Discrete Comput Geom 9:11–38 (1993)

Discrete & Computational
Geometry

Applications of a New Space-Partitioning Technique*

Pankaj K. Agarwal[1] and Micha Sharir[2]

[1] Computer Science Department, Duke University,
Durham, NC 27706, USA

[2] School of Mathematical Sciences, Tel Aviv University,
Tel Aviv, Israel, and
Courant Institute of Mathematical Sciences, New York University,
New York, NY 10012, USA

Abstract. We present several applications of a recent space-partitioning technique of Chazelle, Sharir, and Welzl (*Proceedings of the 6th Annual ACM Symposium on Computational Geometry*, 1990, pp. 23–33). Our results include efficient algorithms for output-sensitive hidden surface removal, for ray shooting in two and three dimensions, and for constructing spanning trees with low stabbing number.

1. Introduction

In a recent paper, Chazelle *et al.* [16] have given a new quasi-optimal technique for simplex range searching in any dimension. The technique is based on a new hierarchical space-partitioning scheme, which we briefly review below. As it turns out, this partitioning scheme has several useful properties that make it applicable to a variety of other problems. Briefly, these properties are:

(i) It yields quasi-optimal query time for simplex range searching, which

* Work on this paper has been supported by DIMACS, an NSF Science and Technology Center, under Grant STC-88-09684. The second author has been supported by Office of Naval Research Grants N00014-89-J-3042 and N00014-90-J-1284, by National Science Foundation Grant CCR-89-01484, and by grants from the U.S.–Israeli Binational Science Foundation, the Fund for Basic Research, administered by the Israeli Academy of Sciences, and the G.I.F., the German–Israeli Foundation for Scientific Research and Development.

matches Chazelle's lower bound up to a factor of $O(n^{\varepsilon})$ for arbitrarily small $\varepsilon > 0$.[1]

(ii) It works in any fixed dimension.

(iii) It admits tradeoff between storage and query time.

(iv) Its preprocessing cost is low—within a factor $O(n^{\varepsilon})$ of the storage used.

Properties (i)–(iv) are noted in [16].

(v) It facilitates the use of multilevel data structures. This allows us to process queries that are expressed as a conjunction of constraints, so that each constraint is handled by accessing a different level in the structure. Similar ideas have been applied by Dobkin and Edelsbrunner [22] and by Guibas *et al.* [26] to an inferior partitioning scheme (in the plane). Chazelle *et al.* [16] do exploit this multilevel property, but not to full generality.

(vi) It can be efficiently dynamized. This property is newly developed here.

In this paper we study several of these applications and derive efficient solutions for them using the "CSW scheme." The results that we obtain are:

Efficient Construction of a Spanning Tree with Low Stabbing Number. Given a set S of n points in $\mathbb{R}^d$, a (straight-edge) spanning tree T on S is said to have *stabbing number* κ if no hyperplane crosses more than κ edges of T. It has been shown in [17] that there always exists a spanning tree on S with stabbing number $\kappa = O(n^{1-1/d})$. Such spanning trees have been applied to simplex range searching [17], [44], computing a single face in arrangements of lines [25], and ray shooting in a collection of segments [1], [19]. In the planar case the fastest known (deterministic) algorithm for the construction of such a tree runs in time $O(n^{3/2} \log^2 n)$ [27] (see also [2], [17], [27], [28], and [44]). No efficient algorithm is known for computing spanning trees of low stabbing number in higher dimensions.

Using the CSW scheme, we derive an improved (deterministic) algorithm that computes a spanning tree with stabbing number $O(n^{1-1/d+\varepsilon})$ in time $O(n \log n)$ for $d = 2, 3$. Thus, although the stabbing number that we get is slightly suboptimal, the time needed to construct the spanning tree is greatly reduced. Moreover, our algorithm allows us to update the tree in amortized time $O(\log^2 n)$, as we insert or delete a point. This is a significant improvement over the previous algorithm by Cheng and Janardan [19] that requires roughly $\sqrt{n}$ amortized time to update the spanning tree.

Ray Shooting Amidst Segments in the Plane. In this basic probem we wish to preprocess such a collection into a data structure that supports fast *ray-shooting* queries, in which we need to determine the first segment, if any, to be hit by a query ray.

Chazelle and Guibas gave an optimal algorithm for the special case where the segments form the boundary of a simple polygon [15]. A considerably simpler algorithm for the same case has recently been proposed by Chazelle *et al.* [12]. If

[1] Throughout this paper ε denotes a positive constant which can be chosen arbitrarily small with an appropriate choice of other constants in the algorithms. We use the notion of *quasi-optimality* to refer to a bound that is slightly inferior to an optimal bound, usually worse by a factor of at most $O(n^{\varepsilon})$. Similar terms, such as *quasi-linearity*, are defined in an analogous manner.

the segments do not form a simple polygon, the problem becomes much harder and no algorithm is known that answers a query in polylogarithmic time using roughly linear space.

In this paper we present a solution that requires $O(n \log^2 n)$ preprocessing and storage, and answers queries in time $O(n^{1/2+\varepsilon})$. Solutions that match (and even slightly improve) the storage and query time are known (see, for instance, [1], [8], and [19]), but a preprocessing as fast as ours appears to be new—previous bounds on the preprocessing cost are close to $O(n^{3/2})$. We can also obtain a whole range of resource bounds, in which we tradeoff storage (and preprocessing) for query time, as in [16]. In particular, we can answer a query in $O(n^{1+\varepsilon}/\sqrt{s})$ time, using $O(s^{1+\varepsilon})$ preprocessing, if we are allowed to use s units of storage.

Our algorithm can also be modified to report all k segments intersected by a query segment in time $O(n^{1+\varepsilon}/\sqrt{s} + k)$, or to count the number of such segments in time $O(n^{1+\varepsilon}/\sqrt{s})$. Another advantage of our algorithm is that, unlike previous algorithms, we can efficiently maintain the structure dynamically as we insert or delete segments.

Improved Output-Sensitive Hidden Surface Removal. Given n horizontal triangles in $\mathbb{R}^3$, we wish to compute their visibility map, as viewed from a point at $z = -\infty$ (this is a representative case of a more general situation, where the viewed objects have a known *depth order* with respect to the viewing point). Overmars and Sharir [36] have given two output-sensitive algorithms—the first algorithm is very simple and runs in time $O(n\sqrt{k} \log n)$, where k is the number of vertices in the visibility map; the second algorithm is quite complicated but runs in time $O(n^{4/3} \log^{2/3} n + k^{3/5} n^{4/5+\varepsilon})$. We first show that the running time of this latter algorithm can be improved using our ray-shooting algorithms, and then describe a simpler algorithm that computes the visibility map in time $O(n^{2/3+\varepsilon} k^{2/3} + n^{1+\varepsilon})$. This algorithm is based on the dynamic version of our ray-shooting technique.

Ray Shooting in 3-Space. Here we are given a collection of n triangles in $\mathbb{R}^3$ and wish to preprocess it into a data structure that supports fast ray-shooting queries, defined in complete analogy with the planar case. This is a central problem in computer graphics, and, unfortunately, is considerably more difficult than its planar counterpart. We obtain a solution that requires $O(n^{16/15+\varepsilon}/s^{4/15})$ query time with s storage, so it yields $O(n^{4/5+\varepsilon})$ query time with roughly linear storage. We also note that other solutions have been given earlier for several special cases of the general problem. For example, Schmitt *et al.* [40] discuss the case of axis parallel rectangles. Agarwal [3] (see also [37]) discusses the case of vertical ray shooting. Cole and Sharir [21] and later Chazelle *et al.* [14] have given efficient algorithms for ray shooting in polyhedral terrains. Recently and independently, de Berg *et al.* [10] derived an alternative technique that yields similar bounds for the general ray-shooting case, as well as efficient algorithms for several special cases. Some related problems (e.g., reporting the set of triangles intersected by a query line, counting the number of such triangles, etc.) have been considered by Pellegrini [39], but his approach fails to give an efficient algorithm for the ray-shooting problem.

This paper is organized as follows. In Section 2 we briefly review the CSW partitioning scheme, and in Section 3 we explain how to dynamize it. Section 4 describes an efficient construction of spanning trees with low stabbing number. Section 5 presents the application of the CSW scheme to planar ray shooting amidst a collection of line segments. This in turn is applied in Section 6 to obtain two improved output-sensitive hidden surface removal algorithms. Section 7 presents applications of the scheme to ray shooting in three dimensions and related problems.

2. The CSW Partitioning Scheme—An Overview

In this section we briefly review the space-partitioning scheme of Chazelle *et al.* [16]. This scheme produces a multilevel data structure, but we describe here only its primary (top-level) structure, which is rather similar to a geometric partition tree. In the following sections we exploit the multilevel characteristic of the data structure, which allows us to attach to its nodes substructures that will generally be different from those used in [16].

Let S be a set of n points in $\mathbb{R}^d$ that we wish to preprocess for efficient half-space range queries. The CSW structure has two types of nodes, "simplex nodes" and "triangulation nodes," which alternate in depth. Each simplex node v of the structure is associated with a subset $S_v \subseteq S$ of n_v points in (some simplex in) d-space. We fix some sufficiently large constant parameter r. We start at the root node of the structure with the entire set S. Using a recent algorithm of Matoušek [29], we construct $O(\log r)$ different triangulations of d-space, each consisting of $O(r^d)$ simplices, so that the following property holds: for any hyperplane h there exists at least one triangulation such that h crosses only $O(r^{d-1})$ simplices of the triangulation and that only $O((n/r)\log r)$ points of S lie in those simplices. We say that such a triangulation is *sparse* for h. We now create $O(\log r)$ children of the root; they are triangulation nodes, and each corresponds to one of these triangulations. For each such child t, and for each simplex τ in the triangulation that t represents, we create a child of t that corresponds to τ; this is a simplex node that has the subset $\tau \cap S$ associated with it.

Here is a brief review of how these triangulations are constructed. We first pass to dual space, where the given points are mapped into n dual hyperplanes. We apply Matoušek's partitioning algorithm [29], with the same parameter r chosen above, to obtain a decomposition Ξ of the dual space into $O(r^d)$ simplices, each meeting only $O(n/r)$ dual hyperplanes. Taking the vertices of these simplices, with the appropriate multiplicity, and mapping them back to the primal space, we obtain a multiset H of $m = O(r^d)$ "representative" hyperplanes that "approximate" well any other hyperplane, where the distance between two hyperplanes is the number of points of S that separate them. That is, for any hyperplane h_0 there is a hyperplane $h \in H$ so that only $O(n/r)$ points of S separate h_0 from h. Using Matoušek's algorithm, we choose r hyperplanes from H and triangulate the arrangement of these hyperplanes, so that each simplex intersects only $O((m/r)\log r)$ hyperplanes of H. A simple counting argument shows that the

average number of points of S in the zone of (i.e., the collection of cells crossed by) a representative hyperplane is $O((n/r)\log r)$. We call a hyperplane $h \in H$, whose dual vertex h^* is associated with some simplex ξ of Ξ, a "good" hyperplane if the triangulation is sparse for all the hyperplanes corresponding to the vertices of ξ. Another simple counting argument shows that at least a fraction of the hyperplanes of H are good. We remove these good hyperplanes, and repeat the sampling process on the remaining hyperplanes of H. We repeat this process a logarithmic number of times until we are left with $O(r)$ hyperplanes. We then choose all these hyperplanes and triangulate their arrangement.

