Computing and verifying depth orders

SIAM J. COMPUT.
Vol. 23, No. 2, pp. 437–446, April 1994

015

COMPUTING AND VERIFYING DEPTH ORDERS*

MARK DE BERG†, MARK OVERMARS†, AND OTFRIED SCHWARZKOPF†

Abstract. A depth order on a set of line segments in 3-space is an order such that line segment a comes before line segment a' in the order when a lies below a' or, in other words, when there is a vertical ray that first intersects a' and then intersects a. Efficient algorithms for the computation and verification of depth orders of sets of n line segments in 3-space are presented. The algorithms run in time $O(n^{4/3+\varepsilon})$, for any fixed $\varepsilon > 0$. If all line segments are axis-parallel or, more generally, have only a constant number of different orientations, then the sorting algorithm runs in $O(n \log^3 n)$ time and the verification takes $O(n \log^2 n)$ time. The algorithms can be generalized to handle triangles and other polygons instead of line segments. They are based on a general framework for computing and verifying linear orders extending implicitly defined binary relations.

Key words. computational geometry, depth orders, three dimensions, linear extensions of partial orders

AMS subject classification. 68Q25

1. Introduction.

Hidden surface removal is an important problem in computer graphics. In a typical setting, we are given a set of nonintersecting polyhedral objects in 3-space and a view point and want to compute which parts of the objects can be seen from the view point.

An efficient way of solving this problem is the *painter's algorithm*; see, for example, [10]. In this algorithm one tries to "paint" the objects in a back-to-front order onto the screen. Thus the objects in the front are painted on top of the objects in the back, resulting in a correct view of the scene. Such a back to front ordering is called a *depth order* of the set of objects. Note that a depth order does not always exist, since there can be *cyclic overlap* among the objects, as is the case for the three triangles shown in Fig. 1. A closely related approach uses

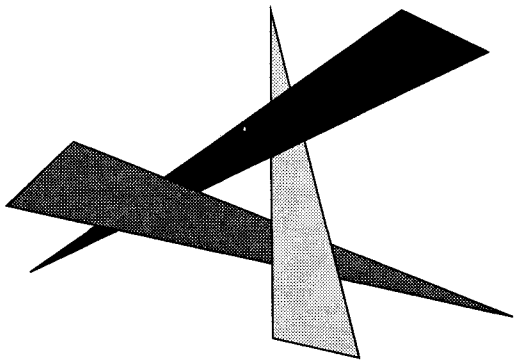

FIG. 1. *Cyclic overlap among triangles.*

a binary space partition tree to obtain a displaying order for the objects in a scene [11]. A binary space partition (BSP) cuts the objects in such a way that there is a depth order in any direction. Unfortunately, the number of fragments and, hence, the size of the resulting BSP tree can be as large as $\Omega(n^2)$ [20]. Hence, this approach can be very wasteful if there is no cyclic overlap in the viewing direction.

The view of a scene consists of a subdivision of the viewing plane into maximal connected regions such that in each region either (some portion of) a single object can be seen or no

*Received by the editors December 16, 1991; accepted for publication (in revised form) April 20, 1993. This research was supported by the ESPRIT Basic Research Action 3075 (project Algorithms and Complexity). The first and second authors were also supported by the Dutch Organization for Scientific Research (N.W.O.).

†Department of Computer Science, Utrecht University, P.O. Box 80.089, 3508 TB Utrecht, the Netherlands.

object is seen. Sometimes it is necessary to compute a combinatorial representation of this so-called *visibility map*. Note that the painter's algorithm does not give us such a combinatorial representation. The combinatorial complexity of the visibility map of a set of objects with n edges in total varies between $O(1)$ and $\Omega(n^2)$. Hence, it would be nice to have an *output-sensitive* algorithm, that is, an algorithm whose running time is dependent on the complexity of the visibility map. Almost all output-sensitive algorithms known to date require that a depth order on the objects is given; see, for example, [14], [16], [19], [21]. Only the recent algorithms of [8], [9] do not need a depth order. The implementation of the latter algorithms, however, is much easier when a depth order is known.

It is thus important to be able to compute depth orders efficiently. This problem was studied by Chazelle et al. [5]. When the objects are lines in 3-space, they noted that a depth order can be obtained by a standard sorting algorithm, because any two lines can be compared (assuming no two have parallel projections). If there is cyclic overlap, however, then the outcome of the sorting algorithm is not a valid depth order. Verifying whether a depth order is valid is no trivial matter though; in [5], Chazelle et al. presented an $O(n^{4/3+\varepsilon})$ time algorithm to verify a given depth order of a set of lines. When the objects are line segments in 3-space, the problem becomes much harder, since not every pair of line segments can be compared. For this case, the best algorithm that was known runs in time $O(n \log n + k)$, where k is the number of intersections in the projection plane, or, in other words, the number of pairs that can be compared directly [5], [18]. Note that k can be $\Theta(n^2)$ and, hence, that the worst-case running time of these algorithms is $\Theta(n^2)$. Even for the case of axis-parallel line segments, it was an open problem to find a depth order in $o(n^2)$ time [21].

