Merging visibility maps

Mark H. Overmars and Micha Sharir

RUU-CS-90-9
March 1990

Utrecht University

Department of Computer Science

Padualaan 14, P.O. Box 80.089,
3508 TB Utrecht, The Netherlands,
Tel. : ... + 31 - 30 - 531454

Merging visibility maps

Mark H. Overmars and Micha Sharir

Technical Report RUU-CS-90-9
March 1990

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

which object is visible at that pixel. These techniques generally have hardware implementations, but they can still be slow when the screen size and the number of objects in the scene are both large [18].

Other techniques have an "object-space" flavor. That is, they try to obtain 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.

Early object-space methods compute this visibility map by projecting all the edges of the given objects down and computing all their intersections. Crude implementations of this approach run in time $O(n^2)$ [3, 9]. More careful implementations run in time $O((n+I)\log n)$, where I denotes the number of intersections between the projected edges [5]. See also [7, 11, 17]. The problem with these methods is that they are insensitive to the output size of the problem. That is, if the visibility map has k edges, we would prefer an algorithm whose running time depends on k so that when k is small the algorithm becomes more efficient. In all the above-mentioned techniques it is possible that k is very small (even a constant) while I is quadratic in n. Thus all these methods might require quadratic time to produce a trivial output.

There have been a few recent solutions that are truly *output sensitive* in the above sense. Some of these techniques deal with the restricted case in which the objects are all horizontal axis-parallel rectangles, and lead to fairly efficient output-sensitive algorithms [1, 6, 16]. Another output-sensitive algorithm has been proposed by Reif and Sen [14], for the special case of a polyhedral terrain (i.e., a piecewise linear surface meeting each vertical line in exactly one point). Some general results have only recently been obtained. In [15] (see also [12]) a method is described that computes the visibility map for a set of n triangles in time $O(n\sqrt{k}\log n)$ where k is the size of the visibility map. The paper [12] also contains a more sophisticated algorithm with a somewhat improved output-sensitive bound on its running time. Also [10] gives a "quasi-output-sensitive" hidden surface removal method; its running time is a sum of weights associated with all intersections of the projected object edges, where the weight of an intersection decreases as the number of objects hiding it from v increases.

In a number of applications, visibility maps for subsets of the objects are already available. For example, in animation a number of objects move with respect to a fixed environment. Typically, the largest part of the complexity of the scene lies in the static environment. One can easily compute the visibility map of the environment beforehand. After this preprocessing, this map is available and only the maps of the moving objects have to be recomputed. The problem now is to merge the different maps which might be intermixed in depth (see figure 1 for an example). This approach is also useful to obtain realistic images—lighting information, texture, etc. can be computed for each of the faces of the separate visibility maps. After the

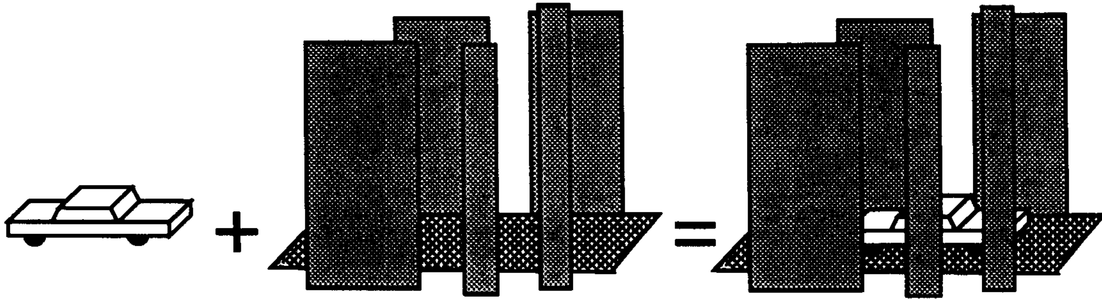

Figure 1: Merging visibility maps.

merge, the appropriate parts of the different maps can be copied without the need to recompute all this information.

The final "merged" scene will consist of parts of the individual maps. In some special cases the problem is relatively easy. For instance, if we merge just two visibility maps, M_1, M_2, and all objects in M_1 are nearer to the viewing point than the objects in M_2, then we can obtain the overall map M as follows. We form the "contour" C_1 of M_1—this is the boundary of the union of the projections of all objects in M_1 and is a portion of the map M_1. We then merge C_1 with M_2, using a plane sweep. This will yield the portion of the overall map that represents objects of M_2 "seen through" objects of M_1. Gluing this portion with faces of M_1 at which objects of the first set are seen, we obtain the overall desired map. We can afford to detect all intersections between edges of C_1 and edges of M_2 because, as is easy to check, each such intersection is a vertex of the final map M. Using the line-sweeping algorithms of [2] or of [8], we can compute M in time $O(n \log n + k)$, where n and k are as above. However, this approach breaks down as soon as the number of maps to be merged is 3 or more, or the maps interleave in depth, so a more sophisticated technique, that manages to avoid having to examine every intersection between map edges, is called for.

In this paper we provide such a technique for the general case of merging visibility maps; it runs in time $O((n + k)z \log^2 n)$, where z is the number of maps to be merged, n is the total size of these maps, and k is the size of the overall visibility map. The method is based on a new kind of "2.5-dimensional" space sweep through all visibility maps simultaneously, related to techniques for computing red-blue line segment intersections of [8], and to the "red-blue merging" technique of [4]. The only assumption our method makes is that a (partial) ordering of the objects by nearness (depth) to the viewing point v exists and is given to us; for example, this is the case when all objects are flat horizontal polygons. (See Paterson and Yao [13] for a recent treatment of this ordering problem.)

The technique can also be used for solving the hidden surface removal problem for particular types of objects. The idea is to split the set of objects into a number

of subsets such that for each subset the individual visibility map is small and can be computed efficiently. We can thus compute the maps for the individual subsets and merge them to obtain the final map in output sensitive manner. We exemplify this idea in two cases. First we show that the visibility map for a set of n horizontal unit discs (or of pairwise disjoint unit spheres) viewed from $z = -\infty$ can be determined in time $O((n + k)\log^2 n)$. Then we show that the visibility map for a set of n horizontal axis-parallel rectangles, viewed as above, can be computed in time $O((n + k)\log^4 n)$. The first result is a substantial improvement over the previous bound of $O(n\sqrt{n}\log n + k)$ given in [12, 15]. The second result is somewhat inferior to the results of [1, 6, 16], where $O((n + k)\log n)$ algorithms are given, but the method is completely different and may have other applications as well.

The paper is organized as follows. In Section 2 we describe our main result of merging visibility maps. In Section 3 we present the applications just mentioned. Finally, in Section 4 we give some conclusions and open problems.

2 The merging method

In this section we describe our main algorithm for merging a number of different visibility maps. Let V be the set of objects in space. We assume that the objects are convex, pairwise disjoint, and have simple shape, so that basic operations, such as computing the intersection between the projections of two objects, can be performed in constant time. Without loss of generality we assume that the viewing point is at $z = -\infty$, which means that we are looking at the objects from below and that they are projected orthogonally on the viewing plane. We also assume that a depth ordering on V exists and is given to us *a priori*. More specifically, we define a binary relation on V by saying that object O_1 lies below object O_2 if there is a vertical line that intersects both objects and its intersection with O_1 lies below its intersection with O_2. We assume that this relation is acyclic. This is the case, for example, when all objects are flat and lie parallel to the xy plane, or when they are pairwise disjoint spheres. In both cases (a linear extension of) the depth ordering can easily be computed in time $O(n\log n)$.

Let $V_1, \ldots, V_z$ form a partition of V into pairwise disjoint subsets, having respective visibility maps $M_1, \ldots, M_z$. Each visibility map is a planar map drawn in the xy-plane. Its faces are maximal connected regions of the plane, in each of which a single object (or no object at all) is seen. The edges of each map are projections of (maximal) visible connected portions of the *silhouettes* of the objects. We will say that edge e *belongs* to object O if e is a portion of the projected silhouette of O. The vertices of each map are points of intersection between two projected visible silhouettes. Assuming general position of the objects, each map vertex p is of degree 3 and looks like a "T-junction". In the special case that all objects are horizontal polygons, their silhouettes are their boundaries. In this case it is convenient to introduce the projections of the object vertices as additional, degree 2, map vertices.

We assume that the map edges are all x-monotone; if not we can always add extra vertices to split the edges into x-monotone pieces. The simplicity of the objects is also assumed to imply that the number of such extra vertices per edge is constant, and that any pair of edges from different visibility maps M_i, M_j intersect at most a constant number of times and that these intersections can be found in constant time. Let M be the visibility map of V, and let k denote the *size* of M, which we will take to mean the number of vertices of M; by Euler's equation, the number of faces and edges of M is also linear in k. Let $n = |M_1| + \cdots + |M_z|$ be the total size of all visibility maps M_j. We will show how to compute M from $M_1, \ldots, M_z$ in time $O((n + k)z \log^2 n)$. Note that z, the number of different maps, is not considered to be a constant!

