Kinetic Binary Space Partitions for Intersecting Segments and Disjoint Triangles

(Extended Abstract)

Pankaj K. Agarwal* Jeff Erickson† Leonidas J. Guibas‡

Abstract

We describe randomized algorithms for efficiently maintaining a binary space partition of continuously moving, possibly intersecting, line segments in the plane, and of continuously moving but disjoint triangles in space. Our two-dimensional BSP has depth $O(\log n)$ and size $O(n \log n + k)$ and can be constructed in expected $O(n \log^2 n + k \log n)$ time, where k is the number of intersecting pairs. We can detect combinatorial changes to our BSP caused by the motion of the segments, and we can update our BSP in expected $O(\log n)$ time per change. Our three-dimensional BSP has depth $O(\log n)$, size $O(n \log^2 n + k')$, construction time $O(n \log^3 n + k' \log n)$, and update time $O(\log^2 n)$ (all expected), where k' is the number of intersections between pairs of edges in the xy-projection of the triangles. Under reasonable assumptions about the motion of the segments or triangles, the expected number of number of combinatorial changes to either BSP is $O(mn\lambda_s(n))$, where m is the number of moving objects and $\lambda_s(n)$ is the maximum length of an (n, s) Davenport-Schinzel sequence for some constant s.

1 Introduction

Hidden surface removal is a fundamental problem in computer graphics. Given a set of objects, a viewpoint, and an image plane, the hidden-surface-removal problem asks for computing the scene visible from the viewpoint on the image plane. A simple and widely used algorithm for hidden surface removal is the so-called *painter's algorithm* [10], which draws the objects on the image plane in a back-to-front order (also known as a *depth order*). Since such an ordering does not always exist, objects may need to be split so that their fragments admit a depth order. In a typical setting one wants to draw the same set of objects repeatedly from different viewpoints, which calls for a data structure to store these objects so that a depth order from a given viewpoint can be computed quickly. Fuchs *et al.* [11] introduced the *binary space partition* (BSP), based on earlier work by Schumacker *et al.* [21], as a data structure for computing a depth order from a viewpoint. Binary space partitions are also used to filter out a small subset of the input objects that comprises those visible from a given viewpoint. Even algorithms that rely on a hardware z-buffer cannot render very large scenes (composed of millions of polygons) in real time, so filtering a small superset of objects visible from a given viewpoint is a critical step in obtaining fast rendering algorithms [23]. Besides these two important applications, BSPs have been used for many other problems in graphics (including ray tracing [16] and shadow generation [7, 8]), solid modeling [15, 24], geometric data repair [14], robotics [4], etc.

A BSP $\mathcal{B}$ for a set S of objects in $\mathbb{R}^d$ is a binary tree, each of whose nodes v is associated with an open convex polytope Δ_v, called the *cell* of v, and the set $S_v = \{s \cap \Delta_v \mid s \in S\}$ of objects clipped to within Δ_v. (In our case S is either a set of segments in the plane or a set of triangles in $\mathbb{R}^3$.) If $S_v = \emptyset$, then v is a leaf of the binary space partition. Otherwise, Δ_v is partitioned into two convex polytopes by a *cutting hyperplane* h_v. We store with v the equation of h_v and the set $\{s \mid s \subseteq h_v, s \in S_v\}$ of objects in S_v that lie on h_v. The polytopes associated with the left and right children of v are $\Delta_v \cap h_v^-$ and $\Delta_v \cap h_v^+$, respectively where h_v^+ and h_v^- denote the open halfspaces bounded by h_v. The size of $\mathcal{B}$ is the the number of nodes in $\mathcal{B}$, together with the storage required to hold the information associated with each node. The efficiency of many BSP-based algorithms depends critically on the number of nodes in the tree. This dependence has motivated several algorithms for constructing BSPs of small size; see [2, 6, 11, 19, 20, 23, 24].

Most of the work to date on BSPs has dealt with static

*Center for Geometric Computing, Computer Science Department, Duke University, Box 90129, Durham, NC 27708–0129, agarwal@cs.duke.edu, http://www.cs.duke.edu/~agarwal. Research supported by National Science Foundation grant CCR-93-01259, by U.S. Army Research Office MURI grant DAAH04-96-1-0013, by a Sloan Fellowship, by a National Science Foundation NYI award and matching funds from Xerox Corporation, and by a grant from the U.S.-Israeli Binational Science Foundation.

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

‡Computer Graphics Laboratory, Computer Science Department, Stanford University, Stanford, CA 94305, guibas@cs.stanford.edu, http://www-graphics.stanford.edu/~guibas. Research partially supported by National Science Foundation grant CCR-9623851 and by U.S. Army Research Office MURI grant 5-23542-A.

objects. In practice, however, the set of objects changes over time — some objects move along continuous paths, some new objects are added, and some old objects are removed. Relatively little attention has been paid so far to developing efficient algorithms for updating a BSP dynamically. In this paper, we investigate the problem of maintaining a BSP as objects, either intersecting segments in the plane or disjoint triangles in space, move along continuous paths. Extant work in the graphics community on maintaining a BSP of moving objects is all based on discretizing time into short intervals. At these discrete times the moving objects are deleted from the BSP and re-inserted in their new positions; see, for example, [17, 25, 9]. Such approaches suffer from the fundamental problem that it is difficult to know how to choose the correct length of the discretization interval. If the interval is too small, then the BSP does not in fact change combinatorially, and the deletion/re-insertion is just wasted computation; on the other hand, if the interval is too big, important intermediate events can be missed, with ill effects on applications using the tree.

A more effective approach is to regard BSP as a *kinetic data structure*, as introduced by Basch *et al.* [5]. (See [3, 12, 18] for some other weaker models for solving kinetic problems.[1]) In the kinetic view each moving object follows a posted flight plan or path. In the binary space partition context, the equations of the cuts made at the nodes of the BSP become continuous functions of time. Combinatorial changes in the BSP (we define this notion precisely later), however, occur only at certain discrete times. We explicitly take advantage of the continuity of the motion of the objects involved so as to update the BSP only when actual events cause the BSP to change combinatorially. Such an approach was first used in a recent paper by Agarwal *et al.* [1] to maintain the BSP of a set of disjoint segments in the plane. In this paper we extend their approach to efficiently maintain a BSP for the considerably more complex case of disjoint triangles in $\mathbb{R}^3$. To achieve this result, we first develop a method to efficiently maintain a BSP for intersecting line segments in the plane; we are then able to "lift" our two-dimensional solution into three dimensions. These are the first kinetic structures proposed for these problems.