We now recurse with this partitioning scheme in each simplex τ of each triangulation (with the set of points contained in the simplex), unless either

(a) τ contains more than $(cn/r)\log r$ points of S, for some appropriate constant c, or
(b) the number of points within τ is less than some parameter σ, chosen as a function of the storage s that we allow for the structure.

We refer to the structure resulting from this recursive process as the *top* part of the whole structure.

In case (a) the simplex becomes a so-called "fat leaf," and the (current level of the) structure is not expanded there any further. In case (b) we expand the structure at τ using the following different scheme. The points of S within τ are dualized to hyperplanes, and we continue the construction in dual space. Applying Matoušek's algorithm [29], we choose a sample of τ hyperplanes in dual space and triangulate their arrangement so that each simplex intersects only $O((\sigma/r)\log r)$ dual hyperplanes. There are $O(r^d)$ simplices in the triangulated arrangement. We associate with each resulting simplex τ the set $H(\tau)$ of hyperplanes that lie strictly above τ (and a similar set of hyperplanes that lie strictly below it). We continue recursively to process, for each simplex τ, the subset of hyperplanes crossing τ, in the same manner. The total storage (and preprocessing cost) required for this procedure is easily seen to be $O(\sigma^{d+\varepsilon})$ (see [16]). We refer to this structure, constructed for "leaf-simplices" of the top part, having fewer than σ points each, as the *bottom* part of the structure.

In two and three dimensions the structure can be somewhat simplified, as follows. As earlier, we choose a sample of r hyperplanes (since $d = 2, 3$, hyperplanes are either lines or planes), but instead of triangulating their arrangement, we now compute its *vertical decomposition*, as defined in [13] (see also [20]). We then find "good" hyperplanes, defined as above, with respect to this map, throw them away, and repeat this step with the remaining hyperplanes. We thus obtain a family of $O(\log r)$ maps. We superimpose these $O(\log r)$ vertical decompositions to obtain a single partitioning $\mathcal{M}$ of d-space. For each cell τ of $\mathcal{M}$, we create a child of the current node of the structure, and associate with it the subset $S \cap \tau$, in complete analogy with the original scheme. We recurse with this partitioning scheme on each child as above. (Note that each cell of the map contains only $O((n/r)\log r)$ points of S, therefore in this case a cell can never be a fat leaf.) Since each hyperplane in H is good for at least one vertical decomposition, it follows easily that all hyperplanes of H are good for $\mathcal{M}$, in the same sense as above. Moreover,

Chazelle *et al.* [16] have proved that, in $\mathbb{R}^2$, $\mathcal{M}$ has $O(r^2 \log r)$ cells and each line intersects $O(r \log r)$ cells of $\mathcal{M}$, and that, in $\mathbb{R}^3$, $\mathcal{M}$ has $O(r^3 \log^6 r)$ cells and each plane intersects $O(r^2 \log^5 r)$ cells of $\mathcal{M}$. At present we do not have sharp bounds for the complexity of the superimposed map $\mathcal{M}$, and of zones of hyperplanes in $\mathcal{M}$, in higher dimensions; even for $d = 3$ the above complexity bounds are derived only for a special variant of the vertical decomposition scheme. This is why we have to maintain $O(\log r)$ different maps for $d \geq 4$. The advantages of using this improved scheme are discussed below. This improvement is not given in [16], and is newly observed here.

As explained in [16], a half-space query—reporting the points lying above (or below) a query hyperplane h—can be answered as follows. We find a sparse triangulation for h at the root of the structure. We then determine all cells of this triangulation that lie fully above h and those that are crossed by h.[2] Simplices of the former type are fully within the query range, whereas simplices of the second type are processed further recursively. (Note that, since the triangulation is sparse for h, no simplex of the second type is a fat leaf, so the query will never be stuck at such a leaf.) If the number of points within a simplex is less than σ, we switch to the second kind (i.e., bottom part) of the data structure. We locate the simplex τ that contains the point h^*, dual to the query hyperplane h, and retrieve the associated set $H(\tau)$. We next recurse in the substructure for τ. In total, we obtain a compact representation of the set of points that lie in our range, as the disjoint union of "canonical" prestored subsets.

There are several issues to observe about the scheme. The first is the storage s (and preprocessing cost) needed for the structure. Actually, this can be set to any desired value between $n^{1+\varepsilon}$ and $n^{d+\varepsilon}$. The larger s is, the more "efficient" a query becomes, where query efficiency is measured as the number of canonical subsets whose union constitutes the query output. Roughly speaking, to achieve storage s, we need to choose σ to be about $(s/n)^{1/(d-1)}$; the precise analysis is given in [16].

The next issue is the potential of the resulting structure to support multilevel substructures. We note that the output to a half-space query is given as the disjoint union of "canonical" subsets, where each subset is either the set of points of S within a simplex in one of the triangulations used in the top part of the structure, or is the primal version of a subset of the dual hyperplanes that pass above (or below) a simplex in the bottom part of the structure. In either case, we can take each such subset, process it further to obtain a second-level data structure, and attach it to the corresponding primary node. We note that the technique in [16] does just that, where the secondary structures are of the same kind as the primary structure; in fact, the structure produced in [16] is $d + 1$ levels deep. In some of the applications given in this paper, we first apply the CSW partitioning scheme recursively for a constant number of levels and then, at each cell, we construct an entirely different structure tailored to the specific problem that we are trying to solve. In our applications the lower-level structures will usually be on a set of

[2] For $d = 2, 3$, there is only one triangulation stored at the root, and we find the cells of this map that lie above h and those that intersect h.

different objects that stand in 1–1 correspondence with the original points of S—see below for examples.

Another issue is the "efficiency" of a query, as defined above. As argued in [16], the total number of canonical subsets that a query collects is $O(n^{1+\varepsilon}/s^{1/d})$. Moreover, the analysis in [16] also shows that even if we perform a second-level query (or the same type as the top-level query) at each of these subsets (actually going down any constant number of levels) the total number of subsets, of any level, that arise is still $O(n^{1+\varepsilon}/s^{1/d})$ (with a larger constant of proportionality). Our multilevel structures will have a similar property, as will be observed below.

Remark 2.1. In two and three dimensions, if $\sigma = O(1)$, that is, if only the top part of the structure is being created, then superimposing the $O(\log r)$ vertical decompositions at each node has two additional advantages:

(i) The space complexity of a k-level structure becomes $O(n \log^{k-1} n)$ assuming that the structure at any fixed level requires only linear space. Similarly, the preprocessing time in this case becomes $O(n \log^k n)$ assuming that the kth level structure can be constructed in $O(n \log n)$ time. In this case a single level of the partitioning is usually called a *geometric partition tree* [32].

(ii) A simplex range query can be answered using only a single-level structure: We simply find all the cells crossed by the $d + 1$ hyperplanes defining the query simplex, and the cells that are fully contained in the simplex. We report all cells of the latter kind and recurse on the cells of the former kind. The bound on the query time is established in the same manner as in [16]. The preprocessing time and storage required for the simplex range searching structure are now $O(n \log n)$ and $O(n)$, respectively.

3. Dynamizing the CSW Partitioning Scheme

In this section we show that the CSW partitioning structure can be maintained dynamically as we insert or delete points. This is a new feature of the partitioning scheme, though the method that we use is similar to older techniques for dynamizing data structures, such as those in [9] and [34]. We show that if the structure is storing n points and is using $n^{1+\varepsilon} \leq s \leq n^{d+\varepsilon}$ space, then a point can be inserted to or deleted from it in $O(s/n^{1-\varepsilon})$ amortized time. (We believe that the amortized bounds can be turned into worst-case bounds using the standard partial reconstruction techniques [34]. However, we feel that the goal is not worth the effort at this point.)

Suppose we want to insert a point p into S (deletions are completely symmetric). We search through the CSW partition tree in a top down fashion. At each node v visited, we perform one of the following two operations:

(i) Reconstruct the subtree rooted at v from scratch, including the lower-level structures stored at v. Suppose m_v is the number of points in S_v, the set of points associated with v, when v was reconstructed last time. We perform this step again if S_v has been visited during m_v/r updates since that last

reconstruction (r is the same constant parameter that controls the construction of the structure).

(ii) Insert p into the secondary structure stored at v, and into some of the children of v if v is not a leaf. If v is a leaf and there are no secondary structures, simply add p to S_v.

If the secondary structure is again a CSW partitioning tree, p is inserted there in a recursive manner. Otherwise we assume that there is an efficient procedure for inserting p into the secondary structure. As mentioned earlier, in some applications the secondary structure is constructed on a different set of objects, in which case we add the object corresponding to p to the secondary structure, instead of p itself. This will become clearer when we consider a concrete example (e.g., see Section 5.2).

Next, if v is not a leaf, p is inserted into some of the children of v. If v is in the bottom part of the structure, we dualize p to the hyperplane p^* and descend to the children of v corresponding to the simplices (of v) that intersect p^*; there are $O(r^{d-1})$ such cells. Otherwise, if v is a simplex node in the top part of the structure, for each of the $O(\log r)$ triangulations stored at v, we determine the simplex τ containing the point p and descend to the child corresponding to τ; in this case p is inserted into $O(\log r)$ "simplex grandchildren" of v.

A point can be deleted from S using the same approach. We now analyze the time spent in inserting or deleting a point. Let $\mathscr{T}_v$ denote the primary structure stored at v (i.e., the subtree rooted at v) and let m_v be the number of points in S_v when $\mathscr{T}_v$ was constructed last time. Since we reconstruct $\mathscr{T}_v$ after m_v/r update operations that visit v, the number n_v of points currently in S_v satisfies the following inequality:

$$m_v\left(1-\frac{1}{r}\right) \le n_v \le m_v\left(1+\frac{1}{r}\right). \tag{3.1}$$

Moreover, $\mathscr{T}_v$ (including the secondary substructures) can be reconstructed in time $O(s_v^{1+\varepsilon})$, where s_v is the storage allocated to v, so we can charge $O(rs_v^{1+\varepsilon}/m_v) = O(s_v^{1+\varepsilon}/n_v)$ time to each update operation that visits v to account for the time spent in reconstructing $\mathscr{T}_v$.