The idea behind the method is the following: We perform a plane sweep from left to right through all visibility maps simultaneously. Whenever the sweepline is at some position L the part of M to the left of L will already have been computed. In fact, we will only show how to compute the vertices of M. The edges and faces can easily be subsequently computed in time $O((n + k)z)$ using the different input visibility maps, or can be maintained during the sweep by a slight modification of the algorithm. Note that the vertices of M are of two different types. They are either vertices of one of the visibility maps M_i or they correspond to visible "T–junction" intersections between edges of different visibility maps.

Rather than maintaining one structure with the sweepline we maintain a separate structure T_i for each visibility map M_i. In T_i we maintain the intersection of the sweepline L with M_i, plus other auxiliary data 'implanted' from other maps. Actually, because the depth relationship between objects is crucial to the calculation of M, we will maintain each T_i as a binary tree, whose leaves represent the objects in V_i ordered by depth, so that each internal node δ is associated with a substructure of T_i that corresponds to the subset of objects of V_i stored at the leaves of the subtree rooted at δ. To determine visible intersections between edges of visibility maps we also insert certain edges of one visibility map into the trees corresponding to other maps, in a manner to be described below.

In more detail, T_i is a binary tree that stores the objects in V_i in its leaves, ordered by depth (we assumed such an order exists and is available to us). With each node δ in T_i we maintain a structure S^i_δ that stores the cross-section of L with the edges of M_i that belong to objects in the subtree rooted at δ. Note that each edge e of map M_i might be stored in up to $O(\log n)$ S^i_δ structures (at all nodes on the path in T_i towards the leaf corresponding to the object to which e belongs). S^i_δ is a simple balanced search tree storing the edges in the order in which they are intersected by the sweepline. So, in fact, every S^i_δ represents a separate sweep structure for the subset of M_i belonging to a particular depth range. In addition, we also store in each S^i_{root} information about the objects of V_i that are visible in M_i along L between adjacent edges. This extra information is required to determine whether vertices of one visibility map are hidden by the objects in the other maps. See figure 2 for an example of a T-structure. The objects in the scene are indicated

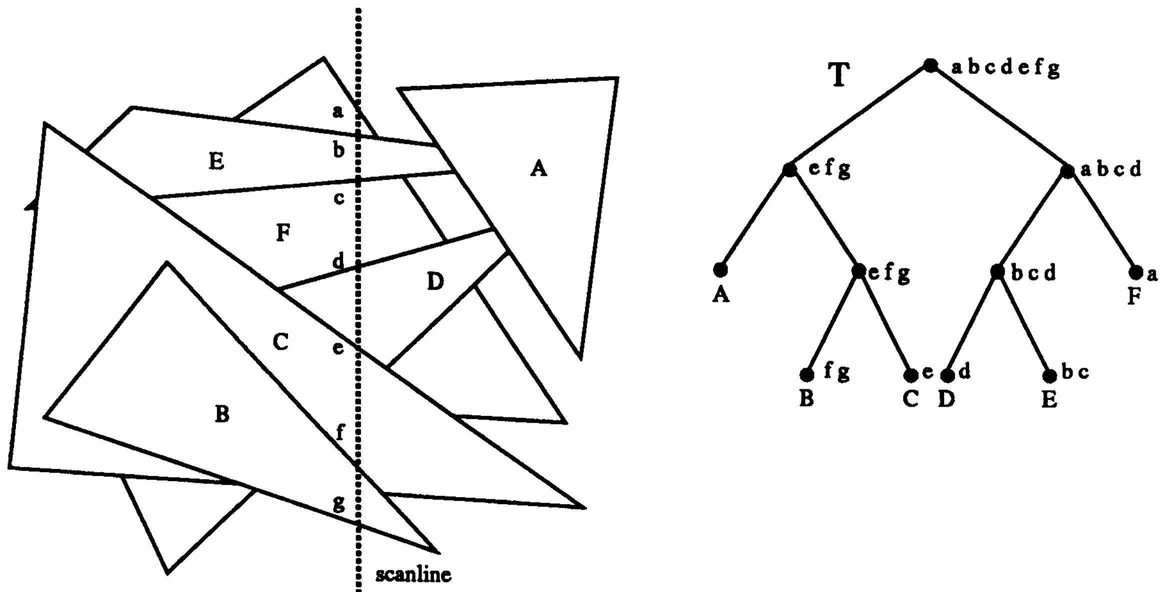

Figure 2: A T-structure.

by uppercase letters, A being the nearest and F the furthest away; the edges of the visibility map are indicated by lowercase letters. At each node of the tree the edges to be stored in the associated structure are indicated.

As event points in our sweep we take initially all vertices of the visibility maps M_i. These are stored, sorted by x-coordinate, in an event queue Q. Later we will add other intersection events to Q. Just maintaining the trees T_i is accomplished as follows. Each initial event point p is a vertex of some map M_i, and requires the insertion or deletion of each edge of M_i incident to p in the corresponding tree T_i. Let e be one of these edges. We search with e in T_i towards the object to which e belongs. For each node δ on the search path we insert or delete e in the tree S^i_δ. This clearly takes time $O(\log n)$ per node δ and, hence, $O(\log^2 n)$ time in total. As there are a total of n edges in all visibility maps and each edge is inserted and deleted once, this yields a total time bound of $O(n \log^2 n)$.

Of course this does not compute M, so we have to do some additional work. As noted above, vertices of M are either vertices of some M_i or are visible T–junctions between edges in different maps. To find the visible vertices of the maps, whenever we reach an event point p being a vertex of some map M_i, we search with p in each S^j_{root} for all $j \neq i$ to find the lowest object of each V_j seen at p. If none of these objects covers p, p is visible and can be reported. This clearly takes time $O(z \log n)$ per node and, hence, at most $O(nz \log n)$ in total.

It remains to determine the visible intersections between edges in different maps. Let p be such an intersection between edge e_i in M_i and edge e_j in M_j. Let e_i be the

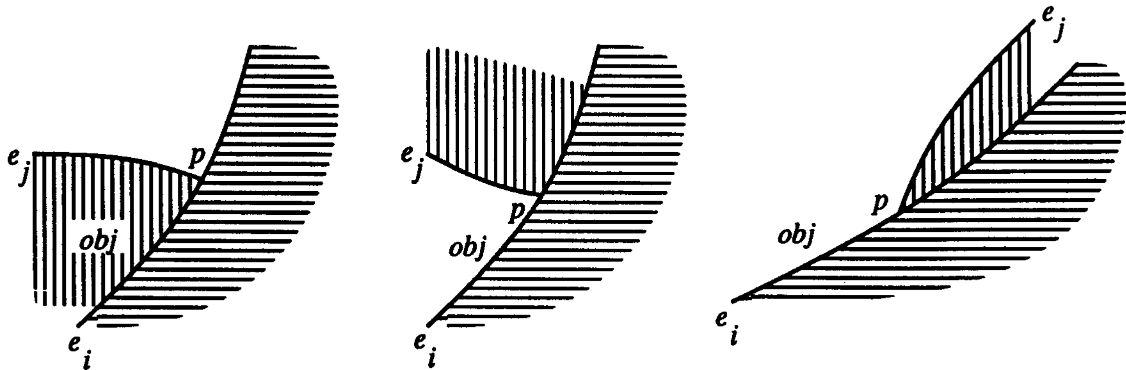

Figure 3: Different types of intersections (looking from below).

lowest of the two edges. So either e_j starts being visible at this point in the sweep or e_j stops being visible (see figure 3). Our technique will be such that the lowest edge e_i will be "responsible" for finding p. To ensure this property, we will insert a visible edge in M (like e_i) into all other trees T_j for $j \neq i$. The method is based on the following lemma that is easy to prove (see also figure 3):

Lemma 2.1 *An intersection p between edges $e_i \in M_i$ and $e_j \in M_j$ $(i \neq j)$ with e_i below e_j is a vertex of M if and only if the following holds: Let obj be the object in V that lies directly above e_i just before the sweepline reaches p. Then e_j must be the projection of the silhouette of either obj or of an object at depth between obj_i and obj, where e_i belongs to obj_i. Moreover e_i must be visible.*

We can thus proceed as follows. First we give a somewhat informal overview of the technique. For each edge e_i that is visible in M and intersected by the sweepline, we compute and maintain the lowest object *obj* above it. This can be done when e_i is inserted into the sweep structures, in time $O(z \log n)$, as follows. Let p be the left endpoint of e_i. In each tree T_j, for $j \neq i$, search with p in the structure S^j_{root} associated with the root, to find the lowest object of V_j that lies at position p. Collecting all these objects over all trees and choosing the lowest one yields the desired object *obj*. (We also have to query M_i, but there it suffices to inspect the two faces of M_i adjacent to e_i.) Assuming e_i is visible at this point, we insert it into all trees T_j $(j \neq i)$ in the following way: We search with the depth of e_i and with the depth of *obj* in T_j to determine those $O(\log n)$ nodes δ of T_j that lie in between the two respective search paths, so that their fathers lie on the search paths (see figure 4). The structures S^j_δ associated with these nodes contain all edges e_j belonging to objects of V_j at depth between e_i and *obj*, as required in the lemma above; moreover, if we restrict ourselves to edges of M_j in the structures of T_j, then each edge appears in exactly one of these subtrees. We insert e_i into each of these structures and check

whether any of its two neighbors in the structure represents an edge of an object in V_j and whether it intersects e_i to the right of the sweepline. In this way we find in each tree T_j at most $O(\log n)$ different intersections. Of all these intersections in all trees we take the leftmost and add it to Q as a new event point. This point is the first visible intersection for e_i, based on the information we have at this moment. We call this intersection the ***branching point*** of e_i and we call the edge e_j that forms this intersection with e_i the ***branch*** of e_i. Later on, a new, nearer branching point might be found, so we leave e_i in the different S-structures into which it has been inserted. Note that e_i might have a neighbor $e_{j'}$ in some S^j-structure that was also inserted there from another map $M_{j'}$; in fact, e_i and $e_{j'}$ could be neighbors in many S-structures. This is allowed by the algorithm (although no intersection between them is computed in this case) and does not cause too much difficulty — see below for details.