Our kinetic structures are based on new randomized algorithms for constructing static BSPs for these two situations. In Section 2 we present a randomized algorithm to construct a BSP for a set S of n stationary (possibly intersecting) segments in the plane. The algorithm constructs a BSP of expected size $O(n \log n + k)$ in expected time $O(n \log^2 n + k \log n)$, where k is the number of intersection points between the segments of S. We then extend this algorithm to construct a BSP of expected size $O(n \log^2 n + k')$ in expected time $O(n \log^3 n + k' \log n)$ for a set Δ of n disjoint triangles in space; here k' is the number of intersection points among the edges of the xy-projections of triangles in Δ. The previous best-known algorithms for triangles are by Agarwal *et al.* [1]. They present a randomized algorithm that constructs a BSP of size $O(n^2)$ in expected time $O(n^2 \log^2 n)$, and a deterministic algorithm that constructs a BSP of size $O((n + k') \log n)$ in time $O((n + k') \log^2 n)$. A shortcoming of all known algorithms (including ours) for constructing a BSP of triangles is that they may construct a BSP of size $\Omega(n^2)$ even though a BSP of size $O(n)$ exists. It is a challenging open problem to construct in polynomial time a BSP whose size is close to optimal (say, within a constant factor).

Neither the algorithms by Agarwal *et al.* nor the original algorithm by Paterson and Yao [19] is suitable for a kinetic data structure because a small motion of one of the objects may cause many or non-local changes to the BSP. In Section 3 we show that our static algorithms can be used to maintain the BSP of a set of moving segments or triangles. As in [1], we assume that the segment motions are *oblivious* to the random bits used by the algorithm. Following Basch *et al.* [5], we also assume that each moving segment has a posted flight plan that gives full or partial information about its current motion. Based on these flight plans, we maintain a priority queue of upcoming events that change the combinatorial structure of our BSP. We can process each such event in $O(\log n)$ time. Whenever a flight plan changes (possibly due to an external agent), our algorithm is notified and updates the global event queue to reflect the change. Furthermore, assuming that the motions of the segments are "pseudo-algebraic" and that m of the n segments are actually moving, we show an $O(mn\lambda_s(n))$ bound on the number of events processed, where $\lambda_s(n)$ is the maximum length of an (n, s) Davenport-Schinzel sequence, for some constant s.

The kinetic structure we propose for intersecting segments in the plane brings to light several important and subtle issues that did not arise in the disjoint segment context of [1]: intersections among the segments have to be detected and maintained; vertical threads from intersection points need to be propagated and their cost analyzed; when a segment becomes vertical during the motion, the entire ordering of intersection points on it flips, thus creating an event that can be expensive to process unless we can keep track of this ordering implicitly; and finally the analysis of the worst possible number of events that have to be processed by the kinetic BSP is substantially more challenging in this case.

[1] Atallah [3] and Ottmann and Wood [18] study kinetic geometric problems in an off-line setting, and Kahan [12] studies some problems under the assumption that the speed of the objects is bounded. The model introduced by Basch *et al.* is on-line and does not assume any bound on the rate at which the objects move.

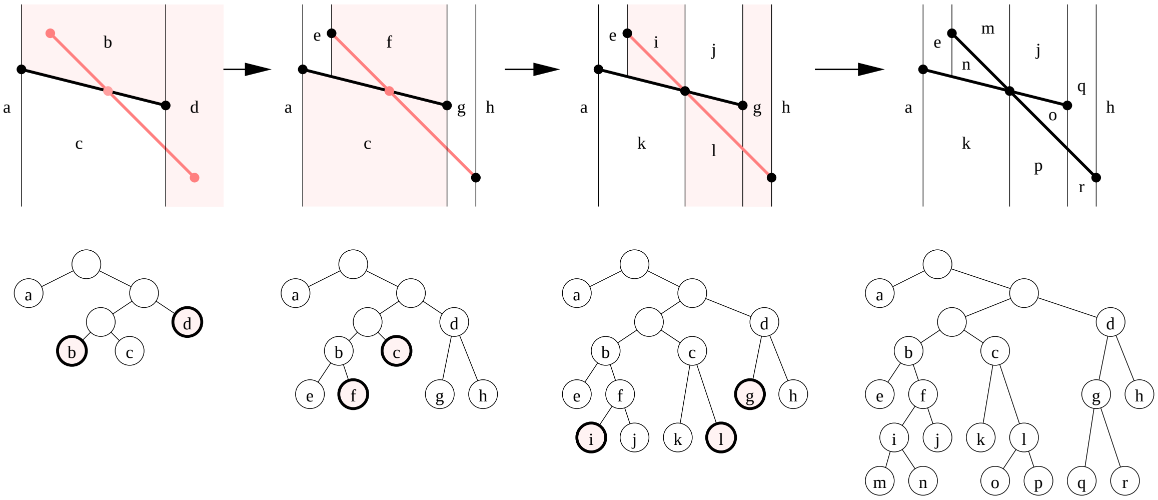


Figure 1. The sequence of cuts made while inserting a segment, and the resulting BSPs after each phase. In each phase, the shaded trapezoids and leaves are being split.

2 Static Algorithms

In this section, we will describe and analyze randomized algorithms for constructing binary space partitions for intersecting line segments in the plane and for disjoint triangles in $\mathbb{R}^3$.

2.1 Intersecting Planar Segments

Let S be a set of line segments in the plane. Our algorithm constructs a BSP $\mathcal{B} = \mathcal{B}(S)$ containing three types of cuts. A *vertex cut* is a vertical cut through an endpoint of a segment. An *intersection cut* is a vertical cut through the intersection point of two segments. An *edge cut* is a cut along (a contiguous subset of) a segment. Vertex cuts and intersection cuts are collectively referred to as *point cuts*. For every node v in $\mathcal{B}$, the corresponding convex polygon Δ_v is a trapezoid with two vertical edges (one of which can have length zero) induced by point cuts, and top and bottom edges induced by edge cuts; some cells are unbounded.

We begin by choosing a random permutation $\langle s_1, s_2, \ldots, s_n \rangle$ of S. Once the permutation is chosen, our algorithm is completely deterministic. We insert the segments one at a time in the chosen order and maintain a BSP of the segments added so far. Whenever $i < j$, we say that s_i has *higher priority* than s_j; segments with higher priority are inserted earlier. Let $\mathcal{B}_i$ denote the BSP after the first i segments are inserted; $\mathcal{B}_0$ is the trivial tree containing one node, and $\mathcal{B}_n = \mathcal{B}$. We insert s_i into $\mathcal{B}_{i-1}$ in three phases.

