Cylindrical Static and Kinetic Binary Space Partitions

Pankaj K. Agarwal* Leonidas J. Guibas† T. M. Murali‡ Jeffrey Scott Vitter§

February 12, 1999

Abstract

We describe the first known algorithm for efficiently maintaining a Binary Space Partition (BSP) for n continuously moving segments in the plane, whose interiors remain disjoint throughout the motion. Under reasonable assumptions on the motion, we show that the total number of times this BSP changes is $O(n^2)$, and that we can update the BSP in $O(\log n)$ expected time per change.

We also consider the problem of constructing a BSP for n static triangles with pairwise-disjoint interiors in $\mathbb{R}^3$. We present a randomized algorithm that constructs a BSP of size $O(n^2)$ in $O(n^2 \log^2 n)$ expected time. We also describe a deterministic algorithm that constructs a BSP of size $O((n+k)\log^2 n)$ and height $O(\log n)$ in $O((n+k)\log^3 n)$ time, where k is the number of intersection points between the edges of the projections of the triangles onto the xy-plane. This is the first known algorithm that constructs a BSP of $O(\log n)$ height for disjoint triangles in $\mathbb{R}^3$.

*Support was provided by National Science Foundation research grant CCR–93–01259, by 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 Corp, and by a grant from the U.S.-Israeli Binational Science Foundation. Address: Center for Geometric Computing, Department of Computer Science, Duke University, Box 90129, Durham, NC 27708–0129. Email: pankaj@cs.duke.edu

†Support was provided in part by National Science Foundation grant CCR–9623851 and by US Army MURI grant DAAH04–96–1–007. Address: Computer Science Department, Gates 374, Stanford University, Stanford, CA 94305. Email: guibas@cs.stanford.edu

‡This research was done when this author was affiliated with Brown University and was a visiting scholar at Duke University. Support was provided in part by National Science Foundation research grant CCR–9522047 and by Army Research Office MURI grant DAAH04–96–1–0013. Address: Computer Science Department, Gates 133, Stanford University, Stanford, CA 94305. Email: murali@cs.stanford.edu

§Support was provided in part by National Science Foundation research grant CCR–9522047, by Army Research Office grant DAAH04–93–G–0076, and by Army Research Office MURI grant DAAH04–96–1–0013. Address: Center for Geometric Computing, Department of Computer Science, Duke University, Box 90129, Durham, NC 27708–0129. Email: jsv@cs.duke.edu

1 Introduction

The Binary Space Partition (BSP, also known as BSP tree), originally proposed by Schumacker et al. [31] and further refined by Fuchs et al. [18], is a hierarchical partitioning of space widely used in several areas, including computer graphics (global illumination [9], shadow generation [12, 13], visibility determination [4, 33], and ray tracing [24]), solid modeling [27, 25, 34], geometric data repair [23], robotics [5], network design [21], and surface simplification [3]. Key to the BSP's success is that it serves both as a model for an object (or a set of objects) and as a data structure for querying the object.

Informally, a BSP $\mathcal{B}$ for a set of objects is a binary tree, where each node v is associated with a convex region Δ_v. The regions associated with the children of v are obtained by splitting Δ_v with a hyperplane. If v is a leaf of $\mathcal{B}$, then the interior of Δ_v does not intersect any object. The regions associated with the leaves of the tree form a convex decomposition of space. The faces of the decomposition induced by the leaves intersect the objects and divide them into fragments; these fragments are stored at appropriate nodes of the BSP. The efficiency of BSP-based algorithms depends on the number of nodes in the tree and on the height of the tree. As a result, several algorithms for constructing BSPs of small size and/or small height have been proposed [4, 10, 18, 28, 29, 33, 34].

In this paper, we study *cylindrical* BSPs in which all the cuts that do not contain any input object are made by hyperplanes parallel to the same fixed direction. We address two problems. The first problem can be formulated as follows: Let S be a set of n interior-disjoint segments in the plane, each moving along a continuous path. We want to maintain the BSP for S as the segments in S move. We assume that the segments move in such a way that they never intersect, except possibly at their endpoints. Most of the work to date deals with constructing a BSP for a set of static segments which do not move. Paterson and Yao propose a randomized algorithm that constructs a BSP of $O(n \log n)$ size in $\Theta(n \log n)$ for a set of n segments in the plane [28]. They also propose a deterministic algorithm, based on a divide-and-conquer approach, that constructs a BSP of size $O(n \log n)$ in $\Theta(n \log n)$ time [28]. Both of these algorithms are not "robust," in the sense that a small motion of one of the segments may cause many changes in the tree, or may cause non-local changes. Therefore, they are ill-suited for maintaining a BSP for a set of moving segments.

There have been a few attempts to update BSPs when the objects defining them move. Naylor describes a method to implement dynamic changes in a BSP, where the static objects are represented by a balanced BSP (computed in a preprocessing stage), and then the moving objects are inserted at each time step into the static tree [26]. Using the same assumption that moving objects are known *a priori*, Torres proposes the augmentation of BSPs with additional separating planes, which may localize the updates needed for deletion and re-insertion of moving objects in a BSP [35]. This approach tries to exploit the spatial coherence of the dynamic changes in the tree by introducing additional cutting planes. Chrysanthou suggests a more general approach, which does not make any distinction between static and moving objects [14]. By keeping additional information about topological adjacencies in the tree, the algorithm performs insertions and deletions of a node in a more

localized way. But all these prior efforts boil down to deleting moving objects from their earlier positions and re-inserting them in their current positions after some time interval has elapsed. Such approaches suffer from the fundamental problem that it is very difficult to know how to choose the correct time interval size: if the interval is too small, then the BSP does not in fact change combinatorially, and the deletion/re-insertion is just wasted computation; if it is too big, then important intermediate changes to the BSP can be missed, which may affect applications that use the tree.

Our algorithm, instead, treats the BSP as a *kinetic data structure*, a paradigm introduced by Basch et al. [6]; see also the survey by Guibas [19]. We view the equations of the cuts made at the nodes of the BSP and the edges and faces of the subdivision induced by the BSP as functions of time. The cuts and the edges and faces of the subdivision change continuously with time. However, "combinatorial" changes in the BSP and in the subdivision (we precisely define this notion later) occur only at certain times. We explicitly take advantage of the continuity of the motion of the objects involved so as to generate updates to the BSP only when actual events cause the BSP to change combinatorially.

In Section 3, we describe a randomized kinetic algorithm for maintaining a BSP for moving segments in the plane. We assume that the segment motions are *oblivious* to the random bits used by the algorithm; our algorithm chooses a random permutation of the segments at the beginning of time, and we assume that no agent determining the motion of the segments has access to any information about this random permutation. Following Basch et al. [6], we assume that each moving segment has a posted flight plan that gives full or partial information about the segment's current motion. Whenever a flight plan changes (possibly due to an external agent), our algorithm is notified and it updates a global event queue to reflect the change. We first derive a randomized algorithm for computing a BSP for a set of static segments, which combines ideas from Paterson and Yao's randomized and deterministic algorithms, but is also robust, in the sense described earlier. The "combinatorial structure" of the BSP constructed by this algorithm changes only when the x-coordinates of a pair of segment endpoints, among a certain subset of $O(n)$ pairs, become equal. We show that under the above assumption on the segment motions, the BSP can be updated in $O(\log n)$ expected time at each such event. We also show that if k of the segments of S move along "pseudo-algebraic" paths, and the remaining segments of S are stationary, then the expected number of changes in the BSP is $O(kn \log n)$. As far as we know, this is the first nontrivial algorithm for maintaining a BSP for moving segments in the plane.

Next, we study the problem of computing a BSP for a set S of n interior-disjoint triangles in $\mathbb{R}^3$. Paterson and Yao [28] describe a randomized incremental algorithm that constructs a BSP of size $O(n^2)$ in expected time $O(n^3)$. They also show that their algorithm can be made deterministic without affecting its asymptotic running time. It has been an open problem whether a BSP for n triangles in $\mathbb{R}^3$ can be constructed in near-quadratic time. Sub-quadratic bounds are known for special cases: Paterson and Yao's algorithm for orthogonal rectangles [29], de Berg's result for fat polyhedra [16], and the technique of Agarwal et al. [2] for fat orthogonal rectangles. However, none of these approaches lead to a

near-quadratic-time algorithm for triangles in $\mathbb{R}^3$. The bottleneck in analyzing the expected running time of the Paterson-Yao algorithm is that no nontrivial bound is known on the number of vertices in the convex subdivision of $\mathbb{R}^3$ induced by the BSP constructed by the algorithm. Known techniques for analyzing randomized algorithms, such as the Clarkson-Shor framework [15] or backwards analysis [32], cannot be used to obtain a near-quadratic bound on this quantity, since the BSP constructed by the algorithm is not canonical; it strongly depends on the order in which triangles are processed.

In Section 4, we present a randomized algorithm that constructs a BSP for S of size $O(n^2)$ in $O(n^2 \log^2 n)$ expected time. Our algorithm is a variant of the Paterson-Yao algorithm. We construct the BSP for S in such a way that there is a close relationship between the BSP and the planar arrangement of lines supporting the edges of the xy-projections of the triangles in S. We use results on ε-nets [20] and on arrangements of lines [17] to bound the expected number of vertices in the convex subdivision of $\mathbb{R}^3$ induced by the BSP and the expected running time of the algorithm.

Finally, we present a deterministic algorithm in Section 5 for constructing a BSP for a set S of n triangles in $\mathbb{R}^3$. If k is the number of intersection points of the xy-projections of the edges of triangles in S, then the algorithm constructs a BSP of size $O((n+k)\log n)$ in time $O((n+k)\log^2 n)$; if $k \ll n^2$, the deterministic algorithm constructs a much smaller BSP than do Paterson and Yao's and our randomized algorithms. Another nice property of our deterministic algorithm is that the height of the BSP it constructs is $O(\log n)$, which is useful for ray-shooting queries, for example. It was an open problem whether BSPs of near-quadratic size and $O(\log n)$ height could be constructed for n triangles in $\mathbb{R}^3$. The height of the BSP constructed by the randomized algorithms (both ours and Paterson and Yao's) can be $\Omega(n)$, e.g., when S is the set of faces of a convex polytope.

