Hidden surface removal for c-oriented polyhedra

M. de Berg, M.H. Overmars

RUU-CS-90-33
November 1990

Utrecht University

Department of Computer Science

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

Hidden surface removal for c-oriented polyhedra

M. de Berg, M.H. Overmars

Technical Report RUU-CS-90-33
November 1990

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

ISSN: 0924-3275

Hidden surface removal for c–oriented polyhedra*

Mark de Berg[†] Mark H. Overmars

Abstract

In this paper we present a new, efficient, output-sensitive method for computing the visibility map of a set of c–oriented polyhedra (polyhedra with their faces and edges in c orientations, for some constant c) as seen from a given viewpoint. For non-intersecting polyhedra with n edges in total, the algorithm runs in time $O((n+k)\log n)$, where k is the complexity of the visibility map. The method can handle cyclic overlap of the polyhedra and perspective views without any problem. The method can even deal with intersecting polyhedra. In the latter case the algorithm runs in time $O((n+k)\log^2 n)$.

1 Introduction

A major algorithmic problem in computer graphics is *hidden surface removal*. In a typical setting of the problem we are given a collection of polyhedral objects in 3-space, and a viewing point p_{view}, and our goal is to construct the view of the given scene, as seen from p_{view}.

Many different solutions to this problem exist. Some of them 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. [22]). Other techniques have an *object-space* flavor. The view of a scene 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. Object-space algorithms compute such a subdivision as a collection of polygonal faces. 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 onto the viewing plane and computing all their intersections. Crude implementations of this approach run in time $O(n^2)$ [5, 11]. More

*This research was partially supported by the ESPRIT Basic Research Action No. 3075 (project ALCOM). Authors addresses: Department of Computer Science, Utrecht University, P.O.Box 80.089, 3508 TB Utrecht, the Netherlands.

†This author was also supported by the Dutch Organization for Scientific Research (N.W.O.).

careful implementations run in time $O((n+I)\log n)$ or $O(n\log n+I+J)$, where I denotes the number of intersections between the projected edges and J is the number of intersections between the projected polygons [7, 10, 13, 20]. 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. Mulmuley [12] 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 p_{view} increases. Still, also this method might require quadratic time to produce a trivial output.

General output-sensitive solutions for the hidden surface removal problem are unavailable to date. All prior existing solutions assume that a depth order of the objects exists (i.e., there is no cyclic overlap among the objects) and, moreover, that this order is known. The most general output-sensitive solutions have been proposed by Overmars and Sharir [14] (see also [15, 21]). They show how to compute the visibility map of a set of n horizontal triangles viewed from a point at $z=\infty$ in time $O(n\sqrt{k}\log n)$ where k is the complexity of the output visibility map. (In fact, a second, better bound is obtained in [14] as well. The method though is very complicated and not very practical.)

Better solutions have been obtained for several special cases. For example, Reif and Sen [19] describe an efficient output-sensitive algorithm for hidden surface removal in a polyhedral terrain. Another special case that has received considerable attention is hidden surface removal in a set of horizontal axis-parallel rectangles (also called the *window rendering problem*). See [7, 10, 16] for several solutions. The best result obtained so far is due to Bern [1] and Goodrich, Overmars and Atallah [8] and runs in time $O((n+k)\log n)$. Güting and Ottmann [10] also studied the hidden surface removal problem for c-oriented sets of horizontal polygons. (A set of polygons or polyhedra is c-oriented if the number of different orientations of the edges is c, for some constant c. This notion was introduced by Güting in [9].) They obtained an $O((n+k)\log^2 n)$ time algorithm. Recently Preparata, Vitter and Yvinec [17] have shown how to compute the perspective view of a set of axis-parallel blocks in space from an arbitrary viewpoint p_{view} in time $O((n+k)\log n\log\log n)$. Their method again assumes that a depth order on the set of faces of the blocks exists and is known.