1. We insert vertex cuts into the cell(s) containing the endpoints of s_i.
2. For each $j < i$ such that s_j intersects s_i, we insert intersection cuts into the cells above and below the point $s_i \cap s_j$.
3. We insert an edge cut along s_i into each cell that intersects the interior of s_i.

Within each of the three phases, the cuts can be inserted in any order. Each leaf of $\mathcal{B}_{i-1}$ is the ancestor of at most four leaves of $\mathcal{B}_i$. An example of our insertion algorithm is illustrated in Figure 1.

Since each interior node is associated with a cut, the size of $\mathcal{B}$ is twice the number of cuts made by our algorithm. There are clearly $2n$ vertex cuts and $2k$ intersection cuts in $\mathcal{B}$. As Paterson and Yao observe [19], the number of edge cuts contributed by a segment s_i is one more than the number of point cuts that cross s_i. Thus, to compute the size of $\mathcal{B}$, it suffices to count the segments crossed by each point cut.

Lemma 2.1. *The expected number of segments crossed by each vertex and intersection cut are $O(\log n)$ and $O(1)$, respectively.*

Proof: The portion of the point cut that extends above (resp. below) an endpoint or an intersection point of p is called the *upper thread* (resp. *lower thread*) of p. The segment containing the other endpoint of the thread is called the thread's *stopper*. Note that the stopper of a vertex thread is inserted before than the segment containing the vertex, and the stopper of an intersection thread is inserted before at least one of the two segments containing the intersection point.

Let p be an endpoint of a segment $s \in S$, and let $\sigma_1, \sigma_2, \ldots, \sigma_m$ be the sequence of segments intersected by a vertical ray shooting upwards (in the positive y-direction) from p. The upper thread of p crosses σ_i if and only if s is inserted before any of the segments $\sigma_1, \ldots, \sigma_i$. Since the segments are inserted in random order, the probability that the upper thread crosses σ_i

is precisely $1/(i+1)$. Thus, the expected number of segments crossed by the upper thread of p is $H_{m+1}-1 \leq H_n - 1 = O(\log n)$.

Next, let p be the intersection point of two segments $s, s' \in S$, and let $\sigma_1, \sigma_2, \ldots, \sigma_m$ be the sequence of segments intersected by a vertical ray shooting upwards from p. The upper thread of p crosses σ_i if and only if both s and s' are inserted before any of the segments $\sigma_1, \ldots, \sigma_i$. Since the segments are inserted in random order, the probability that the upper thread crosses σ_i is precisely $2/(i+2)(i+1)$. The expected number of segments crossed by the upper thread is

$$\sum_{i=1}^{m} \frac{2}{(i+2)(i+1)} = 1 - \frac{2}{m+2} < 1.$$

The lower threads are analyzed symmetrically. □

Lemma 2.2. *The depth of $\mathcal{B}$ is $O(\log n)$ with high probability.*[2]

Proof: First we compute the depth of an arbitrary point p in the plane, that is, the depth of the leaf v whose cell Δ_v contains p. We will say that the cuts associated with the nodes on the path from the root to v *see* the point p. We separately count the vertex cuts, edge cuts, and intersection cuts that see p.

Let $\sigma_1, \sigma_2, \ldots$ be the sequence of segments intersected by a vertical ray shooting upwards (in the positive y-direction) from p. An edge cut through σ_i sees p if and only if σ_i is inserted before any of $\sigma_1, \ldots, \sigma_{i-1}$. This occurs with probability $1/i$. It follows that the total expected number of edge cuts that see p from above is at most $H_n = O(\log n)$. Let X be the number of edge cuts that see p from above. Since X is the sum of independent indicator variables and $\mu = E[X] \leq H_n$, by Chernoff's bound,

$$\Pr[X > \alpha\mu] < \left(\frac{e^{\alpha-1}}{\alpha^\alpha}\right)^\mu = O(n^{-\alpha \ln \alpha + \alpha - 1}),$$

for any constant α [13, p.68]. In particular, for any constant c, we can choose α so that $\Pr[X > \alpha H_n] < n^{-c}$. Similarly, we can bound the number of edge cuts that see p from below.

Now let $\pi_1, \pi_2, \ldots$ be the *left* segment endpoints that lie to the *left* of p. In order for a vertex cut through π_i to see p, the segment containing π_i must be inserted before the segments containing $\pi_1, \ldots, \pi_{i-1}$. By our earlier analysis, the number of such cuts is $O(\log n)$ with high probability. Similar arguments apply to the right endpoints to the left of p and the endpoints to the right of p.

It remains to count the intersection cuts that see p. Every intersection cut χ is defined by a pair of segments (s_i, s_j) where $j > i$. If χ is made at a node v of $\mathcal{B}$, then either the top or the bottom edge of the trapezoid Δ_v is a portion of s_i, and the intersection point $s_i \cap s_j$ lies on this edge. Suppose χ sees p; then p must lie in Δ_v. We call χ *essential* if the lower-priority segment s_j does not intersect the vertical line through p inside Δ_v (*e.g.*, if s_i lies above p and $s_i \cap s_j$ lies to the right of p, then slope of s_j is less than that of s_i); otherwise χ is called a *nonessential*. A nonessential cut that sees p is immediately preceded or followed by a vertex cut, an edge cut, or an essential intersection cut that also sees p. Thus, we only need to count the essential intersection cuts. We will explicitly consider only essential cuts (σ, τ) where σ lies above p and the intersection point $\sigma \cap \tau$ lies to the right of p (and therefore τ has smaller slope than σ); let us refer to such cuts as *north-east* cuts. Other cases are handled symmetrically.

Recall that $\sigma_1, \sigma_2, \ldots$ is the sequence of segments above p. For each segment σ_i, let p_i be the point on σ_i directly above p, and let $\tau_{i1}, \tau_{i2}, \ldots, \tau_{im_i}$ be the sequence of segments that intersect σ_i to the right of p_i and have slope less than σ_i. Any north-east cut is defined by some pair (σ_i, τ_{ij}). If this cut sees p, then σ_i and τ_{ij} must be inserted (in that order) before any of the segments $\sigma_1, \ldots, \sigma_{i-1}, \tau_{i1}, \ldots, \tau_{i,j-1}$. The probability of this event is $1/(i+j)(i+j-1)$. It follows that the expected number of north-east cuts that see p is at most

$$\sum_{i=1}^{n} \sum_{j=1}^{n-i} \frac{1}{(i+j)(i+j-1)} = H_n - 1.$$

Moreover, by our earlier analysis, the number of north-east cuts that see p is $O(\log n)$ with high probability.