We bound the update time in two steps. First we consider the case where v is in the bottom part of the structure, i.e., $m_v \le \sigma$. In this case $s_v = O(n_v^{d+\varepsilon})$ and we visit at most $c_1 r^{d-1}$ children of v for some fixed constant c_1. Moreover, the time charged to each update operation for reconstructing the structure is $O(s_v^{d+\varepsilon}/n_v) = O(n_v^{d-1+\varepsilon})$. Let $U(n_v)$ be the amortized time spent in updating $\mathscr{T}_v$, including the secondary structure; then

$$U(n_v) \le \sum_{i=1}^{c_1 r^{d-1}} U(n_i) + U_s(n_v) + O(n_v^{d-1+\varepsilon}) + O(r^d),$$

where $U_s(n_v)$ is the time spent in updating the secondary structure and n_i is the number of points in the ith child of v. The last term accounts for the time spent in

finding the children of v that we need to visit. Since the number of points at any child of v after the last construction of $\mathcal{T}_v$ is at most $(cm_v/r)\log r$ and we add at most m_v/r points before reconstructing $\mathcal{T}_v$, $n_i \le ((c+1)m_v\ r)\log r$. If we assume inductively that $U_s(n_v) = O(n_v^{d-1+\varepsilon})$, then the solution of the above recurrence is easily seen to be $U(n_v) = O(n_v^{d-1+\varepsilon}) = O(n_v^{\varepsilon}\sigma^{d-1})$, because $n_v \le (1+1/r)\sigma$.

We next consider a simplex node that is in the top part of the structure. Recall that if v is in the top part of the structure, then, for each triangulation child of v, we descend to just one of its children. In this case the time charged to each update operation is $O(rs_v^{1+\varepsilon}/m_v)$. It is easy to check that $s_v = O(n_v\sigma^{d-1+\varepsilon})$, hence the above charge is $O(n_v^{\varepsilon'}\sigma^{d-1})$, where $\varepsilon' = \varepsilon'(r) > 0$ is another positive constant that tends to 0 as r increases. Let $U(n_v)$, $U_s(n_v)$ denote the same amortized times as above; then we have

$$U(n_v) \le \sum_{i=1}^{c_2 \log r} U(n_i) + U_s(n_v) + O(n_v^{\varepsilon'}\sigma^{d-1}) + O(r^d \log r),$$

where c_2 is some constant and $n_i \le ((c+1)n/r)\log r$ is the number of points in the simplex node in which we recurse in the ith triangulation child of v. Assuming that $U_s(n_v) = O(n_v^{\varepsilon'}\sigma^{d-1})$, we can easily show that $U(n_v) = O(n_v^{\varepsilon'}\sigma^{d-1})$. Since σ^{d-1} is about s/n, it easily follows that $U(n) = O(s/n^{1-\varepsilon''})$ for another constant $\varepsilon'' = \varepsilon''(r)$ that can also be made arbitrarily small.

As for the query time, our update algorithm ensures that the number of points associated with any node of the structure does not deviate too much from what it should be in the static case. In particular, at every node v at the top part of the structure, for each hyperplane h there is at least one triangulation such that the number of points in the cells of that triangulation intersected by h is still $O((n_v/r)\log r)$. Similarly, at the bottom part of the structure, the cell containing the point dual to h intersects only $O(n_v/r)\log r)$ hyperplanes. Thus, repeating the analysis of [16], we can verify that the query time is still $O(n^{1+\varepsilon}/s^{1/d})$. In summary, we have

Theorem 3.1. *Given a set of n points in $\mathbb{R}^d$ and a parameter $n^{1+\varepsilon} \le s \le n^{d+\varepsilon}$, we can maintain the CSW partitioning structure, in $O(s/n^{1-\varepsilon})$ amortized time per update, as we insert or delete points, and can answer a half-space (or simplex, or general multilevel) range query, in time $O(n^{1+\varepsilon}/s^{1/d})$.*

An immediate corollary of this theorem is

Corollary 3.2. *Simplex range searching, in any dimension d, can be performed dynamically, using quasi-optimal resources: allowing storage $n^{1+\varepsilon} \le s \le n^{d+\varepsilon}$, preprocessing cost is $O(s^{1+\varepsilon})$, query time is $O(n^{1+\varepsilon}/s^{1/d})$, and insertion and deletion of points costs $O(s/n^{1-\varepsilon})$ amortized time per update.*

4. Spanning Trees with Low Stabbing Number

Let S be a set of n points in $\mathbb{R}^d$. Let T be a straight-edge spanning tree on S. The *stabbing number* of T is the maximum number of edges of T crossed by a

hyperplane. It has been show in [17] and [44] that any such set S has a spanning tree with stabbing number $O(n^{1-1/d})$, and that in the worst case this bound is tight. Such spanning trees have been used in several applications, including range searching [17], implicit representation of faces in arrangements [25], and ray shooting amidst arbitrary collections of segments [1]. Several algorithms have been given for constructing spanning trees with low stabbing number [3], [17], [25], [28], [44], but their time complexity is not very efficient. In the planar case the best (deterministic) algorithm obtained so far is due to Matoušek [27], and runs in time $O(n^{3/2}\log^2 n)$.

In this section we adapt the CSW scheme to obtain an $O(n\log n)$-time algorithm for constructing spanning trees with low stabbing number for points in $\mathbb{R}^d$, $d = 2, 3$. We pay a small price for this significant reduction in time—the stabbing number of the resulting tree is $O(n^{1-1/d+\varepsilon})$ instead of the optimal bound $O(n^{1-1/d})$. For the sake of simplicity we describe the algorithm for $d = 2$; with a few straightforward modifications, the same approach also works for $d = 3$. As in [3] and [28], the tree produced by our algorithm is actually a spanning path.

The algorithm follows the approach of the CSW scheme, but constructs the spanning path directly instead of the data structure produced in [16]. We begin as in [16] with σ taken to be some constant. We choose some appropriate constant parameter r, construct a set of $O(r^2)$ representative lines, and construct $O(\log r)$ planar subdivisions, of $O(r^2)$ cells each, with the property that for each line l there is at least one sparse subdivision, that is, l crosses only $O(r)$ cells of the map and only $O((n/r)\log r)$ points of S lie in those cells. As mentioned in Section 2, each map is the vertical decomposition of the arrangement of some r lines. We now take these $O(\log r)$ triangulations, and superimpose them on each other. This yields one common convex subdivision, $\mathcal{M}$, of the plane with the property that the number of points of S in the cells crossed by any line is $O((n/r)\log r)$. Moreover, the arguments of [16] imply that the overall complexity of $\mathcal{M}$ is $O(r^2\log r)$, and that the number of cells of $\mathcal{M}$ crossed by a line is $O(r\log r)$.

We now construct our desired spanning path T as follows. For each cell τ of $\mathcal{M}$ let $n_\tau = |S\cap\tau|$. We recurse within each τ to construct a spanning path T_τ of $S\cap\tau$ with low stabbing number. (If this set is empty, there is nothing to be done, and if n_τ is less than some constant, we take T_τ to be any spanning path of $S\cap\tau$.) Now we connect all the paths T_τ into the full spanning path T in any convenient manner, adding $O(r^2\log r)$ new edges.

Let $K(n)$ denote the maximum stabbing number that can arise in our construction for a spanning path on a set of n points. Then we have $K(O(1)) = O(1)$, and

$$K(n) \le \sum_{i=1}^{ar\log r} K(n_i) + br^2\log r,$$

so that $\sum_i n_i \le (cn/r)\log r$, for appropriate constants a, b, c. It is easily verified that the solution of this recurrence is $K(n) \le A_\varepsilon n^{1/2+\varepsilon}$ (the constant r is chosen as a function of ε, and the constant of proportionality A_ε depends on r, thus also on ε).

As for the running time, we spend $O(n)$ time to construct the convex subdivision

$\mathcal{M}$, to distribute the points of S among its cells, and to connect all paths T_τ into the full path. Since each subproblem has size $O((n/r)\log r)$, the total time spent is easily seen to be $O(n \log n)$.

Finally, using the dynamization technique of the CSW scheme, we can also update the spanning path in $O(\log^2 n)$ amortized time as we insert or delete a point. For the sake of completeness, we describe the algorithm in more detail.

Apart from maintaining the spanning path, we now also maintain the CSW partition tree, $\mathcal{T}$, that we use to construct the spanning path. At each node $v \in \mathcal{T}$, we implicitly maintain a spanning path Π_v of the points of S_v, whose stabbing number is $O(n_v^{1/2+\varepsilon})$. Suppose we want to insert a point p to S (deletions are symmetric). We follow the path π of $\mathcal{T}$, consisting of nodes whose cells contain p, in a bottom-up fashion and process each node v of this path as follows:

(i) If v is a leaf, add p to S_v and reconstruct the spanning path of S_v in any convenient manner.

(ii) Otherwise, let w be the child of v whose cell contains p. We add p to S_w recursively, and obtain a modified path Π_w that includes p too. We update Π_v by connecting the (possibly new) endpoints of Π_w to the former neighbors of this subpath in Π_v.

(iii) Finally, let m_v be the number of points in S_v, when $\mathcal{T}_v$ was reconstructed last time. If S_v has been visited during m_v/r updates since that last construction, we reconstruct from scratch the subtree rooted at v and also the spanning path Π_v for S_v.

Our update algorithm guarantees that if a node v had m_v points when it was constructed last time, then the number of points in the cells intersected by a line is at most $((c+1)m_v/r)\log r$. As a result, the stabbing number of the spanning path is always $O(n^{1/2+\varepsilon})$. As for the time spent in updating the tree, we reconstruct the spanning path of S_v after m_v/r update operations on S_v, and spend $O(m_v \log m_v)$ time in reconstructing it. Therefore, using the same argument as in Section 3, we get the following recurrence for the amortized update time $U(n_v)$:

$$U(n_v) \leq U\left(\frac{(c+1)n_v}{r}\log r\right) + O(\log n_v) + O(r^2 \log r),$$

whose solution is $O(\log^2 n_v)$. Hence, the spanning tree can be updated in $O(\log^2 n)$ amortized time per insert/delete.