Before proceeding further, we give a formal definition of a BSP. A *binary space partition* $\mathcal{B}$ for a set S of convex $(d-1)$-polytopes in $\mathbb{R}^d$ with pairwise-disjoint interiors is a tree defined as follows: Each node v in $\mathcal{B}$ is associated with a convex d-polytope Δ_v and a set of $(d-1)$-polytopes $S_v = \{s \cap \Delta_v \mid s \in S\}$. The polytope associated with the root is $\mathbb{R}^d$ itself. If S_v is empty, then node v is a leaf of $\mathcal{B}$. Otherwise, we partition Δ_v into two convex polytopes by a *cutting hyperplane* H_v. We refer to the polytope $H_v \cap \Delta_v$ as the *cut* made at v. At v, we store the equation of H_v and the set $\{s \mid s \subseteq H_v, s \in S_v\}$ of polytopes in S_v that lie in H_v. If we let H_v^+ be the positive halfspace and H_v^- be the negative halfspace bounded by 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. The left subtree of v is a BSP for $S_v^- = \{s \cap H_v^- \mid s \in S_v\}$ and the right subtree of v is a BSP for $S_v^+ = \{s \cap H_v^+ \mid s \in S_v\}$. The size of $\mathcal{B}$ is the sum of the number of nodes in $\mathcal{B}$ and the total number of polytopes stored at all the nodes in $\mathcal{B}$.

For our purposes, S is either a set of n segments in the plane or a set of n triangles in $\mathbb{R}^3$. A unifying feature of all the BSPs constructed by our algorithms is that the region Δ_v associated with each node v is a *cylindrical* cell in the sense that Δ_v may contain top and bottom faces that are contained in objects belonging to S, but all other faces are vertical. In the plane, Δ_v is a trapezoid; in $\mathbb{R}^3$, Δ_v may have large complexity, as it can contain many vertical faces.

2 Static Algorithm for Segments

Let S be a set of n non-intersecting segments in the plane. We first describe a randomized algorithm for computing a BSP $\mathcal{B}$ for S when the segments in S are stationary, and then explain how to maintain $\mathcal{B}$ as each segment in S moves along a continuous path.

Our algorithm makes two types of cuts: a *point* cut is a vertical cut through an endpoint of a segment and an *edge* cut is a cut along a segment. Edge cuts are always contained totally within input segments; therefore, they do not cross any other input segment. For each node $v \in \mathcal{B}$, the corresponding polygon Δ_v is a trapezoid; the left and right boundaries of the trapezoid are bounded by point cuts, and the top and bottom boundaries are bounded by edge cuts.

We now describe our static algorithm. We start by choosing a random permutation $\langle s_1, s_2, \ldots, s_n \rangle$ of S. We say that s_i has a *higher priority* than s_j if $i < j$. We add the segments in decreasing order of priority and maintain a BSP for the segments added so far. Let $S^i = \{s_1, s_2, \ldots, s_i\}$ be the set of the first i segments in the permutation. Our algorithm works in n stages. At the beginning of the ith stage, where $i > 0$, we have a BSP $\mathcal{B}^{i-1}$ for S^{i-1}; $\mathcal{B}^0$ consists of a single node v, where Δ_v is the entire plane. In the ith stage, we add s_i and compute a BSP $\mathcal{B}^i$ for S^i as follows:

1. Suppose p and q are the left and right endpoints of s_i, respectively. Let v be the leaf of $\mathcal{B}^{i-1}$ such that Δ_v contains p. We partition Δ_v into two trapezoids Δ_v^- and Δ_v^+ using a point cut defined by p, where Δ_v^- lies to the left of the cut. We create two children w and z of v, with w being the left child of v. We set $\Delta_w = \Delta_v^-$ and $\Delta_z = \Delta_v^+$ and store p at v. We then perform a similar step for q.

2. For each trapezoid Δ_x that intersects s_i, we store s_i at x, and split Δ_x into two trapezoids by making an edge cut along s_i. We again create two children w and z of v, with w being the left child. We set Δ_w to be the sub-trapezoid of Δ_x lying below the cut and Δ_z to be the sub-trapezoid of Δ_x lying above the cut.

The resulting tree is the BSP $\mathcal{B}^i$ for S^i. See Figure 1 for an example of constructing $\mathcal{B}^i$ from $\mathcal{B}^{i-1}$.

This completes the description of our algorithm. Note that once we fix the permutation, the algorithm is deterministic and constructs a unique BSP. In order to analyze the algorithm, we need a few definitions. The vertical segment drawn upwards (resp., downwards) from an endpoint p is referred to as the *upper* (resp., *lower*) *thread* of p. The segment containing the other endpoint of a thread is called the *stopper* of that thread. Note that the priority of the stopper of a thread of p is higher than that of the segment containing p. We can prove the following lemma about each thread:

Lemma 2.1 *Let p be an endpoint of a segment in S. The expected number of segments crossed by each of p's threads is $O(\log n)$.*

Proof: Let $\sigma_1, \sigma_2, \ldots$ be the sequence of segments in S that intersect the top thread ρ of p, sorted in increasing order of the y-coordinates of their intersection with ρ; clearly,

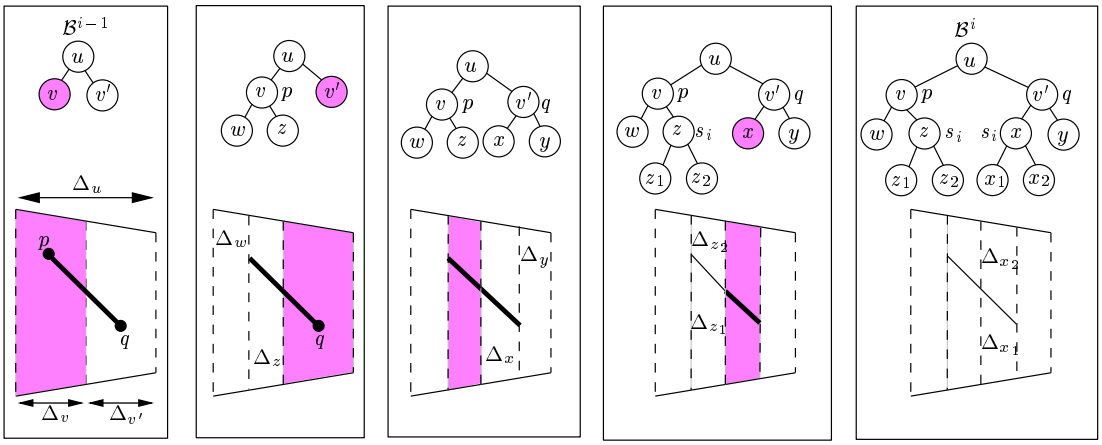


Figure 1: The BSP $\mathcal{B}^{i-1}$, the sequence of cuts made in the ith stage, and the BSP $\mathcal{B}^i$. At each step, the shaded trapezoids are split. Portions of s_i that lie in the interior of a trapezoid are drawn using thick lines. The label next to a node signifies the cut made at that node.

there are at most n segments in this sequence. The segment σ_i is crossed by ρ if and only if s has greater priority than all the segments $\sigma_1, \sigma_2, \ldots, \sigma_{i-1}$. Since $\mathcal{B}$ is constructed by inserting the segments of S in random order, the probability that ρ crosses σ_i is $1/(i+1)$. Therefore the expected number of segments crossing ρ is at most $\sum_{i=1}^{n} 1/(i+1) = O(\log n)$. We can similarly show that the expected number of segments crossing p's lower thread is $O(\log n)$. □

We can use the above lemma to bound the size and height of $\mathcal{B}$.

Theorem 2.2 *The expected size of the BSP constructed by the above algorithm is $O(n \log n)$ and the height of the BSP is $O(\log n)$, where the second bound holds with high probability.*

Proof: In order to bound the size of $\mathcal{B}$, the BSP constructed by the algorithm, it is enough to count the total number of cuts made in $\mathcal{B}$, since a cut is made at each interior node of $\mathcal{B}$. Clearly, there are at most $2n$ point cuts made in $\mathcal{B}$. If an edge cut e is made at a node v, we charge e to the right endpoint of e. Suppose s is the segment in S containing e. The right endpoint of e is either the right endpoint of s or the intersection point of s with a thread of a segment whose priority is higher than s. In this way, we charge each endpoint and the intersection point of a segment and a thread at most once. As a result, Lemma 2.1 implies that the expected total number of edge cuts is $O(n \log n)$, which proves that the expected size of $\mathcal{B}$ is $O(n \log n)$.

To bound $\mathcal{B}$'s height, we first bound the *depth* of an arbitrary point p in the plane, i.e., the number of nodes in the path from the root of $\mathcal{B}$ to the leaf $v \in \mathcal{B}$ such that Δ_v

contains p. We bound the number of nodes on this path that are split by edge cuts and point cuts separately.

Let $\sigma_1, \sigma_2, \ldots$ be the ordered sequence of segments in S intersected by a vertical ray starting at p and pointing in the $(+y)$-direction. An ancestor of v is split by an edge cut through σ_i if and only if σ_i has higher priority than $\sigma_1, \sigma_2, \ldots, \sigma_{i-1}$. This event happens with probability $1/i$. Hence, the expected value of X, the number of ancestors of v that are split by edge cuts, is $H_n = O(\log n)$. We can actually prove that this bound on X holds with high probability. Since X is the sum of independent 0-1 random variables, using Chernoff's bound [22, p. 68], we have that for any constant $\alpha \geq 1$,

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

In particular for any constant $c \geq 3$, we can choose α so that $\Pr[X > \alpha H_n] < 1/n^c$.

We can similarly prove that the number of ancestors of v that are split by point cuts is $O(\log n)$ with high probability. The segments in S and the vertical lines passing through every segment endpoint decompose the plane into $O(n^2)$ trapezoids. Any two points in one of these trapezoids will be contained in the same leaf of any BSP that our algorithm constructs, independent of the permutation we choose at the beginning of the algorithm. Hence, the height of BSP is the maximum depth of $O(n^2)$ points, one in each such trapezoid. Since the depth of each point is $O(\log n)$ with probability $1 - 1/n^c$, where $c \geq 3$, the height of $\mathcal{B}$ is also $O(\log n)$ with probability $1 - 1/n^{c-2}$. This completes the proof of the lemma. □

3 Kinetic Algorithm for Segments

We now describe how to maintain the static BSP as the segments in S move continuously, under the assumption that their interiors remain pairwise disjoint throughout the motion. We parameterize the motion of the segments by time and use t to denote time. For a given time instant t, we will use t^- and t^+ to denote the time instants $t - \varepsilon$ and $t + \varepsilon$, respectively, where $\varepsilon > 0$ is a sufficiently small constant.

Let $s_i \in S$ be a segment with endpoints p and q. We assume that the position of p at time t is $p(t) = (x_p(t), y_p(t))$, where $x_p(t)$ and $y_p(t)$ are continuous functions of time; $q(t)$ is specified similarly. The position of s_i at time t is $s_i(t) = (p(t), q(t))$; if s_i is moving rigidly, then the equations for its endpoints are not independent. Our algorithm and the analysis work even if the endpoints of s_i move independently. Let $S(t)$ denote the set S at time t. We assume that we choose a random permutation π of S once in the very beginning (at $t = 0$), and that π does not change with time. Let $\mathcal{B}(t)$ denote the BSP of $S(t)$ constructed by the static algorithm, using π as the permutation to decide the priority of the segments. We describe an algorithm that updates the BSP under the following assumption:

($\star$) *There is no correlation between the motion of the segments in S and their priorities. Therefore, the chosen permutation π always behaves like a random permutation, and Lemma 2.1 and Theorem 2.2 hold at all times.*

We first give an important definition. The *combinatorial structure* of $\mathcal{B}$ is a binary tree, each of whose nodes v is associated with the set of segments S_v. We will use the combinatorial structure of the BSP crucially in our algorithm.