We conclude that the depth of any point is $O(\log n)$ with high probability.

The segments in S, together with vertical lines through every endpoint and intersection point, split the plane into $O(n^3)$ trapezoids. Two points in the same trapezoid are on the same side of every cut in any BSP that our algorithm constructs, so the depth of $\mathcal{B}$ is the maximum of the depths of only $O(n^3)$ points, one from each trapezoid. Since each of these points has depth $O(\log n)$ with high probability, the depth of $\mathcal{B}$ is also $O(\log n)$ with high probability. □

Theorem 2.3. *Let S be a set of n segments in $\mathbb{R}^2$, and let k be the number of intersecting pairs of segments in S. The BSP $\mathcal{B}(S)$ has expected size $O(n \log n + k)$ and expected depth $O(\log n)$.*

2.2 Disjoint Triangles in Space

Let S be a set of triangles in space, and let E be the set of edges of the triangles in S. For any object s in

[2] *i.e.*, with probability $1 - n^{-c}$ for any constant c, where the constant hidden in the big-Oh depends on c.

$\mathbb{R}^3$, let s^* denote its orthogonal projection onto the xy-plane. We say that two objects in $\mathbb{R}^3$ *overlap* if their xy-projections intersect. Let k denote the number of pairs of edges in E that overlap, or equivalently, the number of intersections between segments in $E^* = \{e^* \mid e \in E\}$.

Just as in the previous section, our algorithm constructs $\mathcal{B} = \mathcal{B}(S)$ by choosing a random permutation $\langle s_1, s_2, \ldots, s_n \rangle$ of S and inserting the triangles one at a time in the chosen order. Our algorithm uses four types of cuts. A *vertex cut* is a cut parallel to the yz-plane through a triangle vertex. An *edge cut* is a cut parallel to the z-axis along (a contiguous subset of) a triangle edge. An *intersection cut* is a cut parallel to the yz-plane through the intersection of an edge of one triangle and an edge cut defined by an earlier triangle.[3] Finally, a *surface cut* is a cut along (a contiguous subset of) a triangle. Again, vertex cuts and intersection cuts are collectively referred to as *point cuts*. For every node v in $\mathcal{B}$, the corresponding polytope Δ_v is a four-sided cylinder whose xy-projection is a trapezoid with two facets defined by point cuts and two defined by edge cuts; the top and bottom faces of the cylinder are defined by surface cuts.

We insert s_i into $\mathcal{B}_{i-1}$ in four phases.

1. We insert vertex cuts into the cell(s) containing the vertices of s_i.
2. For each $j < i$, we insert intersection cuts through any point on the boundary of s_i that overlaps a point on the boundary of s_j.[4]
3. For each edge e of s_i, we insert an edge cut along e into each cell that intersects e.
4. Finally, we insert a surface cut along s_i into each cell that intersects s_i.

Within each phase, the cuts can be inserted in any order. Each leaf of $\mathcal{B}_{i-1}$ is the ancestor of a constant number of (at most 15) leaves of $\mathcal{B}_i$.

Our three-dimensional BSP $\mathcal{B}$ induces a *shadow BSP* $\mathcal{B}^*$ on the projected triangles S^*. The cuts in $\mathcal{B}^*$ are the projections of vertical cuts in $\mathcal{B}$ onto the xy-plane; surface cuts in $\mathcal{B}$ do not contribute to $\mathcal{B}^*$. The shadow BSP is almost exactly the same as the cylindrical BSP described in the previous section for the projected edges E^*. The only difference is that instead of inserting all $3n$ edges in random order, the three edges of each triangle are inserted together. This difference has minimal effect on the analysis; the shadow BSP still has expected size $O(n \log n + k)$ and expected depth $O(\log n)$. It immediately follows that our three-dimensional BSP has only $O(n \log n + k)$ point and edge cuts.

[3] It might be more consistent to call these "overlap cuts", but we want to emphasize the similarity with the two-dimensional case.

[4] Actually, we only need to insert intersection cuts through the intersections of edges of s_i and previous edge cuts, but the extra cuts simplify our analysis.

Lemma 2.4. *The expected number of surface cuts is $O(n \log^2 n + k)$.*

Proof: Let $\mathcal{B}'_i$ be the BSP just before the surface cuts for s_i are inserted. The intersection of s_i with the cells of $\mathcal{B}'_i$ forms a decomposition of s_i into trapezoids. Since the decomposition is a planar graph, the number of trapezoids is at most three times the number of vertices. Every vertex of the decomposition is the intersection of s_i, a point cut in $\mathcal{B}'_i$, and an edge cut in $\mathcal{B}'_i$. Thus, to count the surface cuts, it suffices to count these triple intersection points.

Let p be an arbitrary point in $\mathbb{R}^3$. For any object (or set of objects) X in $\mathbb{R}^3$, Let $X|_p$ denote the intersection of (the objects in) X with the plane Π_p through p normal to the x-axis. The BSP $\mathcal{B}$ induces a *slice BSP* $\mathcal{B}|_p$ for the set of disjoint segments $S|_p$. The cuts in $\mathcal{B}|_p$ are the intersections of the cuts in $\mathcal{B}$ with Π_p. Point cuts in $\mathcal{B}$, and other cuts that do not meet the plane, do not contribute to $\mathcal{B}|_p$. We easily observe that $\mathcal{B}|_p$ is a cylindrical BSP of the disjoint segments $S|_p$, exactly as described by Agarwal *et al.* [1] and in the previous section. To emphasize their connections to cuts in $\mathcal{B}$, we refer to vertex cuts in $\mathcal{B}|_p$ as *endpoint* cuts, since they are induced by edges in S, and edge cuts in $\mathcal{B}|_p$ as *segment* cuts, since they are induced by surface cuts in $\mathcal{B}$. Each slice BSP has expected size $O(n \log n)$ and expected depth $O(\log n)$.

Now consider the triple intersection points contained in the vertex cut through a triangle vertex p. Each of these triple intersection points is also a vertex in the planar decomposition defined by the slice BSP $\mathcal{B}|_p$, and lies within the three-dimensional cell C split by the point cut. These vertices also lie in the two-dimensional cell $C|_p$ split by the endpoint cut through p ($= p|_p$) in $\mathcal{B}|_p$. Since the decomposition of $C|_p$ by $\mathcal{B}|_p$ is a planar graph, the number of vertices is a constant multiple of the number of trapezoids in the decomposition, which is equal to the size of the subtree of $\mathcal{B}|_p$ rooted at the endpoint cut through p.

