Efficient Hidden Surface Removal for Objects with Small Union Size*

Matthew J. Katz† Mark H. Overmars‡ Micha Sharir§

Abstract

Let S be a set of n non-intersecting objects in space for which we want to determine the portions visible from some viewing point. We assume that the objects are ordered by depth from the viewing point (e.g., they are all horizontal and are viewed from infinity from above). In this paper we give two algorithms that compute the visible portions in time $O((U(n)+k)\log^2 n)$, where $U(n')$ is a super-additive bound on the maximal complexity of the union of (the projections on a viewing plane of) any n' objects from the family under consideration, and k is the complexity of the resulting visibility map. Both algorithms use $O(U(n)\log n)$ working storage. The algorithms are useful when the objects are "fat" in the sense that the union of the projection of any subset of them has small (i.e., subquadratic) complexity. We present three applications of these general techniques: (i) For disks (or balls in space) we have $U(n) = O(n)$, thus the visibility map can be computed in time $O((n+k)\log^2 n)$. (ii) For 'fat' triangles (where each internal angle is at least some fixed θ degrees) we have $U(n) = O(n \log\log n)$ and the algorithms run in time $O((n\log\log n + k)\log^2 n)$. (iii) The methods also apply to computing the visibility map for a polyhedral terrain viewed from a fixed point, and yield $O((n\alpha(n)+k)\log n)$ algorithms.

*Work by Mark Overmars has been partially supported by the ESPRIT Basic Research Action No. 3075 (project ALCOM) and by the Dutch Organisation for Scientific Research. Work on this paper by Matthew Katz and Micha Sharir has been supported by a Grant from the G.I.F., the German-Israeli Foundation for Scientific Research and Development. Work by Micha Sharir has also been supported by Office of Naval Research Grant N00014-90-J-1284, by National Science Foundation Grant CCR-89-01484, and by grants from the U.S.-Israeli Binational Science Foundation, and the Fund for Basic Research administered by the Israeli Academy of Sciences.

†School of Mathematical Sciences, Tel Aviv University, Tel Aviv 69978, Israel.

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

§School of Mathematical Sciences, Tel Aviv University, Tel Aviv 69978, Israel, and Courant Institute of Mathematical Sciences, New York University, New York, NY 10012, USA.

1 Introduction

In the past few years much attention has been given in computational geometry to the *hidden surface removal problem*, one of the central problems in computer graphics. In a typical setting of the problem we are given a collection of n non-intersecting polyhedral or other objects in 3-space, and a viewing point v, and our goal is to construct the view of the given scene, as seen from v.

Most solutions to the problem as applied in graphics use an "image-space" approach, in which one tries to calculate, for each pixel in the viewed image, which object is visible at that pixel (see e.g. [26]).

Recently a considerable effort has been made to obtain efficient "object-space" methods that try to compute a discrete combinatorial representation of the view of the scene, whose complexity does not depend on the screen size, but only on the combinatorial complexity of the scene. This view consists of a subdivision of the viewing plane into maximal connected regions in each of which (some portion of) a single object can be seen, or no object is seen. The obtained subdivision is called the *visibility map* of the given collection of objects.

A major challenge in this direction is to obtain *output-sensitive* algorithms, namely algorithms whose running time depends on the actual combinatorial complexity, k, of the visibility map, so that if k is small the algorithms will run more efficiently. Early object-space methods have a running time of $O(n^2)$, independent of the complexity of the resulting visibility map [7, 14]. Other implementations run in time $O((n+I)\log n)$, where I denotes the number of intersections between the projected edges [8, 10, 16, 25], which may also be insensitive to the output size (there are easy examples where $I = \Theta(n^2)$ but k is a constant). Another recent technique [15] uses a randomized incremental approach, leading to expected running time that is expressed as a weighted sum over the I intersection points; however, this technique is also not output-sensitive.

The most general output-sensitive hidden surface removal method to date is due to Overmars and Sharir

[17] (see also [18, 24]). It computes the view of a set of horizontal triangles (or other flat objects with a simple shape), as seen from above, in time $O(n\sqrt{k}\log n)$. Although the method is output-sensitive, the running time is still quite high (a somewhat improved, but more complicated, algorithm is given in [18]; see also [1]). Better results have been obtained for special cases, like axis-parallel rectangles [2, 9, 22], c-oriented polyhedra [4, 10], polyhedral terrains [23], and unit disks [19].

In this paper we develop two new techniques for output-sensitive hidden surface removal. The techniques are fairly general and simple, but their efficiency shows up when the objects have the property that the union of the projections on the viewing plane of any subcollection of j of them has small combinatorial complexity (by 'small' we mean $o(j^2)$, and typically close to linear in j). We refer to objects with this property as being "fat". Let $U(n)$ be a bound on the maximum combinatorial complexity of the union of the projections of any n objects from such a family, and suppose that $U(n)$ is *super-additive*, i.e., $U(n_1)+U(n_2) \leq U(n_1+n_2)$. We show that the view of n such fat objects can be computed in time $O((U(n)+k)\log^2 n)$, using $O(U(n)\log n)$ working storage. Both methods are simple and, hence, potentially practical.