3.1 Critical events

As the segments in S move continuously, the equations of the cuts associated with the nodes of $\mathcal{B}$ also change. At the same time, the edges and vertices of the trapezoids in the subdivision of the plane induced by $\mathcal{B}$ also move. However, the combinatorial structure of $\mathcal{B}$ changes only when the set S_v changes for some node $v \in \mathcal{B}$. Since the segments in S are interior-disjoint and they move continuously, the set S_v changes only when the endpoint of a segment in S_v lies on the left or right edge of Δ_v. See Figure 2 for an example of such an event. We formalize this idea in the following lemma, which is not difficult to prove:

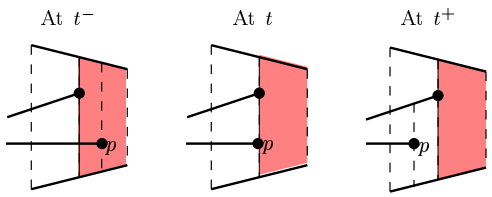


Figure 2: Endpoint p lies on the left edge of Δ_v (the shaded trapezoid) at t. The set S_v changes at time instant t.

Lemma 3.1 *For any time instant t, $\mathcal{B}(t^-)$ and $\mathcal{B}(t^+)$ have different combinatorial structures if and only if there exists a $j > 0$ such that either s_j rotates through a vertical line at time t or there is a leaf $v \in \mathcal{B}^{j-1}(t^-)$ such that an endpoint of s_j lies on the left or right edge of Δ_v at time t.*

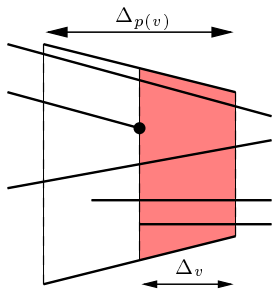


Figure 3: The shaded trapezoid Δ_v is transient.

The above lemma implies that the combinatorial structure of $\mathcal{B}(t)$ changes if and only if for a node $v \in \mathcal{B}(t)$, Δ_v shrinks to a vertical segment; we refer to these instants of time as

critical events. This observation motivates us to call a node v in $\mathcal{B}(t)$ *transient* if Δ_v does not contain any endpoint in its interior and a point cut is made at the parent $p(v)$ of v; we call Δ_v a *transient trapezoid*. See Figure 3. Note that only edge cuts are made at v and its descendants. Thus, transient trapezoids are maximal among all those not containing point cuts. The following corollary to Lemma 3.1 is immediate:

Lemma 3.2 *For any time instant t, $\mathcal{B}(t^-)$ and $\mathcal{B}(t^+)$ have different combinatorial structures if and only if there exists a transient node v in $\mathcal{B}(t^-)$ so that Δ_v becomes a vertical segment at time t.*

Transient nodes have some useful properties that are summarized in the following lemma:

Lemma 3.3 *At any instant t, the set of transient nodes in $\mathcal{B}(t)$ have the following properties. Let v be a transient node in $\mathcal{B}(t)$.*

1. *No proper ancestor of v is transient.*
2. *Only edge cuts are made at the descendants of v (including v itself). The left (resp., right) edge of the trapezoid associated with each descendant of v is a portion of the left (resp., right) edge of Δ_v.*
3. *The expected number of descendants of v is $O(\log n)$.*
4. *The number of transient nodes in $\mathcal{B}(t)$ is at most $4n$.*

Proof: Let p be the endpoint of a segment in S through which the point cut at $p(v)$ is made.

1. No proper ancestor w of v can be transient since Δ_w contains p.
2. Since Δ_v does not contain any endpoints, only edge cuts are made at all the descendants of v. Each segment that intersects Δ_v crosses the left and right boundaries of v. Hence, the left (resp., right) edge of the trapezoids associated with each descendant of v is a portion of the left (resp., right) edge of Δ_v.
3. Each segment that induces an edge cut made at a descendant of v intersects one of p's threads. Hence, by Lemma 2.1, the expected number of descendants of v is $O(\log n)$.
4. A point cut is made at the parent of each transient node. There are $2n$ nodes in $\mathcal{B}$ that are split by point cuts; each such node has two children.

□

Intuitively, transient nodes are the highest nodes in $\mathcal{B}(t)$ that can shrink to a vertical segment, causing a change in the combinatorial structure of $\mathcal{B}(t)$. If a trapezoid contains an endpoint in its interior, it cannot be the next trapezoid to shrink to a segment; and if an edge cut is made at the parent $p(v)$ of a node v and $\Delta_{p(v)}$ does not contain an endpoint,

then $\Delta_{p(v)}$ also shrinks to a segment whenever Δ_v shrinks to a segment. Hence, it suffices to keep track of transient nodes to determine all the instants when the combinatorial structure of $\mathcal{B}(t)$ changes. In the rest of the section, we present our kinetic algorithm motivated by this observation.

3.2 Updating the BSP

For a node v in $\mathcal{B}$, let λ_v (resp., ρ_v) denote the endpoint of a segment in S that induces the point cut containing the left (resp., right) edge of Δ_v. To detect critical events, we maintain the set

$$\Gamma(t) = \{(\lambda_v, \rho_v) \mid v \text{ is a transient node at time } t\}$$

of endpoint pairs inducing the point cuts that bound the left and right edges of each transient node; Lemma 3.3 implies that $|\Gamma(t)| = O(n)$. The elements of $\Gamma(t)$ are certificates that prove that the combinatorial structure of $\mathcal{B}(t)$ is valid. For each pair (λ_v, ρ_v) in $\Gamma(t)$, we use the known flight paths of λ_v and ρ_v to compute the time at which the x-coordinates of λ_v and ρ_v coincide; we store these time values in a global priority queue. In order to expedite the updating of $\mathcal{B}$ at each critical event, we also store some additional information with the nodes in $\mathcal{B}$ and the segments in S:

1. At each node v of $\mathcal{B}$, we store the number c_v of segment endpoints lying in the interior of Δ_v (c_v helps us to determine the new transient trapezoids at an event).

2. For each endpoint p of a segment in S, we maintain the list T_p (resp., B_p) of segments that the upper (resp., lower) thread of p crosses, sorted in the $(+y)$-direction (resp., $(-y)$-direction). As the segments move, we will use these lists to update the stoppers of the threads issuing from the segment endpoints.

We first construct $\mathcal{B}(0)$ using the static algorithm presented in Section 2. Next, we compute the set $\Gamma(0)$ and insert the corresponding critical events in the priority queue. Then we repeatedly remove the next event from the priority queue and update $\mathcal{B}$, Γ, and the priority queue as required. In the rest of the section, we will prove that if the combinatorial structure of $\mathcal{B}$ changes at time t, then we can obtain $\mathcal{B}(t^+)$ from $\mathcal{B}(t^-)$ in $O(\log n)$ expected time. We will also show that at each event, the expected time to update the global event queue is $O(\log n)$.

We now describe the procedure for updating the tree at each critical event. Recall that at each such instant t, there is a segment $s_j \in S$ such that (i) either s_j becomes vertical or (ii) there is a leaf $w \in \mathcal{B}^{j-1}(t^-)$ such that an endpoint p of s_j lies on the left or right edge of Δ_w. We consider each case separately. Let $\mathcal{B}^- = \mathcal{B}(t^-)$ and $\mathcal{B}^+ = \mathcal{B}(t^+)$. For a node $z \in \mathcal{B}^-$, let $\mathcal{B}^-_z$ denote the subtree of $\mathcal{B}^-$ rooted at z; define $\mathcal{B}^+_z$ similarly.

Case (i): s_j is vertical. In this case, v is a transient node in $\mathcal{B}^-$ with the property that λ_v and ρ_v are both endpoints of s_j. See Figure 4. Let u be v's grandparent in $\mathcal{B}^-$; the trapezoid Δ_u contains s_j. Suppose v is in the right subtree of u in $\mathcal{B}^-$. Let v_1 be the left

child of u and let v_2 be the sibling of v in $\mathcal{B}^-$. To obtain $\mathcal{B}^+$, we create a new node v_1' and attach it as the left child of u. The left and right child of v_1' are v_1 and v, respectively. We delete the right child of u and replace it with v_2.

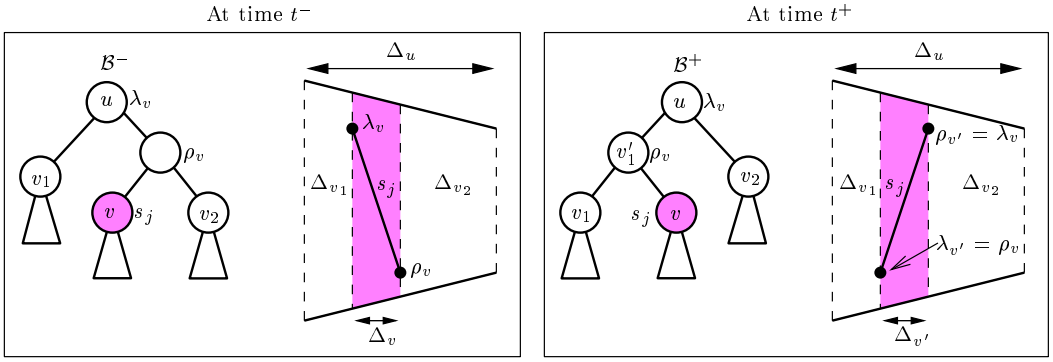


Figure 4: The case when λ_v and ρ_v belong to the same segment.