In the full version of the paper, we show that for any endpoint p in any set S of n disjoint segments in the plane, the expected size of the subtree of $\mathcal{B}(S)$ rooted at the endpoint cut through p is $O(\log^2 n)$. Thus, the expected number of triple intersection points contained in each vertex cut in our three-dimensional BSP is $O(\log^2 n)$. By a similar argument, the expected number of triple intersection points contained in each intersection cut is only a constant. $\square$

Theorem 2.5. *Let S be a set of disjoint triangles in $\mathbb{R}^3$, and let k be the number of intersections between edges of S^*. The BSP $\mathcal{B}(S)$ has expected size $O(n \log^2 n + k)$ and expected depth $O(\log n)$.*

3 Kinetic Algorithms

In this section, we describe how to maintain the BSPs constructed in the previous section, as the set S of input objects moves continuously. Each segment or triangle is specified by the positions of its endpoints. If each object of S is moving rigidly, then not all the coordinates of the vertices of an object are independent. For example, we need three parameters to denote the position of a segment in the plane while the coordinates its two endpoints have four parameters. However, our algorithm, as well as the analysis, works even if we assume that the vertices move independently, that is, even if the segments (or triangles) are not only moving rigidly but they are also expanding and shrinking continuously with time. For the sake of simplicity, let us assume that the segments (resp. triangles) do not shrink to points (resp. line segments). We suppose also that each vertex has a posted *flight path* specifying its position as a continuous function of time.

Both of our static algorithms begin by choosing a random permutation of the objects and inserting them in a BSP in the chosen order. Once this permutation is chosen, it remains fixed for all time. At any moment in time, the BSP maintained by our kinetic algorithm is precisely the BSP that would be constructed by the corresponding static algorithm, given the sequence of objects in their current positions as input. We assume that the flight paths of the vertices are chosen completely independently of the original permutation of the objects.

For any real value t, let t^- and t^+ respectively denote $t-\varepsilon$ and $t+\varepsilon$ for some sufficiently small constant $\varepsilon > 0$.

3.1 Intersecting Line Segments

Let $S = \langle s_1, s_2, \ldots, s_n \rangle$ be a sequence of segments in random order. Let $S(t)$ denote the positions of the segments in S at time t, and let $\mathcal{B}(t)$ denote the BSP of $S(t)$ constructed by our static algorithm.

The combinatorial structure of $\mathcal{B}(t)$ changes at time t if some segment s_i rotates through a vertical line at time t, or if some segment s_i intersects Δ_v at time t^- but not at time t^+, or vice-versa, for some leaf v in $\mathcal{B}_{i-1}$. We refer to such times t as *critical events*. Since the endpoints of segments in S are moving continuously, a segment may leave or enter a cell Δ_v of $\mathcal{B}_{i-1}$ only in one of the following three ways: (1) an endpoint of s_i passes through the left or right edge of Δ_v, (2) an endpoint of s_i passes through the top or bottom edge of Δ_v, or (3) the interior of s_i passes through a vertex of Δ_v. We will refer to these critical events as *vertex*, *intersection*, and *edge* events, respectively. We will call a segment rotating through a vertical line a *flip* event.

Following [1], we say that a node w of $\mathcal{B}$ is *transient* if the parent of w is associated with a point cut and Δ_w contains no vertices or intersection points, so the subtree rooted at w contains only edge cuts. To detect critical events, we maintain three types of *certificates*, which guarantee the combinatorial structure of our BSP: (1) a certificate for each transient node v, which becomes invalid when the cell Δ_v collapses, (2) a certificate for each segment endpoint p_i, which becomes invalid when the next segment above or below p_i changes, and (3) a certificate for each segment s_i, which becomes invalid when s_i flips. We easily observe that the combinatorial structure of our BSP does not change as long as these certificates remain valid, and that at least one certificate becomes invalid at each critical event. We assume that the expiration time of each certificate can be computed in constant time.

The following lemma is easy to prove.

Lemma 3.1. *The combinatorial structure $\mathcal{B}(S)$ changes only at critical events. There are $O(n + k)$ valid certificates at any time, and only $O(1)$ certificates can change at any critical event. Changing the flight plan of a segment s_i changes $O(k_i + 1)$ certificates, where k_i is the number of segments that intersect s_i.*

We maintain a priority queue of critical events, ordered chronologically. At each critical event t, we must transform $\mathcal{B}(t^-)$ into $\mathcal{B}(t^+)$, update the certificates, and update the event queue. In the rest of the subsection we describe how we handle each of the critical events. Due to lack of space, we defer most of the details to the full paper.

3.1.1 Vertex Events

A vertex event occurs when an endpoint p_i of a segment s_i passes through the left or right side of the trapezoid Δ_v for some leaf v of $\mathcal{B}_{i-1}$. Figure 2 illustrates four types of vertex events; all other cases can be obtained by horizontal or vertical reflection, time reversal, or both. In each case, p_i is the right endpoint of s_i, and p_i is moving to the right through a point cut defined by a point above p_i.

Three of the four cases already occur when the segments are disjoint, and were described and analyzed by Agarwal *et al.* [1]. The only new case occurs when p_i passes directly under the intersection point $s_j \cap s_{j'}$, where s_j is the top edge of the cell containing p_i at time t^-, and $s_{j'}$ is the top edge of cell containing p_i at time t^+. This is the first case in Figure 2. However, this can also be handled using the same algorithm. Omitting further details, we conclude:

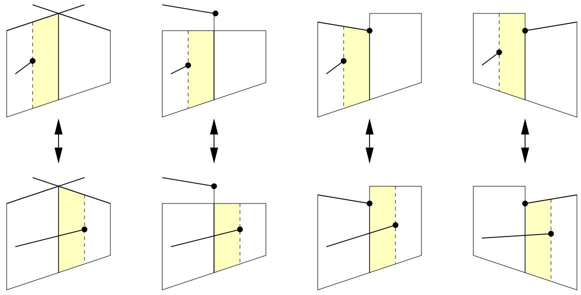


Figure 2. Four types of vertex events. Shaded cells are transient.

Lemma 3.2. *The expected cost of a vertex event is $O(\log n)$. If m the segments are moving along pseudo-algebraic paths and the other $n-m$ remain stationary, there are $O(mn^2)$ vertex events.*

3.1.2 Edge Events

An edge event occurs whenever a segment s_i passes through a corner of a trapezoid Δ_v for some leaf v of $\mathcal{B}_{i-1}$. There are essentially five distinct types of edge events, illustrated in Figure 3(a). All other cases can be obtained from these by horizontal or vertical reflection, time reversal, or both. Due to lack of space we only describe the first illustrated case: a segment s_i passes over the endpoint of s_j, for some $j < i$. The other cases are either easier or can be handled in a similar manner.