In this paper we show that a depth order for a set of line segments in 3-space can be computed in subquadratic time. More specifically, we give an algorithm that computes a depth order in time $O(n^{4/3+\varepsilon})$. We also present an algorithm that verifies a given order in $O(n^{4/3+\varepsilon})$ time. When the line segments are c-oriented, that is, they have only c different orientations for some constant c, then the sorting algorithm runs in $O(n \log^3 n)$ time and verification takes $O(n \log^2 n)$ time. Note that axis-parallel line segments are 3-oriented. The results can be generalized to depth orders for sets of triangles, or other polygons, instead of line segments.

The algorithms that we give are surprisingly simple. They are based on a general framework for computing a linear order extending a relation $(S, \prec)$. It is easy to compute an order in time that is linear in the number of pairs that are related; to this end one sorts the directed graph $\mathcal{G} = (S, E)$ topologically, where $(a, a') \in E$ if and only if $a \prec a'$. This is the approach taken in [5], [18] to sort a set S of n line segments: first compute all pairs of line segments that are related—this can be done in $O(n \log n + k)$ time by computing all intersections in the projection plane—and then sort the corresponding graph $\mathcal{G}$ in $O(n + k)$ time. Note that if $(S, \prec)$ does not contain a cycle then the sorting will succeed, otherwise some cycle will be detected in the graph $\mathcal{G}$. We show that it is not necessary to compute the full graph corresponding to $(S, \prec)$. All that is needed is to have a data structure that answers the following question: Given an element $a \in S$, return a predecessor of a and a successor of a, if they exist. The data structure should allow for the deletion of an element a in S in sublinear time. In cases where the relation is given implicitly—such as for depth orders—this is often possible. Our algorithm uses an interesting form of divide-and-conquer, where the divide-step does not need to be balanced. In fact, the more unbalanced it is, the better the running time of the algorithm.

There is some previous work on the computation of a linear order that extends a partial order. This work is also in the context of depth orders, in particular, depth orders in two-dimensional space [12], [13], [23] and depth orders for spheres in 3-space [22]. Unfortunately, the solutions given in these papers do not generalize to our setting. Of related interest is also

a paper by Kenyon-Mathieu and King [15], who describe an algorithm that verifies whether a given partial order holds on n elements from an unknown total order.

The rest of this paper is organized as follows. In §2 we present our general framework for computing a linear order extending a relation $(S, \prec)$, and in §3 we give an algorithm to verify a given order. In §4 we show how to use these results to compute or verify a depth order for a set of line segments (or triangles, or polygons) in 3-space. We make some concluding remarks in §5.

2. Computing linear extensions. Let $\prec$ be a binary relation defined on a set S of n elements. Note that $\prec$ is not necessarily a partial order, since we do not assume transitivity. This will be useful in our application. In this section it is shown how to compute a linear order extending $(S, \prec)$ or to decide that $(S, \prec)$ contains a cycle. Thus we want to compute an order $a_1, \ldots, a_n$ on the elements in S such that $a_i \prec a_j$ implies $i < j$. The algorithm that we will give for this problem needs a data structure $\mathcal{D}_\prec$ for storing a subset $S' \subseteq S$ that can return a predecessor in S' of a query element $a \in S$. More formally, QUERY$(a, \mathcal{D}_\prec)$ returns an element $a' \in S'$ such that $a' \prec a$ or *NIL* if there is no such element. We call such a query a *predecessor query*. Similarly, we need a structure $\mathcal{D}_\succ$ for *successor queries*. To make our algorithm efficient, the structures should allow for efficient deletions of elements from S' and the preprocessing time should not be too high.

Let us define $\prec_*$ to be the transitive closure of $\prec$ and $\succ_*$ to be the transitive closure of $\succ$. The basic strategy of the algorithm is divide and conquer: we pick a pivot element $a_{\text{piv}} \in S$, partition the remaining elements into a subset $S_\prec$ of elements a that must come before a_{piv} in the order because $a \prec_* a_{\text{piv}}$ and a subset $S_\succ$ of elements that must come after a_{piv} in the desired order because $a_{\text{piv}} \prec_* a$, and recursively sort these sets. Note that not every pair of elements is comparable under $\prec_*$. Hence, except for the subsets $S_\prec$ and $S_\succ$, there is a third subset $S_\approx$ of elements that cannot be compared to a_{piv} under $\prec_*$. This subset should be sorted recursively as well. To find the subsets $S_\prec$ and $S_\succ$ efficiently, the data structures $\mathcal{D}_\prec$ and $\mathcal{D}_\succ$ are used. Consider the subset $S_\prec$. By querying $\mathcal{D}_\prec$ with element a_{piv}, we can find an element a such that $a \prec a_{\text{piv}}$. We delete a from $\mathcal{D}_\prec$ to avoid reporting it more than once and query once more with a_{piv}. Continuing in this manner until the answer to the query is *NIL*, we can find all elements $a \in S$ such that $a \prec a_{\text{piv}}$. However, we want to find all elements a such that $a \prec_* a_{\text{piv}}$. Thus we also have to query $\mathcal{D}_\prec$ with the elements a that we have just found, query with the new elements that we find, and so forth. Whenever we find an element, it is deleted from $\mathcal{D}_\prec$ and we query with it until we have found all predecessors of it (that have not been found before). This way we can compute the set $S_\prec$ with a number of queries in $\mathcal{D}_\prec$ that is linear in the size of $S_\prec$. Notice that when we find a_{piv} as an answer to a query, there must be a cycle in the relation. The subset $S_\succ$ can be found in a similar way, using the data structure $\mathcal{D}_\succ$. The subset $S_\approx$ contains the remaining elements.