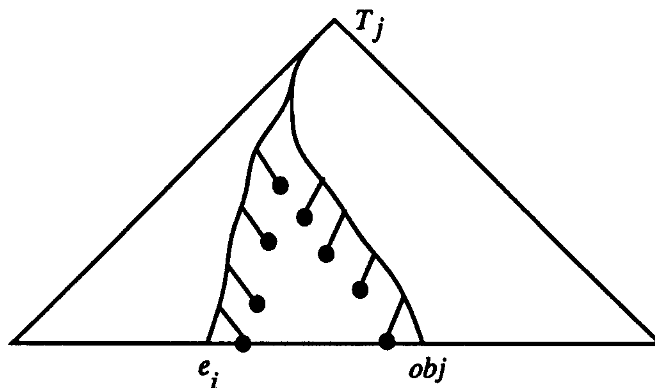

Figure 4: The nodes δ where e_i must be inserted.

So there are three different types of event points: left endpoints of edges, right endpoints of edges and branching points (visible intersections). We now describe in more detail the different steps that have to be taken when the sweepline reaches any of these event points.

To initialize the sweep, we put all left and right endpoints of map edges into the event queue Q, and initialize the trees T_i with empty associated S-structures at each node. In each step of the sweep we remove from the queue Q the leftmost event point p and perform one of the following steps.

To argue that the following procedure is correct, we claim that it maintains the following properties at any instance during the sweep.

(i) Each edge e_i of M_i that is visible along the sweepline is stored at every set S^j_δ, such that $j \neq i$, the depth range of the original edges of S^j_δ lies between the depths of e_i and of its "background" object *obj* as defined above, and the father of δ in T_j does not have this property.

(ii) Any visible intersection between an edge $e_i \in M_i$ and an edge $e_j \in M_j$ is detected and added to Q as soon as (a) the sweep has already reached the rightmost of the left endpoints of both edges, (b) the lower of these two edges, say e_i, is adjacent to e_j in the appropriate structure S_δ^j where they are both stored, and (c) there are no other visible intersections along the lower edge e_i between this point and the sweepline. Moreover, once all these conditions are met, this intersection will not be removed from Q until the sweepline reaches it.

Left endpoint of an edge $e_i \in M_i$.

1. *Determine whether e_i is visible.* This can be done, as explained above, by querying the various structures S_{root}^j for all $j \neq i$, in time $O(z \log n)$.

2. *Add e_i to T_i.* Search with e_i in T_i. For each node δ on the search path add e_i to S_δ^i. Check each of the two neighbors of e_i in S_δ^i. If such a neighbor e does not belong to M_i, check whether e_i intersects it (e_i might be a nearer branch for e; note that by construction all "foreign" edges in S_δ^i precede all original edges in that set in depth order). If so and if the intersection lies before the current nearest branching point of e, remove that branching point from Q and add the new intersection instead. The total time for this step is $O(\log^2 n)$.

3. *If e_i is visible find its nearest branching point.* Let *obj* be the lowest object above e_i (we can get this data from the first step). Insert e_i in all structures T_j ($j \neq i$) according to the condition in property (i) above. If e_i has been inserted into a structure S_δ^j, check its two neighbors there and, if they are original edges of M_j, obtain from them new candidate branching points for e_i as described above. This takes time $O(z \log^2 n)$. The nearest branching point for e_i found (if any) is added to Q.

The total time for an insertion is $O(z \log^2 n)$.

Right endpoint of an edge $e_i \in M_i$.

1. *Remove e_i from T_i.* Search with e_i in T_i. For each node δ on the search path delete e_i from S_δ^i. Check the two former neighbors of e_i in S_δ^i. If one of them is in M_i and the other is not, this might introduce a new nearest branching point. We check this and replace the previous branching point for the lower neighbor (which is necessarily the "foreign" edge) in Q by the new one, if necessary. The time required is $O(\log^2 n)$.

2. *If e_i is visible remove it from the other structures.* So we remove e_i from $O(z \log n)$ S-structures. In each of these structures new neighbors appear that, if in different maps, have to be checked for new nearest branching points. This takes time $O(\log n)$ per S-structure.

Thus a deletion takes time $O(z \log^2 n)$ as well.

Branching point.

A branching point can easily be viewed as one (or two) deletions followed by two (or one) insertions (where the lower edge e_i will be deleted and reinserted). We can thus implement this step as a combination of insertion and deletion steps, as explained above. Hence this step also requires time $O(z \log^2 n)$. (Note that re-insertion of e_i is an essential step of the algorithm and is not done merely for convenience — the object *obj* below e_i will generally change at a branching point, so e_i will have to be moved to different S-structures to preserve property (i).) Finally, the branching point is reported as a new vertex of M.

We claim that properties (i) and (ii) are maintained throughout the execution of the sweep. Property (i) is easily seen to hold by construction. As to property (ii), we will prove it by induction on the left-to-right order of the new vertices of M. Let p be a visible intersection between a lower edge $e_i \in M_i$ and a higher edge $e_j \in M_j$. Let S_δ^j be the unique S-structure in T_j where both edges are stored. Clearly, slightly to the left of p all conditions (a)–(c) of (ii) will be satisfied for this pair. Let us move to the left as much as possible until the first time that one of these conditions fails. Let L be the position of the sweepline when this happens.

If (a) fails, we are at the point of inserting e_i or e_j; since (b) still holds, p will be detected and (c) ensures that p is added to Q (and also that it is not removed later until the sweep reaches p).

If (b) fails, a third edge e_k, that did lie between e_i and e_j slightly to the left of L, has either just intersected e_i or e_j or has its right endpoint on L. In the later case our deletion step will detect p and add it to Q. In the former case, suppose first that $e_k \in M_j$, so that it had to intersect e_i (and not e_j) at some point $q \in L$. Since e_i is visible at this point and e_k lies between e_i and its background object *obj* (by property (i)), it follows that q is visible, so our induction hypothesis implies that q will be detected and processed as a branching point for e_i. Since e_i is re-inserted at this point, the intersection p with its new neighbor e_j will be detected. It remains to consider the case where e_k does not belong to M_j. In this case both e_i and e_k are foreign edges, necessarily lying below all original edges of S_δ^j. If e_k intersects e_j on L then again this must be a branching point for e_k which, by induction, is detected and processed at L. This will cause re-insertion of e_j, at which the point p will be detected. Finally, suppose e_k intersects e_i at some $q \in L$. Since both e_i and e_k are visible at this point, and since e_i is visible to the right of L, it follows that q is a visible intersection and a branching point for e_i. Thus by induction q is detected and processed (in a different tree T_k), which causes e_i to be re-inserted, leading again to the detection of p.

It remains to consider the case where (c) fails at L. Thus e_i has a branching point q on L. By induction, this point will be detected and processed at L, which

will cause e_i to be reinserted, so that p will be detected at this time.

This completes the proof of correctness of the procedure. Since there are $O(n)$ left and right endpoints and k branching points, we obtain the following summary result:

Theorem 2.2 *Let V be a set of objects in 3-space with the properties assumed above. Let $V_1, \ldots, V_z$ form a partition of V into subsets with respective visibility maps $M_1, \ldots, M_z$, having total size n. The visibility map M of V can be computed from $M_1, \ldots, M_z$ in time $O((n+k)z \log^2 n)$, using $O(nz \log n)$ storage.*

3 Applications

In this section we will show how to use the method described above to solve some special instances of the hidden surface removal problem. The idea here is to split the set of objects into a number of different subsets such that for each subset the visibility map has a small size. If we can compute the visibility map for each of the subsets efficiently, then we can obtain the overall map by merging the separate maps using the result of the previous section.

First we apply the method to a set of horizontal unit discs (or of unit spheres) viewed from $z = -\infty$. The previously best known result for this case runs in time $O(n\sqrt{n} \log n + k)$ (see [15]). Note that the size of the visibility map of a set of unit discs can still be as large as $\Omega(n^2)$.