Suppose s_i passes over the left endpoint of s_j at time t, for some $j < i$, and s_i intersects s_j at time t^+. We now have to add intersection cuts at two leaves $\mathcal{B}_{i-1}$; see Figure 3(a). Let v be one of these leaves. Suppose v is the left child of its parent u in $\mathcal{B}(t^-)$. We create a new node w, associated with the new intersection cut, make v the right child of w, and make w the new left child of u. We also create a new node v_L and make it the left child of w. See Figure 3(b). We create a new subtree, rooted at v_L, which contains the edge cuts that intersect the left edge of Δ_v, using the following recursive algorithm. Here left(v) and right(v) are the left and right children of v, and cut(v) is the cut associated with v.

```
LeftShave(v) :
    Create a new node v_L
    if cut(v) is an edge cut
        cut(v_L) ← cut(v)
        left(v_L) ← LeftShave(left(v))
        right(v_L) ← LeftShave(right(v))
    else if cut(v) is a point cut
        v_L ← LeftShave(left(v))
    return v_L
```

The edge cuts in the new subtree are induced by exactly the edges that intersect the new intersection cut. By Lemma 2.1, the expected number of edges is a constant. Lemma 2.2 implies that the path from v to any leaf has $O(\log n)$ edges with high probability. It follows that the expected time to create the new subtree is $O(\log n)$.

We may also have to insert a new edge cut into v_L, by creating a new node with a leaf as its right child and v_L as its left child. We conclude:

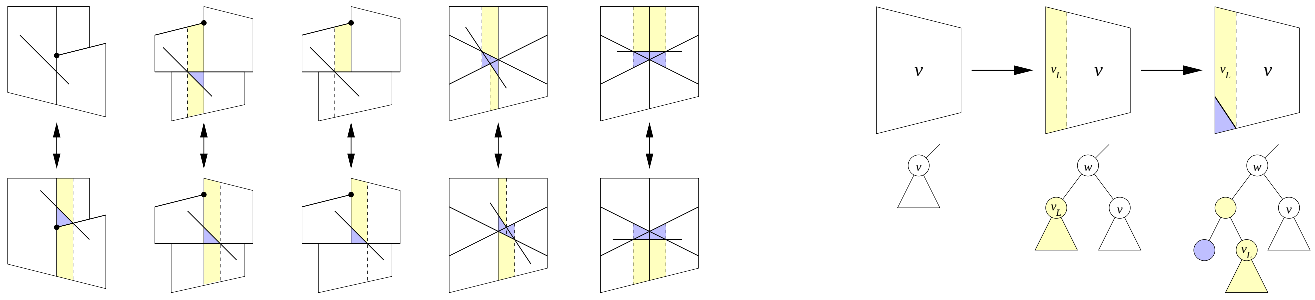
Lemma 3.3. *The expected cost of an edge event is $O(\log n)$.*

If m of the segments are moving along pseudo-algebraic paths, there are clearly at most $O(mn^3)$ edge events, since each such event involves at most four segments, of which at least one must be moving. We can construct a *sequence* S of n segments, m of which are translating along horizontal paths, such that $\mathcal{B}(S)$ must change $\Omega(mn^3)$ times. However, the expected number of edge events for any *set* of segments is significantly smaller.

Lemma 3.4. *If m segments in S move along pseudo-algebraic trajectories, and the remaining $m-n$ segments remain stationary, the expected number of edge events is $O(mn\lambda_{s+2}(n))$ for some positive integer s.*

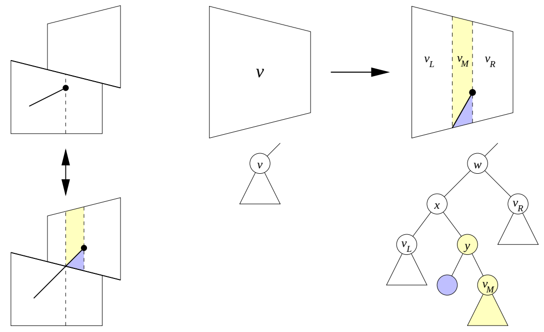
Proof (sketch): There are four different types of combinatorial changes that can cause an edge event: two intersection points switching their x-coordinate order, an endpoint and an intersection point switching their x-coordinate order, a segment passing over an endpoint, or a segment passing over an intersection point. The second, third, and fourth types of changes can occur only $O(mn^2)$ times, regardless of the insertion order of the segments. It therefore suffices to bound the edge events involving two intersection points.

Fix two segments $s_i, s_j \in S$, and let $l_{ij}(t)$ denote the vertical line through their intersection point at time t, or the empty set if s_i and s_j do not intersect at time t. Let $\Gamma_{ij}(t)$ be the y-coordinates of the intersection of $l_{ij}(t)$ and $S(t)$. (If s_i and s_j do not intersect at time t, then $\Gamma_{ij}(t)$ is empty.) If we plot $\Gamma_{ij}(t)$ as a function of t, the result is a collection of monotone Jordan arcs in the (y,t)-plane. Call this collection of Jordan arcs Γ_{ij}. Because the segments are moving pseudo-algebraically, each segment $s_k \in S \setminus \{s_i, s_j\}$ contributes a constant number of arcs to Γ_{ij}, which we will label σ_k. Each intersection point $\sigma_k \cap \sigma_l$ corresponds to a time at which the two intersection points $s_i \cap s_j$ and $s_k \cap s_l$ have the same x-coordinate, and thus each double intersection event in which one of the intersection points is $s_i \cap s_j$ corresponds to an intersection point $\sigma_k \cap \sigma_l$. However, not every intersection point of Γ_{ij} corresponds to an event. The expected number of intersection points that correspond to an event is only $O(\lambda_{s+2}(n))$, where the constant s is the maximum number of times that two arcs in Γ_{ij} intersect, and $\lambda_{s+2}(n)$ is the maximum



(a) (b)

Figure 3. (a) Five types of edge events. (b) Shaving the left side of a node. Lightly shaded cells are transient; darker cells are leaves.



(a) (b)

Figure 4. (a) An intersection event. (b) Splitting a node. Lightly shaded cells are transient; darker cells are leaves.

length of an $(n, s+2)$ Davenport-Schinzel sequence [22], a (very) slightly superlinear function of n. Summing over all pairs i, j, the total number of events is only $O(n^2\lambda_{s+2}(n))$.