There is one major problem with this approach: we cannot ensure that the partitioning is balanced, that is, that the sets $S_\prec$, $S_\succ$, and $S_\approx$ have about the same size. Fortunately, we can circumvent this if we make the following two observations. First, we note that we need not treat the subset $S_\approx$ separately. We can put the elements of $S_\approx$ in either $S_\prec$ or $S_\succ$, as long as we do it consistently, that is, as long as we put all elements in the same set. It seems that this only makes things worse because the partitioning gets more unbalanced. But now we observe that it is enough to find the smaller of the two subsets $S_\prec$ and $S_\succ$. The remaining elements—which can be elements of $S_\approx$—are all put into one set. It is possible to find the smaller of the two subsets $S_\prec$ and $S_\succ$—without computing the complete larger set as well—with a number of queries that is linear in its size, by doing a "tandem search": alternatingly, find an element of $S_\prec$ and an element of $S_\succ$ until the computation of one of the two subsets has been completed. Thus we partition S into two subsets in time that is dependent on the size of the smaller of the

two subsets. This means that the more unbalanced the partitioning is, the faster it is performed, leading to a good worst-case running time for the algorithm. There is one problem left that we have not addressed so far: we cannot afford to build the data structures that we need for the recursive call for the large set from scratch. Fortunately, we can obtain these structures from those that we have at the end of the tandem search by reinserting and deleting certain elements.

The algorithm for computing an ordering on $(S, \prec)$ first builds the data structures $\mathcal{D}_\prec$ and $\mathcal{D}_\succ$ on the set S and then calls the procedure ORDER, with the set S and these two data structures as arguments. Below follows a detailed description of this procedure, whose output is a linear order extending $(S, \prec)$ if one exists, and which detects a cycle otherwise. The algorithm maintains two queues $\mathcal{Q}_\prec$ and $\mathcal{Q}_\succ$, which store the elements of $S_\prec$ (respectively, $S_\succ$) for which we have not yet found all predecessors (respectively, successors). The procedure ENQUEUE adds an element to a queue. Similarly, DEQUEUE deletes an element from the queue. An element a is deleted from the data structure $\mathcal{D}_\prec$ by calling DELETE$(a, \mathcal{D}_\prec)$; a deletion from $\mathcal{D}_\succ$ is performed with a similar call. To delete all elements in a set A, we simply write DELETE$(A, \mathcal{D}_\prec)$.

The two main steps in the algorithm are steps 4 and 5. In step 4 of the algorithm the tandem search is performed; step 4(i) computes a new element of $S_\prec$, and step 4(ii) computes a new element of $S_\succ$. In step 5 the two sets that result from the partitioning are sorted recursively; to this end we first construct the data structures that are needed in the recursive calls. For the larger of the two sets the new data structures are obtained from the existing data structures, and for the smaller set the data structures are built from scratch.

ORDER(S,$\mathcal{D}_\prec$, $\mathcal{D}_\succ$)
1. **if** $|S| > 1$ **then** perform steps 2–6 **else** stop (S is already sorted).
2. Make $S_\prec \leftarrow \emptyset$ and $S_\succ \leftarrow \emptyset$, and initialize two empty queues $\mathcal{Q}_\prec$ and $\mathcal{Q}_\succ$.
3. Pick an arbitrary pivot element $a_{\text{piv}} \in S$; ENQUEUE$(a_{\text{piv}}, \mathcal{Q}_\prec)$; ENQUEUE$(a_{\text{piv}}, \mathcal{Q}_\succ)$.
4. **while** both $\mathcal{Q}_\prec$ and $\mathcal{Q}_\succ$ are nonempty
   do (i) $a \leftarrow$ DEQUEUE$(\mathcal{Q}_\prec)$; $a' \leftarrow$ QUERY$(a, \mathcal{D}_\prec)$.
   if $a' \neq NIL$
   then if $a' = a_{\text{piv}}$
   then Stop and report that there is a cycle.
   else ENQUEUE$(a, \mathcal{Q}_\prec)$; ENQUEUE$(a', \mathcal{Q}_\prec)$; DELETE$(a', \mathcal{D}_\prec)$.
   $S_\prec \leftarrow S_\prec \cup \{a'\}$.
   (ii) Compute a new element $a' \in S_\succ$ in a similar way, using $\mathcal{Q}_\succ$ and $\mathcal{D}_\succ$.
5. **if** $\mathcal{Q}_\prec$ is empty (hence, $S_\prec$ is the smaller set)
   then Reinsert the elements of $S_\succ$ into $\mathcal{D}_\succ$.
   DELETE$(S_\prec \cup \{a_{\text{piv}}\}, \mathcal{D}_\succ)$; DELETE$(a_{\text{piv}}, \mathcal{D}_\prec)$.
   Build new predecessor and successor structures $\mathcal{D}'_\prec$ and $\mathcal{D}'_\succ$ for the set $S_\prec$.
   ORDER$(S_\prec, \mathcal{D}'_\prec, \mathcal{D}'_\succ)$.
   ORDER$(S - \{a_{\text{piv}}\} - S_\prec, \mathcal{D}_\prec, \mathcal{D}_\succ)$.
   else Compute the data structures for the recursive calls as above, reversing the roles of $S_\prec$, $\mathcal{D}_\prec$ and $S_\succ$, $\mathcal{D}_\succ$, and sort $S_\succ$ and $S - S_\succ - \{a_{\text{piv}}\}$ recursively.
6. Concatenate $S_\prec$, a_{piv}, and $S_\succ$ to form the ordered list for S.