A similar technique applies in three dimensions, using the bounds established in [16] on the complexity of the partition obtained by superimposing vertically decomposed arrangements of planes in 3-space. We omit the details, which are rather straightforward. In summary, we have shown

Theorem 4.1. *Given a set S of n points in $\mathbb{R}^d$, for $d = 2, 3$, we can construct, in time $O(n \log n)$, a spanning path T on S with stabbing number $O(n^{1-1/d+\varepsilon})$. Moreover, the path can be updated, as points are being inserted into or deleted from S, in $O(\log^2 n)$ amortized time per update.*

Remarks 4.2. (i) The recurrence on $K(n)$ also applies to the time needed to find all the edges of T crossed by a query line or plane, hence this time is also $O(n^{1-1/d+\varepsilon})$ for $d = 2, 3$.

(ii) If we choose $r = n^{\varepsilon}$ in the above algorithm, the stabbing number $K(n)$ improves to $O(n^{1-1/d}(\log n)^{c_\varepsilon \log\log n})$, where c_ε is a constant depending on the value of ε. However, the running time now becomes $O(n^{1+\varepsilon})$.

(iii) This algorithm does not extend to $d > 3$ because it crucially depends on each node of the structure storing only a single partition, which is obtained by superimposing $O(\log r)$ different partitions. As already mentioned, in higher dimensions we do not have sharp bounds for the complexity of the map resulting by such a superposition. This is the only missing ingredient for extending our algorithm to higher dimensions.

5. Ray Shooting in the Plane

The problem studied in this section is one of the basic problems in computational geometry. Let $\mathcal{G} = \{e_1, \ldots, e_n\}$ be a collection of n line segments in the plane. We wish to preprocess $\mathcal{G}$ so that, given any query ray ρ, we can efficiently compute the first inersection, if any, of ρ with the segments in $\mathcal{G}$. The problem has been studied by Chazelle and Guibas [15] for the special case where the segments of $\mathcal{G}$ form the boundary of a simple polygon (see also [12]). A solution for the general case has been given by Agarwal [1]; it uses $O(n^{3/2}\log^{4.33} n)$ preprocessing, $O(n\log^4 n)$ storage, and $O(\sqrt{n\alpha(n)}\log^2 n)$ query time. This has been slightly improved by Bar Yehuda and Fogel [8]; their solution takes $O((n\alpha(n))^{3/2})$ preprocessing, $O(n\log^2 n)$ storage, and $O(\sqrt{n\alpha(n)}\log n)$ query time (the bounds can be slightly improved if the segments are nonintersecting). The recent results of Matoušek [27] and of Cheng and Janardan [19] also lead to improved performance.

Although no similar lower bounds are known for the problem, it is conjectured that the above technique is quasi-optimal in terms of query time, assuming quasi-linear storage. Nevertheless, the preprocessing cost is rather high. Using alternative techniques, Guibas *et al.* [26] have obtained a solution that requires only close to linear preprocessing time and storage, but the bound obtained for the query time is worse—only about $O(n^{2/3})$ (the technique of Dobkin and Edelsbrunner [22] mentioned above can also be adapted to yield similar results). An alternative technique, which also gives suboptimal query time, has been given by Overmars *et al.* [35]. It is based on storing the given segments in a partition tree constructed on their endpoints.

In this section we combine the better of the two worlds, obtaining a solution that is efficient both in terms of preprocessing cost and of query time—it uses $O(n\log^2 n)$ preprocessing time and storage, and $O(n^{1/2+\varepsilon})$ query time. As in the previous section, we lose slightly in terms of query time, but gain considerably in terms of preprocessing. In addition, our technique allows space/query-time tradeoff and can be dynamized efficiently.

We first present an algorithm for ray shooting amidst a collection of lines. We

need it as a subroutine to the general algorithm which is given next. In Section 5.2 we consider ray shooting among segments and finally, in Section 5.3, we explain how to modify our algorithm to report or to count the segments intersected by a query segment. We also discuss the issues of space/query-time tradeoff and of dynamization.

5.1. *Ray Shooting Among Lines*

In this section we assume that the elements of $\mathcal{G}$ are full lines. Dualize each line of $\mathcal{G}$ to a point, resulting in a set $\mathcal{G}^*$ of n points. Construct a single level CSW partitioning structure on $\mathcal{G}^*$ with $\sigma = O(1)$, as described in Section 2. For a node v of the tree, let $\mathcal{G}_v$ be the set of lines corresponding to the points of $\mathcal{G}_v^*$. We construct, as a secondary structure, the lower and the upper envelopes of $\mathcal{G}_v$, that is, the faces in the arrangement of $\mathcal{G}_v$ that lie respectively above and below all the lines of $\mathcal{G}_v$. If we have already computed the lower and upper envelopes at all the children of a node v, then the lower and upper envelopes of $\mathcal{G}_v$ can be computed in $O(|\mathcal{G}_v|)$ time, so the overall preprocessing time and the space required are $O(n \log n)$.

Given a query ray ρ, let s be its origin point and let l be the line supporting ρ. To answer the ray-shooting query, we query the primary structure with the half-plane lying above s^*, the line dual to s. Let $\mathcal{G}_v^*$ be a canonical subset in the output of the query, then s lies below all the lines dual to the points of $\mathcal{G}_v^*$. As a result, the first intersection point of ρ and the lines of $\mathcal{G}_v$ lies on the lower envelope of $\mathcal{G}_v$. Since the lower envelope is a convex polygon, the first intersection point can be computed, in logarithmic time, by a binary search. By repeating this step for all canonical subsets, we can find the first intersection point of ρ and the lines of $\mathcal{G}$ that lie above s. We now repeat the same procedure but query with the half-plane lying below s^*, and choose the intersection point that lies nearest to s.

The correctness of the algorithm is obvious. As for the query time, we spend logarithmic time at each canonical subset, and, since the query output consists of $O(n^{1/2+\varepsilon})$ canonical subsets, the query time is $O(n^{1/2+\varepsilon})$.

Theorem 5.1. *Given a collection of n lines, we can preprocess it, in time $O(n \log n)$, into a data structure of size $O(n \log n)$, so that a ray-shooting query can be answered in $O(n^{1/2+\varepsilon})$ time.*

Remark 5.2. The above technique can also be used to compute efficiently the face in the arrangement of $\mathcal{G}$ that contains a query point; see [25] for more details.

Since the CSW scheme admits space/query-time tradeoff, we can reduce the query time by allowing more space, as follows: We choose an appropriate value of σ and construct the corresponding CSW structure. If a node v is in the bottom part of the structure, then, for each child w of v (corresponding to a triangle τ of the subdivision stored at v), we store the lower envelope of the lines (dual to the points of S_v^*) lying above τ and the upper envelope of the lines lying below τ.

Answering a query is done as above. Following the analysis of the CSW scheme, we conclude that if we allow s storage, we can preprocess the lines in $O(s^{1+\varepsilon})$ time so that a ray-shooting query can be answered in $O(n^{1+\varepsilon}/s^{1/2})$ time.

Finally, since the upper and lower envelopes of a set of n lines can be maintained dynamically in $O(\log^2 n)$ time per insert/delete [33], Theorem 3.1 implies

Theorem 5.3. *Given a collection $\mathcal{G}$ of n lines and a storage parameter $n^{1+\varepsilon} \leq s \leq n^{2+\varepsilon}$, we can preprocess $\mathcal{G}$, in time $O(s^{1+\varepsilon})$, into a data structure of size s, so that a ray-shooting query can be answered in time $O(n^{1+\varepsilon}/\sqrt{s})$. Moreover, lines can be inserted to or deleted from the structure in amortized time $O(s/n^{1-\varepsilon})$ per update.*

5.2. *Ray Shooting Among Segments*

In this section we present two algorithms for ray shooting among a collection of (possibly intersecting) segments. The first algorithm constructs a standard multi-level CSW structure, while the second algorithm applies the CSW scheme in a somewhat different manner. In particular, the segments on which the secondary structure is constructed at each node v are not necessarily in 1–1 correspondence with the points of the primary structure associated with v; see below for details. Moreover, although both algorithms reduce the problem to ray shooting among lines, the basic difference is the way in which the reduction works. The first algorithm uses the observation that if the left and right endpoints of a segment e lie on opposite sides of the line containing the ray ρ, then ρ intersects e if and only if it intersects the line containing e. The second algorithm, on the other hand, is based on the fact that if the starting point of ρ is in a triangle τ, then ρ hits a segment e, whose endpoints lie outside τ, at a point inside τ, if and only if ρ hits the line containing the segment e at such an interior point. The advantage of the second algorithm is that if we choose $\sigma = O(1)$, its preprocessing time and space requirements are better than those of the first algorithm by a factor of $\log n$. Another advantage of the second algorithm is that it can be extended to a collection of nonintersecting Jordan arcs (see [4]).

5.2.1. First Solution. Let L be the set of the left endpoints of the segments in $\mathcal{G}$. Preprocess L for half-plane range queries, as described in Section 2, allowing quasi-linear storage. We next take each of the canonical subsets of L that are produced in the primary data structure and process it further as follows: Let L' be such a subset, and let R' be the set of right endpoints of the segments whose left endpoints are in L'. We preprocess R' for half-plane range queries, and attach the resulting data structure to L' as a secondary structure. This is not the end yet: We next take each secondary canonical subset R'', extend its corresponding segments to full lines, and process them for ray shooting using the technique described in the preceding section. Note that, altogether, we have constructed a 4-level structure. By Theorem 5.1, the auxiliary structure of a second-level canonical subset of m points requires $O(m \log m)$ space and preprocessing. The overall storage and preprocessing are thus $O(n \log^3 n)$ (see Remark 2.1(i)).

Given a ray ρ, let l be the line supporting ρ, and let s be the origin of ρ. We first query the primary data structure with the half-plane lying above l, then query the secondary data structure of each canonical subset of the output with the half-plane lying below l. We next query again the first-level structure with the half-plane lying below l, and the secondary structures with the half-plane lying above l. Let $\mathscr{G}''$ be a secondary canonical subset in the output of the query so far. We know that l hits all segments in $\mathscr{G}''$ because their endpoints lie on different sides of l. Thus extending these segments to full lines will not change the first one to be hit by ρ. We thus continue at this level with the ray-shooting procedure described in the preceding section, applied to each of these subsets $\mathscr{G}''$. We collect the outputs of all these "subqueries," and choose the one nearest to s as the final output.