Let V be a set of n unit discs, parallel to the xy-plane. We divide (conceptually) the xy plane into a grid of small squares of size $1/2 \times 1/2$ each. For each disc we determine the squares that intersect its xy projection. In this way we get a set of $O(n)$ (partially) covered squares and for each square sq we have a set V_{sq} of discs whose projections (partially) cover it. As the projection of each disc (partially) covers only $O(1)$ squares, the total size of all sets is linear. For each square sq we determine the lowest disc c_{sq} whose projection completely covers sq (if any). From V_{sq} we remove c_{sq} and all discs that lie above c_{sq} (they will not be visible in sq). c_{sq} will be the background disc in sq (or, if no such disc exists, there is no background disc).

Now each V_{sq} consists of discs partially covering sq and, when computing the visibility map of each V_{sq} restricted to sq (filling the open areas with the background disc), they together form the visibility map of the entire set of discs.

So we now restrict our attention to computing the visibility map of a single set V_{sq}, restricted to sq. The crucial observation is that no disc in V_{sq} can have its center inside sq. We split V_{sq} into four subsets: V_l contains the discs with center to the left of sq, V_r contains the discs with center to the right of sq, V_a contains the remaining discs with center above sq and V_b contains the remaining discs with center below sq.

Lemma 3.1 *Let V be a set of horizontal unit discs whose xy-projections have centers to the left of a line ℓ. The visibility map of V to the right of ℓ has linear*

size.

Proof. The result is based on the easy observation that the boundaries of each pair of unit discs can intersect at most once to the right of ℓ. As a result each disc boundary can contribute at most one arc to the visibility map to the right of ℓ. If not, there would be some disc D whose bounding circle C contributes at least two arcs to the visibility map. Between these arcs C must be covered by at least one other disc, D'. This however is impossible, because the boundaries of D' and D can intersect at most once to the right of ℓ. □

As a result, for each of the sets V_l, V_r, V_a and V_b the visibility map inside sq has linear size. These maps can be computed in time $O(n' \log^2 n')$, where n' is the size of V_{sq}, using the following divide and conquer technique. We describe this procedure for the set V_l. Partition V_l into two subsets of roughly equal size, V_l^1, V_l^2, so that all discs in V_l^1 are lower than all discs in V_l^2. Recursively compute the visibility maps M_1, M_2 of V_l^1, V_l^2 respectively. Let C_1 be the contour of M_1, as defined in the introduction. We then merge C_1 with M_2 to obtain the portion of the overall visibility map representing discs of V_l^2 "seen through" the discs of V_l^1. Gluing this map to M_1 yields the overall map. The merge is accomplished by a sweep, making use of the observation (also noted in the introduction) that every intersection between an edge of C_1 and an edge of M_2 must be a vertex of the overall map. Since all these maps have linear size, the merging can be done in time $O(n' \log n')$, so the overall algorithm takes time $O(n' \log^2 n')$.

The four visibility maps of the subsets V_l, V_r, V_a, V_b can then be merged in time $O(n' + k') \log^2 n')$ using Theorem 2.2, where k' is the size of the visibility map inside sq. since the sum of the k' for all squares is k and the sum of the n' is n we obtain:

Theorem 3.2 *Given a set of n unit discs, parallel to the xy-plane, their visibility map, when viewed from $z = -\infty$, can be determined in time $O((n+k) \log^2 n)$, where k is the size of the map.*

This technique of partitioning a set of objects into subsets with small individual visibility maps can also be applied to other types of objects. As an example, consider hidden surface removal in a set of axis-parallel horizontal rectangles, again viewed from $z = -\infty$. We will show how to split a set of n such rectangles into $O(\log^2 n)$ subsets such that in each subset the visibility map has size linear in the number of rectangles in the subset. This is based on the following lemma which is easy to prove.

Lemma 3.3 *Let V be a set of n axis-parallel horizontal rectangles whose projections all contain a common point p. Then the visibility map of V has linear size and can be computed in time $O(n \log n)$.*

Our approach will be to split the set of rectangles into subsets V_i, such that each V_i is a union of smaller subsets W_{ij}, so that (i) all rectangles in a single subset W_{ij} have a point in common, and (ii) for each fixed i, the subsets W_{ij} are separated from each other by parallel lines. The above lemma implies that the visibility map of each subset V_i is linear.

Let us first consider a set V of rectangles all intersecting a vertical line ℓ. We can split V into $O(\log n)$ subsets each with a linear-size visibility map as follows. Let p be a point on ℓ such that at most half of the rectangles lie completely above p and at most half lie completely below p. Split V into three sets. A_1 is the set of rectangles that contain p. V_1 is the set of rectangles above p and V_2 the set of rectangles below p. A_1 is our first subset with linear visibility map. Split V_1 by a point p_1 on ℓ into two subsets in the same manner that V has been split by p, and split V_2 similarly by a point p_2. A_2 contains the rectangles in V_1 that contain p_1 and the rectangles in V_2 that contain p_2. Because V_1 and V_2 are separated, the visibility map of A_2 will again be linear. We are left with four subsets of rectangles. Each of these is split into two subsets by a point as above, and A_3 is taken to contain the rectangles that contain any of these four points, and so on. In this way we obtain at most $\log n$ sets, each with linear visibility map.

Let us now consider a set of arbitrary (axis-parallel) rectangles. Let ℓ be a vertical line such that at most half the rectangles lie completely to the left of ℓ and at most half the rectangles lie completely to the right of ℓ. Let A^1 be the set of rectangles that intersect ℓ. Let V_1 be the set of rectangles to the left of ℓ and V_2 the set of rectangles to the right of ℓ. We split A^1 in the manner described above into $O(\log n)$ subsets $A_1^1, A_2^1, A_3^1, \ldots$. Then we split V_1 with a vertical line ℓ_1 and V_2 with a vertical line ℓ_2 into (at most) halves. Let B^1 be the set of rectangles that intersect ℓ_1 and B^2 the set of rectangles that intersect ℓ_2. Both sets are split, in the manner described above, into $O(\log n)$ subsets, $B_1^1, B_2^1, \ldots$ and $B_1^2, B_2^2, \ldots$ respectively. Then we merge B_i^1 with B_i^2 to obtain sets A_i^2, for $i = 1, 2, \ldots$. It is easily seen that each set A_i^2 has linear visibility map. We are now left with four sets of rectangles. Each is split with a vertical line, the sets of rectangles that intersect each line are split into $O(\log n)$ subsets, and we merge quadruplets of these subsets to obtain subsets $A_1^3, A_2^3, \ldots$ In this way we obtain $O(\log n)$ groups of $O(\log n)$ subsets, each with linear visibility map. The whole subdivision can easily be constructed in time $O(n \log^2 n)$.

Lemma 3.4 *Let V be a set of n axis-parallel horizontal rectangles viewed from $z = -\infty$. V can be split in time $O(n \log^2 n)$ into $O(\log^2 n)$ subsets such that for each subset the visibility map has linear size.*

Now we can apply Theorem 2.2 to obtain the following result:

Theorem 3.5 *Given a set of n axis-parallel rectangles as above, their visibility map of size k can be constructed in time $O((n + k) \log^4 n)$.*

This result is clearly much worse than the results in [1, 6, 16], where $O((n + k) \log n)$ algorithms are given, but the method is completely different.

4 Conclusion

In this paper we have presented a method for merging visibility maps efficiently. The result is important e.g. in computer animation when objects move in a fixed environment. The result also has applications for some special cases of the hidden surface removal problem.

A number of open problems do remain. First of all it might be possible to improve the running time of the algorithm, e.g. by applying certain fractional cascading techniques on the data structures used. Another challenge is to find other special cases (besides unit discs and axis-parallel rectangles) of the hidden surface removal problem that can be solved efficiently using this method. In particular, is it always possible to split a set of arbitrary sized discs into (a small number of) subsets with low complexity of the individual visibility maps.

References

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

[2] B. Chazelle and H. Edelsbrunner, An optimal algorithm for intersecting line segments in the plane, *Proc. 29th IEEE Symp. on Foundations of Computer Science*, 1988, pp. 590–600.

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

[4] H. Edelsbrunner, L. Guibas and M. Sharir, The complexity and construction of many faces in arrangements of lines or of segments, *Discrete Comput. Geom.* **5** (1990), 161–196.

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

[6] 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*, 1990, to appear.

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

[8] 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.

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

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

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

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

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

[14] 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.

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

[16] F.P. Preparata, J.S. Vitter and M. Yvinec, Computation of the axial view of a set of isothetic parallelepipeds, Techn. Rep. LIENS-88-1, Lab. d'Informatique de L'Ecole Normal Supérieure, 1988.