The restriction of the availability of a depth order is a severe one. Even in a simple set of axis-parallel blocks cyclic overlap can occur at many places. See Figure 1 for an example. Moreover, when no cyclic overlap occurs it is in general still difficult to obtain a valid depth order. In this paper we present a first output-sensitive hidden surface removal algorithm that can deal with cyclic overlap. The method extends and improves the results of [1, 8, 10, 17] and computes the visibility map of a c-oriented set of polyhedra in time $O((n+k)\log n)$ where n is the number

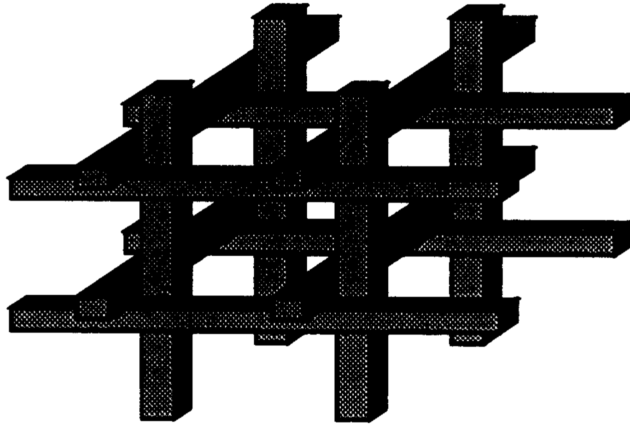

Figure 1: Blocks with cyclic overlap.

of edges of the polyhedra and k is the size of the visibility map. The polyhedra are allowed to have holes. Hence, the method easily solves cases like the one depicted in Figure 1 and even situations as depicted in Figure 2. Both parallel and perspective views can be computed. The method can even be extended to deal with intersecting polyhedra at the cost of only a small increase in the time bound.

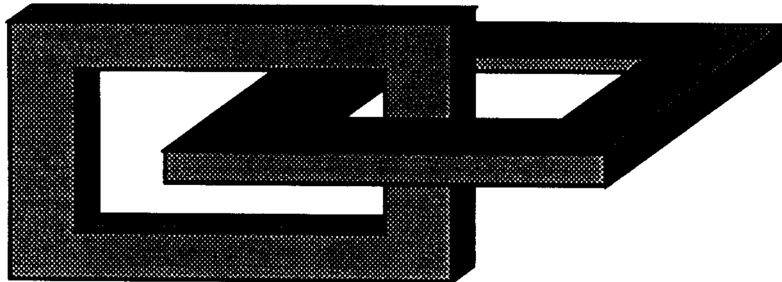

Figure 2: A possible configuration.

The method is not very complicated and, hence, potentially practical. The basic idea is to first compute the visible vertices and next trace the visibility map using shooting queries. This approach has been used before in [15]. In the non-intersecting case, two data structures are needed for this method: one for answering visibility queries for points and the second for performing 2.5-dimensional shooting queries. In the intersecting case a third structure is needed that can answer so-called penetration queries.

The paper is organized as follows. In section 2 we describe the global strategy of the method for the case of non-intersecting polyhedra. This method transforms

the hidden surface removal problem to two types of queries: visibility queries and shooting queries. We also state the result for non-intersecting polyhedra. In sections 3 and 4 new data structures are presented for these two query problems in a c–oriented set of polyhedra. In section 5 intersecting polyhedra are treated. A structure for penetration queries is presented and it is shown how to use this structure to solve the hidden surface removal problem for intersecting polyhedra. In section 6 we adapt the method to perspective views. Finally, in section 7 we make some concluding remarks and give directions for further research.

2 Outline of the method

In this section we will describe the global algorithm that computes the visibility map of a set S of non-intersecting polyhedra in 3–space. (In section 5 it is shown how intersecting polyhedra can be handled.) The method works for any set of non-intersecting polyhedra; the fact that the polyhedra are c–oriented will only be used in the next section to obtain efficient data structures that support the algorithm. p_{view} will be the viewpoint and $\mathcal{P}$ the projection plane. Here we will treat parallel projections only. Hence, the viewpoint p_{view} lies at infinity. In section 6 we show how to extend our method to perspective projections. With $\mathcal{M}(S)$ we denote the visibility map of S (as seen from p_{view}). $\mathcal{M}(S)$ forms a polygonal decomposition of $\mathcal{P}$ in maximal regions where a single face (or no face at all) is visible. We will restrict ourselves to computing the edges of $\mathcal{M}(S)$. The polygons that are visible inside the polygonal regions can easily be maintained during the computations.