Concerning complexity, we pay logarithmic time at each fourth-level structure, and the number of such structures that are retrieved during the query processing is $O(n^{1/2+\varepsilon})$, as follows from the discussion in Section 2. The resulting query time is thus $O(n^{1/2+\varepsilon})$. Hence, we can conclude

Theorem 5.4. *Given a set of n arbitrary segments in the plane, we can preprocess it, in time $O(n \log^3 n)$, into a data structure of size $O(n \log^3 n)$, so that, for any query ray ρ, the first segment that ρ hits can be found in time $O(n^{1/2+\varepsilon})$.*

5.2.2. Second Solution. We now describe the second algorithm for ray shooting among segments. Let S be the set of endpoints of segments of $\mathscr{G}$. We construct a one-level quasi-linear CSW structure on S. Recall that each of its nodes v is associated with a subset $S_v \subseteq S$ and a cell Δ_v. We also associate with v a set of segments $\mathscr{G}_v$. A segment e belongs to $\mathscr{G}_v$ if at least one of its endpoints is in S_w, where w is the parent of v, and Δ_v intersects e but does not contain any of its endpoints. We extend the segments of $\mathscr{G}_v$ to full lines and preprocess them for ray shooting as described in Section 5.1. It can be checked that $\sum_{v \in \mathscr{T}} |\mathscr{G}_v| = O(n \log n)$. Since the secondary structure of v requires $O(|\mathscr{G}_v| \log |\mathscr{G}_v|)$ space and preprocessing, the overall storage and preprocessing time are $O(n \log^2 n)$.

To answer a ray-shooting query we trace the query ray ρ through the subdivision, M, stored at the root, as follows. Let Δ_v be the cell of M containing the origin point of ρ. We use the substructure stored at the child v to determine whether ρ intersects a segment of $\mathscr{G}$ inside Δ_v, and, if the answer is positive, we compute the first intersection point and return it as the output. If not, we visit the next cell Δ_z of M intersected by ρ and move the origin point to the first intersection point of ρ and Δ_z.

The first intersection point of ρ and $\mathscr{G}$ lying inside Δ_v is computed in two steps. The segments of $\mathscr{G}$ that intersect Δ_v can be classified as either

(i) "long" segments whose endpoints lie outside Δ_v, or
(ii) "short" segments, one of whose endpoints lies in Δ_v.

We compute the first segment of each class hit by ρ and then choose the one that lies nearer to the origin of ρ.

The first short segment hit by ρ inside Δ_v is computed by searching through

the structure of v recursively. To compute the first long segment hit by ρ, we exploit the following observation: ρ hits a long segment e inside Δ_v if and only if it intersects the line containing e inside Δ_v. Thus, using the secondary structure stored at v, we can determine the first long segment hit by ρ inside Δ_v. (The secondary structure stores only segments that have at least one endpoint in the cell of the parent of v; nevertheless, at the root all long segments in Δ_v have this property, and the property continues to hold, by construction, at all other nodes as well.) Since we spend $O(n_v^{1/2+\varepsilon})$ time for finding the first long segment hit by ρ and visit $O(r \log r)$ cells of the subdivision stored at the root, we get the following recurrence for the query time

$$Q(n) \leq \sum_{i=1}^{cr \log r} Q(n_i) + O(n^{1/2+\varepsilon} r \log r),$$

where c is some constant and n_i is the number of short segments in the ith cell crossed by ρ. It follows that $\sum_{i=1}^{cr \log r} n_i = O((n/r) \log r)$. Hence, the solution of the above recurrence is easily shown to be $O(n^{1/2+\varepsilon})$. We thus obtain

Theorem 5.5. *Given a collection of n segments in the plane, we can preprocess it, in time $O(n \log^2 n)$, into a data structure of size $O(n \log^2 n)$, so that a ray-shooting query can be answered in $O(n^{1/2+\varepsilon})$ time.*

Remark 5.6. If $\mathcal{G}$ is a collection of nonintersecting segments, then we can slightly improve the structure, as follows. At each node v we store the following secondary structure. We clip the segments of $\mathcal{G}_v$ to within Δ_v and process their arrangement for fast point location. The first segment of $\mathcal{G}_v$ hit by ρ inside Δ_v can then be computed in $O(\log n_v)$ time by locating the origin of ρ among the clipped segments and then by testing a constant number of segments for possible intersection with ρ; see [35] for details. Since the secondary structure requires only $O(n_v)$ storage and $O(n_v \log n_v)$ preprocessing, the overall storage and preprocessing in this case become $O(n \log n)$ and $O(n \log^2 n)$, respectively.

5.2.3. Space/Query-Time Tradeoff. Since the first solution basically follows the CSW scheme at each level, Theorem 5.3 implies that, if we allow s storage for the structure, the query time can be improved to $O(n^{1+\varepsilon}/\sqrt{s})$. Furthermore, the structure supports insertions and deletions of segments in amortized time $O(s/n^{1-\varepsilon})$ per update. To achieve similar enhancements of the second solution, we have to construct the bottom part of the structure too in a somewhat different way. Let v be a node of the CSW structure such that $|S_v| \leq \sigma$. Let $\mathcal{G}_v$ denote the set of segments, one of whose endpoints is in S_v. We dualize the segments of $\mathcal{G}_v$ to double wedges; let $\mathcal{G}_v^*$ denote the set of resulting double wedges. Using the algorithm of [29], we choose a subset of r double wedges of $\mathcal{G}_v^*$ and triangulate their arrangement so that each triangle intersects the boundary of at most $O((|\mathcal{G}_v|/r) \log r)$ double wedges. For each triangle τ, we create a child w associated with τ and recursively construct the structure on the double wedges whose boundaries intersect τ. We also take the segments dual to the double wedges of $\mathcal{G}_v^*$ that contain

τ, and extend them to full lines; let H_w denote the set of these segments, and let L_w denote the set of resulting lines. We construct the arrangement $\mathscr{A}(L_w)$ and store it at w as the secondary structure.

A ray-shooting query is answered in the same way as earlier except that at each node v in the bottom part of the structure we do the following: We dualize the line l containing ρ to the point l^*, and locate it in the subdivision stored at v. Let w be the child corresponding to the cell containing l^*. If a segment $e \in \mathscr{G}_v$ intersects l, then the double wedge dual to e either intersects τ or contains τ. For the first type of segments, we recursively search through the primary structure of w. For the second type of segments, we locate the cell $f \in \mathscr{A}(L_w)$ containing the origin point s of ρ. Since l intersects all segments of H_w, the first intersection point of ρ and the segments of $\mathscr{G}_w$ is the same as that of f and ρ. However, f is convex, so the first intersection point of f and ρ can be computed, in $O(\log n)$ time, by a straightforward binary search.

Following the same analysis as in [16], the query time is easily seen to be $O(n^{1+\varepsilon}/\sqrt{s})$. Finally, the arrangement of L_w can be updated in $O(|L_w| \log |L_w|)$ time as we insert or delete a line, see [18]. Hence, the overall structure can be maintained dynamically in $O(s/n^{1-\varepsilon})$ amortized time per update. We therefore have

Theorem 5.7. *Given a collection of n segments and a storage parameter $n^{1+\varepsilon} \leq s \leq n^{2+\varepsilon}$, we can preprocess the collection, in $O(s^{1+\varepsilon})$ time, into a data structure of size s, so that a ray-shooting query can be answered in $O(n^{1+\varepsilon}/\sqrt{s})$ time. Moreover, the structure supports insertions and deletions of segments in amortized time $O(s/n^{1-\varepsilon})$ per update.*

5.3. *Intersection Queries*

Consider the following problem: "Given a set $\mathscr{G}$ of n segments in the plane, preprocess it into a data structure, so that the segments of $\mathscr{G}$ intersected by a query segment can be reported efficiently." We show that our ray-shooting structure can be modified to handle this problem. We only describe how to modify the first solution; the second solution can also be modified in a similar manner.

We construct the first two levels of the structure as in Section 5.2.1; for each canonical subset of the second-level structure, we extend the corresponding set of segments to full lines. We dualize these lines to points and preprocess them for triangle range searching. We thus obtain a three-level structure.

To answer a query, we proceed in the same way as for ray-shooting queries. Recall that a segment g of a secondary canonical subset $\mathscr{G}''$ in the output of the query intersects the query segment e if and only if the line l containing g intersects e. In dual setting it is the same as saying that the double wedge dual to e contains the point dual to l. Therefore, using the third-level structure, we can report (or count the number of) all segments of $\mathscr{G}''$ intersecting e. Once again, we can derive space/query-time tradeoff, and we can maintain the structure dynamically. We thus obtain

Theorem 5.8. *Given a collection $\mathcal{G}$ of n segments in the plane and a parameter $n^{1+\varepsilon} \leq s \leq n^{2+\varepsilon}$, we can preprocess $\mathcal{G}$ into a data structure of size s, in time $O(s^{1+\varepsilon})$, so that we can report all k segments of $\mathcal{G}$ intersecting a query segment in time $O(n^{1+\varepsilon}/s^{1/2} + k)$, or can count the number of such segments in time $O(n^{1+\varepsilon}/s^{1/2})$. As above, insertions and deletions of segments can be performed in amortized time $O(s/n^{1-\varepsilon})$ per update.*

6. Efficient Output-Sensitive Hidden Surface Removal

In a recent series of papers, Overmars and Sharir [35]–[38], [41] have studied the problem of output-sensitive hidded surface removal. In this problem we are given a collection Δ of n nonintersecting triangles in 3-space, and we want to compute the portions of these triangles that are visible from some viewing point z. In order to make our algorithms work, we have to assume that the triangles have a known *depth order* with respect to z (see [41] for more details). For simplicity, we assume that the triangles in Δ are all horizontal, and that the viewing point is at $z = -\infty$. In this case the xy-projections of the visible portions form a planar map $\mathcal{M}$, known as the visibility map of the triangles. If the combinatorial complexity (say, the number of vertices) of $\mathcal{M}$ is k, the goal is to compute $\mathcal{M}$ in time that depends on k (and on n) so that when k is small the algorithm runs faster. Ideally, we would like the time complexity to be something like $O(f(n) + kg(n))$, where $f(n)$ is subquadratic and $g(n)$ is small, say polylogarithmic in n. This however appears to be difficult to achieve in general. Overmars and Sharir initially gave two output-sensitive solutions, the first [41] is very simple and its running time is $O(n\sqrt{k}\log n)$. An improved algorithm is given in [37]. It is a fairly complicated algorithm but its time complexity, $O(n^{4/3}\log^{2/3} n + k^{3/5}n^{4/5+\varepsilon})$, is an improvement over the first algorithm.[3] In this section we describe two algorithms for this problem. In our first algorithm we follow the general approach of the second algorithm of [37], and plug into it our ray-shooting technique described in the previous section. This results in a further improved algorithm whose running time is $O(n^{2/3+2\varepsilon}k^{2/3-\varepsilon} + n^{4/3}\log^{2/3} n)$. We then present a different algorithm, based on the dynamic version of our ray-shooting technique, whose running time is $O(n^{2/3-\varepsilon}k^{2/3+\varepsilon} + n^{1+\varepsilon})$. This algorithm is significantly simpler than the previous algorithm.