We present three applications of these techniques:

- If the given objects are horizontal disks (or, for that matter, pairwise disjoint balls) viewed from any direction from infinity, then $U(n) = O(n)$ [11]. In this case our techniques yield algorithms with running time $O((n+k)\log^2 n)$.

- If the given objects are horizontal 'fat' triangles, namely triangles whose angles are all at least some fixed angle θ, which are viewed from any direction from infinity, then $U(n) = O(n \log\log n)$ [13]. In this case our techniques yield algorithms with running time $O((n\log\log n + k)\log^2 n)$.

- Finally we consider the case of viewing a polyhedral terrain from any fixed point. Here one has $U(n) = O(n\alpha(n))$, where $\alpha(n)$ is the extremely slowly growing inverse of Ackermann's function [6]. In this case our techniques yield algorithms with running time $O((n\alpha(n)+k)\log n)$. (The simpler structure of the visibility map in this case facilitates a saving of a $\log n$ factor in the time bound.)

Finally, a caveat for the reader: like most of the papers cited so far, our techniques assume a depth order among the viewed objects, which is easy to compute and which excludes cyclic overlaps among them. Problems in which such an order is not available or does not exist are much harder to handle (see e.g. [5, 20] for the extra techniques that may be required; see also [4] for a technique that does not require such an order, although it applies only in some special cases).

The paper is organized as follows. In Section 2 we describe the first algorithm. In Section 3 we analyze its run-time and show how to improve the storage to the bound given above. In Section 4 we describe the second algorithm. In Section 5 we present the applications listed above. The paper is concluded in Section 6 with a discussion of our results and some open problems.

2 The First Algorithm

We first present a simpler version of the method where we do not optimize the working storage. This version is really simple — it involves two divide-and-conquer passes over the objects ordered by depth from the viewing point. At each recursive call we compute the union, intersection, or difference of two planar regions, using standard line-sweeping methods. In this version the working storage is $O((U(n)+k)\log n)$. Optimizing the storage requires a more careful handling of the recursive process.

As a first step the method sorts the objects by depth order and stores them in this order in the leaves of a balanced binary tree $\mathcal{T}$, the nearest object in the leftmost leaf. For each node δ of $\mathcal{T}$ we compute the following two maps:

- U_δ — the union of the projections of the objects in the subtree $\mathcal{T}_\delta$ of $\mathcal{T}$ rooted at δ.

- V_δ — the visible portions of U_δ, i.e. the subset of U_δ consisting of those points that are not contained in the projection of any nearer object (stored in $\mathcal{T}$ to the left of δ).

Both U_δ and V_δ are planar regions, possibly with holes. Their boundary consists of portions of projected edges of the original objects. Clearly $V_\delta \subseteq U_\delta$. In following the description of the algorithm, it is helpful to visualize U_δ as a new nominal object obtained by "squashing" all objects stored below δ onto some common in-between plane and gluing them together. V_δ can be thought of as the portions of the new object that are visible in the standard sense.

Once we have computed V_δ for each node δ in the tree we are done, because for each leaf δ, V_δ consists precisely of those parts of the object stored in this leaf that are visible. So reporting V_δ for all leaves gives the entire visibility map (those V_δ's can easily be glued together in a final step of the algorithm to obtain the global visibility map).

Computing U_δ for all nodes is quite easy. We do this in a bottom-up manner by first computing the unions

for all leaves (being the objects themselves) and then merging unions towards the root, using the fact that

$$U_\delta = U_{\mathrm{lson}(\delta)} \cup U_{\mathrm{rson}(\delta)}.$$

Merging two unions is done by computing all intersections between their boundaries. Note that any such intersection point is necessarily a vertex of the overall union. This can be done using e.g. the red-blue intersection algorithm of Mairson and Stolfi [12] in time $O((u_{\mathrm{lson}(\delta)} + u_{\mathrm{rson}(\delta)}) \log n + u_\delta)$, where u_δ denotes the complexity of U_δ. For our purpose we can as well use the standard intersection algorithm by Bentley and Ottmann [3] (see also [21]) without increasing the overall asymptotic time complexity.

After computing the union U_δ at each node, we compute V_δ for all nodes in a top-down manner, starting at the root and working our way down the tree. The method is based on the following lemma:

Lemma 2.1 *The maps V_δ stored at nodes δ satisfy the following equations:*

$$V_{\mathrm{root}} = U_{\mathrm{root}}$$

$$V_{\mathrm{lson}(\delta)} = V_\delta \cap U_{\mathrm{lson}(\delta)}$$

$$V_{\mathrm{rson}(\delta)} = V_\delta - U_{\mathrm{lson}(\delta)} .$$

Proof. The first equation is easy, because the whole union of the set of objects is obviously visible in the sense defined above — there is no nearer object to hide it. The second equation follows from the fact that $U_{\mathrm{lson}(\delta)}$ can only be covered by objects that also cover U_δ. Moreover, $U_{\mathrm{lson}(\delta)}$ is a subset of U_δ. V_δ can be interpreted as the window through which we can see U_δ and, hence, the portions of $U_{\mathrm{lson}(\delta)}$ that can be seen are exactly those that lie inside V_δ. The third equation follows from the fact that $V_\delta - U_{\mathrm{lson}(\delta)}$ consists of those points of $U_{\mathrm{rson}(\delta)}$ that are not hidden by objects stored to the left of δ or below $\mathrm{lson}(\delta)$; by definition, these points constitute $V_{\mathrm{rson}(\delta)}$. □

