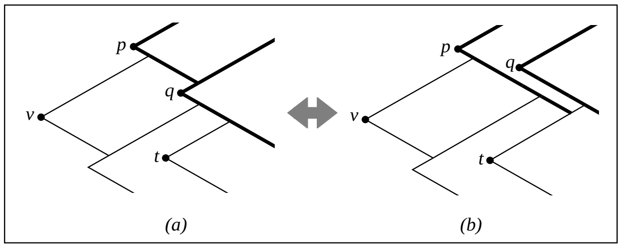

Figure 7: A 60° event. (a→b) q enters $Dom(p)$; (b→a) q exits $Dom(p)$.

In the second case, q exits $Dom(p)$. See Figure 7. The pseudo-code in this case just inverts the action performed in the first case:

1. If $q \in Cands(p)$ then

(a) Let t be the point such that $p \in \mathit{Parents}_a(t)$. Delete p from $\mathit{Parents}_a(t)$ and insert p into $\mathit{Parents}_a(q)$.

(b) Split off from $\mathit{Parents}_b(p)$ the points whose edges are incident to $\mathit{Dom}(p)$ below q, and join them onto the top of $\mathit{Parents}_b(q)$.

(c) Let v be the new rightmost point of $\mathit{Parents}_b(p)$, if any, or else the point such that $p \in \mathit{Cands}(v)$. Delete q from $\mathit{Cands}(p)$ and insert q into $\mathit{Cands}(v)$.

Lemma 3.5 *After the preceding procedure for updating the Cands(), Parents(), and lcand() fields when two points of S exchange in the $+60°$-order, the data structure correctly represents the maxima diagram for the current configuration of S, and the lcand() fields are correct.*

Proof: Consider first the case in which q enters $\mathit{Dom}(p)$. If $p \notin \mathit{Parents}_a(q)$, then $q \notin \mathit{Maxima}(p)$ and the exchange of p and q in the $+60°$-order does not affect the maxima diagram at all. No data structure updates are necessary.

If $p \in \mathit{Parents}_a(q)$, the maxima diagram changes, but only in the vicinity of p and q. The exchange of p and q does not change the targets of points to the right of p. Only the lower target of p needs to be updated; Step 1b takes care of this. Of the points to the left of p, only those with q as their upper target (i.e., members of $\mathit{Parents}_b(q)$) need to have their targets changed to p. Step 1c does this. The *Cands*() set changes only for p (because q enters it) and for the point v whose $\mathit{Cands}(v)$ set q leaves; Step 1a does this. The "leftmost" fields are updated in the *Cands*() binary trees during the modification, so $\mathit{lcand}(p)$ and $\mathit{lcand}(v)$ are correctly maintained. The *Cands*() and *Parents*() lists are enough to specify the combinatorial structure of the maxima diagram; since they are correctly maintained, so is the maxima diagram.

In the case in which q exits $\mathit{Dom}(p)$, the changes to the maxima diagram are the inverse of those in the first case. The update procedure for this case inverts the action of the first update procedure, and hence is correct. $\square$

The procedure for exchanging two points in the $-60°$-order is symmetric to the one for $+60°$-order exchanges. There are $O(n^2)$ exchanges in each of the three orders.

It is clear that each of the update operations needed to restore the auxiliary data structures *Cands*(), *Parents*(), and *lcand*() takes $O(\log n)$ time: each involves a constant number of standard operations on balanced binary trees.

The final element of our kinetic data structure is a kinetic tournament on the $3n$ distances corresponding to $(p, \mathit{lcand}(p))$ pairs (this adds $3n$ certificates to our KDS). The root of the tournament tree contains the closest pair at any time during the running of the algorithm. Note that when $\mathit{lcand}(p)$ changes, it triggers a discontinuity of the associated distance in the kinetic tournament, but bounds like those in Section 1 apply even in this case.

Theorem 3.6 *The kinetic data structure for the closest pair problem is efficient, responsive, compact, and local.*

Proof: When the n points of S move according to pseudo-algebraic functions of time, the total number of external events is roughly quadratic, in the worst case (consider n points moving on a line; the number of closest pairs is at least the number of intersections between trajectories). The worst-case number of internal events is only a logarithmic factor more: the number of exchanges in each of the three orders is $O(n^2)$, since any pair of points undergoes only a constant number of exchanges under pseudo-algebraic motions; there are $O(n^2)$ changes to *lcand*() values over the life of the algorithm; the number of vertices on the lower envelope of the $(p, lcand(p))$ pairs is $O(n^2\beta(n))$ where $\beta(n)$ is an extremely slowly growing function [34]; and the kinetic tournament processes only a logarithmic factor more events than appear on the lower envelope of the pairwise distances. Hence the KDS is efficient.

The processing of an event involves $O(\log n)$ operations on the structure and the scheduling of $O(\log n)$ events in the event queue. Hence the KDS is responsive.

There are only $O(n)$ events in the event queue at any time: one for each of the $3n - 3$ order certificates, and $O(n)$ for the kinetic tournament. Hence the KDS is compact.