6.1. *First Algorithm*

To explain our improvement we give a very brief sketch of the technique of [37], and refer the reader to that paper for more details. Let the given triangles be $\Delta_1, \ldots, \Delta_n$, in the order of increasing height.

[3] The bound just stated is a slight improvement over the bound given in [37], obtained by applying Matoušek's partitioning algorithm [29].

- We first find all the vertices of the triangles that are visible from $z = -\infty$. This can be accomplished, for instance, by a batching technique of Agarwal [3] in time $O(n^{4/3} \log^{2/3} n)$.
- Project all triangles vertically on the xy plane to obtain a planar arrangement of overlapping triangles.
- Choose some parameter r in an appropriate manner (detailed below), and apply the partitioning algorithm of Agarwal [2] or of Matoušek [29] to obtain a decomposition of the plane into $O(r^2)$ triangular cells, each of which meets only $O(n/r)$ projected triangle edges. This can be done in time $O(nr)$ [29].
- For each cell τ we compute separately the portion of the map $\mathcal{M}$ clipped to within τ. We first find all triangles whose projections fully contain τ; the lowest of these triangles serves as a "background" triangle for τ and all triangles lying higher than it can be ignored at τ.
- We next take the set $K^{(\tau)}$ of projected triangles that have an edge that crosses τ (and lie lower than the background triangle), and represent it as a minimum-height binary tree B storing the triangles of that set in its leaves in the order of increasing height. For each node ξ of B we take the set K_ξ of triangles stored at the leaves of the subtree rooted at ξ, and preprocess the edges of their projections (clipped to within τ) for fast ray shooting as in the preceding section.
- Next we compute the visibility map over the edges of the cell τ. This is an easy one-dimensional lower envelope calculation, that can be performed in time $O((n/r) \log(n/r))$.
- We now compute the "inner" vertices of $\mathcal{M}$ over τ by performing ray shooting along projections of visible edges in $\mathcal{M}$, starting from the boundary of τ, $\partial\tau$, and going inward. These shootings are somewhat involved (see [37] for details)—a shooting along the projection of an edge e of some triangle Δ needs to be performed only either in the collection of the projections of the triangles that lie below Δ or in the collection of the projections of the triangles that lie between Δ and the triangle lying directly above e. However, each such collection is the disjoint union of $O(\log n)$ subtrees of our tree B, and each such subtree has been processed for fast ray shooting, so we perform these shootings separately, and the nearest of all the outputs is the answer to our query.
- There are several additional technical issues that have to be addressed. For instance, there are "spurious" ray shootings that do not discover any vertex of $\mathcal{M}$ but shoot from one side of τ to the other. Another issue is that the value of r is a function of the output size k, which is not known to us in advance. However, all these problems are handled efficiently in [37], and we tackle them in the same manner; we refer the reader to that paper for more details.

Our improved solution follows the same overall approach just outlined. The only difference is that we use the planar ray-shooting technique described in the previous section—as noted, this is the only known technique that achieves both

quasi-linear preprocessing time and quasi-optimal query time, and both characteristics are useful in the above hidden surface removal algorithm.

Following the analysis given in [37] and plugging the complexity bounds for our ray-shooting technique, the overall time used by the algorithm is easily seen to be

$$O\left(n^{4/3}\log^{2/3} n + n^{1+\varepsilon}r^{1-\varepsilon} + k\left(\frac{n}{r}\right)^{1/2+\varepsilon}\right).$$

Hence, choosing $\lceil r = k^{2/3}/n^{1/3}\rceil$ (in the same manner as in [37]), we obtain

Theorem 6.1. *Hidden surface removal in a collection of n horizontal triangles in 3-space, viewed from $z = -\infty$ (or, more generally, of n nonintersecting triangles with a known depth order with respect to an arbitrary viewing point), can be performed in time $O(n^{4/3}\log^{2/3} n + n^{2/3+2\varepsilon}k^{2/3-\varepsilon})$, where k is the size of the resulting visibility map of the triangles.*

6.2. Second Algorithm

We now present our second algorithm that is based on the dynamic ray-shooting technique developed in the preceding section. It constructs the visibility map incrementally by adding the triangles one by one in the nondecreasing order of their z-coordinates. Suppose we have computed $\mathcal{M}_i$, the visibility map of $\Delta_1, \ldots, \Delta_i$ and we are about to add Δ_{i+1}. Since we are adding the triangles in the increasing order of their heights, Δ_{i+1} cannot obscure any portion of $\Delta_1, \ldots, \Delta_i$, so Δ_{i+1} can appear only in $\mathbb{R}^2 - U_i$, where U_i is the union of the projections of $\Delta_1, \ldots, \Delta_i$. We refer to the boundary of U_i as the *contour* of $\mathcal{M}_i$, and denote it by C_i. Let ζ_i denote the number of segments forming C_i, and let $\zeta = \max_i \zeta_i$. (Note that we always have $\zeta \leq k$.) For the sake of convenience, we denote by Δ_i both the triangle and its xy-projection.

The processing of Δ_{i+1} consists of computing $\mathcal{M}_{i+1}$ and C_{i+1} from $\mathcal{M}_i$ and C_i, respectively. The new vertices of $\mathcal{M}_{i+1}$, that is, the ones that were not in $\mathcal{M}_i$, are either the vertices of Δ_{i+1} or the intersection points of C_i and $\partial\Delta_{i+1}$. Moreover, every intersection point of C_i and $\partial\Delta_{i+1}$ is a vertex of $\mathcal{M}_{i+1}$. Once we know the new vertices of $\mathcal{M}_{i+1}$, its new edges can be computed in a straightforward manner. We maintain the following three data structures on C_i:

- We preprocess the edges of C_i for efficient (dynamic) ray-shooting queries as described in Section 5.2.1.
- We preprocess the edges of C_i, as described in Section 5.3, so that the intersections between the edges of C_i and a query segment can be reported quickly.
- We preprocess the left endpoints of the edges of C_i for triangle range searching queries so that the left endpoints lying in Δ_{i+1} can be computed quickly.

Remark 6.2. Recall that the first two data structures are almost the same except at the third level, so we can combine them into one structure by storing two

auxiliary structures at each node of the second level. Nevertheless, for the sake of clarity, we assume that we have two separate structures.

The updating of $\mathcal{M}_i$ and C_i consists of the following three steps:

(i) *Computing the new vertices of* $\mathcal{M}_i$: For each edge e of Δ_{i+1} we compute the intersection points of e and C_i using the second data structure. These points partition e into subsegments that alternately lie inside and outside U_i. Thus if e intersects C_i, we can also easily decide which endpoints of e appear in $\mathcal{M}_{i+1}$. If e does not intersect C_i, we shoot a ray ρ from one endpoint a of e along the line containing e in the direction of e. The ray-shooting query can be answered using the first data structure. If ρ does not intersect C_i, both endpoints of e appear in $\mathcal{M}_{i+1}$, otherwise we easily determine from the first hit of ρ and C_i whether both endpoints appear in $\mathcal{M}_{i+1}$ or both do not appear.

(ii) *Deleting the edge of* $C_i - C_{i+1}$: For each edge $e \in C_i$, $e \cap \Delta_{i+1}$ does not appear in C_{i+1}, so if e intersects Δ_{i+1}, we replace e by $e - \Delta_{i+1}$; $e - \Delta_{i+1}$ may consist of two connected components. We delete e from C_i, insert the connected components of (the closure of) $e - \Delta_{i+1}$ back to C_i, and update all structures accordingly.

As for finding the edges e that intersect Δ_{i+1}, e satisfies at least one of the following two conditions: (i) e intersects $\partial\Delta_{i+1}$ or (ii) the left endpoint of e lies in Δ_{i+1}. The first type of edges have already been determined in step (i), and the second type of edges can be obtained using our third range searching data structure.

(iii) *Adding the edges of* $C_{i+1} - C_i$: Let ab be an edge of Δ_{i+1}. If ab does not intersect C_i but a appears in $\mathcal{M}_{i+1}$, then ab is a new edge of C_{i+1}. On the other hand, if $\chi_1, \chi_2, \ldots, \chi_m$ are the intersection points of C_i and ab, then either $a\chi_1, \chi_2\chi_3, \chi_4\chi_5, \ldots$ or $\chi_1\chi_2, \chi_3\chi_4, \ldots$ are new edges of C_{i+1}. We add these edges to C_{i+1} and update the data structures accordingly.

Repeating this procedure for all triangles in order, we obtain the final visibility map $\mathcal{M}$. We now analyze the time spent in adding Δ_{i+1}. Let k_{i+1} be the number of new vertices in $\mathcal{M}_{i+1}$, and let β_{i+1} be the number of edges of C_i that are fully contained in Δ_{i+1}. Obviously, the number of edges of C_i that intersect Δ_{i+1} is at most $k_{i+1} + \beta_{i+1}$.

By Theorems 5.7 and 5.8, step (i) can be executed in time $O((\zeta_i/\sigma)^{1/2+\varepsilon} + k_{i+1})$, where σ is the parameter that controls the storage size of the CSW scheme, whose value will be chosen later. In step (ii) at most $k_{i+1} + \beta_{i+1}$ edges are deleted and at most $2k_{i+1}$ edges are inserted, so the number of updates performed on each structure is at most $\beta_{i+1} + 3k_{i+1}$. As a result, step (ii) can be accomplished in (amortized) time $O((\beta_{i+1} + k_{i+1})\zeta^{\varepsilon}\sigma)$. Finally, we insert at most $k_{i+1} + 3$ edges in step (iii), so this step requires $O(k_{i+1}\zeta^{\varepsilon}\sigma)$ (amortized) time.