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

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

Merging visibility maps

Mark H. Overmars and Micha Sharir

RUU-CS-90-9
March 1990

Utrecht University

Department of Computer Science

Padualaan 14, P.O. Box 80.089,
3508 TB Utrecht, The Netherlands,
Tel. : ... + 31 - 30 - 531454

Merging visibility maps

Mark H. Overmars and Micha Sharir

Technical Report RUU-CS-90-9
March 1990

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

Merging Visibility Maps*

Mark H. Overmars[†] Micha Sharir[‡]

Abstract

Let V be a set of objects in space for which we want to determine the portions visible from a particular point of view v. Assume V is subdivided in subsets $V_1, \ldots, V_z$ and the *visibility maps* $M_1, \ldots M_z$ of these subsets from point v are known. We show that the visibility map M for V can be computed by merging $M_1, \ldots, M_z$ in time $O((n+k)z \log^2 n)$ where n is the total size (number of edges, vertices and faces) of the visibility maps $M_1, \ldots, M_z$ and k is the size of M.

This result has important applications e.g. in animation where objects move with respect to a fixed environment. It also leads to efficient algorithms for special cases of the hidden-surface removal problem. For example, we obtain a method for hidden surface removal in a set of unit spheres, viewed from infinity, that runs in time $O((n+k)\log^2 n)$.

1 Introduction

An important problem in computer graphics is *hidden surface removal.* In a typical setting of the problem we are given a collection of 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.

Many solutions have been developed to date. Some of them use an "image-space" approach, in which one tries to calculate, for each pixel in the viewed image,

---

*Work by the first author has been partially supported by the ESPRIT II Basic Research Actions Program of the EC, under contract No. 3075 (project ALCOM). Work by the second author has been partially supported by Office of Naval Research Grant N00014-87-K-0129, by National Science Foundation Grant CCR-89-01484, and by grants from the U.S.-Israeli Binational Science Foundation, the NCRD - the Israeli National Council for Research and Development, and the Fund for Basic Research administered by the Israeli Academy of Sciences. Both authors wish to acknowledge additional support by DIMACS, a Science and Technology Center under National Science Foundation Grant STC-88-09648.

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

‡School of Mathematical Sciences, Sackler Faculty of Exact Sciences, Tel Aviv University, 69978 Tel Aviv, Israel, and Courant Institute of Mathematical Sciences, New York University, 251 Mercer Street, New York, NY 10012, U.S.A.

which object is visible at that pixel. These techniques generally have hardware implementations, but they can still be slow when the screen size and the number of objects in the scene are both large [18].

Other techniques have an "object-space" flavor. That is, they try to obtain 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.

Early object-space methods compute this visibility map by projecting all the edges of the given objects down and computing all their intersections. Crude implementations of this approach run in time $O(n^2)$ [3, 9]. More careful implementations run in time $O((n+I)\log n)$, where I denotes the number of intersections between the projected edges [5]. See also [7, 11, 17]. The problem with these methods is that they are insensitive to the output size of the problem. That is, if the visibility map has k edges, we would prefer an algorithm whose running time depends on k so that when k is small the algorithm becomes more efficient. In all the above-mentioned techniques it is possible that k is very small (even a constant) while I is quadratic in n. Thus all these methods might require quadratic time to produce a trivial output.

There have been a few recent solutions that are truly *output sensitive* in the above sense. Some of these techniques deal with the restricted case in which the objects are all horizontal axis-parallel rectangles, and lead to fairly efficient output-sensitive algorithms [1, 6, 16]. Another output-sensitive algorithm has been proposed by Reif and Sen [14], for the special case of a polyhedral terrain (i.e., a piecewise linear surface meeting each vertical line in exactly one point). Some general results have only recently been obtained. In [15] (see also [12]) a method is described that computes the visibility map for a set of n triangles in time $O(n\sqrt{k}\log n)$ where k is the size of the visibility map. The paper [12] also contains a more sophisticated algorithm with a somewhat improved output-sensitive bound on its running time. Also [10] gives a "quasi-output-sensitive" hidden surface removal method; its running time is a sum of weights associated with all intersections of the projected object edges, where the weight of an intersection decreases as the number of objects hiding it from v increases.

In a number of applications, visibility maps for subsets of the objects are already available. For example, in animation a number of objects move with respect to a fixed environment. Typically, the largest part of the complexity of the scene lies in the static environment. One can easily compute the visibility map of the environment beforehand. After this preprocessing, this map is available and only the maps of the moving objects have to be recomputed. The problem now is to merge the different maps which might be intermixed in depth (see figure 1 for an example). This approach is also useful to obtain realistic images—lighting information, texture, etc. can be computed for each of the faces of the separate visibility maps. After the

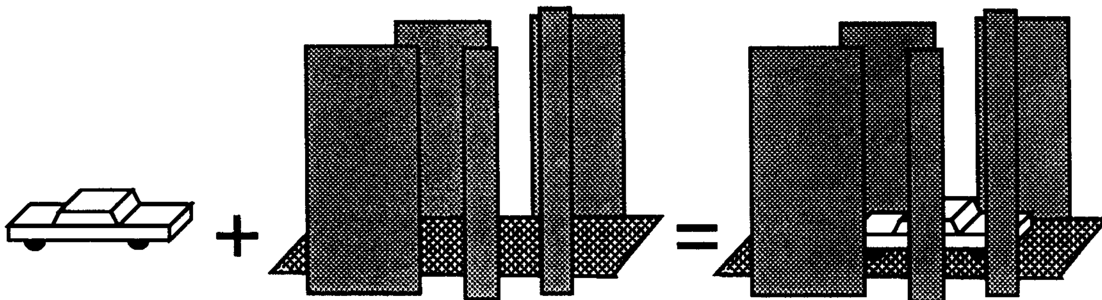

Figure 1: Merging visibility maps.

merge, the appropriate parts of the different maps can be copied without the need to recompute all this information.

The final "merged" scene will consist of parts of the individual maps. In some special cases the problem is relatively easy. For instance, if we merge just two visibility maps, M_1, M_2, and all objects in M_1 are nearer to the viewing point than the objects in M_2, then we can obtain the overall map M as follows. We form the "contour" C_1 of M_1—this is the boundary of the union of the projections of all objects in M_1 and is a portion of the map M_1. We then merge C_1 with M_2, using a plane sweep. This will yield the portion of the overall map that represents objects of M_2 "seen through" objects of M_1. Gluing this portion with faces of M_1 at which objects of the first set are seen, we obtain the overall desired map. We can afford to detect all intersections between edges of C_1 and edges of M_2 because, as is easy to check, each such intersection is a vertex of the final map M. Using the line-sweeping algorithms of [2] or of [8], we can compute M in time $O(n \log n + k)$, where n and k are as above. However, this approach breaks down as soon as the number of maps to be merged is 3 or more, or the maps interleave in depth, so a more sophisticated technique, that manages to avoid having to examine every intersection between map edges, is called for.

In this paper we provide such a technique for the general case of merging visibility maps; it runs in time $O((n + k)z \log^2 n)$, where z is the number of maps to be merged, n is the total size of these maps, and k is the size of the overall visibility map. The method is based on a new kind of "2.5-dimensional" space sweep through all visibility maps simultaneously, related to techniques for computing red-blue line segment intersections of [8], and to the "red-blue merging" technique of [4]. The only assumption our method makes is that a (partial) ordering of the objects by nearness (depth) to the viewing point v exists and is given to us; for example, this is the case when all objects are flat horizontal polygons. (See Paterson and Yao [13] for a recent treatment of this ordering problem.)

The technique can also be used for solving the hidden surface removal problem for particular types of objects. The idea is to split the set of objects into a number

of subsets such that for each subset the individual visibility map is small and can be computed efficiently. We can thus compute the maps for the individual subsets and merge them to obtain the final map in output sensitive manner. We exemplify this idea in two cases. First we show that the visibility map for a set of n horizontal unit discs (or of pairwise disjoint unit spheres) viewed from $z = -\infty$ can be determined in time $O((n + k)\log^2 n)$. Then we show that the visibility map for a set of n horizontal axis-parallel rectangles, viewed as above, can be computed in time $O((n + k)\log^4 n)$. The first result is a substantial improvement over the previous bound of $O(n\sqrt{n}\log n + k)$ given in [12, 15]. The second result is somewhat inferior to the results of [1, 6, 16], where $O((n + k)\log n)$ algorithms are given, but the method is completely different and may have other applications as well.

The paper is organized as follows. In Section 2 we describe our main result of merging visibility maps. In Section 3 we present the applications just mentioned. Finally, in Section 4 we give some conclusions and open problems.

2 The merging method

In this section we describe our main algorithm for merging a number of different visibility maps. Let V be the set of objects in space. We assume that the objects are convex, pairwise disjoint, and have simple shape, so that basic operations, such as computing the intersection between the projections of two objects, can be performed in constant time. Without loss of generality we assume that the viewing point is at $z = -\infty$, which means that we are looking at the objects from below and that they are projected orthogonally on the viewing plane. We also assume that a depth ordering on V exists and is given to us *a priori*. More specifically, we define a binary relation on V by saying that object O_1 lies below object O_2 if there is a vertical line that intersects both objects and its intersection with O_1 lies below its intersection with O_2. We assume that this relation is acyclic. This is the case, for example, when all objects are flat and lie parallel to the xy plane, or when they are pairwise disjoint spheres. In both cases (a linear extension of) the depth ordering can easily be computed in time $O(n \log n)$.