Case (ii): λ_v and ρ_v are endpoints of different segments. Assume that $\rho_v = p$ (resp., $\lambda_v = q$) is the right endpoint of the segment s_i (resp., s_j), that s_i lies above s_j, that the priority of s_i is higher than that of s_j (i.e., $i < j$), and that the x-coordinate of q is less than the x-coordinate of p at t^-; see Figure 5. We now describe how we update $\mathcal{B}(t)$ for this case. We will show later how to relax these assumptions. Let u and w be the leaves of $\mathcal{B}^{i-1}(t^-)$ and $\mathcal{B}^{j-1}(t^-)$, respectively, at which the point cuts through p and q, respectively, are made. At time t^-, a point cut made through p divides Δ_w into two trapezoids. One of these trapezoids is Δ_v, which is transient at time t^-. By our assumptions about the event, v is the right child of w, and w lies in the left subtree of u. Let u_L be the left child of u, and let w_L be the left child of w. Let x be the leaf of $\mathcal{B}^{j-1}(t^-)$ that contains q at time t^+; x lies in the right subtree of u. Since the combinatorial structures of $\mathcal{B}^{j-1}(t^-)$ and $\mathcal{B}^{j-1}(t^+)$ are identical, x is a leaf of $\mathcal{B}^{j-1}(t^+)$ too. Let $s_k \in S$ be the segment containing the top edge of Δ_x; clearly, $k \geq i$. At time t^+, as q leaves the trapezoid Δ_w and enters Δ_x, Δ_{w_L} expands to Δ_w, Δ_v disappears, and Δ_x is split at t^+ into two trapezoids: a new trapezoid $\Delta_{v'}$ and the portion of Δ_x lying to the right of the cut through q. At time t^-, Δ_w is split by a point cut through q and Δ_{w_L} is split by an edge cut along s_j, while at time t^+, Δ_w is split by an edge cut along s_j. Therefore $\mathcal{B}_w^+$ is the same as $\mathcal{B}_{w_L}^-$. To obtain $\mathcal{B}^+$, we execute the following steps:

1. We search in the right subtree of u to locate the leaf x of $\mathcal{B}^{j-1}(t^-)$ such that Δ_x contains q at time t^+.

2. We delete the node w from $\mathcal{B}^-$, and if w was a left (resp., right) child of its parent $p(w)$, we make w_L the new left (resp., right) child of $p(w)$.

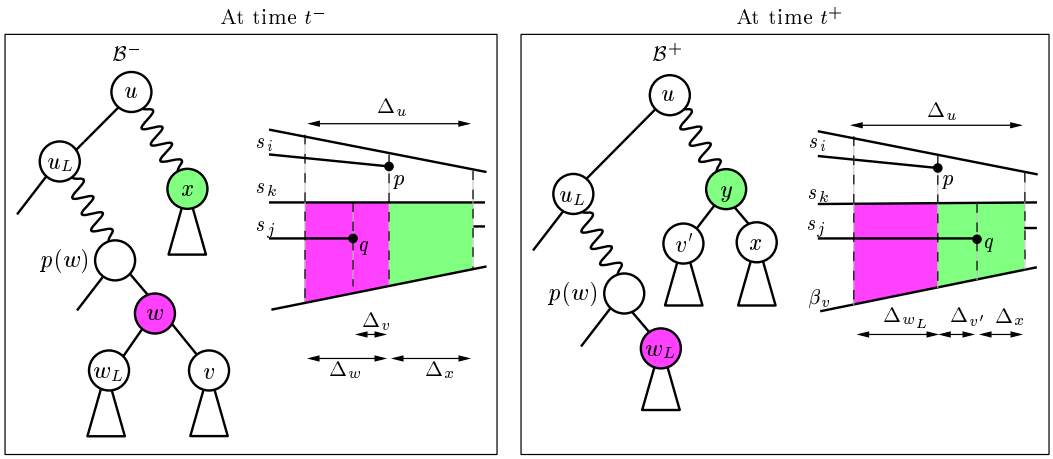


Figure 5: The case when λ_v and ρ_v are endpoints of different segments. Arrows mark the horizontal extents of the trapezoids.

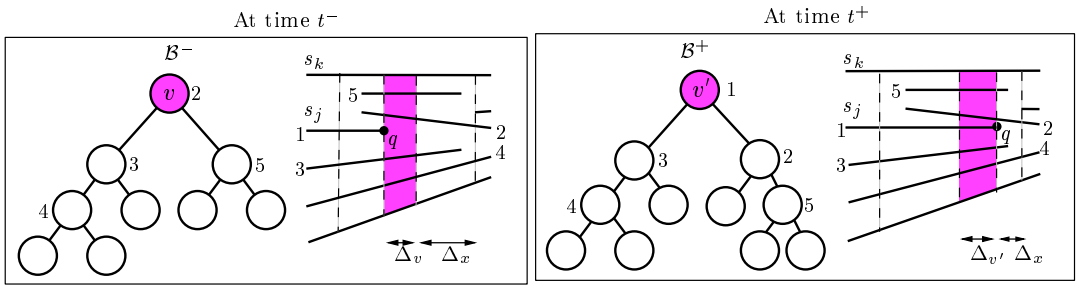


Figure 6: The edge cuts made in $\mathcal{B}_v^-$ and $\mathcal{B}_{v'}^+$. Segments crossing Δ_v and $\Delta_{v'}$ are labelled with their priorities. The label next to a node is the priority of the segment containing the edge cut made at that node.

3. We construct the subtree $\mathcal{B}^+_{v'}$ by determining the set C of segments that intersect $\Delta_{v'}$ (at time t^+) and by making edge cuts through the segments in C in decreasing order of priority. There are two cases to consider:

   (a) The segment s_k contains the top edge of Δ_v: See Figure 6. The set C consists of s_j and the set of segments intersecting Δ_v (at time t^-). We find these segments by traversing all the nodes of $\mathcal{B}^-_v$.

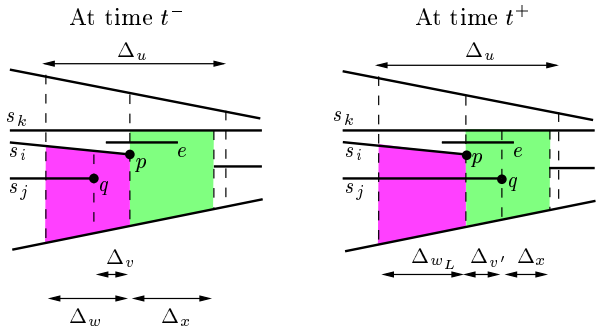


Figure 7: The case when the top edges of Δ_w and Δ_x are contained in different segments. The segment e is intersected by the top thread of q at t^+ but not at t^-.

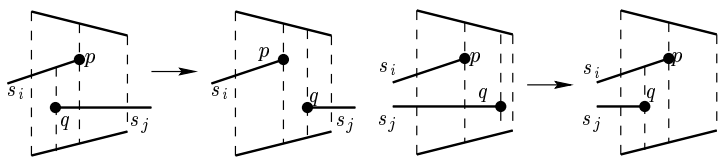
   (b) The segment s_i contains the top edge of v: See Figure 7. We set s_k to be the stopper of the upper thread of q. As in the previous case, we include s_j and the segments inducing the edge cuts made in $\mathcal{B}^-_v$ in C. In addition, C contains all segments that appear before s_k in the upper thread of p. We determine these segments by traversing T_p, the list of segments that cross the upper thread of p. Note that these segments also cross the upper thread of q at t^+.

   Finally, we insert s_j into B_p, the list of segments in S crossed by the lower thread of p, and update T_q.

4. We attach $\mathcal{B}^+_{v'}$ to a descendant of $p(x)$, the parent of x in $\mathcal{B}^-$, as follows: We create a node y and associate the point cut through q with it. The left and right subtrees of y in $\mathcal{B}^+$ are $\mathcal{B}^+_{v'}$ and $\mathcal{B}^-_x$, respectively. If x is the left (resp., right) child of $p(x)$, then we add y as the new left (resp., right) child of $p(x)$.

5. We update the set $\Gamma(t^+)$. For a node z, the number c_z of endpoints lying in the interior of Δ_z, changes only if z lies along the paths in $\mathcal{B}^+$ from u to the nodes $p(w)$ and y. For such a node z, if $c_z = 0$ at t^+ and if $\Delta_{p(z)}$ is split by a point cut, we add (λ_z, ρ_z) to the list $\Gamma(t^+)$. On the other hand, if $c_z \neq 0$ at t^+ but z is transient at t^- (z must be an ancestor of x in $\mathcal{B}^-$), we delete (λ_v, ρ_z) from $\Gamma(t^+)$. We also update the priority queue to reflect the changes to $\Gamma(t^+)$.

Other cases: We now show how we relax the assumptions we made earlier about the relative positions of s_i and s_j and their priorities. For each case, we show how a simple transformation reduces it to one of the earlier cases.



(a) (b)

Figure 8: Some other cases that arise when different segments interact in a critical event.

1. If q is the left endpoint of s_j, the update procedure is the same, except that at time t^+ we do not make an edge cut through s_j in $\mathcal{B}^+_{v'}$. See Figure 8(a).

2. If the x-coordinate of q is greater than the x-coordinate of p at time t^-, we "go back in time." Figure 8(b). The node x is again the leaf of $\mathcal{B}^{j-1}(t^-)$ such that Δ_x contains q at t^+. We can reconstruct $\mathcal{B}^+_{v'}$ as before: if the same segment contains the top edge of Δ_v and Δ_x, the same set of segments intersects $\mathcal{B}^-_v$ and $\mathcal{B}^+_{v'}$ (except that s_j does not intersect $\mathcal{B}^+_{v'}$); otherwise, among the segments that intersect $\mathcal{B}^-_v$, only segments with priority at most i intersect $\mathcal{B}^+_{v'}$. In the second case, we also update T_q accordingly. The other changes to $\mathcal{B}$ are similar to the cases we have handled; the details are not difficult to work out.

3. If p is the left endpoint of s_i, we reflect S about the y-axis.

4. If s_i lies below s_j, we reflect S about the x-axis.

5. If the priority of s_i is less than the priority of s_j, we swap the roles of s_i and s_j.

This completes the description of our procedure for processing critical events. We now analyze the running time of the update procedure. Assumption ($\star$) implies that Lemma 2.1 and Theorem 2.2 hold at times t^-, t, and t^+. We spend $O(\log n)$ time in Step 1, since we traverse a path in $\mathcal{B}$ to find the node x. It is clear that Step 2 takes $O(1)$ time. In Step 3, we find the segments crossing $\Delta_{v'}$ and construct $\mathcal{B}^+_{v'}$ in $O(\log n)$ expected time, since the expected size of $\mathcal{B}^-_v$ is $O(\log n)$ (by Lemma 3.3) and the expected number of segments in T_p is $O(\log n)$ (by Lemma 2.1). It is clear that Step 4 takes $O(1)$ time. Finally, in Step 5, we process $O(\log n)$ nodes lying in two paths in the tree. By Lemma 3.3, each of the two paths contains at most one transient node. Hence, we insert or delete at most two events from the priority queue, which implies that Step 5 takes $O(\log n)$ time. We thus obtain the main result of this section:

Theorem 3.4 *At each critical event, we can update $\mathcal{B}(t)$ in $O(\log n)$ expected time.*

Note that this theorem makes our BSP a kinetic data structure that is responsive, efficient, local, and compact, in the sense defined by Basch et al. [6].