Hence C_{i+1} can be computed, and the various structures can be updated after inserting Δ_{i+1}, in time

$$O\left(\left(\frac{\zeta_i}{\sigma}\right)^{1/2+\varepsilon} + (\beta_{i+1} + k_{i+1})\zeta^{\varepsilon}\sigma\right).$$

Observe that each intersection point of C_i and $\partial\Delta_{i+1}$ creates at most one new edge along C_i, that the edges of C_i lying completely inside Δ_{i+1} do not appear in C_{i+1}, and that at most $k_{i+1}+3$ edges of C_{i+1} occur along $\partial\Delta_{i+1}$. Hence

$$\zeta_{i+1}-\zeta_i \le 2k_{i+1}-\beta_{i+1}+3 \quad \text{or} \quad \beta_{i+1} \le \zeta_i-\zeta_{i+1}+2k_{i+1}+3.$$

The total time spent in adding Δ_{i+1} is thus

$$O\left(\left(\frac{\zeta_i}{\sigma}\right)^{1/2+\varepsilon}+(k_{i+1}+\zeta_i-\zeta_{i+1})\zeta^{\varepsilon}\sigma\right).$$

Thus, $T(n)$, the overall time spent in computing $\mathcal{M}$, is

$$\begin{aligned}T(n) &= \sum_{i=0}^{n-1} O\left(\left(\frac{\zeta_i}{\sigma}\right)^{1/2+\varepsilon}+(k_{i+1}+\zeta_i-\zeta_{i+1})\zeta^{\varepsilon}\sigma\right)\\ &= O\left(n\left(\frac{\zeta}{\sigma}\right)^{1/2+\varepsilon}+\left(\sum_{i=1}^{n}k_i+\sum_{i=0}^{n-1}(\zeta_i-\zeta_{i+1})\right)\zeta^{\varepsilon}\sigma\right)\\ &= O\left(n\left(\frac{\zeta}{\sigma}\right)^{1/2+\varepsilon}+k\zeta^{\varepsilon}\sigma\right)\end{aligned}$$

because $\sum_{i=1}^{n} k_i = k$. We now choose $\sigma = \lceil n^{2/3}\zeta^{1/3}/k^{2/3}\rceil$, so that

$$T(n) = O(n^{2/3+\varepsilon'}k^{1/3}\zeta^{1/3}+k\zeta^{\varepsilon}+n^{1+\varepsilon'}), \tag{6.1}$$

where ε' is different from ε but still arbitrarily small positive constant. Since $\zeta \le k$ and $k = O(n^2)$, we obtain $T(n) = O(n^{2/3+\varepsilon'}k^{2/3}+n^{1+\varepsilon'})$.

Choosing the value of σ is somewhat tricky, since we do not know in advance the value of k. As in [37], we guess the value of k (and thus of σ). We start with some appropriate value of k, say $n^{1/2}$. Let k_{cur} be the current estimated value of k. We compute the value of σ using k_{cur}. If, at any stage, k becomes larger than k_{cur}, we double the value of k_{cur}, recompute the value of σ, and rerun the algorithm. The overall time complexity of the algorithm is easily seen to be asymptotically the same as in (6.1). We thus conclude generalizing as in Theorem 6.1:

Theorem 6.3. *Given a set of n nonintersecting triangles in $\mathbb{R}^3$ with a known depth order with respect to a given viewing point, their visibility map can be computed in time $O(n^{2/3+\varepsilon}k^{2/3}+n^{1+\varepsilon})$, where k is the output size.*

7. Ray Shooting in Three Dimensions

Let $\mathcal{T} = \{T_1, \ldots, T_n\}$ be a collection of n triangles in $\mathbb{R}^3$. We wish to preprocess it into a data structure that supports fast *ray-shooting* queries, where each such query asks for the first triangle, if any, intersected by a query ray. Note that we

do not put any restriction on the query ray—it can emanate from any point in any direction; nor do we require any special structure of the given collection of triangles—our technique can even handle intersecting triangles. We show that, allowing s storage, we can preprocess $\mathscr{T}$ in time $O(s^{1+\varepsilon})$ into a data structure of size s, so that a ray-shooting query can be answered in time $O(n^{16/15+\varepsilon}/s^{4/15})$.

Let $e_1, \ldots, e_{3n}$ be the edges of the triangles in $\mathscr{T}$, and let l_j denote the line containing e_j, for $j = 1, \ldots, 3n$. Let ρ be a query ray, and let λ be the line containing it. Let T be a triangle of $\mathscr{T}$, let e_1, e_2, e_3 denote its edges, and suppose that they lie respectively on the lines l_1, l_2, and l_3. We orient the lines l_i so that T lies to the right of each of them. The line λ intersects T if and only if it has the same *relative orientation* with respect to the three lines l_i. The relative orientation of two oriented lines l, λ in 3-space is defined to be the orientation of the simplex $abcd$, where $a, b \in l$, $c, d \in \lambda$, l is oriented from a to b, and λ is oriented from c to d. Equivalently, it is also the sign of the inner product between the two vectors in 6-space representing the *Plücker's coordinates* of the two lines. More details concerning Plücker's coordinates and relative orientations can be found in [14], [42], and [43].

These observations suggest the following approach. We describe a solution that uses quasi-linear storage; the general case can be handled using the tradeoff properties of the CSW scheme, but we derive below a somewhat improved tradeoff. We first take one edge from each triangle $T \in \mathscr{T}$, and form the collection $\mathscr{L}_1$ of the lines containing these edges and oriented as above. We map each line l in $\mathscr{L}_1$ to its Plücker point $\pi(l)$ in projective 5-space (see [14] for more details). We apply the CSW partitioning scheme to the resulting collection, $\mathscr{P}_1$, of points in 5-space. For each canonical subset of Plücker points, or equivalently of lines in 3-space, we take the corresponding subset of triangles of $\mathscr{T}$, pick a second edge in each, form the corresponding set of oriented lines containing these edges, transform them into Plücker points, and apply the CSW scheme to these points. We attach the resulting data structure as a secondary substructure at the corresponding node of the primary structure. We then repeat the same process once more for each canonical subset in any secondary substructure, where now we use the third edge of each corresponding triangle of $\mathscr{T}$.

We have thus obtained a 3-level structure, which uses $O(n^{1+\varepsilon})$ space and takes $O(n^{1+\varepsilon})$ time to construct. We use this structure to (partially) process a ray-shooting query as follows. We map the line λ containing our query ray ρ into its *Plücker's hyperplane* $\varpi(\lambda)$ in projective 5-space (again, see [14] for details). We query with this hyperplane our structure to obtain all triples of lines in 3-space, each triple containing the edges of a single triangle in $\mathscr{T}$, so that the three lines in a triple have the same relative orientation with respect to λ. As explained in [14], any such triple corresponds to a triple of Plücker points all lying on the same side of $\varpi(\lambda)$; those triples are easily obtained using our structure. In other words, the query output consists of all triangles stabbed by λ. Since we are in 5-space, the properties of the CSW scheme imply that the query time is $O(n^{4/5+\varepsilon})$, and that the output of our query consists of $O(n^{4/5+\varepsilon})$ pairwise disjoint canonical subsets of triangles.

We next have to shoot along ρ within each of these canonical subsets and

report the hit point nearest to the origin of ρ. Let $\mathscr{T}'$ be one of these canonical subsets. Extend each triangle in $\mathscr{T}'$ to a full plane, and preprocess the resulting collection of planes for fast ray shooting. This can be done as follows. Dualize the collection of planes to obtain a set of points in 3-space. Apply the CSW partitioning scheme to this set. Take the origin point a of ρ, dualize it to a plane a^*, and query the resulting data structure with a^*. This gives us a collection of $O(n^{2/3+\varepsilon})$ canonical subsets of dual points, each lying either fully above or fully below a^*. For each of these canonical subsets we go back to the primal space, and conclude that a lies either above all the corresponding planes, or below all of them. We can therefore precompute the upper and lower envelopes of this collection of planes, each being an unbounded convex polyhedron, and preprocess each of them for fast ray shooting, using the hierarchical representation of convex polyhedra given by Dobkin and Kirkpatrick [23], [24].

Using the properties of the CSW scheme, it is easy to verify that the overall time for processing a ray-shooting query is $O(n^{4/5+\varepsilon})$. The preprocessing time and storage are both $O(n^{1+\varepsilon})$.

The method just presented achieves quasi-linear storage (and preprocessing). In order to obtain space/query-time tradeoff, we use the recent result of [7], which implies that the overall combinatorial complexity of all the cells in an arrangement of r hyperplanes in 5-space, which are intersected by the *Plücker surface* (i.e., the quadratic surface is the image of the space of all lines in 3-space under the Plücker transformation), is $O(r^4 \log r)$.

Consider a node v in the bottom part of one of the first three levels of the CSW structure constructed above. Normally, we need to store at v a complete arrangement of r dual hyperplanes. However, since any query point, being the Plücker image of some line in 3-space, lies on the Plücker surface, it suffices to store at v only (the triangulation of) the cells crossed by that surface (see [39]). The total number of simplices in that portion of the arrangement is thus only $O(r^4 \log r)$. Recall that at the fourth level we apply the CSW scheme in 3-space, so a node in the bottom part of the fourth-level structure stores only $O(r^3)$ simplices. The analysis of the space/query-time tradeoff of the CSW scheme implies that the parameter σ should be chosen to be about $(s/n)^{1/3}$, rather than $(s/n)^{1/4}$, and the query time is still about $(n/\sigma)^{4/5}$. Working out the details, we easily verify that the query time becomes $O(n^{16/15+\varepsilon}/s^{4/15})$. We summarize our results in

Theorem 7.1. *Given a collection of n triangles in $\mathbb{R}^3$ and storage size s that can vary between $n^{1+\varepsilon}$ and $n^{4+\varepsilon}$, we can preprocess the collection into a data structure of size s, in time $O(s^{1+\varepsilon})$, which supports ray-shooting queries in time $O(n^{16/15+\varepsilon}/s^{4/15})$.*

The above algorithm can be modified to report (or to count the number of) all triangles intersected by a query segment e as follows. We construct the first three levels of the structure as in the ray-shooting structure. Recall that a triangle of a third-level canonical subset of the query output intersects the query segment e if and only if the plane containing the triangle intersects e. Therefore, for every third-level canonical subset, we extend its triangles to full planes, dualize them to

points, and process these points for simplex range searching (actually for double-wedge range searching). To answer a query, we dualize the segment e to the double wedge e^* in $\mathbb{R}^3$ and query the relevant fourth-level structures to report or count the points that lie in e^*. Hence, we obtain

Theorem 7.2. *Given a collection of n triangles in $\mathbb{R}^3$ and storage size s that can vary between $n^{1+\varepsilon}$ and $n^{4+\varepsilon}$, we can preprocess the collection into a data structure of size s, in time $O(s^{1+\varepsilon})$, so that all k triangles intersected by a query segment can be reported in time $O(n^{16/15+\varepsilon}/s^{4/15} + k)$, or the number of such triangles can be counted in time $O(n^{16/15+\varepsilon}/s^{4/15})$.*

Remark 7.3. Recently and independently, de Berg *et al.* [10] have obtained a somewhat different algorithm, also based on the CSW scheme, for ray shooting in 3-space, that can answer a ray-shooting query among a set of n triangles in $O(\log n)$ time using $O(n^{4+\varepsilon})$ space and preprocessing.

8. Discussion

The favorable properties of the CSW partition scheme makes it an ideal tool for various applications that are closely related to simplex range searching and that require the use of multilevel structures. This paper demonstrates the usefulness of the CSW scheme for obtaining improved solutions for several fundamental problems in computational geometry, such as constructing a spanning tree with low stabbing number, ray shooting in two and three dimensions, and output-sensitive hidden surface removal. We feel that these applications only "scratch the surface" of a large collection of problems of this sort that can benefit from the CSW scheme.