As a preliminary step in our algorithm we remove all backfaces of the polyhedra. (A backface of a polyhedron is a face that lies 'on the back side' of the polyhedron, i.e., whose face normal points away from p_{view}, and therefore can never be visible.) This reduces the amount of work in the rest of the algorithm. Removing these backfaces can easily be done in linear time by checking the normals of the faces. Let F denote the remaining set of polygonal faces. Let E be the set of all edges of faces in F and V be the set of all vertices of these faces. (Multiple edges and vertices are counted only once.) We consider the faces and the edges as being open, i.e., the boundary of a face and the endpoints of an edge are not included in the face and edge. We assume that for each edge we know its endpoints and the incident faces and for each vertex the incident edges (at most three because the polyhedra are axis-parallel) and the incident faces. To compute the visibility map $\mathcal{M}(S)$ we can restrict our attention to F, E and V. The edges of $\mathcal{M}(S)$ are parts of the projection of edges in E and the vertices are either projections of visible vertices in V (not hidden by any face in F) or visible intersections between the projected edges in E. For a face f, edge e or vertex v we denote the projection onto $\mathcal{P}$ by $\overline{f}$, $\overline{e}$ and $\overline{v}$, respectively. Similarly $\overline{F}$, $\overline{E}$ and $\overline{V}$ are the sets of projected faces, edges and vertices. We assume that the scene is non-degenerated in the sense that no two vertices in V project onto the same point of $\overline{V}$ and no vertex in V projects onto the

interior of any edge in $\overline{E}$. The methods can be adapted to degenerate cases but we leave the details to the reader.

In a first phase of the algorithm we compute those vertices in V that are visible; the projections onto $\mathcal{P}$ of these vertices are vertices of $\mathcal{M}(S)$. In the second phase the connected components of $\mathcal{M}(S)$ are computed by 'ray shooting' along the edges of $\mathcal{M}(S)$, starting at these visible vertices. This way the other vertices of $\mathcal{M}(S)$, which are visible intersections between edges in $\overline{E}$, are discovered. (This approach to compute visibility maps was also used by Overmars and Sharir in [15].) The correctness of this approach rests on the observation that each connected component of $\mathcal{M}(S)$ contains the projection of at least one visible vertex in V. Take, e.g., the leftmost vertex of the component. It is easily seen that this must be the projection of a visible vertex in V (assuming polyhedra do not intersect). We will now give a more detailed description of the two phases of the algorithm.

In the first phase we have to determine which of the vertices in V are visible. We will solve this problem by building a data structure on the set F that can answer the following *visibility query*:

> Given a visible query point q, report the face in F that lies immediately below q in the viewing direction (or report that no face lies below q).

The face immediately below q in the viewing direction is the face that one sees when standing at q and looking in the viewing direction. More precisely, consider the ray starting at the viewpoint and passing through q. Then the face immediately below q is the first face hit by this ray after passing through q.

Now let $v = (v_x, v_y, v_z)$ be a vertex in V. Lift v in the direction of the viewpoint to obtain a point v' above the polyhedral scene. Then clearly v is visible iff the face immediately below v' also lies below v. (Recall that faces are open and that we assume that there are no degenerate cases.) Thus by performing a visibility query with all (lifted) vertices in V we can determine which ones are visible.