We say that the trajectories followed by a set of segments are *pseudo-algebraic* if the segments move so that each pair of endpoints exchanges y-order only $O(1)$ times. A special case of pseudo-algebraic trajectories is when the trajectories of all the endpoints are constant-degree polynomials. If the trajectories of k of the segments in S are pseudo-algebraic and the remaining segments are stationary, then the total number of event points is $O(kn)$. We spend $O(\log n)$ expected time to maintain $\mathcal{B}(t)$ at each event point. Hence, we obtain the following corollary to Theorem 3.4:

Corollary 3.5 *Let S be a set of n segments in the plane, and let $G \subseteq S$ be a set of k segments. Suppose each segment of G moves along a pseudo-algebraic trajectory and the remaining segments of S are stationary, the total expected time spent in maintaining $\mathcal{B}$ is $O(kn \log n)$.*

4 BSPs for Triangles: A Randomized Algorithm

In this section we describe a randomized algorithm for constructing a BSP $\mathcal{B}$ of expected size $O(n^2)$ for a set of n triangles with pairwise-disjoint interiors in $\mathbb{R}^3$. The expected running time of the algorithm is $O(n^2 \log^2 n)$. We describe the algorithm in Section 4.1 and analyze its performance in Section 4.2.

4.1 The randomized algorithm

We start with some definitions. For an object s in $\mathbb{R}^3$, let s^* denote the xy-projection of s. Let E be the set of edges of the triangles in S, and let E^* denote the set $\{e^* \mid e \in E\}$. Let $\mathcal{L}$ be the set of lines in the xy-plane supporting the edges in E^*. We choose a random permutation $\langle \ell_1, \ell_2, \ldots, \ell_{3n} \rangle$ of $\mathcal{L}$, and add the lines one by one in this order to compute $\mathcal{B}$. Let $\mathcal{L}^i = \{\ell_1, \ell_2, \ldots \ell_i\}$.

The algorithm works in $3n$ stages. In the ith stage, we add ℓ_i and construct a top subtree[1] $\mathcal{B}^i$ of $\mathcal{B}$ by refining the leaves of $\mathcal{B}^{i-1}$; $\mathcal{B}^0$ consists of one node (corresponding to $\mathbb{R}^3$) and $\mathcal{B}^{3n}$ is $\mathcal{B}$. As usual, we associate a convex polytope Δ_v with each node v of $\mathcal{B}^{i-1}$. If v is a leaf of $\mathcal{B}^{i-1}$ and no triangle in S intersects the interior of Δ_v, i.e., $S_v = \emptyset$, then v is a leaf of $\mathcal{B}$ and we do not refine it further. Otherwise, we partition Δ_v into two cells; these two cells are leaves of $\mathcal{B}^{i+1}$.

Before describing the ith stage of the algorithm in detail, we explain the structure of $\mathcal{B}^i$. We need a few definitions first. At a node v of $\mathcal{B}$, the cutting plane H_v may support a triangle $s \in S$ such that $H_v \cap \Delta_v \subseteq s$, i.e., the portion of H_v that lies in the interior of Δ_v is contained in s. Such a cutting plane is referred to as a *free cut* and s is called a *free*

[1] A *top* subtree of a tree is one that includes the root of the tree.

triangle. We say that a leaf v of $\mathcal{B}^i$ (or the cell Δ_v) is *active* if a triangle in S intersects the interior of Δ_v (i.e. $S_v \neq \emptyset$); similarly, we say that a face f in the line arrangement $\mathcal{A}(\mathcal{L}^i)$ is *active* if a segment in E^* intersects the interior of f. For each active leaf v in $\mathcal{B}^i$, the algorithm ensures that Δ_v satisfies the following properties:

(P1) If a triangle $s \in S$ intersects the interior of Δ_v, then the boundary of s also intersects the interior of Δ_v.

(P2) The cell Δ_v is a vertical section of the cylinder $\{(p, z) \mid p \in f, z \in \mathbb{R}\}$, for exactly one active face f of $\mathcal{A}(\mathcal{L}^i)$; the vertical section may be truncated by triangles of S at the top and bottom. See Figure 9.

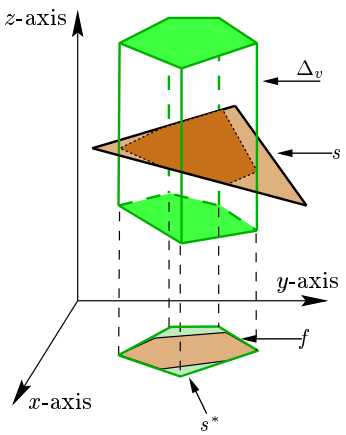


Figure 9: An active face f and an active cell $\Delta_v \in \mathbf{\Delta}(f)$ that is a vertical section of the cylinder erected on f. The boundary of triangle s intersects Δ_v and the boundary of s^* intersects f.

In order to execute each stage efficiently, we maintain the following additional information:

(i) For each active cell $\Delta \in \mathcal{B}^i$, we store the subset $S_\Delta \subseteq S$ of triangles that intersect the interior of Δ.

(ii) We maintain the arrangement $\mathcal{A}(\mathcal{L}^i)$ as a planar graph [17]. For each active face $f \in \mathcal{A}(\mathcal{L}^i)$, we maintain the set $\mathbf{\Delta}(f)$ of those active cells in $\mathcal{B}^i$ that lie inside the cylinder $\{(p, z) \mid p \in f, z \in \mathbb{R}\}$. Note that by Properties (P1) and (P2), a face $f \in \mathcal{A}(\mathcal{L}^i)$ is active if and only if $\mathbf{\Delta}(f) \neq \emptyset$.

We now describe the ith stage in detail. In this stage, we make a vertical cut along the vertical plane H_i supporting ℓ_i, followed by a number of free cuts as follows: Let H_i^+ (resp., H_i^-) be the positive (resp., negative) halfspace supported by H_i.

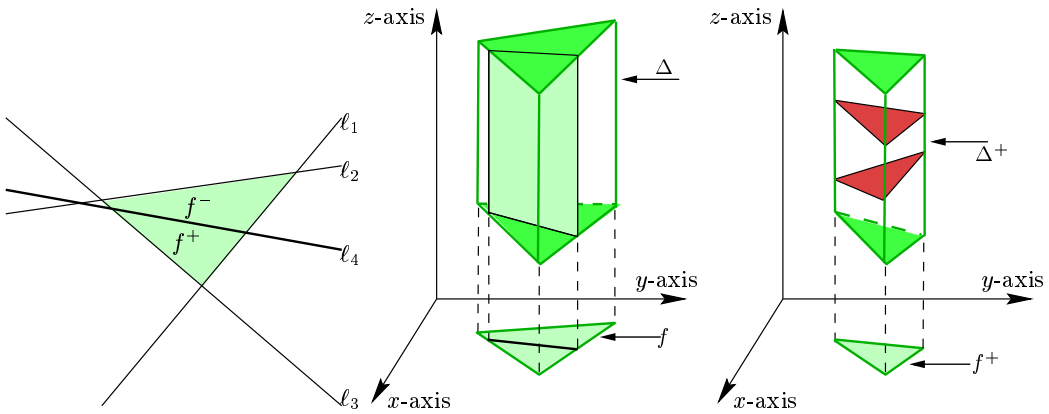


Figure 10: (a) Tracing ℓ_4 (the thick line) through the faces of $\mathcal{A}(\mathcal{L}^3)$. The face f is shaded. (b) Splitting cell $\Delta \in \mathbf{\Delta}(f)$ by H_4, the vertical plane containing ℓ_4. (c) The free cuts in F_{Δ^+} are ordered by z-coordinate.

1. We trace ℓ_i through the faces of $\mathcal{A}(\mathcal{L}^{i-1})$. For each face $f \in \mathcal{A}(\mathcal{L}^{i-1})$ intersected by ℓ_i, we use ℓ_i to split f into two faces f^+ and f^-. See Figure 10(a). Next, we partition each active cell $\Delta \in \mathbf{\Delta}(f)$ into two cells $\Delta^+ = \Delta \cap H_i^+$ and $\Delta^- = \Delta \cap H_i^-$ (see Figure 10(b)) and execute the following two steps on Δ:

2. We compute the set $S_{\Delta^+} \subseteq S_\Delta$ of triangles that intersect the interior of Δ^+. We also compute the set $F_{\Delta^+} \subseteq S_{\Delta^+}$ of triangles whose boundaries do not cross Δ^+. Similarly, we compute the sets S_{Δ^-} and F_{Δ^-} for Δ^-.

3. We split Δ^+ into a set Ψ of $|F_{\Delta^+}| + 1$ cells by making free cuts along each of the triangles in F_{Δ^+}. The cells in Ψ can be ordered by z-coordinate. Since the triangles in S are pairwise-disjoint, each triangle $s \in S_{\Delta^+} \setminus F_{\Delta^+}$ intersects a unique cell $\Delta' \in \Psi$. We compute Δ' by performing a binary search, and add s to $S_{\Delta'}$. For each cell $\Delta' \in \Psi$, we add Δ' to the set $\mathbf{\Delta}(f^+)$ if $S_{\Delta'} \neq \emptyset$. Next, we repeat the same procedure for Δ^-.

Whenever we split a three-dimensional cell into two, we add two children to the corresponding node in $\mathcal{B}^{i-1}$ and store the necessary information with the newly created nodes. The resulting tree is $\mathcal{B}^i$. The cuts made in Step 3 ensure that $\mathcal{B}^i$ satisfies property (P1). $\mathcal{B}^i$ satisfies property (P2) since the cuts made in Step 1 are vertical. Note that a triangle $s \in S$ does not intersect the interior of any cell after the three lines supporting the edges of s^* have been processed.

Remark: The free cuts made in Step 3 are crucial in keeping the size of the BSP quadratic. If, instead, we simply erect vertical planes as we do in Step 1 of the algorithm, and make cuts along a triangle $s \in S$ only when all three lines supporting the xy-projections of the edges of s have been added, then there are instances of input triangles for which our algorithm will construct a BSP of $\Omega(n^3)$ size regardless of the initial permutation of the triangles.

4.2 Analysis of the algorithm

We first bound the expected size of $\mathcal{B}$. A similar proof is used by Paterson and Yao [28] to analyze their randomized algorithm for constructing BSPs for triangles in $\mathbb{R}^3$. The cuts made by the algorithm partition each triangle in S into a number of sub-polygons; each such sub-polygon is contained in the cutting plane of some node in $\mathcal{B}$ and is stored at that node. Let $\nu(S)$ be the total number of polygons stored at the nodes of $\mathcal{B}$. The following lemma bounds the size of $\mathcal{B}$ in terms of $\nu(S)$.

Lemma 4.1 *The size of $\mathcal{B}$ is at most $17\nu(S)$.*

Proof: Recall that the size of $\mathcal{B}$ is defined to be the sum of the number of nodes in $\mathcal{B}$ and the total number of triangles stored at all the nodes in $\mathcal{B}$. To bound the size of $\mathcal{B}$, we count the number of nodes in $\mathcal{B}$ and then add $\nu(S)$. Let $\nu(E)$ be the number of segments into which the edges of the triangles in S are partitioned by the cuts in $\mathcal{B}$.