Since the original submission of the paper, there has been progress in the design of more efficient range searching techniques, which can be used instead of the CSW scheme in some of our applications [11], [30], [31]. One such improved structure is a geometric partition tree due to Matoušek [31] (see also [30]), which can answer a half-space range query in $\mathbb{R}^d$ in time $O(n^{1-1/d})$ using $O(n)$ space and $O(n \log n)$ preprocessing. Matoušek has also shown that combining this structure along with Chazelle's recent cutting algorithm [11], a d-dimensional multilevel search structure can be constructed, for any $n \le m \le n^d$, which uses $O(m \log^{c_1} m)$ storage, requires $O(m^{1+\varepsilon})$ preprocessing, and can answer a range query in time $O((n/m^{1/d}) \log^{c_2} n)$, for appropriate constants c_1, c_2 that depend on the number of levels in the multilevel structure. Using this scheme, instead of the CSW partitioning scheme, we can slightly improve the performance of most of the algorithms described in this paper. For example, we can construct a spanning tree of stabbing number $O(n^{1-1/d})$ in time $O(n^{1+\varepsilon})$, or we can replace the n^{ε} factor with a polylogarithmic factor in the bounds on the query time and the storage requirement of our ray-shooting algorithms.

There are several open problems raised by the results in this paper. One problem is to prove that our two-dimensional ray-shooting algorithms are indeed close to

optimal. Another open problem is to improve further the ray-shooting technique in 3-space. Allowing only quasi-linear storage, can we bring the query time down to $O(n^{2/3+\varepsilon})$, as in the case of planes? Recently, Agarwal and Matoušek have developed a data structure that can answer a ray-shooting query among triangles in 3-space in time $O(n^{3/4+\varepsilon})$ [6]. In another paper [5], they show that a ray-shooting query among half-planes can be answered in $O(n^{2/3+\varepsilon})$ time, but this improved technique does not extend to triangles.

Since many of the applications (here and in [16]) are important in practice, there is the issue of obtaining simplified algorithms. The CSW scheme and the algorithms that we have developed here, based on that scheme, are still too complex to be useful in practice. Can alternative practical, but still quasi-optimal, partition schemes, or solutions based on totally different ideas be obtained?

Acknowledgments

The authors wish to thank Jirka Matoušek and two anonymous referees for several useful comments.

References

[1] P. K. Agarwal, Ray shooting and other applications of spanning trees with low stabbing number, *SIAM J. Comput.* **21** (1992), in press.

[2] P. K. Agarwal, Partitioning arrangements of lines, I: An efficient deterministic algorithm, *Discrete Comput. Geom.* **5** (1990), 449–483.

[3] P. K. Agarwal, Partitioning arrangements of lines, II: Applications, *Discrete Comput. Geom.* **5** (1990), 533–573.

[4] P. K. Agarwal, M. van Kreveld, and M. Overmars, Searching and storing curved objects, *Proc. 7th ACM Symp. on Computational Geometry*, 1991, pp. 41–50. (Also to appear in *J. Algorithms*.)

[5] P. K. Agarwal and J. Matoušek, Ray shooting and parametric search, *Proc. 24th ACM Symp. on Theory of Computing*, 1992.

[6] P. K. Agarwal and J. Matoušek, Range searching with non-linear objects, in preparation, 1991.

[7] B. Aronov and M. Sharir, On the zone of a surface in a hyperplane arrangement, *Proc. 2nd Workshop on Algorithms and Data Structures*, Lecture Notes in Computer Science, Vol. 519, Springer-Verlag, Berlin, 1991, pp. 13–19.

[8] R. Bar Yehuda and S. Fogel, Good splitters with applications to ray shooting, *Proc. 2nd Canadian Conf. on Computational Geometry*, 1990, pp. 81–85. (Also to appear in *Algorithmica*.)

[9] J. Bentley and J. Saxe, Decomposable searching problems, I: Static-to-dynamic transformation, *J. Algorithms*, **1** (1980), 301–358.

[10] M. de Berg, D. Halperin, M. Overmars, J. Snoeyink, and M. van Kreveld, Efficient ray shooting and hidden surface removal, *Proc. 7th Symp. on Computational Geometry*, 1991, pp. 21–30.

[11] B. Chazelle, An optimal computing convex hull algorithm and new results on cuttings, *Proc. 32nd IEEE Symp. on Foundations of Computer Science*, 1991, pp. 29–38.

[12] B. Chazelle, H. Edelsbrunner, M. Grigni, L. Guibas, J. Hershberger, M. Sharir, and J. Snoeyink Ray shooting in polygons using geodesic triangulations, *Proc. 18th International Colloquium on Automata, Languages, and Programming*, 1991, pp. 661–673.

[13] B. Chazelle, H. Edelsbrunner, L. Guibas, and M. Sharir, A singly-exponential stratification scheme for real semi-algebraic varieties and its applications, *Proc. 16th International Colloquium on Automata, Languages and Programming*, 1989, pp. 179–192.

[14] B. Chazelle, H. Edelsbrunner, L. Guibas, M. Sharir, and J. Stolfi, Lines in space: Combinatorics

and algorithms, Technical Report 491, Dept. of Computer Science, New York University, February 1990.

[15] B. Chazelle and L. Guibas, Visibility and intersection problems in plane geometry, *Discrete Comput. Geom.* **4** (1989), 551–589.

[16] B. Chazelle, M. Sharir, and E. Welzl, Quasi-optimal upper bounds for simplex range searching and new zone theorems, *Proc. 6th ACM Symp. on Computational Geometry*, 1990, pp. 23–33.

[17] B. Chazelle and E. Welzl, Quasi-optimal range searching in spaces of finite Vapnik–Chervonenkis dimensions, *Discrete Comput. Geom.* **4** (1989), 467–489.

[18] S. W. Cheng and R. Janardan, New results on dynamic planar point location, *Proc. 31st IEEE Symp. on Foundations of Computer Science*, 1990, pp. 96–105.

[19] S. W. Cheng and R. Janardan, Space efficient ray shooting and intersection searching: Algorithms, dynamization, and applications, *Proc. 2nd SIAM–ACM Symp. on Discrete Algorithms*, 1991, pp. 7–16.

[20] K. Clarkson, H. Edelsbrunner, L. Guibas, M. Sharir, and E. Welzl, Combinatorial complexity bounds for arrangemens of curves and spheres, *Discrete Comput. Geom.* **5** (1990), 99–160.

[21] R. Cole and M. Sharir, Visibility problems for polyhedral terrains, *J. Symbolic Comput.* **7** (1989), 11–30.

[22] D. Dobkin and H. Edelsbrunner, Space searching for intersecting objects, *J. Algorithms* **8** (1987), 348–361.

[23] D. Dobkin and D. Kirkpatrick, A linear algorithm for determining the separation of convex polyhedra, *J. Algorithms* **6** (1985), 381–392.

[24] D. Dobkin and D. Kirkpatrick, Determining the separation of preprocessed polyhedra: a unified approach, *Proc. 17th International Colloquium on Automata, Languages, and Programming*, 1990, pp. 400–413.

[25] H. Edelsbrunner, L. Guibas, J. Hershberger, R. Seidel, M. Sharir, J. Snoeyink, and E. Welzl, Implicitly representing arrangements of lines and of segments, *Discrete Comput. Geom.* **4** (1989), 433–466.

[26] L. Guibas, M. Overmars, and M. Sharir, Ray shooting, implicit point location, and related queries in arrangements of segments, Technical Report 433, Courant Institute, New York University, 1989.

[27] J. Matoušek, More on cutting hyperplanes and spanning trees with low crossing number, Technical Report B-90-2, Freie Universität Berlin, 1990.

[28] J. Matoušek, Spanning trees with low crossing numbers, *Inform. Théoret. Applic.* **25** (1991), 103–123.

[29] J. Matoušek, Approximations and optimal geometric divide-and-conquer, *Proc. 23rd ACM Symp. on Theory of Computing*, 1991, 506–511.

[30] J. Matoušek, Efficient partition trees, *Proc. 7th ACM Symp. on Computational Geometry*, 1991, pp. 1–9.

[31] J. Matoušek, Range searching with efficient hierarchical cuttings, *Proc. 8th ACM Symp. on Computational Geometry*, 1992, to appear.

[32] K. Mehlhorn, *Data Structures and Algorithms, 3: Multi-Dimensional Searching and Computational Geometry*, Springer-Verlag, Berlin, 1984.

[33] M. Overmars and J. van Leeuwen, Maintenance of configurations in the plane, *J. Comput. System Sci.* **23** (1981), 166–204.

[34] M. Overmars and J. van Leeuwen, Worst-case optimal insertion and deletion methods for decomposable searching, *Inform. Process. Lett.* **12** (1981), 168–173.

[35] M. Overmars, H. Schipper, and M. Sharir, Storing line segments in partition trees, *BIT* **30** (1990), 385–403.

[36] M. Overmars and M. Sharir, Output-sensitive hidden surface removal, *Proc. 30th IEEE Symp. on Foundations of Computer Science*, 1989, pp. 598–603.

[37] M. Overmars and M. Sharir, An improved technique for output-sensitive hidden surface removal, Technical Report RUU-CS-89-32, Computer Science Department, University of Utrecht, December 1989. (To appear in *Algorithmica*.)

[38] M. Overmars and M. Sharir, Merging visibility maps, *Comput. Geom. Theory Applic.* **1** (1991), 35–50.

[39] M. Pellegrini, Combinatorial and algorithmic analysis of stabbing and visibility problems in 3-dimensional space, Ph.D. Thesis, New York University, 1991.
[40] A. Schmitt, H. Müller, and W. Leister, Ray tracing algorithms—theory and algorithms, in *Theoretical Foundations of Computer Graphics and CAD* (ed. R. Earnshaw), NATO Series, Springer-Verlag, Berlin, 1988, pp. 997–1030.
[41] M. Sharir and M. Overmars, A simple output-sensitive hidden surface removal algorithm, *ACM Trans. Graphics* **11** (1992), 1–11.
[42] D. M. H. Sommerville, *Analytical Geometry in Three Dimensions*, Cambridge University Press, Cambridge, 1951.
[43] J. Stolfi, Primitives for computational geometry, Ph.D. Thesis, Stanford University, 1989.
[44] E. Welzl, Partition trees for triangle counting and other range searching problems, *Proc. 4th ACM Symp. on Computational Geometry*, 1988, pp. 23–33.

Received March 1, 1991, and in revised form January 8, 1992.