The following lemma proves the correctness of our algorithm.

LEMMA 2.1. *Procedure* ORDER *outputs a linear order extending* $(S, \prec)$ *if it exists and detects a cycle otherwise.*

Proof. It is straightforward to see that the algorithm never claims to have found a cycle that does not exist. It remains to show that if ORDER outputs a list $a_1, \ldots, a_n$ then this list is

a correct ordering. Assume for a contradiction that $a_i \succ a_j$ for some $i < j$. Then, at some stage of the algorithm, a_i must have been put into $S_\prec$, whereas a_j was put into $S_\succ$, or a_i was put into $S_\prec$ and a_j was the pivot element a_{piv}, or a_i was the pivot element a_{piv} and a_j was put into $S_\succ$. The second and third case both imply that there is a cycle containing a_{piv}, and we can easily verify that step 3 never fails to discover a cycle containing the pivot element. We thus consider the first case: If $\mathcal{Q}_\prec$ is empty after step 3 then all predecessors of a_i have been found, including a_j. Hence, a_j would have been put into $S_\prec$ instead of $S_\succ$. (It may also happen that a_j is put into both sets, but in that case the algorithm would have reported a cycle containing the pivot element.) Similarly, if $\mathcal{Q}_\succ$ is empty then a_i would have been put into $S_\succ$. □

Next we prove a bound on the running time of the algorithm. Let us for the sake of simplicity assume that the query time of $\mathcal{D}_\prec$ and the query time of $\mathcal{D}_\succ$ are equal, and let this time be denoted by $Q(n)$. Similarly, let the time to build these structures on n elements be $B(n)$, and let $D(n)$ denote the time for a deletion.

LEMMA 2.2. *The procedure* ORDER *runs in* $O([B(n) + n(Q(n) + D(n))] \log n)$ *time. The running time reduces to* $O(B(n) + n[Q(n) + D(n)])$ *if* $B(n)/n + Q(n) + D(n) = \Omega(n^\alpha)$ *for some constant* $\alpha > 0$.

Proof. Since all other operations in the procedure can be done in constant time, the time that we spend is dominated by the operations on the structures $\mathcal{D}_\prec$ and $\mathcal{D}_\succ$. Furthermore, if the size of the smaller of the two subsets $S_\prec$ and $S_\succ$ is m, then we perform at most $2m + 2$ queries and deletions on these structures in step 4 of the procedure. Restoring a data structure to a situation from the past, which we do in step 5, can be done without extra asymptotic overhead if we record all the changes. Finally, we perform m deletions in step 5, and we build new data structures for the smaller set. This adds up to $B(m) + O(1 + m)[Q(n) + D(n)]$ in total for the partitioning.

Next we argue that $m \leq n/2$ if the partitioning is successful, that is, if no cycle is found at this point. Suppose that $m > n/2$. Then there must be an element $a \in S_\prec \cap S_\succ$. But this means that a_{piv} will be found as a predecessor or a successor (whichever happens first) and a cycle is detected. Trivially, an unsuccessful partitioning happens at most once, giving a one-time cost of $O(n[Q(n) + D(n)])$.

It follows that the total running time $T(n)$ can be bounded by the recursion

$$T(n) \leq \max_{0 \leq m \leq n/2} \{B(m) + O(1 + m)[Q(n) + D(n)] + T(m) + T(n - m - 1)\},$$

which solves to the claimed time. □

Combining the two lemmas above, we obtain the following theorem.

THEOREM 2.3. *The procedure* ORDER *runs in* $O([B(n) + n(Q(n) + D(n))] \log n)$ *time and outputs an ordered list if* $(S, \prec)$ *does not contain a cycle or finds a cycle otherwise. The running time reduces to* $O(B(n) + n[Q(n) + D(n)])$ *if* $B(n)/n + Q(n) + D(n) = \Omega(n^\alpha)$ *for some constant* $\alpha > 0$.

Remark. With a little extra effort, the algorithm can output a witness cycle, when $(S, \prec)$ cannot be ordered. To this end, we keep track of the successor (predecessor) of each element that we put into $S_\prec$ ($S_\succ$). This extra information enables us to "walk back" when we find a_{piv} in step 3 of the algorithm and report the elements of the cycle.