Let $V_1, \ldots, V_z$ form a partition of V into pairwise disjoint subsets, having respective visibility maps $M_1, \ldots, M_z$. Each visibility map is a planar map drawn in the xy-plane. Its faces are maximal connected regions of the plane, in each of which a single object (or no object at all) is seen. The edges of each map are projections of (maximal) visible connected portions of the *silhouettes* of the objects. We will say that edge e *belongs* to object O if e is a portion of the projected silhouette of O. The vertices of each map are points of intersection between two projected visible silhouettes. Assuming general position of the objects, each map vertex p is of degree 3 and looks like a "T–junction". In the special case that all objects are horizontal polygons, their silhouettes are their boundaries. In this case it is convenient to introduce the projections of the object vertices as additional, degree 2, map vertices.

We assume that the map edges are all x-monotone; if not we can always add extra vertices to split the edges into x-monotone pieces. The simplicity of the objects is also assumed to imply that the number of such extra vertices per edge is constant, and that any pair of edges from different visibility maps M_i, M_j intersect at most a constant number of times and that these intersections can be found in constant time. Let M be the visibility map of V, and let k denote the *size* of M, which we will take to mean the number of vertices of M; by Euler's equation, the number of faces and edges of M is also linear in k. Let $n = |M_1| + \cdots + |M_z|$ be the total size of all visibility maps M_j. We will show how to compute M from $M_1, \ldots, M_z$ in time $O((n+k)z\log^2 n)$. Note that z, the number of different maps, is not considered to be a constant!

The idea behind the method is the following: We perform a plane sweep from left to right through all visibility maps simultaneously. Whenever the sweepline is at some position L the part of M to the left of L will already have been computed. In fact, we will only show how to compute the vertices of M. The edges and faces can easily be subsequently computed in time $O((n+k)z)$ using the different input visibility maps, or can be maintained during the sweep by a slight modification of the algorithm. Note that the vertices of M are of two different types. They are either vertices of one of the visibility maps M_i or they correspond to visible "T–junction" intersections between edges of different visibility maps.

Rather than maintaining one structure with the sweepline we maintain a separate structure T_i for each visibility map M_i. In T_i we maintain the intersection of the sweepline L with M_i, plus other auxiliary data 'implanted' from other maps. Actually, because the depth relationship between objects is crucial to the calculation of M, we will maintain each T_i as a binary tree, whose leaves represent the objects in V_i ordered by depth, so that each internal node δ is associated with a substructure of T_i that corresponds to the subset of objects of V_i stored at the leaves of the subtree rooted at δ. To determine visible intersections between edges of visibility maps we also insert certain edges of one visibility map into the trees corresponding to other maps, in a manner to be described below.

In more detail, T_i is a binary tree that stores the objects in V_i in its leaves, ordered by depth (we assumed such an order exists and is available to us). With each node δ in T_i we maintain a structure S^i_δ that stores the cross-section of L with the edges of M_i that belong to objects in the subtree rooted at δ. Note that each edge e of map M_i might be stored in up to $O(\log n)$ S^i_δ structures (at all nodes on the path in T_i towards the leaf corresponding to the object to which e belongs). S^i_δ is a simple balanced search tree storing the edges in the order in which they are intersected by the sweepline. So, in fact, every S^i_δ represents a separate sweep structure for the subset of M_i belonging to a particular depth range. In addition, we also store in each S^i_{root} information about the objects of V_i that are visible in M_i along L between adjacent edges. This extra information is required to determine whether vertices of one visibility map are hidden by the objects in the other maps. See figure 2 for an example of a T-structure. The objects in the scene are indicated

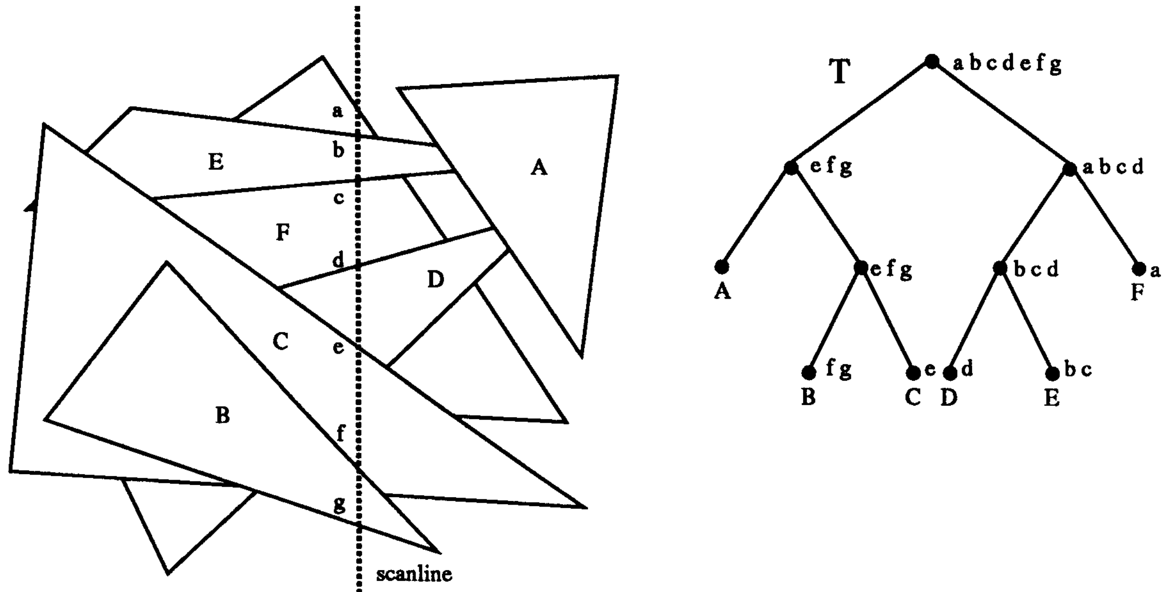

Figure 2: A T-structure.

by uppercase letters, A being the nearest and F the furthest away; the edges of the visibility map are indicated by lowercase letters. At each node of the tree the edges to be stored in the associated structure are indicated.

As event points in our sweep we take initially all vertices of the visibility maps M_i. These are stored, sorted by x-coordinate, in an event queue Q. Later we will add other intersection events to Q. Just maintaining the trees T_i is accomplished as follows. Each initial event point p is a vertex of some map M_i, and requires the insertion or deletion of each edge of M_i incident to p in the corresponding tree T_i. Let e be one of these edges. We search with e in T_i towards the object to which e belongs. For each node δ on the search path we insert or delete e in the tree S^i_δ. This clearly takes time $O(\log n)$ per node δ and, hence, $O(\log^2 n)$ time in total. As there are a total of n edges in all visibility maps and each edge is inserted and deleted once, this yields a total time bound of $O(n \log^2 n)$.

Of course this does not compute M, so we have to do some additional work. As noted above, vertices of M are either vertices of some M_i or are visible T–junctions between edges in different maps. To find the visible vertices of the maps, whenever we reach an event point p being a vertex of some map M_i, we search with p in each S^j_{root} for all $j \neq i$ to find the lowest object of each V_j seen at p. If none of these objects covers p, p is visible and can be reported. This clearly takes time $O(z \log n)$ per node and, hence, at most $O(nz \log n)$ in total.

It remains to determine the visible intersections between edges in different maps. Let p be such an intersection between edge e_i in M_i and edge e_j in M_j. Let e_i be the

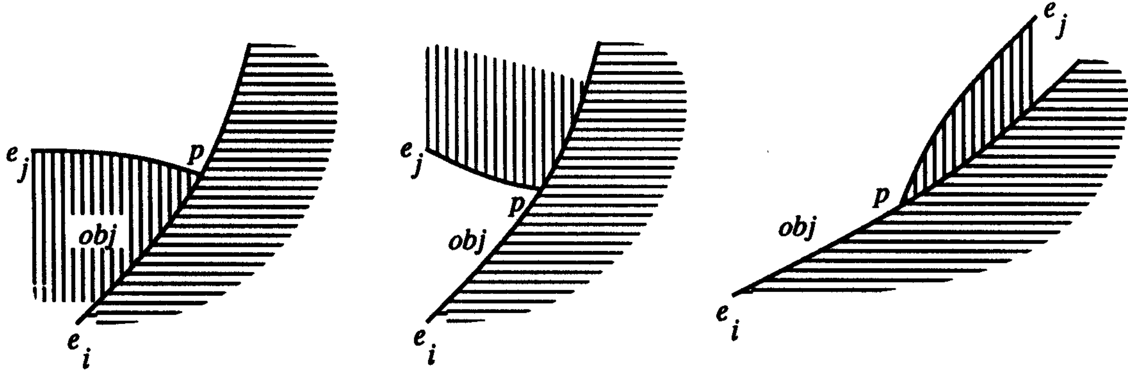

Figure 3: Different types of intersections (looking from below).