We obtain an improved bound by further classifying the events according to which of the segments s_i, s_j, s_k, s_l are stationary and which are moving. We omit further details from this extended abstract. □

3.1.3 Intersection Events

An intersection event occurs when an endpoint of s_i passes through s_j, for some $j < i$. Up to reflections and time reversal, there is only one type of intersection event, illustrated in Figure 4(a): the right endpoint p_i of s_i passes upwards through s_j, creating a new intersection cut. The event affects two leaves of $\mathcal{B}_{i-1}$, one above and one below s_j.

Updating the lower node is trivial; we only have to change the vertex cut to an intersection cut. To update the upper node v, we first split the subtree rooted at v into three subtrees v_L, v_M, v_R containing the cuts to the left, between, and to the right of the new point cuts, respectively, using an algorithm similar to the one we used for edge events. To complete the update, we then perform some pointer manipulation, which is illustrated in Figure 4(b). We omit further details.

Lemma 3.5. *The expected cost of an intersection event is $O(\log n)$. If m of the segments move along pseudo-algebraic paths and the other $n-m$ remain stationary, there are $O(mn)$ intersection events.*

3.1.4 Flip Events

If the segment s_i flips over, we must update the subtrees rooted at every leaf v of $\mathcal{B}_{i-1}$ whose cell intersects s_i. Fortunately, each of these subtrees can be updated using the same algorithm. The various cases (up to reflection) are illustrated in Figure 5(a). Let x be the grandchild of v whose cell lies between the two point cuts and is associated with the edge cut through s_i. Note that the subtree rooted at x contains only edge cuts and intersection cuts through s_i. The y-coordinate order of the edge cuts in this subtree remains unchanged by the flip event, but the x-coordinate order of the intersection cuts is exactly reversed. Thus, in addition to the usual constant amount of pointer manipulation, shown in Figure 5(b), we also traverse the subtree rooted at x, swapping the children of every descendant associated with an intersection cut. If we are careful to avoid descendants of x whose cells do not intersect s_i, the number of edges we traverse to flip the subtree is a constant times the number of edges s_j that intersect s_i, with $j > i$. Altogether, we modify a constant number of nodes for every segment that intersects s_i.

Lemma 3.6. *The expected cost of a flip event is $O(k_i \log n)$, where k_i is the number of segments in S that intersect the flipping segment. If m of the segments move along pseudo-algebraic paths and the other $n-m$ remain stationary, there are $O(m)$ flip events, and the total time spent handling them is $O(k' \log n)$, where k' is the number of pairs of segments that ever intersect.*

Unlike every other kind of event, which we can handle in $O(\log n)$ time, the worst-case cost of a flip event is $O(n \log n)$. However, by storing some extra information with the BSP, we can reduce the cost of a flip event to $O(\log n)$. We construct the *augmented* BSP $\widetilde{\mathcal{B}}(S)$ as follows. If a segment s_i induces two point cuts

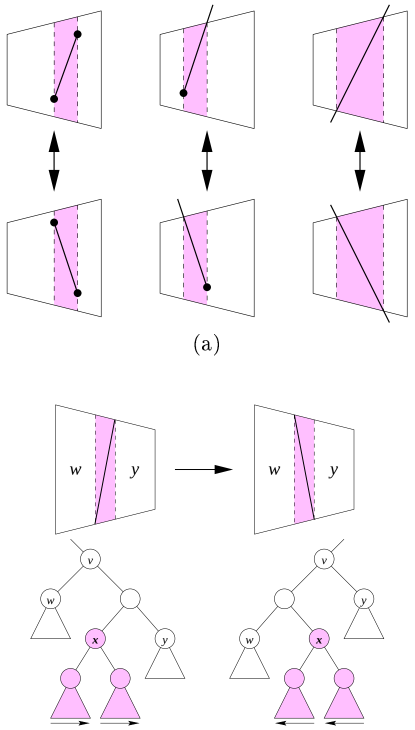


Figure 5. (a) Possible effects of a flip event. (b) How to flip a node. Shaded cells contain no segment endpoints.

inside a leaf of $\mathcal{B}_{i-1}$, instead of inserting two binary nodes, we insert a *ternary* node whose three children correspond to the cells to the left, between, and to the right of the cuts. This lets us avoid the local pointer manipulation. To avoid swapping the children of intersection cuts, we mark one endpoint of s_i, and let the sense of an intersection cut defined by $s_i \cap s_j$, where $i < j$, depend on whether the marked endpoint is further to the right or to the left. If the marked endpoint is to the left, then intersection cuts are handled normally: the left child of an intersection cut corresponds to the points geometrically to the left of the cut. If the marked endpoint is to the right, then intersection cuts are interpreted "backwards": the left child of an intersection cut corresponds to the points geometrically to the *right* of the cut. We now easily observe that flip events do not change $\widetilde{\mathcal{B}}$ at all. Moreover, with only minor changes to the algorithm, we can use $\widetilde{\mathcal{B}}$ in any application of true binary space partition trees, since it implicitly encodes the structure of $\mathcal{B}$. Each vertex, intersection or edge event can create or destroy at most one ternary node, but this has minimal effect on our analysis; Lemmas 3.2–3.5 still apply.

Putting everything together, we obtain the following main result.

Theorem 3.7. *If m segments in a set S of n segments move along pseudo-algebraic trajectories, and the remaining $n - m$ segments remain stationary, the total expected time spent maintaining $\mathcal{B}(S)$ (or $\widetilde{\mathcal{B}}(S)$) is $O(mn\lambda_{s+2}(n)\log n)$ for some positive integer s.*

3.2 Disjoint Triangles in Space

We now briefly describe how to extend the algorithm for segments to maintain the BSP $\mathcal{B} = \mathcal{B}(S)$ of a set S of n disjoint triangles in $\mathbb{R}^3$.

The combinatorial structure of $\mathcal{B}$ changes when a triangle rotates through a vertical plane, an edge of a triangle rotates through a line parallel to the yz-plane, a triangle s_i enters a cell Δ_v of $\mathcal{B}_{i-1}$, or a triangle s_i leaves a cell Δ_v of $\mathcal{B}_{i-1}$. Since we assume the triangles in S to be always disjoint, a triangle s_i can leave or enter a cell of $\mathcal{B}_{i-1}$ only in one of two ways: a vertex of s_i passes through a vertical face of Δ_v or an edge of s_i passes through a vertical edge of Δ_v. The former corresponds to vertex and intersection events of segments in the plane, and the latter corresponds to edge events. An edge of s_i rotating through a line parallel to the yz-plane is the same as the flip event (Section 3.1.4). The only new event is a triangle rotating through a vertical plane, which we will refer to as a *flop* event. In this case we update the subtrees rooted at every leaf of $\mathcal{B}_{i-1}$ whose cell intersects s_i. It can be shown that the expected number of cells to be updated is $O(n\alpha(n)\log n)$ and that each can be updated in $O(1)$ time; here $\alpha(n)$ is the inverse Ackermann function.