We apply this lemma to compute the regions V_δ, starting at the root and working our way down. To compute $V_{\mathrm{lson}(\delta)}$ (resp. $V_{\mathrm{rson}(\delta)}$) we simply compute the intersection (resp. difference) of V_δ and $U_{\mathrm{lson}(\delta)}$ using any of the techniques above, say the red-blue intersection algorithm of [12]. This takes time $O((u_{\mathrm{lson}(\delta)} + v_\delta) \log n + v_{\mathrm{lson}(\delta)})$, where v_δ denotes the complexity of V_δ. (Note that in both cases any intersection between the boundaries of V_δ and $U_{\mathrm{lson}(\delta)}$ must be a vertex of the resulting intersection or difference.)

This concludes the description of (the simpler version of) the algorithm. In the following section we will slightly modify the algorithm so as to reduce its working storage. After we have computed the regions V_δ at all nodes of the tree, we simply collect (and properly glue) the regions computed at the leaves, to construct the whole visibility map. Note that the algorithm is very simple and only requires as a subroutine an implementation of the red-blue intersection algorithm (or some other intersection algorithm like the one in [3]), suitable for computing unions, intersections, and differences between two regions in the plane.

3 Analysis of the First Algorithm

It immediately follows from the above description that the total time required for the algorithm, after the initial sorting and construction of the tree (which requires time $O(n \log n)$), is bounded by

$$\sum_\delta O((u_\delta + v_\delta) \log n) = O(\log n) \cdot (\sum_\delta u_\delta + \sum_\delta v_\delta) \ . \quad (1)$$

So we have to estimate both $\sum_\delta u_\delta$ and $\sum_\delta v_\delta$. As indicated in the introduction, we assume that the objects involved are "fat" in the sense that the complexity of the union of (the xy-projections of) any subset of n' objects is bounded by (the subquadratic function) $U(n')$ which we also assume to be super-additive. Now let n_δ denote the number of objects in the subtree rooted at δ. Then clearly

$$\sum_\delta u_\delta \ \leq \ \sum_\delta U(n_\delta) = \sum_{d=0}^{\log n} \sum_{\delta \text{ at depth } d} U(n_\delta) = \sum_{d=0}^{\log n} O(U(n)) = O(U(n) \log n) \ . \quad (2)$$

Estimating v_δ is slightly more complicated. The bound is based on the following lemma:

Lemma 3.1 *Any vertex of V_δ is a vertex of $V_{\delta'}$ for some leaf δ' in the subtree rooted at δ.*

Proof. V_δ has four different types of vertices: visible vertices of U_δ, visible intersections between the boundaries of U_δ and the projection of a nearer object, visible vertices of nearer objects that lie inside U_δ, and visible intersections between the (projections of the) boundaries of two nearer objects, which lie inside U_δ. All of these are obviously vertices of the final visibility map. It remains to show that there exists an object stored in the subtree rooted at δ, such that the intersection shows up as a vertex of the individual visibility map of the object. This claim is immediate for vertices of the first or second type, because each of them is either an original

vertex of an object stored below δ, or the intersection of the boundary of such an object with the boundary of another higher object. For a vertex v of the third or fourth type, note that U_δ must be visible on some side of v in a sufficiently small neighborhood, which means that an object stored below δ is visible there. Hence v is a vertex of $V_{\delta'}$ for the leaf δ' that stores this object. □

As stated above, the collection of maps V_δ over all leaves forms together the full visibility map. Moreover, as in the proof of the preceding lemma, it is easily verified that each vertex of the map can appear in at most two 'leaf-regions' V_δ. As a result we have:

$$\sum_{\delta \text{ a leaf}} v_\delta = O(k) .$$

It follows from the above lemma that the overall complexity of the maps V_δ on each level of the tree is also $O(k)$. Hence,

$$\sum_\delta v_\delta = \sum_{d=0}^{\log n} \sum_{\delta \text{ at depth } d} v_\delta = \sum_{d=0}^{\log n} O(k) = O(k \log n) . \qquad (3)$$

This leads to the following result:

Proposition 3.2 *Given a set of n non-intersecting objects, such that the union of the projections on a viewing plane of any n' of them has complexity $U(n')$, where $U(n')$ is super-additive (and hopefully subquadratic), the visibility map of the objects can be computed in time $O((U(n)+k)\log^2 n)$.*

Proof. This follows immediately from equation (1), plugging in the results of equations (2) and (3). □

Remark. As noted earlier, this technique is rather general — it only requires a (known) depth ordering of the objects relative to the viewing point. It also applies when $U(n)$ is large, up to quadratic, except that the result is then much less exciting.

It remains to analyze the amount of working storage required by the algorithm. Unfortunately, using the method as described above, the amount of required working storage becomes $O((U(n)+k)\log n)$. To reduce this we have to modify the method slightly.