A point is involved in $O(\log n)$ certificates. Each point p of S belongs to at most six order certificates and to $O(\log n)$ comparison certificates that maintain the "leftmost" fields in *lcand*() trees. It belongs to at most three $(p, lcand(p))$ pairs (one for each of the three rotations). Likewise, each p is $lcand(q)$ for at most three different qs, one for each rotation; this is because for a single rotation, the *Cands*() sets are all disjoint. Each active $(p, lcand(p))$ pair participates in $O(\log n)$ events in the kinetic tournament. Hence the KDS is local. □

4 Conclusion and further issues

We have presented a new framework for maintaining attributes (configuration functions) of objects in continuous motion. This framework introduces an on-line, combinatorial approach to changes in the configuration function, avoids a discretization of time, and sets the ground for using sophisticated algorithmic techniques to maintain these configurations in what we call kinetic data structures. We measure the quality of a KDS by its responsiveness, efficiency, locality, and compactness. By working through three examples, we have demonstrated the generality of the kinetization procedure, which transforms a static algorithm into its kinetic counterpart. Moreover, the algorithms described in this paper have been implemented, showing that the framework as well as the algorithms are valuable in practice [11].

In conclusion, we mention a few of the numerous issues that need further work.

Although in the analyses of the two examples discussed in this paper (convex hull and closest pair) we have assumed that each point follows a fixed pseudo-algebraic flight plan, in general it is important to make the number of flight plan changes (globally, or on a per object basis) a parameter of the analysis. This will become necessary, even if our actual objects never change flight plans, whenever we want to compose kinetizations. For example, the separation of the closest pair among continuously moving points changes continuously, even if the actual pairs realizing the distance change from time to time. If this distance itself is to become an input to another kinetic algorithm, its flight plan has to be updated whenever the underlying realizing pair changes. An instance of this phenomenon is already present inside our kinetization of the closest pair algorithm in Section 3.

Experiments on random inputs showed that our kinetic convex hull algorithm has an overhead of internal events that is of the same order as the number of external events, whereas our kinetic closest pair algorithm always processes $\Theta(n^2)$ internal events. Hence, ideally, the measure of efficiency should not compare the worst-case number of internal events to the worst-case number of external events, but the worst-case ratio of the actual number of internal events to the actual number of external events for any flight plan. It appears much more difficult to develop good algorithms with respect to this measure. Even if an exact analysis is difficult, heuristics that prune unneeded internal events are likely to prove important in practice.

We can view our kinetization process as starting from a proof of correctness of a static configuration function, and then "animating this proof through time." Not all proofs are equally good for this use. Our locality requirement favors proofs that have a small number of predicates involving each particular datum. Thus it will generally be advantageous to start with "shallow proofs"—proofs of small depth—for the static problem, such as one gets, for example, from *parallel* algorithms for solving the static version. Techniques already developed in parallel computational geometry [6] or in parametric searching [5] may prove to be useful.

In a real time system, it is possible that there is not sufficient time to process an event completely before the next event appears. If kinetic structures are to be used in such a context, it is crucial to be able to maintain partially correct structures, with a mechanism for processing multiple events efficiently and correctly as a batch.

Acknowledgments. We wish to thank Pankaj Agarwal, Rajeev Motwani, G.D. Ramkumar, Craig Silverstein, and Li Zhang for useful discussions, and Robert-Paul Berretty for pointing out an error in a first version of this paper. Julien Basch, Craig Silverstein, and Li Zhang implemented the algorithms described in this paper. Leonidas Guibas acknowledges support by ARO–MURI grant 5-23542, and NSF grants CCR-9215219 and IRI-9306544.

References

[1] P. Agarwal, J. Erickson, and L. Guibas. Kinetic binary space partitions for triangles. In *Proc. 9th ACM-SIAM Sympos. Discrete Algorithms*, pages 107–116, January 1998.

[2] P. K. Agarwal, L. J. Guibas, J. Hershberger, and E. Veach. Maintaining the extent of a moving point set. In *Proceedings of the 5th Workshop on Algorithms and Data Structures*, pages 31–44. Springer-Verlag, 1997. Lecture Notes in Computer Science 1272.

[3] P. K. Agarwal, L. J. Guibas, T. Murali, and J. Vitter. Cylindrical static and kinetic binary space partitions. In *Proc. 13th Annu. ACM Sympos. Comput. Geom.*, pages 39–48, 1997.

[4] P. K. Agarwal, O. Schwarzkopf, and M. Sharir. The overlay of lower envelopes and its applications. *Discrete Comput. Geom.*, 15:1–13, 1996.

[5] P. K. Agarwal, M. Sharir, and S. Toledo. Applications of parametric searching in geometric optimization. *J. Algorithms*, 17:292–318, 1994.

[6] S. G. Akl and K. A. Lyons. *Parallel Computational Geometry.* Prentice Hall, 1993.