3. Verifying linear extensions. In this section it is shown how to verify a given order for a relation $(S, \prec)$. Notice that different orders can be valid for $(S, \prec)$, so it does not suffice to compute a valid order and compare it to the given order. The algorithm uses a straightforward divide-and-conquer approach. It relies on the existence of a data structure $\mathcal{D}_\prec$ for predecessor queries. Unlike in the previous section, however, this data structure need not be dynamic. The algorithm we describe next has a list $\mathcal{L} = \{a_1, \ldots, a_n\}$ as input. For this list we have to test

whether it corresponds to a valid order. It will report that $\mathcal{L}$ is not sorted or run quietly when $\mathcal{L}$ is a valid ordering for $(S, \prec)$.

VERIFY($\mathcal{L}$)
if $|\mathcal{L}| > 1$
then Let $\mathcal{L}_1 = \{a_1, \ldots, a_{\lfloor n/2 \rfloor}\}$ and $\mathcal{L}_2 = \{a_{\lfloor n/2 \rfloor + 1}, \ldots, a_n\}$.
Build a data structure $\mathcal{D}_\prec$ for predecessor queries on $\mathcal{L}_2$.
for $i = 1$ **to** $\lfloor n/2 \rfloor$
do if QUERY(a_i, $\mathcal{D}_\prec$)$\neq$ *NIL*
then Stop and report that $\mathcal{L}$ is not sorted
VERIFY($\mathcal{L}_1$); VERIFY($\mathcal{L}_2$)

The correctness of the procedure is proved as follows. If $\mathcal{L}$ does not correspond to a valid order, then, by definition, there are elements a_i, a_j such that $a_i \prec a_j$ and $i > j$. Now either $i > \lfloor n/2 \rfloor$ and $j \leq \lfloor n/2 \rfloor$, or $i, j \leq \lfloor n/2 \rfloor$, or $i, j > \lfloor n/2 \rfloor$. The first case is tested by querying with the elements of $\mathcal{L}_1$ in the data structure $\mathcal{D}_\prec$, and the second and third possibilities are tested with the recursive calls for $\mathcal{L}_1$ and $\mathcal{L}_2$, respectively. The following theorem is now straightforward. As before, $B(n)$ denotes the time needed to build the structure $\mathcal{D}_\prec$ on a set of n elements, and $Q(n)$ denotes the query time.

THEOREM 3.1. *The procedure* VERIFY *verifies in* $O([B(n) + nQ(n)] \log n)$ *time whether a list* $\mathcal{L}$ *corresponds to an order for* $(S, \prec)$. *The running time of the procedure reduces to* $O(B(n) + nQ(n))$ *if* $B(n)/n + Q(n) = \Omega(n^\alpha)$ *for some constant* $\alpha > 0$.

Remark. Observe that if the procedure reports that $\mathcal{L}$ is not ordered, then it can report a witness pair a_i, a_j of elements such that $i < j$ and $a_j \prec a_i$. If the structure $\mathcal{D}_\prec$ is dynamic, then the algorithm can even report all conflicting pairs. When we test an element $a_i \in \mathcal{L}_1$, we just remove each element $a_j \in \mathcal{L}_2$ that conflicts with a_i from $\mathcal{D}_\prec$ and report the pair a_i, a_j, until no more conflicting elements are found. Then we reinsert the elements of $\mathcal{L}_2$ into $\mathcal{D}_\prec$ and test the next element of $\mathcal{L}_1$ in the same way.

4. Application to depth orders.

4.1. Depth orders for line segments. Let S be a set of n line segments in 3-space, and let $\vec{d}$ be the viewing direction. (The adaptation of the algorithms to "perspective depth orders," that is, depth orders with respect to a point, is straightforward.) We want to find a depth order on S for direction $\vec{d}$. In other words, we want to find a linear order extending the relation $(S, \prec)$, where $a \prec a'$ when there is a ray into direction $\vec{d}$ that first intersects line segment a' and then intersects line segment a. When $a \prec a'$, we say that a lies *behind* a' or that a' lies *in front of* a. Observe that $\prec$ is not necessarily a transitive relation. To apply Theorem 2.3, we need dynamic data structures that store a set $S' \subset S$ of line segments and enable us to find a line segment in S' lying behind (respectively, in front of) a query line segment. Define the *curtain* of a line segment into direction $\vec{d}$ to be the set of points q in 3-space such that there is a ray into direction $\vec{d}$ that first intersects a and then intersects q. If we want to find a line segment in S' lying in front of a query line segment a, we just have to check whether a intersects one of the curtains hanging from the line segments in S' and report the line segment holding that curtain. See Fig. 2. Finding a line segment lying behind a query line segment can be done in a similar way. Agarwal and Matoušek [2] have shown that intersection queries in a set of n curtains can be answered in time $O(n^{1/3})$ with a structure that uses $O(n^{4/3+\varepsilon})$ space and has an amortized update time of $O(n^{1/3+\varepsilon})$. (Note that the fact that the update time is amortized does not cause any difficulties for the analysis of the time bound.) If both the line segments holding the curtains and the query line segments are c-oriented, that is, they have only c different orientations for some constant c, then queries can be answered in time

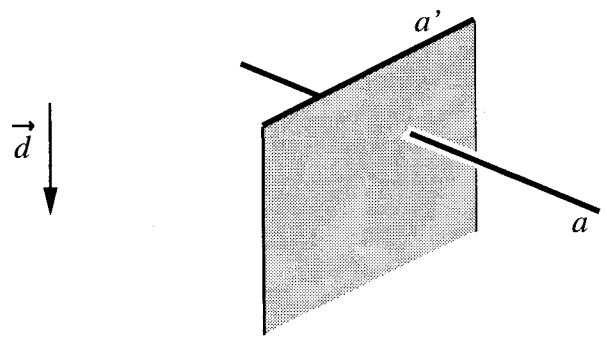

FIG. 2. *Line segment a lies behind line segment a' and, hence, intersects its curtain.*

$O(\log^2 n)$ with a structure using $O(n \log n)$ space and with $O(\log^2 n)$ update time; see de Berg [7]. Combining this with Theorem 2.3 gives us the following result.