First we construct the whole tree, together with the U_δ's for all nodes. All U_δ's at any particular level of the tree use $O(U(n))$ overall storage, so the total tree uses so far $O(U(n)\log n)$ storage. Next we recursively traverse the tree in preorder, computing the V_δ for all nodes, in the following way:

- if δ is a leaf, output V_δ; otherwise,
- compute $V_{\text{lson}(\delta)}$ from $U_{\text{lson}(\delta)}$ and V_δ;
- recursively treat the left subtree;
- remove $V_{\text{lson}(\delta)}$ (it is no longer required);
- compute $V_{\text{rson}(\delta)}$ from $U_{\text{lson}(\delta)}$ and V_δ;
- recursively treat the right subtree;
- remove $V_{\text{rson}(\delta)}$.

As a result, at any time during the algorithm we only store the regions V_δ along a single path of the tree, i.e., for at most $O(\log n)$ nodes. It remains to bound the size of one V_δ. Let U be the union of the projections of all the objects that lie nearer than U_δ (i.e. objects that are stored in the tree to the left of the subtree rooted at δ). Any vertex of V_δ is either a vertex of U_δ, or a vertex of U, or an intersection point between the boundaries of U_δ and U, and, hence, a vertex of $U_\delta \cup U$. The total number of these vertices is clearly bounded by $O(U(n))$. This leads to our main result:

Theorem 3.3 *Given a set of n non-intersecting objects in space and a viewing point z (that may be at infinity), such that there exists a known (and easily computable) depth ordering of the objects with respect to z, and such that the union of the projections of any n' of the objects on a viewing plane has complexity $U(n')$, where $U(n')$ is super-additive (and subquadratic), then the visibility map, as seen from z, can be computed in time $O((U(n)+k)\log^2 n)$, using $O(U(n)\log n)$ working storage.*

4 The Second Algorithm

In this section we present another algorithm for computing the visibility map M. It also computes M in time $O((U(n)+k)\log^2 n)$ and working storage $O(U(n)\log n)$. It is a variant of a previous technique of the authors [19].

The first part of the algorithm is identical to the first part of the first algorithm. We sort the objects by depth and store them in the leaves of a balanced binary tree $\mathcal{T}$, the nearest object in the leftmost leaf. Then for each node δ of $\mathcal{T}$, we compute U_δ, the union of the projections of the objects in the subtree whose root is δ. We do this in a bottom-up manner as described in Section 2.

The second part of the algorithm resembles the merging algorithm of [19]. In this part a left-to-right plane sweep is performed through the regions U_δ simultaneously such that at any point during the sweep, the part of the visibility map M to the left of the sweepline has already been computed.

A few comments before we begin our presentation: O_δ denotes the set of objects stored in the leaves of $\mathcal{T}_\delta$, the subtree of $\mathcal{T}$ whose root is δ. The edges of the

boundary of a region U_δ are portions of the projected edges of the objects in O_δ. We will say that *edge e of U_δ belongs to object O* if e is a portion of a projected edge of O. A vertex of the boundary of a region U_δ is either a projected vertex of one of the objects in O_δ, or an intersection point between two projected edges of two different objects in O_δ. We refer to the latter type of vertices as the 'interesting' vertices of U_δ. For simplicity of exposition, we assume that the edges of the objects are straight segments, though the technique described below is also applicable, with appropriate modifications, in more general settings.

As mentioned, we perform a plane sweep through the regions U_δ simultaneously. Instead of maintaining one Y-structure we maintain for each region U_δ a separate Y-structure Y_δ. Y_δ is a balanced binary tree that stores in its nodes the edges of the boundary of U_δ that are currently cut by the sweepline L. We also 'implant' in Y_δ certain other edges that are currently visible and belong to objects that lie nearer than the objects in O_δ. It is the 'responsibility' of these 'foreign' edges to detect the vertices of M that are formed as intersection points between themselves and the edges of U_δ — see below.

Initially, the event points for the sweep are the vertices of the regions U_δ, which are stored, sorted by x-coordinate, in a priority queue Q. With such an event point we record the node of $\mathcal{T}$ from which it comes. A point p may appear as many as $\log n$ times in Q, since all nodes δ on the path from the root to the lowest common ancestor of the leaves that store the objects defining p could have p as a vertex of their region U_δ. During the sweep new event points, referred to as (candidates for) branching points, are added to Q. A new event point is either an 'interesting' vertex of M, or a candidate for such a vertex, which is removed later before it reaches the front of Q.

An event point p initially in Q is either an extreme point of two projected edges of some object $O \in O_\delta$, or is an intersection point between two projected edges of two different objects in O_δ. In the former case, either two edges of O begin, two edges of O end, or one edge of O begins and another edge of O ends. In the latter case, one of the edges stops appearing on the boundary of U_δ and the second starts appearing there.

Temporarily, assume that event points are not added to the queue during the sweep. When the sweepline reaches some event point, p, we determine the two relevant edges of the boundary of U_δ (the region p comes from), and delete/insert them from/into Y_δ. If q is a point with x-value between the previous event point and the current one, we can easily determine in $O(\log n)$ whether q belongs to the region U_δ, for any node δ, by searching in Y_δ. Therefore, by searching in $O(\log n)$ Y-structures, we can determine whether a point a of an object O that belongs to this range is visible — we simply search in the Y-structures associated with the left sons of the nodes along the path from the root of $\mathcal{T}$ to the leaf storing O that are not on the path. Using a similar technique, we can also determine which object (if any) lies immediately behind a; we refer to such an object as the *background* object of a. To find this object, we search in the Y-structures of the right sons of the nodes along the path from the root to the leaf storing O that are not on the path, starting from the son representing the nearest depth range and proceeding to the son representing the farthest depth range until we reach a son δ' whose associated region covers a. Now we move down $\mathcal{T}_{\delta'}$ towards the leftmost leaf that is behind a. These two types of queries — testing a point for being visible, and computing its background object — cost $O(\log^2 n)$ each. The entire sweep takes $O(U(n)\log^2 n)$, because handling an event point, i.e., updating the appropriate Y-structure requires $O(\log n)$ time, and there are $O(U(n)\log n)$ event points.