We first bound the number of leaves in $\mathcal{B}$ in terms of $\nu(S)$ and $\nu(E)$. Let v be the parent of a leaf in $\mathcal{B}$. Either a free cut or a vertical cut through an edge of a triangle in S is made at v. This implies that the number of leaves in $\mathcal{B}$ is at most $2(\nu(S) + \nu(E))$. To bound this quantity, for each triangle $s \in S$, consider the arrangement $\mathcal{A}_s$ on s formed by the intersection of s and the cutting planes in $\mathcal{B}$. Let e_s be the number of edges in $\mathcal{A}_s$ that are portions of edges of s and let f_s be the number of faces in $\mathcal{A}_s$. Since at most three edges of the boundary of a face in $\mathcal{A}_s$ are also portions of the edges of s, we have $e_s \leq 3f_s$. Summing over all triangles $s \in S$, we have $\nu(E) \leq 3\nu(S)$. Hence, the number of leaves in $\mathcal{B}$ is at most $8\nu(S)$, which implies that the number of nodes in $\mathcal{B}$ is at most $16\nu(S)$, thus proving the lemma. □

Thus, it suffices to bound the expectation $E\big[\nu(S)\big]$ to bound the expected size of $\mathcal{B}$. To that end, we count the expected number of new sub-polygons created in the ith stage, and sum the result over all stages. We bound the number ν_s^i of new sub-polygons into which a triangle $s \in S$ is partitioned by the cuts made in the ith stage, and sum the resulting bound over all triangles in S. Note that the vertical cuts made in the ith stage are contained in the vertical plane H_i containing ℓ_i.

Fix a triangle $s \in S$. For $1 \leq k \leq i$, let $\lambda_k = H_k \cap s$ be the segment formed by the intersection of H_k and s, and let Λ be the set of resulting segments. Note that the endpoints of each λ_k lie on the boundary of s. To calculate $E\big[\nu_s^i\big]$, consider the segment arrangement $\mathcal{A}(\Lambda)$ on s. We call a face of $\mathcal{A}(\Lambda)$ a *boundary* face if it is adjacent to an edge of s; otherwise, it is an *interior* face. See Figure 11. Recall that for a leaf $v \in \mathcal{B}^{i-1}$, we

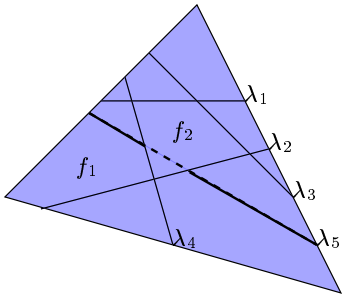


Figure 11: The arrangement $\mathcal{A}(\{\lambda_1, \lambda_2, \lambda_3, \lambda_4\})$ on triangle s (the shaded triangle). The face f_1 is a boundary face and the face f_2 is an interior face. The segment λ_5 does not partition f_2.

partition the cell Δ_v only if Δ_v is active. Property (P1) implies that the cuts made in the $(i-1)$th stage do not cross the interior of any interior face of $\mathcal{A}(\Lambda)$, since such a face cannot intersect the interior of any active cell Δ_v. Hence, ν_s^i is the number of boundary faces of $\mathcal{A}(\Lambda)$ that are intersected by λ_i. (If property (P1) did not hold, ν_s^i would be *all* the regions of $\mathcal{A}(\Lambda^i)$ that are intersected by λ_i.)

For $1 \leq k \leq i$, let $\mu(\Lambda, k)$ denote the number of boundary faces in the arrangement $\mathcal{A}(\Lambda \setminus \{\lambda_k\})$ that are intersected by λ_k. Observe that the sum $\sum_{1 \leq k \leq i} \mu(\Lambda, k)$ equals the total number of edges bounding the boundary faces of $\mathcal{A}(\Lambda)$. By a result of Bern et al. [8], the total number of edges of the boundary faces of $\mathcal{A}(\Lambda)$ is $O(i)$. Hence,

$$\sum_{1 \leq k \leq i} \mu(\Lambda, k) = O(i).$$

Since ℓ_i is chosen randomly from the set $\mathcal{L}^i$, λ_i can be any of the lines $\lambda_1, \lambda_2, \ldots, \lambda_i$ with equal probability. Therefore,

$$E[\nu_s^i] = \frac{1}{i} \sum_{1 \leq k \leq i} \mu(\Lambda, k) = O(1).$$

Hence, the total number of pieces created in the ith stage is $O(n)$. Summing over i, we find that the total number of sub-polygons into which the triangles in S are partitioned into over the course of the entire algorithm is $O(n^2)$. The following lemma is immediate:

Lemma 4.2 *The expected size of the BSP constructed by the algorithm is $O(n^2)$.*

Remark: Since each cell $\Delta_v \in \mathcal{B}$ is cylindrical, each vertex p of Δ_v is contained in one of the triangles s that contains the non-vertical faces of Δ_v. In fact, p is a vertex of the arrangement $\mathcal{A}(\{\lambda_1, \lambda_2, \ldots, \lambda_{3n}\})$ on s defined above. Thus, the above argument implies

that the expected value of the total number of vertices of the nodes of $\mathcal{B}$ is also $O(n^2)$. However, the height of $\mathcal{B}$ can be $\Omega(n)$, e.g., if the triangles in S form a convex polytope.

Before analysing the running time of the algorithm, we establish a relation between the projected edges intersecting an active face $f \in \mathcal{A}(\mathcal{L}^{i-1})$ and the triangles intersecting the cells in $\mathbf{\Delta}(f)$. For such an active face f, let k_f be the number of projected edges in E^* that intersect the interior of f. By Property (P1), if a triangle $s \in S$ intersects the interior of a cell $\Delta \in \mathbf{\Delta}(f)$, i.e., $s \in S_\Delta$, then the boundary of s also intersects the interior of Δ. Therefore, an edge of s^* intersects the interior of f. Since s intersects the interior of only one cell in $\mathbf{\Delta}(f)$, we obtain

$$\sum_{\Delta \in \mathbf{\Delta}(f)} |S_\Delta| \leq k_f. \tag{4.1}$$

We now analyze the expected running time of the algorithm. We count the time spent during the ith stage in inserting the line ℓ_i and then add this time over all stages of the algorithm. The zone theorem [11, 17] implies that in Step 1 of the algorithm, we spend $O(i)$ time in tracing ℓ_i through $\mathcal{A}(\mathcal{L}^{i-1})$. While processing an active face f of $\mathcal{A}(\mathcal{L}^{i-1})$ that intersects ℓ_i, for each cell $\Delta \in \mathbf{\Delta}(f)$, we spend $O(1)$ time in Step 1 and $O(|S_\Delta|)$ time in Step 2. In Step 3, for each triangle $s \in S_{\Delta^+} \setminus F_{\Delta^+}$, we spend $O(\log |F_{\Delta^+}|)$ time in the binary search used to find the cell in the set Ψ that intersects s. Hence, the total time spent in Step 3 for the face f is $O(|S_\Delta| \log |S_\Delta|)$. Thus, (4.1) implies that the total time spent in processing f is

$$\sum_{\Delta \in \mathbf{\Delta}(f)} O(|S_\Delta| \log |S_\Delta|) = O(k_f \log k_f).$$

Let Z be the set of all active faces of $\mathcal{A}(\mathcal{L}^{i-1})$ that are intersected by ℓ_i. The total time spent in the ith stage is

$$\sum_{f \in Z} O(k_f \log k_f).$$

We now bound this sum. If we denote the number of vertices on the boundary of a face f by $|f|$, then by the result of Bern et al. [8], we have $\sum_{f \in Z} |f| = O(i)$. Consider the vertical decomposition $\mathcal{A}^{\|}(\mathcal{L}^{i-1})$ of $\mathcal{A}(\mathcal{L}^{i-1})$. Each face $f \in \mathcal{A}(\mathcal{L}^{i-1})$ is decomposed into $O(|f|)$ trapezoids in $\mathcal{A}^{\|}(\mathcal{L}^{i-1})$. By standard random-sampling arguments, the expected number of edges in E^* that intersect the interior of any such trapezoid is $O((n \log i)/i)$. This implies that for a face $f \in \mathcal{A}(\mathcal{L}^{i-1})$, the expected value of k_f is $O(|f|(n \log i)/i)$. Hence, the expected time spent in the ith stage is

$$\sum_{f \in Z} O(k_f \log k_f) = \sum_{f \in Z} O\left(|f| \left(\frac{n \log i}{i}\right) \log n\right) = O(n \log^2 n),$$

which implies the following theorem:

Theorem 4.3 *Let S be a set of n non-intersecting triangles in $\mathbb{R}^3$. We can compute a BSP for S of expected size $O(n^2)$ in expected time $O(n^2 \log^2 n)$.*

5 BSPs for Triangles: A Deterministic Algorithm

In this section, we describe a deterministic algorithm for computing a BSP for a set S of n triangles in $\mathbb{R}^3$. As in the previous section, let E denote the set of edges of triangles in S, and let $E^* = \{e^* \mid e \in E\}$ be the set of xy-projections of the edges in E. Let k be the number of intersections between the edges in E^*. Our algorithm constructs a BSP $\mathcal{B}$ of size $O((n+k)\log^2 n)$ and height $O(\log n)$ in $O((n+k)\log^3 n)$ time.

As in the previous section, each node v of $\mathcal{B}$ is associated with a cylindrical cell Δ_v, but the top and bottom faces of Δ_v are now trapezoids. Let Δ_v^* denote the xy-projection of the top (or bottom) face of Δ_v; two of the edges of Δ_v^* are parallel to the y-axis. Before presenting our algorithm, we give some definitions.

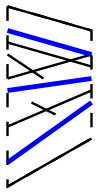


Figure 12: Anchored segments in E_v^* (these segments are drawn thick).

Let E_v^* be the set of segments in E^* that intersect the interior of Δ_v^* and are clipped within Δ_v^*. A segment $\gamma \in E_v^*$ is called *anchored* if its endpoints lie on the two parallel edges of Δ_v^* and if it does not intersect any other segment of E_v^*. Figure 12 shows an example. The anchored segments in E_v^* can be linearly ordered by y-coordinate (since they are disjoint). Let A_v be the set of anchored segments in E_v^* and let P_v be the set of intersection points between the segments of E_v^*. Finally, let $F_v \subseteq S_v$ be the set of all free triangles in S_v. Recall that a triangle $s \in S_v$ is free with respect to Δ_v if no edge of s intersects the interior of Δ_v; s partitions Δ_v into two cylindrical cells. Since Δ_v is a cylindrical cell, the triangles in F_v can be sorted by their z-coordinates.