THEOREM 4.1. *Given a set S of n line segments in 3-space and a viewing direction $\vec{d}$, one can compute a depth order on S for direction $\vec{d}$ or decide that there is cyclic overlap among the line segments, in time $O(n^{4/3+\varepsilon})$, for any fixed $\varepsilon > 0$. If the line segments are c-oriented then the time bound improves to $O(n \log^3 n)$.*

To verify a given depth order for a set of line segments in 3-space, we use the results of §3. In the general case, we again use Agarwal and Matoušek's structure for predecessor and successor queries. In the c-oriented case, we can use a more efficient structure than we used for computing depth orders: because the structure need not be dynamic, we can use the structure of de Berg and Overmars [9], which has $O(\log n)$ query time and $O(n \log n)$ preprocessing time. We immediately obtain the following theorem.

THEOREM 4.2. *It is possible to verify a depth order on a given set S of n line segments in 3-space for a viewing direction $\vec{d}$ in time $O(n^{4/3+\varepsilon})$, for any fixed $\varepsilon > 0$. If the line segments are c-oriented then the time bound improves to $O(n \log^2 n)$.*

4.2. Depth orders for triangles. To extend our results to triangles instead of line segments, we only need to adapt the data structures for predecessor and successor queries. Let us discuss the structure for successor queries; to obtain a structure for predecessor queries we only have to reverse the roles of "behind" and "in front of."

A triangle t is in front of another triangle t' if and only if at least one of the following conditions holds: (i) an edge of t is in front of an edge of t', (ii) t is in front of a vertex of t', or (iii) a vertex of t is in front of t'. (A vertex v is in front of a triangle t' if there is a ray into the viewing direction that first intersects v and then intersects t'.) We already know how to find the triangles t' that satisfy condition (i) for a query triangle t. The triangles satisfying conditions (ii) and (iii) can be found as follows. Consider condition (ii) and assume, to simplify the description, that the viewing direction is the negative z-direction. Project all vertices onto the xy-plane, and let $\bar{t}$ be the projection onto the xy-plane of a query triangle t. To find a vertex in front of t we select all vertices whose projections are contained in $\bar{t}$ in a small number of groups; for such a group we can think of t as being a plane, and the question becomes that of reporting a point in a half-space in 3-space. The latter query can be answered in $O(\log n)$ time, using the half-space emptiness structure of Agarwal et al. [1]. This structure uses $O(n^{1+\varepsilon})$ preprocessing and has $O(n^{\varepsilon})$ update time. The selection can be done using a three-level partition tree: each level filters out those vertices lying on the appropriate side of the line through one of the three edges of $\bar{t}$. We use the partition tree of Matoušek [17], which allows for queries and updates

in time $O(n^{1/3})$ (respectively, $O(n^{1/3+\varepsilon})$), and which uses $O(n^{4/3+\varepsilon})$ preprocessing time and space. Because the preprocessing and the query times in such a multi-level partition tree are essentially determined by the least efficient level, the preprocessing time, query time, and update time of the total structure remain $O(n^{4/3+\varepsilon})$, $O(n^{1/3})$, and $O(n^{1/3+\varepsilon})$, respectively. See [2], [3], [6], [17] for further details on the analysis of multilevel partition trees. The structure for condition (iii) is the same (up to some dualizations) as the structure for (ii) that we just described. We conclude that a dynamic structure for predecessor queries in a set of triangles exists with $O(n^{1/3})$ query and $O(n^{1/3+\varepsilon})$ update time, using $O(n^{4/3+\varepsilon})$ preprocessing time and space.

For the c-oriented case, where the edges of the triangles have only c different orientations for some constant c, we can use structures from [7]. There it is shown that a vertex in front of a c-oriented query triangle can be found in $O(\log^3 n)$ time, with a structure that uses $O(n \log^2 n)$ space and has $O(\log^3 n)$ update time. A triangle in front of a query vertex can be found in $O(\log^2 n \log\log n)$ time, using $O(n \log^2 n)$ space and with $O(\log^2 n \log\log n)$ update time. Furthermore, finding a vertex in front of an axis-parallel rectangle can be done with a structure whose query and update time are $O(\log^2 n)$.

The above combined with Theorems 2.3 and 3.1 lead to the following result.