In the second phase we have to compute the rest of $\mathcal{M}(S)$. This is done as follows. Let $\overline{v}$ be some known vertex of $\mathcal{M}(S)$ (after phase 1 we know the vertices that correspond to visible vertices in V and during phase 2 we detect new vertices). Let $\overline{e_1}, \overline{e_2}, \ldots$ be the at most c edges of $\mathcal{M}(S)$ that end at $\overline{v}$. We will take care that we always know the initial portions of these edges. (For a visible vertex $\overline{v}$ reported in phase 1, these are simply the initial portions of the edges in $\overline{E}$ that end in $\overline{v}$.) For each such edge $\overline{e}$ we want to determine the other endpoint $\overline{w}$ in $\mathcal{M}(S)$ (if this other endpoint is not already known). The process is repeated with $\overline{w}$ and the edges ending there, etc. Thus, starting at some visible vertex $\overline{v}$, the whole connected component of $\mathcal{M}(S)$ containing $\overline{v}$ is computed. Because each connected component of $\mathcal{M}(S)$ contains at least one visible vertex, the entire visibility map is computed this way. (To avoid computing edges more than once, we only repeat the process with those edges incident on $\overline{w}$ whose initial portion lies to the right of $\overline{w}$. Furthermore, if $\overline{w}$

has another incident edge to its left we will only continue if the edge from which we arrived lies closer to the viewing point. It can be shown that this way every edge of $\mathcal{M}(S)$ is reported exactly once.)

To compute $\overline{w}$ we proceed in the following way: Let $\overline{\rho}$ be the ray starting at $\overline{v}$ along (i.e. in the direction of) $\overline{e}$. Clearly $\overline{w}$ is the intersection of $\overline{\rho}$ with some edge in $\overline{E}$ or $\overline{w}$ is the projection of an endpoint of the edge e in E whose projection contains $\overline{e}$. More precisely, $\overline{w}$ is the first such point that is visible. Define v to be the point on e whose projection is $\overline{v}$. Furthermore, let ρ be the ray starting at v along e. Finally, let the ray ρ^* be defined as follows: if e is incident upon two faces (recall that we already removed backfaces) then $\rho^* = \rho$, otherwise ρ^* is the projection of ρ onto the face immediately below v. See Figure 3. Observe that the projection of ρ^* is also $\overline{\rho}$.

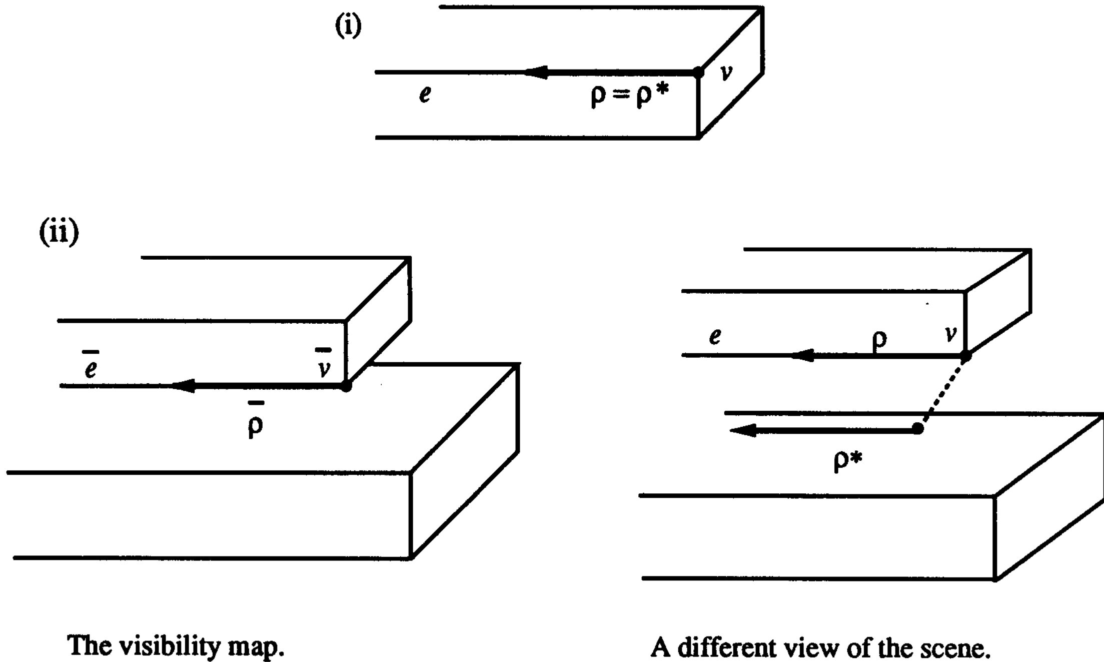

Figure 3: The two possibilities for ρ^*: (i) $\rho^* = \rho$ or (ii) ρ^* is the projection of ρ onto in the face below v.