Of course this procedure does not compute any new vertex of M, so we must do some more work. Let q be an 'interesting' vertex of M. q is an intersection point between (the projections of) two edges of two different objects. The nearer edge e_i is visible both to the left and to the right of q, and the farther edge e_j is visible at one side of q and not visible at the other side of q (where the nearer object is hiding it). We also refer to points like q as branching points, and associate them with the nearer edges. The nearer edge e_i will be 'responsible' for finding the vertex q. To facilitate this, we insert e_i into the Y-structures of certain nodes δ, including a node for which e_j is part of the boundary of U_δ. The edge e_i will search for intersections between it and adjacent edges of Y_δ in a manner that will guarantee the detection of q. See below for more details.

We now present the (second part of the) algorithm for computing M in detail. Actually, we only describe how to compute the vertices of M; the other features of M are easily found afterwards without increasing the asymptotic complexity.

Let p denote the current event point at the front of Q. We distinguish between the original event points and the event points that were added after the sweep action has begun. If p is an original event point, let δ denote the node of $\mathcal{T}$ from which it comes. Apply two of the following four procedures (Procedures A–D) according to the underlying case. If p is a branching point, apply Procedure E (which is based on the former procedures).

A. δ is a leaf of $\mathcal{T}$, and p is the left endpoint of an edge e of the object O_i stored in δ:

1. Add e to Y_δ.
   If one of the neighbors e' of e in Y_δ is an implanted edge, check whether e' intersects e. If it does and

e' does not yet have a candidate branching point in Q, add this intersection point to Q, and in the leaf storing the object of the implanted edge e', set a pointer attached to e' to this new candidate branching point in Q. If the implanted edge e' already has a candidate branching point, check whether the new point appears earlier in the sweep than the existing point. If it is, remove the existing point from Q, insert the new point into Q, and update the appropriate pointer accordingly.

2. Determine whether p is visible.
(This is done as explained above. The presence of implanted edges in the Y-structures requires some slight modifications of the search as described in the remark below.)

3. If p is visible,

   - Output p as a vertex of M.
   - Find O_j, the background object of p (if it exists).
     (This is done as explained above, with some modifications to handle the presence of implanted edges, as described below.)
   - 'Implant' e in the Y-structures of the nodes of $\mathcal{T}$ that are sons of nodes on the paths to the leaves of O_i and O_j and their depth range is strictly between O_i and O_j (there are at most $2 \log n$ such nodes), and in the Y-structure of the leaf storing O_j. During this action check for candidate branching points for e and update Q and the appropriate pointer accordingly. In other words, assume e has just been implanted in some Y-structure and f is one of its neighbors (in this Y-structure). If f is not an implanted edge and e intersects it, a new candidate branching point for e has been discovered. Through the pointer attached to e, determine whether this new candidate branching point appears earlier in the sweep than the one currently in Q (if there exists such a point). If it is, remove the existing point from Q, insert the new point into Q, and update the pointer accordingly. If this new candidate branching point is the first candidate branching point found for e, simply insert it into Q and set the pointer accordingly.

Remark: Let e be an implanted edge in Y_δ, where δ is an internal node of $\mathcal{T}$. Since the depth range of δ is strictly between the object containing e and the background of e, it easily follows that (the projection of) e lies outside U_δ. Hence, when searching for visibility or for a background object, if the interval along Y_δ containing the query point p is bounded by at least one implanted edge, we conclude that p lies outside U_δ. Thus the presence of implanted edges in the Y-structures of internal nodes does not pose any difficulty for these searches. If δ is a leaf of $\mathcal{T}$, then Y_δ has at most 2 non-implanted edges (of the object stored at δ), so searching among them can be done in constant time.

B. δ is an internal node of $\mathcal{T}$, and p is the left endpoint of an edge e of the boundary of U_δ:

1. Add e to Y_δ and check for candidate branching points as described in the first step of Procedure A.

C. δ is a leaf of $\mathcal{T}$, and p is the right endpoint of an edge e of the object O_i stored in δ:

1. Remove e from Y_δ and check for a candidate branching point, that is, if exactly one of the two new neighbors is an implanted edge and it intersects the other, act accordingly, as in Procedure A.

2. If p is visible (equivalently, if there exists implants of e), output p as a vertex of M, and remove the implants of e. During this action check for candidate branching points at the nodes where the implants are removed, as above.

D. δ is an internal node of $\mathcal{T}$, and p is the right endpoint of an edge e of the boundary of U_δ:

1. Remove e from Y_δ and check for a candidate branching point, as above.

E. δ is a branching point of an edge e of object O_i, and the branching (farther) edge is f of object O_j:

1. Output p as a vertex of M.

2. We treat p as the right endpoint of e (the portion of e that ends at p), and apply Procedure C at the leaf storing O_i.