[7] M. J. Atallah. Some dynamic computational geometry problems. *Comput. Math. Appl.*, 11:1171–1181, 1985.

[8] J. Basch, J. Erickson, L. J. Guibas, J. Hershberger, and L. Zhang. Kinetic collision detection between two simple polygons. To appear in *Proc. 10th Sympos. Discrete Algorithms*, 1999.

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

[10] J. Basch, L. J. Guibas, and G.D. Ramkumar. Sweeping lines and line segments with a heap. In *Proc. 13th Annu. ACM Sympos. Comput. Geom.*, pages 469–471, 1997.

[11] J. Basch, L. J. Guibas, C. D. Silverstein, and L. Zhang. A practical evaluation of kinetic data structures. In *Proc. 13th Annu. ACM Sympos. Comput. Geom.*, pages 388–390, 1997.

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

[13] J. L. Bentley and T. A. Ottmann. Algorithms for reporting and counting geometric intersections. *IEEE Trans. Comput.*, C-28:643–647, 1979.

[14] S. N. Bespamyatnikh. An optimal algorithm for closest pair maintenance. In *Proc. 11th Annu. ACM Sympos. Comput. Geom.*, pages 152–161, 1995.

[15] K. Q. Brown. Comments on "Algorithms for reporting and counting geometric intersections". *IEEE Trans. Comput.*, C-30:147–148, 1981.

[16] P. B. Callahan and S. Rao Kosaraju. Algorithms for dynamic closest-pair and n-body potential fields. In *Proc. 6th ACM-SIAM Sympos. Discrete Algorithms (SODA '95)*, pages 263–272, 1995.

[17] T. H. Cormen, C. E. Leiserson, and R. L. Rivest. *Introduction to Algorithms.* The MIT Press, Cambridge, Mass., 1990.

[18] O. Devillers, M. Golin, K. Kedem, and S. Schirra. Revenge of the dog: Queries on Voronoi diagrams of moving points. In *Proc. 6th Canad. Conf. Comput. Geom.*, pages 122–127, 1994.

[19] H. Edelsbrunner and E. Welzl. Constructing belts in two-dimensional arrangements with applications. *SIAM J. Comput.*, 15:271–284, 1986.

[20] J. Erickson, L. J. Guibas, J. Stofi, and L. Zhang. Separation-sensitive kinetic collision detection for convex objects. To appear in *Proc. 10th Sympos. Discrete Algorithms*, 1999.

[21] J.-J. Fu and R. C. T. Lee. Voronoi diagrams of moving points in the plane. *Internat. J. Comput. Geom. Appl.*, 1(1):23–32, 1991.

[22] M. Golin, R. Raman, C. Schwarz, and M. Smid. Randomized data structures for the dynamic closest-pair problem. In *Proc. 4th ACM-SIAM Sympos. Discrete Algorithms*, pages 301–310, 1993.

[23] L. Guibas, J. S. B. Mitchell, and T. Roos. Voronoi diagrams of moving points in the plane. In *Proc. 17th Internat. Workshop Graph-Theoret. Concepts Comput. Sci.*, volume 570 of *Lecture Notes in Computer Science*, pages 113–125. Springer-Verlag, 1991.

[24] J. Hershberger. Finding the upper envelope of n line segments in $O(n \log n)$ time. *Inform. Process. Lett.*, 33:169–174, 1989.

[25] J. Hershberger and S. Suri. Applications of a semi-dynamic convex hull algorithm. *BIT*, 32:249–267, 1992.

[26] S. Kapoor and M. Smid. New techniques for exact and approximate dynamic closest-point problems. In *Proc. 10th Annu. ACM Sympos. Comput. Geom.*, pages 165–174, 1994.

[27] M. C. Lin and J. F. Canny. Efficient algorithms for incremental distance computation. In *Proc. IEEE Internat. Conf. Robot. Autom.*, volume 2, pages 1008–1014, 1991.

[28] T. Ottmann and D. Wood. Dynamical sets of points. *Comput. Vision Graph. Image Process.*, 27:157–166, 1984.

[29] M. H. Overmars and J. van Leeuwen. Maintenance of configurations in the plane. *J. Comput. Syst. Sci.*, 23:166–204, 1981.

[30] M. K. Ponamgi, M. C. Lin, and D. Manocha. Incremental collision detection for polygonal models. In *Proc. 11th Annu. ACM Sympos. Comput. Geom.*, pages V7–V8, 1995.

[31] F. P. Preparata and M. I. Shamos. *Computational Geometry: An Introduction.* Springer-Verlag, New York, NY, 1985.

[32] T. Roos. Voronoi diagrams over dynamic scenes. *Discrete Appl. Math.*, 43:243–259, 1993.

[33] M. Sharir. Almost tight upper bounds for lower envelopes in higher dimensions. *Discrete Comput. Geom.*, 12:327–345, 1994.

[34] M. Sharir and P. K. Agarwal. *Davenport-Schinzel Sequences and Their Geometric Applications.* Cambridge University Press, New York, 1995.