5.1 The deterministic algorithm

In a pre-processing step, we compute all k intersection points of E^* in $O((n+k)\log n)$ time using the Bentley-Ottman sweep-line algorithm [7, 30]. Our algorithm then constructs $\mathcal{B}$ in a top-down fashion by maintaining a top subtree of $\mathcal{B}$. We say that a leaf v of the subtree is *active* if $S_v \neq \emptyset$. Note that v is active even if S_v contains free triangles; in Section 4, if a leaf v is active, then S_v does contain any free triangles. We store the set of all active leaves of the current subtree in a list. For each active leaf v, we maintain the sets P_v, A_v, F_v, and S_v. Note that we can easily compute the set E_v^* from S_v.

At each step of the algorithm, we choose an active leaf v, compute the cutting plane H_v, and use H_v to split Δ_v into two cells Δ_w and Δ_z. If S_w (resp., S_z) is nonempty, we mark w (resp., z) as being active. Before describing how we compute H_v, we specify how

we determine the sets P_w, F_w, S_w, and A_w (the procedure is symmetric for z):

P_w: Let $p \in P_v$ be the intersection point of e_1^* and e_2^*, where e_1 and e_2 are triangle edges; $p \in P_w$ if both e_1 and e_2 intersect Δ_w and p is contained in Δ_w^*.

F_w: Let s be a triangle in S_v. If s intersects Δ_w but none of the edges of s intersects the interior of Δ_w, then $s \in F_w$.

S_w: Let s be a triangle in S_v. If an edge of s intersects the interior of Δ_w, then $s \in S_w$.

A_w: Let $e^* \in E_v^*$, where e is an edge of a triangle in S_v. There are two cases to consider: (i) If $e^* \in A_v$ and e intersects Δ_w, then $e^* \in A_w$. (ii) If $e^* \notin A_v$, e intersects Δ_w, and no point in P_w is contained in e^*, then $e^* \in A_w$. To detect the second case, for each edge $e^* \in E_v^*$, we store the set of points in P_v that are contained in e^* (these points are formed by the intersection of e^* and other segments in E_v^*.

It is clear that these sets can be computed for both w and z in $O(|P_v| + |F_v| + |S_v| + |A_v|)$ time.

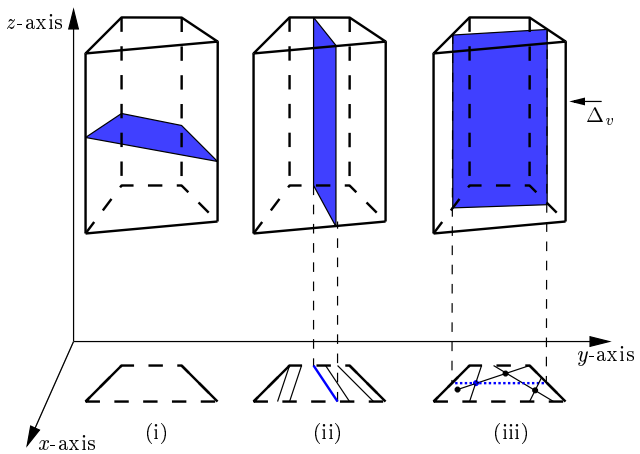


Figure 13: Cuts made in the deterministic algorithm: (i) Free cut, (ii) Cut parallel to the z-axis through an anchored segment, and (iii) Cut parallel to the yz-plane through a vertex of $\mathcal{A}(E_v^*)$.

We now describe how we compute the cutting plane H_v. Our algorithm uses three kinds of cuts: A *face* cut is a plane containing a triangle in S (all face cuts will be free cuts), an *edge* cut is a vertical plane erected on an anchored segment in A_v, and a *point* cut is a plane perpendicular to the x-axis passing through a point in P_v. See Figure 13. We choose the cutting plane H_v as follows:

1. $F_v \neq \emptyset$: Recall that the triangles in F_v are totally ordered by z-coordinate. We set H_v to be the face cut containing the median triangle s of F_v. See Figure 13(i). Since s

does not intersect any triangle of S_v, each triangle of $S_v \setminus \{s\}$ belongs to either S_w or S_z. The free triangles in F_v and the anchored segments in A_v can be similarly partitioned.

2. $F_v = \emptyset$ and $A_v \neq \emptyset$: Recall that the anchored segments are totally ordered by y-coordinate. We set H_v to be the edge cut induced by the median anchored segment in A_v. See Figure 13(ii). Since no segment in A_v intersects H_v, the anchored segments of A_v can be partitioned between A_w and A_z.

3. $F_v = \emptyset, A_v = \emptyset$: We set H_v to be the point cut through the vertex in P_v with the median y-coordinate. See Figure 13(iii).

5.2 Analysis of the algorithm

We now analyze the algorithm. We first bound the size of $\mathcal{B}$, then the running time of the algorithm, and finally the height of $\mathcal{B}$. Let v be a node in $\mathcal{B}$ such that P_v contains p intersection points, A_v contains a anchored segments, and F_v contains f free triangles; clearly S_v contains $O(p+a+f)$ triangles. Let $\mathcal{B}_v$ denote the subtree of $\mathcal{B}$ rooted at v. Set

$$S(p,a,f) = \max_v |\mathcal{B}_v|,$$

where the maximum is taken over all nodes v with $|P_v| = p, |A_v| = a$, and $|F_v| = f$. We bound $S(p,a,f)$ by setting up a recurrence for it. Suppose H_v, the cutting plane at v, partitions Δ_v into two cells Δ_w and Δ_w. Let $p_w = |P_w|, a_w = |A_w|$, and $f_w = |F_w|$; define p_z, a_z, and f_z similarly. Note that P_w and P_z are disjoint subsets of P_v; therefore, $p_w + p_z \leq p$.[2] We consider three cases:

$f \neq 0$: H_v is a free cut containing the median free triangle in F_v. Since H_v is a free cut, it does not intersect any triangles in S_v. Hence, we have $a_w + a_z = a$, and $f_w, f_z \leq f/2$.

$f = 0, a \neq 0$: H_v is an edge cut erected on the median anchored segment in A_v; therefore $a_v, a_z \leq a/2$. Since H_v is an edge cut, it may intersect triangles in S_v, creating free triangles in F_w and F_z. However, there are $O(p+a)$ triangles in S_v; hence, we have that $f_w + f_z = O(p+a)$.

$f = 0, a = 0$: H_v is a point cut defined by the vertex in P_v with median y-coordinate, which implies that $p_w, p_z \leq p/2$. Since H_v may intersect triangles in S_v, both Δ_w and Δ_z can contain anchored segments and free triangles. Since there are $O(p)$ edges in E_v^* and $O(p)$ triangles in S_v, we have $a_w, a_z, f_w, f_z = O(p)$.

In the first case, the size of node v is 2, since the free triangle inducing H_v is stored at v. In the other two cases, the size of node v is 1. The above discussion implies that we can write the following recurrence for $S(p,a,f)$:

$$S(p,a,f) \leq S(p_w, a_w, f_w) + S(p_z, a_z, f_z) + 2, \tag{5.1}$$

[2] It is possible that a point $p \in P_v$ is not an element of either P_w or P_z, for example, when p is the intersection of two projected triangle edges e_1^* and e_2^* but only e_1 intersects Δ_w.

where $p_w + p_z \leq p$, and

1. $a_w + a_z = a$, and $f_w, f_z \leq f/2$, if $f \neq 0$;
2. $a_v, a_z \leq a/2$, and $f_w + f_z = O(p + a)$, if $f = 0, a \neq 0$;
3. $p_w, p_z \leq p/2$, and $a_w, a_z, f_w, f_z = O(p)$, if $f = 0, a = 0$.

Using mathematical induction, we can prove that the solution to this recurrence is

$$S(p, a, f) = O(p \log^2 p + (p + a) \log a + f).$$

Since the root node of $\mathcal{B}$ has $n + k$ intersection points, no anchored segments, and no free triangles, we obtain the following lemma:

Lemma 5.1 *The size of $\mathcal{B}$ is $O((n + k) \log^2 n)$.*

We now analyze the running time of the algorithm. As we have noted earlier, at each node v, we can choose H_v and perform the operations to split v in $O(p + a + f)$ time. If $T(p, a, f)$ denotes the maximum time taken by our algorithm to construct the subtree of $\mathcal{B}$ rooted at a node v with $|P_v| = p, |A_v| = a$, and $|F_v| = f$ (the maximum is taken over all such nodes v), we have

$$T(p, a, f) = T(p_w, a_w, f_w) + T(p_z, a_z, f_z) + O(p + a + f),$$

where $p_w, a_w, f_w, p_z, a_z, f_z$ satisfy the same conditions as in (5.1). Using mathematical induction, we can prove that the solution to the above recurrence is

$$T(p, a, f) = O(p \log^3 p + (p + a) \log(p + a) \log a + (p + a + f) \log f).$$

Thus, we obtain the following lemma:

Lemma 5.2 *The time taken by our algorithm to construct $\mathcal{B}$ is $O((n + k) \log^3 n)$.*

We complete the analysis by bounding the height of $\mathcal{B}$. We need to make two modifications to the algorithm. The first modification will enable us to prove an $O(\log^3 n)$ bound on the height of $\mathcal{B}$ and the second change will help us improve this bound to the desired $O(\log n)$ bound.

Bounding the height of $\mathcal{B}$ at $O(\log^3 n)$

Recall that E^* is the set of xy-projections of the edges of the triangles in S. Imagine executing Steps 2 and 3 of the algorithm on the edges of E^* and restricting the cuts made in these steps to lie in the xy-plane. Thus, we are constructing a BSP $\mathcal{C}$ in the plane for E^*. Let E' be the set of segments that the edges of E^* are partitioned into by these cuts made in $\mathcal{C}$; by definition, the segments in E' do not intersect, except at the endpoints. We modify the algorithm so that it uses the triangles in S and the edges in E' to construct $\mathcal{B}$. We

first need a few definitions. For a node $v \in \mathcal{B}$, we define E'_v to be the set of edges in E' that intersect the interior of Δ^*_v and are clipped to Δ^*_v. We say that a segment $\gamma \in E'_v$ is *anchored* if its endpoints lie on the two parallel edges of ${\Delta_v}^*$. As before, we can linearly order the anchored segments in E'_v by y-coordinate (since they are disjoint). Let A'_v be the set of anchored segments in E'_v and let P'_v be the set of those endpoints of the edges in E'_v that lie in the interior of Δ^*_v (i.e., if an endpoint p of a segment in E'_v lies on the boundary of Δ^*_v, then $p \notin P'_v$).

With these definitions in hand, we are now ready to describe the first modification to our algorithm: In a pre-processing phase, we construct the BSP $\mathcal{C}$ as described above. Next, we run the algorithm on the triangles in S as described in Section 5.1, except that we use the sets F_v, A'_v, and P'_v to decide how to split a cell Δ_v.