THEOREM 4.3. *Given a set S of n triangles in 3-space and a viewing direction $\vec{d}$, one can compute a depth order on S for direction $\vec{d}$ or decide that there is cyclic overlap among the triangles, in time $O(n^{4/3+\varepsilon})$, for any fixed $\varepsilon > 0$. If the triangles are c-oriented then the time bound improves to $O(n \log^4 n)$, and if the objects are axis-parallel rectangles then the algorithm takes $O(n \log^3 n \log\log n)$ time.*

To verify a given depth order for an arbitrary set of triangles, we use the same structures as for computing a depth order. In the c-oriented case, however, we can save some logarithmic factors by using static structures instead of dynamic ones. In the algorithm of §3 we have to test whether the triangles in a list $\mathcal{L}_1$ do not lie in front of any triangle in a list $\mathcal{L}_2$. Testing whether there is an edge of a triangle in $\mathcal{L}_1$ that lies in front of an edge of a triangle in $\mathcal{L}_2$ can be done in $O(n \log n)$ time, as in §4.1. To test for conflicts corresponding to conditions (ii), we build a structure on the triangles in $\mathcal{L}_1$ that reports the first triangle that is hit by a query ray starting from infinity into the viewing direction. Next, we shoot rays from infinity into the viewing direction toward each vertex of all triangles in $\mathcal{L}_2$; when we know the first triangle that is hit by the ray toward a certain vertex, we can decide if there is any triangle in front of the vertex. There exists a structure that answers these ray shooting queries in $O(\log n)$ time, after $O(n \log n)$ preprocessing [9]. Hence, in $O(n \log n)$ time we can decide if there is a triangle in $\mathcal{L}_1$ that is in front of some vertex of a triangle in $\mathcal{L}_2$. To test condition (iii) we build a similar structure on the triangles in $\mathcal{L}_2$ (only this time for query rays into the opposite viewing direction), and we query with vertices of triangles in $\mathcal{L}_1$. This leads to the following theorem.

THEOREM 4.4. *It is possible to verify a given depth order on a set S of n triangles in 3-space for a direction $\vec{d}$ in time $O(n^{4/3+\varepsilon})$, for any fixed $\varepsilon > 0$. If the triangles are c-oriented then the time bound improves to $O(n \log^2 n)$.*

4.3. Extension to polygons. Consider the case where we want to compute a depth order for a set of polygons in 3-space, instead of a set of triangles. Let n be the total number of vertices of the polygons. First, we triangulate every polygon, which can be done in $O(n)$ time in total [4]. Observe that one polygon is behind another polygon if and only if one of the triangles in the triangulation of the first polygon is behind one of the triangles of the second polygon. Hence, we can use the same data structures as before to find predecessors and successors. However, if the polygons do not have constant complexity, then there is a

slight problem; the triangles that correspond to the same polygon must stay together in the ordering, so when we find one triangle as a predecessor or successor we have to report the other triangles as well. This is problematic, because the number of other triangles can be large. Suppose that during our tandem search we suddenly have to add a very large polygon to one of the subsets; if we find out in the next step that the other subset is complete, then we have spent a lot of time that we cannot charge to the smaller subset. An elegant solution to this problem can be obtained if we realize that we can choose any particular pivot element we like. Hence, we can choose the polygon with the largest complexity as pivot element. The tandem search for the sets $S_\prec$ and $S_\succ$ now proceeds as follows. We find successors and predecessors using the data structures for triangles. However, when we find a large polygon for, say, $S_\prec$, we first allow $S_\succ$ to catch up. Thus we search for successors until the complexity of $S_\succ$—that is, the total number of vertices of all polygons in $S_\succ$—is greater than the complexity of $S_\prec$. When this happens, we start querying for predecessors again, and so forth, until one of the subsets is completed. This way the extra work that we have to do, caused by adding a large polygon to what turns out to be the larger set, is bounded by the time spent on one polygon. Since the pivot polygon is chosen to be the largest polygon in the set, we can charge this extra work to the pivot polygon. Clearly, each polygon is charged at most once this way, because in the recursive calls we do not consider the pivot element anymore. Thus the asymptotic running time of the algorithm remains the same, and we have the following theorem.

THEOREM 4.5. *Given a set S of polygons in 3-space with n vertices in total and a viewing direction $\vec{d}$, one can compute a depth order on S for direction $\vec{d}$ or decide that there is cyclic overlap among the polygons, in time $O(n^{4/3+\varepsilon})$, for any fixed $\varepsilon > 0$. If the polygons are c-oriented then the time bound improves to $O(n \log^4 n)$, and if the polygons are axis-parallel then the algorithm takes $O(n \log^3 n \log\log n)$ time.*

The adaptation of the verification procedure to polygons is fairly straightforward, and we leave it as an (easy) exercise to the reader.

THEOREM 4.6. *It is possible to verify a given depth order on a set S of polygons in 3-space with n vertices in total, for a viewing direction $\vec{d}$, in time $O(n^{4/3+\varepsilon})$, for any fixed $\varepsilon > 0$. If the polygons are c-oriented then the time bound improves to $O(n \log^2 n)$.*

5. Concluding remarks. We have shown that it is possible to compute a depth order for a set of line segments in 3-space in subquadratic time. More specifically, a depth order can be computed in $O(n^{4/3+\varepsilon})$ time in the general case and in $O(n \log^3 n)$ time in the c-oriented case. It is also possible to verify a given depth order, and the results can be extended to polygons instead of line segments. Our algorithms are based on a general framework to compute or verify a linear order extending an implicitly defined binary relation, which might have other applications as well.