We say that an edge e' in E *passes above* ρ^* if there is a ray from the viewing point that first (or simultaneously) intersects e' and then intersects ρ^*. Thus, $\overline{e'}$ has to intersect $\overline{\rho}$ and 'at this intersection point' e' has to be closer to the viewpoint. We now claim the following:

Lemma 2.1 *$\overline{w}$ is either the first intersection of $\overline{e}$ with the projection of an edge that passes above ρ^* or, if there is no such edge, $\overline{w}$ is the projection of an endpoint of e.*

Proof. Suppose $\overline{w}$ is not the projection of an endpoint of e. Then $\overline{w}$ must be the intersection of $\overline{e}$ with some other edge $\overline{e'}$. From the definition of ρ^* it is clear that

e' must pass above ρ^*: if it would pass below ρ^* then the intersection point cannot be visible. (In the case where $\rho^* = \rho$ (case (i) in Figure 3) it would be hidden by the two faces incident to e and in the other case (case (ii) in Figure 3) it is hidden by the face below v.)

It remains to show that the first intersection point $\overline{w}$ of $\overline{\rho}$ with an edge passing above ρ^* must be visible. Suppose for a contradiction that $\overline{w}$ is hidden by some face f. From the definition of $\overline{w}$ it is clear that f must lie above ρ^*. Hence, because $\overline{w}$ is the *first* intersection with an edge passing above ρ^*, $\overline{\rho}$ cannot intersect an edge of $\overline{f}$ before $\overline{w}$. (Here we use the fact that the polyhedra do not intersect.) Therefore $\overline{v}$ must be contained in $\overline{f}$. We already know that f must lie above ρ^*. Since v is visible this implies that f has to lie between v and ρ^*, contradicting the definition of ρ^*. □

Because the endpoints of e are readily available, we will concentrate on the computation of the first intersection of $\overline{\rho}$ with the projection of an edge that passes above ρ^*. Notice that the computation of ρ^* itself is trivial if e is incident upon two faces in F. If this is not the case, we have to find the face immediately below v, i.e., we have to perform a visibility query with v. Once we have computed ρ^*, we need a structure that can answer the following *ray shooting query*:

> Given a ray ρ^*, report the first intersection of the projection of ρ^* with the projection of an edge that passes above ρ^*.

If the projection of this edge e' intersects $\overline{e}$ then $\overline{w} = \overline{e} \cap \overline{e'}$, otherwise $\overline{w}$ is the projection of an endpoint of e.

Next we give a more formal description of the algorithm. In this algorithm we only perform shooting queries with rightward directed rays. (Along vertical edges we only shoot upward. For convenience we will continue to say 'to the right of' when we mean 'greater in the lexicographical order'.) Moreover, when we arrive at a vertex we only continue if the edge from which we arrived is visible to the right of the vertex. This is to avoid that edges are computed more than once.

1. Build the data structures for visibility queries in F and for ray shooting queries in E.

2. Compute the visible vertices by performing visibility queries with every vertex in V. Store the projections of the visible vertices in lexicographical order in a priority queue Q.