By using an analysis similar to the one described earlier, we can show that the size of $\mathcal{C}$ is $O((n+k)\log n)$. Similarly, we can show that the size of $\mathcal{B}$ is still $O((n+k)\log^2 n)$ and that it can be constructed in $O((n+k)\log^2 n)$ time. We now prove that the height of $\mathcal{B}$ is $O(\log^3 n)$. Consider a root-to-leaf path π in $\mathcal{B}$. Since we make the median point cut in Step 3, there are $O(\log n)$ nodes in π that are split by a point cut. Let u and w two such consecutive nodes in π (i.e., no node between u and w in π is split by a point cut. Since we make the median edge cut in Step 2, there are $O(\log n)$ nodes in π between u and w where an edge cut is made.[3] Similarly, if u' and w' are consecutive nodes in π that are split by edge cuts, there are $O(\log n)$ nodes in π between u' and w' (a free cut is made at all such nodes). Hence, the length of π is $O(\log^3 n)$, which implies that the height of $\mathcal{B}$ is $O(\log^3 n)$.

Improving the height of $\mathcal{B}$ to $O(\log n)$

We now describe the second modification to the algorithm, which will enable us to prove an $O(\log n)$ bound on the height of $\mathcal{B}$. We define the *measure* μ_v of a node $v \in \mathcal{B}$ to be the number of endpoints in P'_v. Our goal is to ensure that for every node $v \in \mathcal{B}$, there is no descendant w of v such that $\mu_w > 2\mu_v/3$ and there are more than a constant (we specify this constant below) number of nodes between v and w. Clearly, we can prove the desired $O(\log n)$ bound on the height of $\mathcal{B}$ if we achieve this goal. To do so, we modify Steps 1 and 2 as follows:

1. $F_v \neq \emptyset$:
   (a) We set H_v to be the face cut containing a triangle $t \in F_v$ such that H_v splits Δ_v into two regions Δ_w and Δ_z such that $\mu_w, \mu_z \leq 2\mu_v/3$, if such a triangle t exists.
   (b) Otherwise, we compute two consecutive triangles $t_1, t_2 \in F_v$, split Δ_v into two regions Δ_w and Δ_z using the face cut containing t_1, and split Δ_z into two regions Δ_x and Δ_y using the face cut containing t_2 such that $\mu_x > 2\mu_v/3$.

[3] Without the above modification to the algorithm, we cannot make this claim. In the original algorithm, a face cut made at a node between u and w may cause a "new" anchored segment γ to appear; γ will induce an edge cut at a node between u and w, eventhough γ is not one of the anchored segments created by the point cut at u.

It is not difficult to prove that one of these two cases is always true. We similarly modify Step 2. Let $\mathcal{B}$ be the BSP constructed by the algorithm with the above modifications to Steps 1 and 2. We now prove a lemma that implies that the height of $\mathcal{B}$ is $O(\log n)$. We first need a simple definition: if v is a node of $\mathcal{B}$ and w is a descendant of v in $\mathcal{B}$, then the *distance* between v and w is the number of nodes in the path from v to w (not including v and w).

Lemma 5.3 *Let v be a node in $\mathcal{B}$. If w is a descendant of v and the distance between w and v is six, then $\mu_w \leq 2\mu_v/3$.*

Proof: We first define some notation that will be useful in the proof. For a node $v \in \mathcal{B}$, consider the subtree rooted at v such that all leaves of the subtree have measure at most $2\mu_v/3$ and all interior nodes of the subtree have measure greater than $2\mu_v/3$. We use d_v to denote the height of this subtree (the height of a tree is the maximum distance between the root and a leaf of the tree). We claim that for any node $v \in \mathcal{B}$, $d_v \leq 6$. Clearly, the lemma is true if we prove this claim.

If we apply Step 1(a), Step 2(a), or Step 3 at v, then $d_v = 0$ since the measure of child of v is at most $2\mu_v/3$. Suppose we apply Step 1(b) or Step 2(b) at v. The cuts made at either step split Δ_v into three regions Δ_w, Δ_x, and Δ_y such that $\mu_x > 2\mu_v/3$; therefore, $d_v = d_x + 2$. We observe that either F_x or A'_x must be empty. If we apply Step 1(b) at v, we split Δ_v using the face cuts containing two *consecutive* triangles in F_v; therefore, $F_x = \emptyset$. Similarly, if we apply Step 2(b) at v, we split Δ_v using the edge cuts containing two *consecutive* anchored segments in A'_v; therefore, $A'_x = \emptyset$.

We now prove a bound on d_x. If we apply Step 1(a), Step 2(a), or Step 3 at x, we have $d_x = 0$. Let us now consider the other possibilities (i.e., we split Δ_x using Step 1(b) or Step 2(b)). Let x' be the grandchild of x such that $\mu_{x'} > 2\mu_x/3$, which implies that $d_x = d_{x'} + 2$. We consider the two possible cases:

1. $F_x \neq \emptyset$ and $A'_x = \emptyset$: We apply Step 1(b) at x. Using the observation we made above, we see that $F_{x'} = \emptyset$. Further, $A'_{x'} = \emptyset$ since the face cuts that split Δ_x do not create any new anchored edges. Therefore, we split x' using a point cut, which implies that $d_x = 2$.

2. $F_x = \emptyset$ and $A'_x \neq \emptyset$: We apply Step 2(b) at x. Using the observation we made earlier, we see that $A'_{x'} = \emptyset$. Applying the argument of the previous case to x', we have $d_{x'} = 2$, which implies that $d_x = 4$.

The above argument shows that $d_v \leq 6$ for all nodes $v \in \mathcal{B}$. □

Combining the last three lemmas, we state the main result of this section:

Theorem 5.4 *Let S be a set of n triangles in $\mathbb{R}^3$, and let k be the number of intersection points of the xy-projections of the edges of S. We can compute a BSP of size $O((n+k)\log^2 n)$ and height $O(\log n)$ for S in $O((n+k)\log^3 n)$ time.*

6 Conclusions

In this paper, we first presented an efficient algorithm to maintain a BSP of a set of moving segments in the plane. Currently, we do not know any non-trivial lower bounds for this problem. Agarwal et al. [1] have extended our result and developed an algorithm to maintain BSPs for moving triangles in $\mathbb{R}^3$.

We have also presented algorithms for constructing BSPs for triangles in $\mathbb{R}^3$. The randomized algorithm constructs a BSP of worst-case optimal size and runs in near-optimal time in the worst case. The deterministic algorithm is near-optimal in the worst-case. However, for inputs such as terrains that actually arise in practice, the number of intersections between the xy-projections of the triangles is likely to be near-linear. In such cases, our deterministic algorithm constructs BSPs of near-linear size.

There are many interesting open questions regarding BSPs. First of all, our deterministic algorithm is likely to construct BSPs of near-linear size for terrains and urban landscapes, which are common in computer graphics and geographic information systems, but might not be very good for data sets in other application domains (e.g., CAD design). Proving near-linear bounds on BSP size in models that capture the geometric structure of such inputs will be very useful. Secondly, all our algorithms for triangles in $\mathbb{R}^3$ construct BSPs of $\Omega(n^2)$ size even if an $O(n)$ size BSP exists. This raises the question of constructing a BSP of optimal or near-optimal size for triangles in $\mathbb{R}^3$. It is not known whether the problem is *NP*-hard.

References

[1] P. K. Agarwal, J. Erickson, and L. J. Guibas, Kinetic binary space partitions for intersecting segments and disjoint triangles, *Proc. 9th ACM-SIAM Symposium on Discrete Algorithms*, 1998, pp. 107–116.

[2] P. K. Agarwal, E. F. Grove, T. M. Murali, and J. S. Vitter, Binary space partitions for fat rectangles, *Proc. 37th Annu. IEEE Sympos. Found. Comput. Sci.*, October 1996, pp. 482–491.

[3] P. K. Agarwal and S. Suri, Surface approximation and geometric partitions, *Proc. 5th ACM-SIAM Sympos. Discrete Algorithms*, 1994, pp. 24–33.

[4] 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.

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

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

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

[8] M. Bern, D. Eppstein, P. Plassman, and F. Yao, Horizon theorems for lines and polygons, in: *Discrete and Computational Geometry: Papers from the DIMACS Special Year* (J. Goodman, R. Pollack, and W. Steiger, eds.), *DIMACS Series in Discrete Mathematics and Theoretical Computer Science*, Vol. 6, American Mathematical Society, Providence, RI, 1991, pp. 45–66.

[9] A. T. Campbell, *Modeling Global Diffuse Illumination for Image Synthesis*, Ph.D. Thesis, Dept. of Computer Sciences, University of Texas, Austin, 1991.

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

[11] B. Chazelle, L. J. Guibas, and D. T. Lee, The power of geometric duality, *BIT*, 25 (1985), 76–90.

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

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

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

[15] K. L. Clarkson and P. W. Shor, Applications of random sampling in computational geometry, II, *Discrete Comput. Geom.*, 4 (1989), 387–421.

[16] M. de Berg, Linear size binary space partitions for fat objects, *Proc. 3rd Annu. European Sympos. Algorithms*, *Lecture Notes Comput. Sci.*, Vol. 979, Springer-Verlag, 1995, pp. 252–263.

[17] H. Edelsbrunner, *Algorithms in Combinatorial Geometry*, Springer-Verlag, Heidelberg, West Germany, 1987.

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

[19] L. J. Guibas, Kinetic data structures: A state of the art report, in: *Robotics: An Algorithmic Perspective* (P. K. Agarwal, L. E. Kavraki, and M. T. Mason, eds.), 1998, pp. 191–209.

[20] D. Haussler and E. Welzl, Epsilon-nets and simplex range queries, *Discrete Comput. Geom.*, 2 (1987), 127–151.

[21] C. Mata and J. S. B. Mitchell, Approximation algorithms for geometric tour and network design problems, *Proc. 11th Annu. ACM Sympos. Comput. Geom.*, 1995, pp. 360–369.

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

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

[24] 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, February 1986.

[25] B. F. Naylor, SCULPT: an interactive solid modeling tool, *Proc. Graphics Interface '90*, 1990, pp. 138–148.

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

[27] B. F. Naylor, J. Amanatides, and W. C. Thibault, Merging BSP trees yields polyhedral set operations, *Proc. SIGGRAPH 90*, *Comput. Graph.*, Vol. 24, ACM SIGGRAPH, 1990, pp. 115–124.

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

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

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

[31] 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.

[32] R. Seidel, Backwards analysis of randomized geometric algorithms, in: *New Trends in Discrete and Computational Geometry* (J. Pach, ed.), Springer-Verlag, Heidelberg, 1993, pp. 37–68.

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

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

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