When a depth order is needed as input to a hidden surface removal algorithm, we are not done if we detect a cycle: the cycles should be removed by cutting the objects into smaller pieces. Moreover, we would like to use as few cuts as possible. As mentioned in the introduction, binary space partitions are a way of cutting the objects to obtain a depth order, but there is no guarantee that the number of pieces in this scheme is small [20]. We leave the computation of the minimum (or a small) number of cuts as an open problem. See [5] for an initial study of these problems.

REFERENCES

[1] P. K. AGARWAL, D. EPPSTEIN, AND J. MATOUŠEK, *Dynamic half-space reporting, geometric optimization, and minimum spanning trees*, Proc. 33rd Annual IEEE Symposium on Foundations of Computer Science, 1992, pp. 80–89.

[2] P. K. AGARWAL AND J. MATOUŠEK, *Ray shooting and parametric search*, Proc. 24th Annual ACM Symposium Theory Comput., 1992, pp. 517–526.

[3] P. K. AGARWAL AND M. SHARIR, *Applications of a new space partitioning scheme*, Discrete Comput. Geom., 9 (1993), pp. 11–38.

[4] B. CHAZELLE, *Triangulating a simple polygon in linear time*, Discrete Comput. Geom., 6 (1991), pp. 485–524.

[5] B. CHAZELLE, H. EDELSBRUNNER, L. J. GUIBAS, R. POLLACK, R. SEIDEL, M. SHARIR, AND J. SNOEYINK, *Counting and cutting cycles of lines and rods in space*, Comput. Geom. Theory Appl., 1 (1992), pp. 305–323.

[6] B. CHAZELLE, M. SHARIR, AND E. WELZL, *Quasi-optimal upper bounds for simplex range searching and new zone theorems*, Proc. 6th Annual ACM Symposium Comput. Geom., 1990, pp. 23–33.

[7] M. DE BERG, *Dynamic output-sensitive hidden surface removal for c-oriented polyhedra*, Comput. Geom. Theory Appl., 2 (1992), pp. 119–140.

[8] M. DE BERG, D. HALPERIN, M. OVERMARS, J. SNOEYINK, AND M. VAN KREVELD, *Efficient ray shooting and hidden surface removal*, Proc. 7th Annual ACM Symposium Comput. Geom., 1991, pp. 21–30.

[9] M. DE BERG AND M. OVERMARS, *Hidden surface removal for c-oriented polyhedra*, Comput. Geom. Theory Appl., 1 (1992), pp. 247–268.

[10] J. D. FOLEY, A. VAN DAM, S. K. FEINER, AND J. F. HUGHES, *Computer Graphics: Principles and Practice*, Addison-Wesley, Reading, MA, 1990.

[11] H. FUCHS, Z. M. KEDEM, AND B. NAYLOR, *On visible surface generation by a priori tree structures*, Comput. Graph., 14 (1980), pp. 124–133.

[12] L. GUIBAS, M. OVERMARS, AND M. SHARIR, *Ray shooting, implicit point location, and related queries in arrangements of segments*, Report 433, Department of Computer Science, New York University, New York, NY, March 1989.

[13] L. J. GUIBAS AND F. F. YAO, *On translating a set of rectangles*, Computational Geometry, F. P. Preparata, ed., Advances in Computing Research 1, JAI Press, London, England, 1983, pp. 61–77.

[14] R. H. GÜTING AND T. OTTMANN, *New algorithms for special cases of the hidden line elimination problem*, Comput. Vision Graph. Image Process., 40 (1987), pp. 188–204.

[15] C. KENYON-MATHIEU AND V. KING, *Verifying partial orders*, Proc. 21st Annual ACM Symposium Theory Comput., 1989, pp. 367–374.

[16] M. J. KATZ, M. H. OVERMARS, AND M. SHARIR, *Efficient hidden surface removal for objects with small union size*, Comput. Geom. Theory Appl., 2 (1992), pp. 223–234.

[17] J. MATOUŠEK, *Efficient partition trees*, Proc. 7th Annual ACM Symposium Comput. Geom., 1991, pp. 1–9.

[18] O. NURMI, *On translating a set of objects in two- and three-dimensional spaces*, Comput. Vision Graph. Image Process., 36 (1986), pp. 42–52.

[19] M. OVERMARS AND M. SHARIR, *Output-sensitive hidden surface removal*, Proc. 30th Annual IEEE Symposium on Foundations of Computer Science, 1989, pp. 598–603.

[20] M. S. PATERSON AND F. F. YAO, *Efficient binary space partitions for hidden-surface removal and solid modeling*, Discrete Comput. Geom., 5 (1990), pp. 485–503.

[21] F. P. PREPARATA, J. S. VITTER, AND M. YVINEC, *Output-sensitive generation of the perspective view of isothetic parallelepipeds*, Algorithmica, 8 (1992), pp. 257–283.

[22] G. TOUSSAINT, *Some collision avoidance problems between spheres*, Proc. International Conf. Systems, Man, Cybernetics, 1985.

[23] F. F. YAO, *On the priority approach to hidden-surface algorithms*, Proc. 21st Annual IEEE Symposium on Foundations of Computer Science, 1980, pp. 301–307.