3. Now we treat p as the left endpoint of e (the portion of e that begins at p), and apply Procedure A at the leaf storing O_i.
(The removal and reinsertion of e is necessary since the background object of e has changed — it is now either O_j or, if O_j was the background object, a new background object lying farther than O_j.)

4. If at p f disappears, treat p as its right endpoint and apply step 2 of Procedure C at the leaf storing O_j. Also, if f has a candidate branching point in Q, remove it. If at p f (re)appears treat p as its

left endpoint and apply step 3 of Procedure A at the leaf storing O_j.

Remark. Event points coming from internal nodes of $\mathcal{T}$ are handled by Procedure B and Procedure D. These procedures merely insert/delete the appropriate edges of U_δ into/from Y_δ (and check for candidate branching points), and leave the rest to the other procedures. Let p be a visible event point coming from an internal node δ. If p is an endpoint of an edge of an object in O_δ, Procedure A or Procedure C will output it when handling the copy of p coming from the leaf storing this object. If p is an intersection point between two edges of two different objects in O_δ, Procedure E will output it as a branching point of the nearer edge.

Lemma 4.1 *The algorithm correctly computes all the vertices of M.*

Proof. Consider the leftmost vertex, v, of M which is not detected by the algorithm (assuming there is one). If v is a visible vertex of some object O, it is found by Procedure A (if v is a left endpoint of an edge of O) or by Procedure C (if v is a right endpoint of an edge of O). Assume v is an 'interesting' vertex of M, that is, v is a visible intersection point between edges e and f of objects O_i and O_j, respectively, where O_i is nearer than O_j. We will prove that v is inserted into Q as a (final) candidate branching point of e.

Let v' be the vertex of M lying on e to the left of v and nearest to it along e. By assumption, the algorithm has detected v' as a vertex of M. Regardless of which type of vertex v' is, Procedure A (perhaps called by Procedure E) will have (re)inserted e into various Y-structures with v' as its left endpoint. Let O_k be the background object of v'. It is easily seen that the object O_j bounded by f is either O_k or an object whose depth is between those of O_i and O_k. In the former case, since e is implanted at the leaf containing O_k, v will be detected there. In the latter case, there is a node δ with O_j in its object set so that e is implanted into Y_δ. It is easily verified that, just slightly to the left of v, the two edges e and f are adjacent edges in Y_δ, so that e is an implanted edge and f is an edge of U_δ. Go left from v until the first time where this property no longer holds. It is easily seen that at this point the sweepline passes either through v', or through the point w most recently handled along f, or through another vertex z that lies in Y_δ between e and f. It is easily checked that z is a vertex of U_δ. In either case, it is easily verified that the action taken by the algorithm at this point (v', w, or z) detects v. For example, suppose that the relevant vertex is v'. Whatever action was taken by the algorithm at v', it (re)inserts e into Y_δ, and then checks for candidate branching points for e (as in Step 3 of procedure A), thereby detecting v. Similar reasoning applies in the other cases.

To complete the proof we must show that the algorithm does not output points that are not vertices of M, i.e., it does not output vertices or intersections of the projections of the objects that are not visible. Consider a point p which reaches the front of Q and is output by the algorithm. If p is an original event point, only Procedure A and Procedure C could have output it and these procedures carry out a visibility test before they output a point. If p is not an original event point, it is a branching point of some edge e. e was visible at the point p' where p was inserted for the final time into Q. If at some point q between p' and p e has disappeared, q must have been a branching point of some nearer edge g. When processing q the implants of e were removed and so was the candidate branching point p of e (step 4 of Procedure E). We conclude that e remains visible at p, thus p is a vertex of M. □

Thus, we obtain the following theorem which is analogous to Theorem 3.3.

Theorem 4.2 *Given a set of n non-intersecting objects satisfying the conditions of Theorem 3.3, the second algorithm computes the visibility map in time $O((U(n)+k)\log^2 n)$, using $O(U(n)\log n)$ working storage.*

Proof. The sweepline stops at $O(U(n)\log n + k)$ event points. The processing of any event point originating from the objects at the leaves or added to the queue as a branching point requires $O(\log^2 n)$ time, while the processing of any other event point (originating from the unions at internal nodes) requires only $O(\log n)$ time. Thus the number of event points whose handling requires $O(\log^2 n)$ time is $O(n+k)$, and the total time complexity of the algorithm is therefore $O((n+k)\log^2 n + U(n)\log n \cdot \log n) = O((U(n)+k)\log^2 n)$.

At any moment during the sweep there are at most $O(n)$ branching points, because every object edge can have at most one branching point in the queue. Thus, the total working storage used by the algorithm is dominated by the total size of the regions U_δ, namely $O(U(n)\log n)$. □

5 Applications

In this section we present the three applications mentioned in the introduction. In the first application we have a set of non-intersecting balls in space viewed from infinity at some direction, say from above. The view of such a set is the same as the view from above of a set

of horizontal disks. The best known result for output-sensitive hidden surface removal in such a set is due to Sharir and Overmars [24] who give a method that runs in time $O(n\sqrt{n}\log n + k)$. In the special case of unit disks considered in [19] a method is given that runs in time $O((n+k)\log^2 n)$. Here we apply our techniques to obtain the same improved running time for the case of disks of arbitrary radii.