Each vertex, intersection, and edge event can be handled in $O(\log^2 n)$ expected time, each edge flip event in $O(n \log n)$ expected time, and each face flop event in $O(n\alpha(n)\log n)$ expected time. We can augment the three dimensional BSP so that edge flip events only require $O(\log n)$ time, and face flop events cost nothing. Hence, we obtain the following result.

Theorem 3.8. *If m triangles in a set S of n triangles move along pseudo-algebraic trajectories, and the remaining $n - m$ segments remain stationary, the total expected time spent maintaining $\mathcal{B}(S)$ (or $\widetilde{\mathcal{B}}(S)$) is $O(mn\lambda_{s+2}(n)\log^2 n)$ for some positive integer s.*

4 Open Problems

Our static algorithm for triangles in $\mathbb{R}^3$ is optimal in the worst case, since there are sets of triangles for which every BSP has size $\Omega(n^2)$. However, like earlier algorithms [1, 19, 23], there are inputs for which our algorithm will build a BSP of quadratic size, even when a linear-size BSP is possible. How quickly can one construct binary space partitions whose size is optimal, or within a constant factor of optimal? No polynomial-time algorithm is currently known, even for disjoint line segments in the plane.

Our kinetic algorithms are *responsive*; that is, each critical event causes changes the BSP only locally, and we can update the tree in only polylogarithmic time. However, the number of events may be larger than necessary. For example, a different BSP tree might process fewer events for the same motions. Could *a priori* knowledge of the motions be exploited in this way? Conversely, can we prove lower bounds on the number of combinatorial changes that any kinetic BSP must undergo in the worst case, or their total cost, under some specific class of motions?

The analysis of our kinetic algorithms assumes that the motion of the objects is completely independent of the random permutation used to construct the original static BSP. This assumption may not be totally realistic. Flight paths can be specified interactively by user input. By observing the response time of the algorithm, a malicious user could learn which motions cause significant delays, and invalidate our independence assumption by performing only those motions. However, the algorithm could then respond by evolving the BSP according to a random walk in the space all permutations defining the BSP. An interchange of the positions of two adjacent segments along such a permutation can be reflected in the BSP quickly, and a sequence of such interchanges is known to be rapidly mixing. Exactly how to schedule and analyze such random interchanges and how to argue that one can defeat a malicious user this way remain interesting open problems.

Acknowledgments. The authors would like to thank João Comba and T. M. Murali for useful discussions.

References

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

[2] J. M. Airey. *Increasing Update Rates in the Building Walkthrough System with Automatic Model-space Subdivision and Potentially Visible Set Calculations*. Ph.D. thesis, Dept. of Computer Science, University of North Carolina, Chapel Hill, 1990.

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

[4] C. Ballieux. Motion planning using binary space partitions. Tech. Rep. Inf/src/93-25, Utrecht University, 1993.

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

[6] T. Cassen, K. R. Subramanian, and Z. Michalewicz. Near-optimal construction of partitioning trees by evolutionary techniques. *Proc. Graphics Interface '95*, pp. 263–271, 1995.

[7] N. Chin and S. Feiner. Near real-time shadow generation using BSP trees. *Comput. Graph.* 23:99–106, 1989. Proc. SIGGRAPH '89.

[8] N. Chin and S. Feiner. Fast object-precision shadow generation for areal light sources using BSP trees. *Comput. Graph.* 25:21–30, Mar. 1992. Proc. 1992 Sympos. Interactive 3D Graphics.

[9] Y. Chrysanthou. *Shadow Computation for 3D Interaction and Animation*. Ph.D. thesis, Queen Mary and Westfield College, University of London, 1996.

[10] J. D. Foley, A. van Dam, S. K. Feiner, J. F. Hughes, and Phillips. *Introduction to Computer Graphics*. Addison-Wesley, Reading, MA, 1993.

[11] H. Fuchs, Z. M. Kedem, and B. Naylor. On visible surface generation by a priori tree structures. *Comput. Graph.* 14:124–133, 1980. Proc. SIGGRAPH '80.

[12] S. Kahan. A model for data in motion. *Proc. 23th Annu. ACM Sympos. Theory Comput.*, pp. 267–277, 1991.

[13] R. Motwani and P. Raghavan. *Randomized Algorithms*. Cambridge University Press, New York, NY, 1995.

[14] T. M. Murali and T. A. Funkhouser. Consistent solid and boundary representations from arbitrary polygonal data. *Proc. 1997 Sympos. Interactive 3D Graphics*, 1997.

[15] B. Naylor, J. A. Amanatides, and W. Thibault. Merging BSP trees yields polyhedral set operations. *Comput. Graph.* 24:115–124, Aug. 1990. Proc. SIGGRAPH '90.

[16] B. Naylor and W. Thibault. Application of BSP trees to ray-tracing and CSG evaluation. Technical Report GIT-ICS 86/03, Georgia Institute of Tech., School of Information and Computer Science, Feb. 1986.

[17] B. F. Naylor. Interactive solid geometry via partitioning trees. *Proc. Graphics Interface '92*, pp. 11–18, 1992.

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

[19] M. S. Paterson and F. F. Yao. Efficient binary space partitions for hidden-surface removal and solid modeling. *Discrete Comput. Geom.* 5:485–503, 1990.

[20] M. S. Paterson and F. F. Yao. Optimal binary space partitions for orthogonal objects. *J. Algorithms* 13:99–113, 1992.

[21] R. A. Schumacker, R. Brand, M. Gilliland, and W. Sharp. Study for applying computer-generated images to visual simulation. Tech. Rep. AFHRL–TR–69–14, U.S. Air Force Human Resources Laboratory, 1969.

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

[23] S. J. Teller. *Visibility Computations in Densely Occluded Polyhedral Environments*. Ph.D. thesis, Dept. of Computer Science, University of California, Berkeley, 1992.

[24] W. C. Thibault and B. F. Naylor. Set operations on polyhedra using binary space partitioning trees. *Comput. Graph.* 21:153–162, 1987. Proc. SIGGRAPH '87.

[25] E. Torres. Optimization of the binary space partition algorithm (BSP) for the visualization of dynamic scenes. *Eurographics '90*, pp. 507–518, 1990.