lowest of the two edges. So either e_j starts being visible at this point in the sweep or e_j stops being visible (see figure 3). Our technique will be such that the lowest edge e_i will be "responsible" for finding p. To ensure this property, we will insert a visible edge in M (like e_i) into all other trees T_j for $j \neq i$. The method is based on the following lemma that is easy to prove (see also figure 3):

Lemma 2.1 *An intersection p between edges $e_i \in M_i$ and $e_j \in M_j$ $(i \neq j)$ with e_i below e_j is a vertex of M if and only if the following holds: Let obj be the object in V that lies directly above e_i just before the sweepline reaches p. Then e_j must be the projection of the silhouette of either obj or of an object at depth between obj_i and obj, where e_i belongs to obj_i. Moreover e_i must be visible.*

We can thus proceed as follows. First we give a somewhat informal overview of the technique. For each edge e_i that is visible in M and intersected by the sweepline, we compute and maintain the lowest object *obj* above it. This can be done when e_i is inserted into the sweep structures, in time $O(z \log n)$, as follows. Let p be the left endpoint of e_i. In each tree T_j, for $j \neq i$, search with p in the structure S^j_{root} associated with the root, to find the lowest object of V_j that lies at position p. Collecting all these objects over all trees and choosing the lowest one yields the desired object *obj*. (We also have to query M_i, but there it suffices to inspect the two faces of M_i adjacent to e_i.) Assuming e_i is visible at this point, we insert it into all trees T_j $(j \neq i)$ in the following way: We search with the depth of e_i and with the depth of *obj* in T_j to determine those $O(\log n)$ nodes δ of T_j that lie in between the two respective search paths, so that their fathers lie on the search paths (see figure 4). The structures S^j_δ associated with these nodes contain all edges e_j belonging to objects of V_j at depth between e_i and *obj*, as required in the lemma above; moreover, if we restrict ourselves to edges of M_j in the structures of T_j, then each edge appears in exactly one of these subtrees. We insert e_i into each of these structures and check

whether any of its two neighbors in the structure represents an edge of an object in V_j and whether it intersects e_i to the right of the sweepline. In this way we find in each tree T_j at most $O(\log n)$ different intersections. Of all these intersections in all trees we take the leftmost and add it to Q as a new event point. This point is the first visible intersection for e_i, based on the information we have at this moment. We call this intersection the *branching point* of e_i and we call the edge e_j that forms this intersection with e_i the *branch* of e_i. Later on, a new, nearer branching point might be found, so we leave e_i in the different S-structures into which it has been inserted. Note that e_i might have a neighbor $e_{j'}$ in some S^j-structure that was also inserted there from another map $M_{j'}$; in fact, e_i and $e_{j'}$ could be neighbors in many S-structures. This is allowed by the algorithm (although no intersection between them is computed in this case) and does not cause too much difficulty — see below for details.

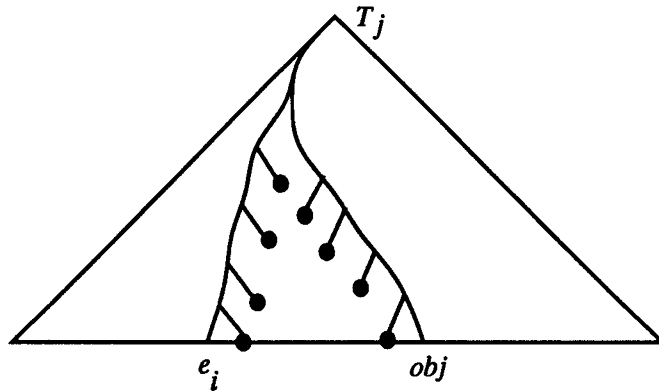

Figure 4: The nodes δ where e_i must be inserted.

So there are three different types of event points: left endpoints of edges, right endpoints of edges and branching points (visible intersections). We now describe in more detail the different steps that have to be taken when the sweepline reaches any of these event points.

To initialize the sweep, we put all left and right endpoints of map edges into the event queue Q, and initialize the trees T_i with empty associated S-structures at each node. In each step of the sweep we remove from the queue Q the leftmost event point p and perform one of the following steps.

To argue that the following procedure is correct, we claim that it maintains the following properties at any instance during the sweep.

(i) Each edge e_i of M_i that is visible along the sweepline is stored at every set S^j_δ, such that $j \neq i$, the depth range of the original edges of S^j_δ lies between the depths of e_i and of its "background" object *obj* as defined above, and the father of δ in T_j does not have this property.

(ii) Any visible intersection between an edge $e_i \in M_i$ and an edge $e_j \in M_j$ is detected and added to Q as soon as (a) the sweep has already reached the rightmost of the left endpoints of both edges, (b) the lower of these two edges, say e_i, is adjacent to e_j in the appropriate structure S_δ^j where they are both stored, and (c) there are no other visible intersections along the lower edge e_i between this point and the sweepline. Moreover, once all these conditions are met, this intersection will not be removed from Q until the sweepline reaches it.

Left endpoint of an edge $e_i \in M_i$.

1. *Determine whether e_i is visible.* This can be done, as explained above, by querying the various structures S_{root}^j for all $j \neq i$, in time $O(z \log n)$.

2. *Add e_i to T_i.* Search with e_i in T_i. For each node δ on the search path add e_i to S_δ^i. Check each of the two neighbors of e_i in S_δ^i. If such a neighbor e does not belong to M_i, check whether e_i intersects it (e_i might be a nearer branch for e; note that by construction all "foreign" edges in S_δ^i precede all original edges in that set in depth order). If so and if the intersection lies before the current nearest branching point of e, remove that branching point from Q and add the new intersection instead. The total time for this step is $O(\log^2 n)$.

3. *If e_i is visible find its nearest branching point.* Let *obj* be the lowest object above e_i (we can get this data from the first step). Insert e_i in all structures T_j ($j \neq i$) according to the condition in property (i) above. If e_i has been inserted into a structure S_δ^j, check its two neighbors there and, if they are original edges of M_j, obtain from them new candidate branching points for e_i as described above. This takes time $O(z \log^2 n)$. The nearest branching point for e_i found (if any) is added to Q.

The total time for an insertion is $O(z \log^2 n)$.

Right endpoint of an edge $e_i \in M_i$.

1. *Remove e_i from T_i.* Search with e_i in T_i. For each node δ on the search path delete e_i from S_δ^i. Check the two former neighbors of e_i in S_δ^i. If one of them is in M_i and the other is not, this might introduce a new nearest branching point. We check this and replace the previous branching point for the lower neighbor (which is necessarily the "foreign" edge) in Q by the new one, if necessary. The time required is $O(\log^2 n)$.

2. *If e_i is visible remove it from the other structures.* So we remove e_i from $O(z \log n)$ S-structures. In each of these structures new neighbors appear that, if in different maps, have to be checked for new nearest branching points. This takes time $O(\log n)$ per S-structure.

Thus a deletion takes time $O(z \log^2 n)$ as well.

Branching point.

A branching point can easily be viewed as one (or two) deletions followed by two (or one) insertions (where the lower edge e_i will be deleted and reinserted). We can thus implement this step as a combination of insertion and deletion steps, as explained above. Hence this step also requires time $O(z \log^2 n)$. (Note that re-insertion of e_i is an essential step of the algorithm and is not done merely for convenience — the object *obj* below e_i will generally change at a branching point, so e_i will have to be moved to different S-structures to preserve property (i).) Finally, the branching point is reported as a new vertex of M.

We claim that properties (i) and (ii) are maintained throughout the execution of the sweep. Property (i) is easily seen to hold by construction. As to property (ii), we will prove it by induction on the left-to-right order of the new vertices of M. Let p be a visible intersection between a lower edge $e_i \in M_i$ and a higher edge $e_j \in M_j$. Let S^j_δ be the unique S-structure in T_j where both edges are stored. Clearly, slightly to the left of p all conditions (a)–(c) of (ii) will be satisfied for this pair. Let us move to the left as much as possible until the first time that one of these conditions fails. Let L be the position of the sweepline when this happens.

If (a) fails, we are at the point of inserting e_i or e_j; since (b) still holds, p will be detected and (c) ensures that p is added to Q (and also that it is not removed later until the sweep reaches p).

If (b) fails, a third edge e_k, that did lie between e_i and e_j slightly to the left of L, has either just intersected e_i or e_j or has its right endpoint on L. In the later case our deletion step will detect p and add it to Q. In the former case, suppose first that $e_k \in M_j$, so that it had to intersect e_i (and not e_j) at some point $q \in L$. Since e_i is visible at this point and e_k lies between e_i and its background object *obj* (by property (i)), it follows that q is visible, so our induction hypothesis implies that q will be detected and processed as a branching point for e_i. Since e_i is re-inserted at this point, the intersection p with its new neighbor e_j will be detected. It remains to consider the case where e_k does not belong to M_j. In this case both e_i and e_k are foreign edges, necessarily lying below all original edges of S^j_δ. If e_k intersects e_j on L then again this must be a branching point for e_k which, by induction, is detected and processed at L. This will cause re-insertion of e_j, at which the point p will be detected. Finally, suppose e_k intersects e_i at some $q \in L$. Since both e_i and e_k are visible at this point, and since e_i is visible to the right of L, it follows that q is a visible intersection and a branching point for e_i. Thus by induction q is detected and processed (in a different tree T_k), which causes e_i to be re-inserted, leading again to the detection of p.