To apply our methods we need a bound on the union of a set of n (arbitrary) disks in the plane. It is well-known [11] that such a union has linear complexity, i.e., $U(n) = O(n)$. Now applying Theorem 3.3 or Theorem 4.2 we obtain:

Theorem 5.1 *Given a set of n non-intersecting balls in space, the view of this set from above (or any other direction) can be computed in time $O((n+k)\log^2 n)$, using $O(n\log n)$ storage.*

Note that the bound $U(n) = O(n)$ applies also to *pseudodisks*, i.e. planar regions with the property that the boundaries of any pair of them intersect in at most 2 points. Hence the preceding theorem can be extended to the case of objects whose projections on the viewing plane behave like pseudodisks, assuming the shape of each object is not too complicated.

As an application of this extension, consider the case of a set of n non-intersecting convex homothetic objects (i.e., objects that are translated and scaled copies of a fixed convex object). Here again, the boundaries of the projections of any pair of the objects, in any parallel view, intersect at most twice, so that the union has linear size. The depth ordering can be computed as in the case of balls or disks. Hence we have:

Theorem 5.2 *Given a set of n non-intersecting convex homothetic objects in space, the (parallel) view of this set from any direction can be computed in time $O((n+k)\log^2 n)$, using $O(n\log n)$ storage.*

Next consider a set of horizontal 'fat' triangles viewed in parallel from any direction. A set of triangles is called fat when there exists some positive constant θ such that any internal angle of the triangles is at least θ. For such a set of triangles it is proven by Matoušek et al. [13] that the union has complexity at most $O(n\log\log n)$. Note that the parallel projections of a set of fat triangles need not in general be fat, but it is still the case that the union of any subfamily of n' of these projections has complexity $O(n'\log\log n')$. (To see this, project the triangles in the required direction, but make the viewing plane horizontal.) Hence, we can apply Theorem 3.3 or Theorem 4.2 to obtain the following result:

Theorem 5.3 *Given a set of n horizontal fat triangles, the view of this set from any direction can be computed in time $O((n\log\log n + k)\log^2 n)$, using $O(n\log n\log\log n)$ storage.*

Finally consider the case of a polyhedral terrain Σ with n faces, viewed from some fixed point a lying above it. A *polyhedral terrain* is the graph of a piecewise linear continuous function $z = \Sigma(x, y)$. It has been shown in [6] that the faces of Σ can be ordered by depth with respect to a (although it might be necessary to cut some faces of Σ to ensure that the resulting order is indeed acyclic). Cole and Sharir [6] give an efficient technique for implicitly computing the visibility map. Reif and Sen [23] give an output-sensitive construction of the map that runs in time $O((n+k)\log n\log\log n)$. Their technique, which is based on dynamic ray-shooting in monotone polygonal chains, is fairly complicated. Using our much simpler algorithms we can obtain faster solutions.

To apply our techniques, imagine that we replace Σ by a collection of semi-unbounded vertical prisms, each consisting of all points lying below a face of Σ. Obviously, the visibility map from a does not change by this transformation. The prisms have the fatness property, since the union of the projections of any n' of them has complexity $U(n') = O(n'\alpha(n'))$ (see [6] for details). We can thus apply Theorem 3.3 or Theorem 4.2 to the modified scene. In this case we can even improve the bound on the running time by a factor of $\log n$. Indeed, in the first algorithm, the regions U_δ and V_δ are all monotone polygons, and it is easily checked that each of the Boolean operations on them performed by the algorithm can be done in linear time. In the second algorithm, each Y_δ structure contains only one edge of U_δ (because of the monotonicity). We also claim that each Y_δ contains at most one implanted edge at a time. Indeed, suppose Y_δ contains two such edges, e, e', with e' lying higher than e. Since Σ is a terrain, it is easily checked that the depth ranges between e and its background and between e' and its background are disjoint, which is impossible by definition. Hence the cost of accessing a Y_δ structure is constant. The computation of the sets U_δ can be done in total time $O(n\alpha(n)\log n)$, using the same technique as in [6]. The $O(n\alpha(n)\log n)$ initial events that are put in the priority queue can be obtained in sorted order as the sets U_δ are constructed, so that we can retrieve them from the queue at constant time per event point. Putting all these observations together, it follows that the time of both algorithms can be improved to $O((n\alpha(n)+k)\log n)$. We thus have:

Theorem 5.4 *The visibility map of a polyhedral terrain consisting of n faces, viewed from some fixed point above it, can be computed in time $O((n\alpha(n)+k)\log n)$ and working storage $O(n\alpha(n)\log n)$.*

6 Conclusion

In this paper we have presented two new methods for computing the visibility map of a set of non-intersecting objects in 3-space. They run in time $O((U(n) + k)\log^2 n)$ and use $O(U(n)\log n)$ working storage, where $U(n')$ is the maximum complexity of the union of the projections on a viewing plane of any subset of n' of the objects, and k is the complexity of the output visibility map. The methods are quite simple, apply to general scenes where a depth ordering of the objects is available, and are efficient whenever $U(n)$ is small. This is the case for sets of fat objects like disks (balls), fat triangles, homothets, and polyhedral terrains. This condition might also occur for many sets of non-fat objects. It is also worth noting that for any set of objects $U(n) = O(n+I)$ where I is the number of intersections in the projection. Hence, even for non-fat objects, the time bound is never worse than $O((n+I)\log^2 n)$ which is only a factor $\log n$ worse than the techniques in [25]. Although we did not exploit this observation, it is interesting to note that our techniques also apply when the objects can be split into a small number of subfamilies so that within each subfamily the union complexity is small. An example where this observation can be applied is the case of axis-parallel horizontal rectangles (see [19] for details), although the resulting algorithm would be inferior to the best known solutions for this case.

Of course, the main open problem that remains is to find an output-sensitive algorithm that is efficient for general objects in space. Another open problem is to improve still further our technique. For instance, can the running time be reduced to $O((U(n)+k)\log n)$ (as in the case of polyhedral terrains)?

References

[1] P.K. Agarwal and M. Sharir, Applications of a new space partitioning technique, manuscript (1991).

[2] M. Bern, Hidden surface removal for rectangles, *J. Comp. Syst. Sciences* **40** (1990), 49–69.

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

[4] M.T. de Berg and M.H. Overmars, Hidden surface removal for axis-parallel polyhedra, *Proc. 31st IEEE Symp. on Foundations of Computer Science*, 1990, pp. 252–261.

[5] B. Chazelle, H. Edelsbrunner, L. Guibas, R. Pollack, R. Seidel, M. Sharir and J. Snoeyink, Counting and cutting cycles of lines and rods in space, *Proc. 31st IEEE Symp. on Foundations of Computer Science*, 1990, pp. 242–251.

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

[7] F. Dévai, Quadratic bounds for hidden line elimination, *Proc. 2nd ACM Symp. on Computational Geometry*, 1986, pp. 269–275.

[8] M.T. Goodrich, A polygonal approach to hidden line elimination, *Proc. 25th Allerton Conf. on Communication, Control and Computing*, 1987, pp. 849–858.

[9] M.T. Goodrich, M.J. Atallah and M.H. Overmars, An input-size/output-size trade-off in the time-complexity of rectilinear hidden surface removal, *Proc. ICALP'90*, Springer-Verlag, Lecture Notes in Computer Science 443, 1990, pp. 689–702.

[10] R.H. Güting and T. Ottmann, New algorithms for special cases of the hidden line elimination problem, *Comp. Vision, Graphics and Image Processing* **40** (1987), 188–204.

[11] K. Kedem, R. Livne, J. Pach and M. Sharir, On the union of Jordan regions and collision-free translational motion amidst polygonal obstacles, *Discrete Comput. Geom.* **1** (1986), 59–71.

[12] H. Mairson and J. Stolfi, Reporting and counting intersections between two sets of line segments, *Theoretical Foundations of Computer Graphics and CAD*, R.A. Earnshaw, Ed., NATO ASI Series, Vol F-40, Springer Verlag, 1988, pp. 307–326.

[13] J. Matoušek, J. Pach, M. Sharir, S. Sifrony and E. Welzl, Fat triangles determine linearly many holes, Tech. Report 174/90, Eskenasy Institute of Computer Sciences, Tel Aviv University, May 1990.

[14] M. McKenna, Worst-case optimal hidden surface removal, *ACM Trans. Graphics* **6** (1987), 19–28.

[15] K. Mulmuley, An efficient algorithm for hidden surface removal, I, *Computer Graphics* **23** (1989), 379–388.

[16] O. Nurmi, A fast line-sweep algorithm for hidden line elimination, *BIT* **25** (1985), 466–472.

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

[18] M.H. Overmars and M. Sharir, An improved technique for output-sensitive hidden surface removal, Techn. Rept. RUU-CS-89-32, Dept. of Computer Science, Utrecht University, 1989.

[19] M.H. Overmars and M. Sharir, Merging visibility maps, *Computational Geometry, Theory and Applications* (1991), to appear.

[20] M. Paterson and F.F. Yao, Binary space partitions with applications to hidden surface removal and solid modeling, *Proc. 5th ACM Symp. on Computational Geometry*, 1989, pp. 23–32.

[21] F.P. Preparata and M.I. Shamos, *Computational Geometry, an Introduction,* Springer-Verlag, New York, 1985.

[22] F.P. Preparata, J.S. Vitter and M. Yvinec, Computation of the axial view of a set of isothetic parallelepipeds, *ACM Trans. Graphics* **9** (1990), 278–300.

[23] J. Reif and S. Sen, An efficient output-sensitive hidden surface removal algorithm and its parallelization, *Proc. 4th ACM Symp. on Computational Geometry*, 1988, pp. 193–200.

[24] M. Sharir and M.H. Overmars, A simple output-sensitive algorithm for hidden surface removal, *ACM Trans. Graphics*, 1990, to appear.

[25] A. Schmitt, Time and space bounds for hidden line and hidden surface algorithms, *Eurographics '81*, pp. 43–56.

[26] I.E. Sutherland, R.F. Sproull and R.A. Schumacker, A characterization of ten hidden-surface algorithms, *Computing Surveys* **6** (1974), 1–25.