It remains to consider the case where (c) fails at L. Thus e_i has a branching point q on L. By induction, this point will be detected and processed at L, which

will cause e_i to be reinserted, so that p will be detected at this time.

This completes the proof of correctness of the procedure. Since there are $O(n)$ left and right endpoints and k branching points, we obtain the following summary result:

Theorem 2.2 *Let V be a set of objects in 3-space with the properties assumed above. Let $V_1, \ldots, V_z$ form a partition of V into subsets with respective visibility maps $M_1, \ldots, M_z$, having total size n. The visibility map M of V can be computed from $M_1, \ldots, M_z$ in time $O((n+k)z\log^2 n)$, using $O(nz\log n)$ storage.*

3 Applications

In this section we will show how to use the method described above to solve some special instances of the hidden surface removal problem. The idea here is to split the set of objects into a number of different subsets such that for each subset the visibility map has a small size. If we can compute the visibility map for each of the subsets efficiently, then we can obtain the overall map by merging the separate maps using the result of the previous section.

First we apply the method to a set of horizontal unit discs (or of unit spheres) viewed from $z = -\infty$. The previously best known result for this case runs in time $O(n\sqrt{n}\log n + k)$ (see [15]). Note that the size of the visibility map of a set of unit discs can still be as large as $\Omega(n^2)$.

Let V be a set of n unit discs, parallel to the xy-plane. We divide (conceptually) the xy plane into a grid of small squares of size $1/2 \times 1/2$ each. For each disc we determine the squares that intersect its xy projection. In this way we get a set of $O(n)$ (partially) covered squares and for each square sq we have a set V_{sq} of discs whose projections (partially) cover it. As the projection of each disc (partially) covers only $O(1)$ squares, the total size of all sets is linear. For each square sq we determine the lowest disc c_{sq} whose projection completely covers sq (if any). From V_{sq} we remove c_{sq} and all discs that lie above c_{sq} (they will not be visible in sq). c_{sq} will be the background disc in sq (or, if no such disc exists, there is no background disc).

Now each V_{sq} consists of discs partially covering sq and, when computing the visibility map of each V_{sq} restricted to sq (filling the open areas with the background disc), they together form the visibility map of the entire set of discs.

So we now restrict our attention to computing the visibility map of a single set V_{sq}, restricted to sq. The crucial observation is that no disc in V_{sq} can have its center inside sq. We split V_{sq} into four subsets: V_l contains the discs with center to the left of sq, V_r contains the discs with center to the right of sq, V_a contains the remaining discs with center above sq and V_b contains the remaining discs with center below sq.

Lemma 3.1 *Let V be a set of horizontal unit discs whose xy-projections have centers to the left of a line ℓ. The visibility map of V to the right of ℓ has linear*

size.

Proof. The result is based on the easy observation that the boundaries of each pair of unit discs can intersect at most once to the right of ℓ. As a result each disc boundary can contribute at most one arc to the visibility map to the right of ℓ. If not, there would be some disc D whose bounding circle C contributes at least two arcs to the visibility map. Between these arcs C must be covered by at least one other disc, D'. This however is impossible, because the boundaries of D' and D can intersect at most once to the right of ℓ. □

As a result, for each of the sets V_l, V_r, V_a and V_b the visibility map inside sq has linear size. These maps can be computed in time $O(n' \log^2 n')$, where n' is the size of V_{sq}, using the following divide and conquer technique. We describe this procedure for the set V_l. Partition V_l into two subsets of roughly equal size, V_l^1, V_l^2, so that all discs in V_l^1 are lower than all discs in V_l^2. Recursively compute the visibility maps M_1, M_2 of V_l^1, V_l^2 respectively. Let C_1 be the contour of M_1, as defined in the introduction. We then merge C_1 with M_2 to obtain the portion of the overall visibility map representing discs of V_l^2 "seen through" the discs of V_l^1. Gluing this map to M_1 yields the overall map. The merge is accomplished by a sweep, making use of the observation (also noted in the introduction) that every intersection between an edge of C_1 and an edge of M_2 must be a vertex of the overall map. Since all these maps have linear size, the merging can be done in time $O(n' \log n')$, so the overall algorithm takes time $O(n' \log^2 n')$.

The four visibility maps of the subsets V_l, V_r, V_a, V_b can then be merged in time $O(n' + k') \log^2 n')$ using Theorem 2.2, where k' is the size of the visibility map inside sq. since the sum of the k' for all squares is k and the sum of the n' is n we obtain:

Theorem 3.2 *Given a set of n unit discs, parallel to the xy-plane, their visibility map, when viewed from $z = -\infty$, can be determined in time $O((n+k) \log^2 n)$, where k is the size of the map.*

This technique of partitioning a set of objects into subsets with small individual visibility maps can also be applied to other types of objects. As an example, consider hidden surface removal in a set of axis-parallel horizontal rectangles, again viewed from $z = -\infty$. We will show how to split a set of n such rectangles into $O(\log^2 n)$ subsets such that in each subset the visibility map has size linear in the number of rectangles in the subset. This is based on the following lemma which is easy to prove.

Lemma 3.3 *Let V be a set of n axis-parallel horizontal rectangles whose projections all contain a common point p. Then the visibility map of V has linear size and can be computed in time $O(n \log n)$.*

Our approach will be to split the set of rectangles into subsets V_i, such that each V_i is a union of smaller subsets W_{ij}, so that (i) all rectangles in a single subset W_{ij} have a point in common, and (ii) for each fixed i, the subsets W_{ij} are separated from each other by parallel lines. The above lemma implies that the visibility map of each subset V_i is linear.

Let us first consider a set V of rectangles all intersecting a vertical line ℓ. We can split V into $O(\log n)$ subsets each with a linear-size visibility map as follows. Let p be a point on ℓ such that at most half of the rectangles lie completely above p and at most half lie completely below p. Split V into three sets. A_1 is the set of rectangles that contain p. V_1 is the set of rectangles above p and V_2 the set of rectangles below p. A_1 is our first subset with linear visibility map. Split V_1 by a point p_1 on ℓ into two subsets in the same manner that V has been split by p, and split V_2 similarly by a point p_2. A_2 contains the rectangles in V_1 that contain p_1 and the rectangles in V_2 that contain p_2. Because V_1 and V_2 are separated, the visibility map of A_2 will again be linear. We are left with four subsets of rectangles. Each of these is split into two subsets by a point as above, and A_3 is taken to contain the rectangles that contain any of these four points, and so on. In this way we obtain at most $\log n$ sets, each with linear visibility map.

Let us now consider a set of arbitrary (axis-parallel) rectangles. Let ℓ be a vertical line such that at most half the rectangles lie completely to the left of ℓ and at most half the rectangles lie completely to the right of ℓ. Let A^1 be the set of rectangles that intersect ℓ. Let V_1 be the set of rectangles to the left of ℓ and V_2 the set of rectangles to the right of ℓ. We split A^1 in the manner described above into $O(\log n)$ subsets $A_1^1, A_2^1, A_3^1, \ldots$. Then we split V_1 with a vertical line ℓ_1 and V_2 with a vertical line ℓ_2 into (at most) halves. Let B^1 be the set of rectangles that intersect ℓ_1 and B^2 the set of rectangles that intersect ℓ_2. Both sets are split, in the manner described above, into $O(\log n)$ subsets, $B_1^1, B_2^1, \ldots$ and $B_1^2, B_2^2, \ldots$ respectively. Then we merge B_i^1 with B_i^2 to obtain sets A_i^2, for $i = 1, 2, \ldots$. It is easily seen that each set A_i^2 has linear visibility map. We are now left with four sets of rectangles. Each is split with a vertical line, the sets of rectangles that intersect each line are split into $O(\log n)$ subsets, and we merge quadruplets of these subsets to obtain subsets $A_1^3, A_2^3, \ldots$ In this way we obtain $O(\log n)$ groups of $O(\log n)$ subsets, each with linear visibility map. The whole subdivision can easily be constructed in time $O(n \log^2 n)$.

Lemma 3.4 *Let V be a set of n axis-parallel horizontal rectangles viewed from $z = -\infty$. V can be split in time $O(n \log^2 n)$ into $O(\log^2 n)$ subsets such that for each subset the visibility map has linear size.*

Now we can apply Theorem 2.2 to obtain the following result:

Theorem 3.5 *Given a set of n axis-parallel rectangles as above, their visibility map of size k can be constructed in time $O((n + k) \log^4 n)$.*

This result is clearly much worse than the results in [1, 6, 16], where $O((n + k) \log n)$ algorithms are given, but the method is completely different.