3. **while** Q not empty

   do Remove the vertex $\overline{v}$ that is smallest in the lexicograhical order from Q. If v is a visible vertex then let $e_1, e_2, \ldots$ be the at most c edges that are incident to v; otherwise let e_1 and e_2 be the two edges such that $\overline{v} = \overline{e_1} \cap \overline{e_2}$.

for each e_i that is visible to the right of $\overline{v}$
do Compute the ray ρ^* as described above and perform a ray shooting query to find the first edge e_i' passing above ρ^*. If $\overline{e_i} \cap \overline{e_i'} \neq \emptyset$ then $\overline{w} = \overline{e_i} \cap \overline{e_i'}$ else $\overline{w}$ is the projection of the right endpoint of e_i. Report $\overline{vw}$ as an edge of $\mathcal{M}(S)$. If $\overline{e_i}$ is visible to the right of $\overline{w}$ then insert w into Q.

This leads to:

Theorem 2.2 *The view of a c–oriented set of polyhedra with n edges in total can be computed in time $O((n+k)\log n)$, where k is the size of the visibility map. The algorithm uses $O(n \log n)$ space.*

Proof. Let us first prove the correctness of the algorithm. Thus we must show that every edge of $\mathcal{M}(S)$ is computed exactly once. We will prove this by induction on the lexicograhical order of the left vertices of the edges of $\mathcal{M}(S)$. So consider an edge $\overline{e}$ and assume that all other edges to the left of $\overline{e}$ have been computed correctly.

If the left vertex $\overline{v}$ of $\overline{e}$ is a visible vertex then this vertex is inserted into Q in step 2 and, hence, $\overline{e}$ will be computed eventually. Moreover, a vertex is inserted into Q in step 3 only when it lies on an edge that is visible both to the left and to the right of the vertex. Clearly this can never be true for a visible vertex and, hence, $\overline{e}$ is not computed more than once.

If $\overline{v}$ is not a visible vertex then exactly one of the edges defining $\overline{v}$ is visible to the left as well as to the right of v. By induction this edge has been computed exactly once. Therefore $\overline{v}$ has been inserted exactly once into Q and $\overline{e}$ will be computed exactly once.

It remains to prove the time and space bounds. In the next sections data structures are described that answer visibility and ray shooting queries in $O(\log n)$ time after $O(n \log n)$ preprocessing. Since we perform $O(n)$ visibility queries in step 2 and $O(k)$ visibility and ray shooting queries in step 3, the time taken by the algorithm is clearly bounded by $O((n+k)\log n)$. To prove the space bound we note that the structures presented in the next sections for visibility and ray shooting queries both use $O(n \log n)$ space and that the number of points in Q is always bounded by $O(n)$. The latter fact can be seen as follows. There are at most n visible vertices so consider the points in Q that are not a visible vertex. At any time during the algorithm the edges of $\mathcal{M}(S)$ that are to the left of and incident to these points are intersected by a common vertical line (because the points are treated from left to right). It thus follows that there cannot be more than n such points in Q. □

3 Visibility queries

In this section we will present a data structure that answers a visibility query in the set F of faces of a c–oriented set of polyhedra efficiently. (A set of polyhedra

is c–oriented if the number of different orientations of the edges of the polyhedra is bounded by a constant c. For example, a set of axis-parallel polyhedra is 3–oriented.) Recall that a visibility query asks for the face in F immediately below a visible query point $q = (q_x, q_y, q_z)$.

First we split each face into a number of quadrilaterals. This is done by adding extra edges that are parallel to the yz–plane from every vertex to its opposite edge. Using the recent algortihm of Chazelle [2] this can be done in linear time, but for our purposes any simple $O(n \log n)$ algorithm is good enough. Now each quadrilateral has two sides (its *left* and its *right* side) that are parallel to the yz-plane and a *top* and a *bottom* side. (Some quadrilaterals are degenerate, i.e. a triangle, and have only one edge that is parallel to the yz-plane.) The resulting set Q of quadrilaterals is then partitioned into c^2 subsets $Q_1, \ldots, Q_{c^2}$ according to the orientation of their top and bottom edges: two quadrilaterals are in the same subset iff their top sides are parallel and their bottom sides are parallel. Since the set of polyhedra is c–oriented this results in $O(c^2)$ subsets. For each subset Q_i we build a separate structure. To find the quadrilateral, and thus the face, immediately below a query point we perform a query in each structure. Of the $O(c^2)$ answers found we select the one closest to the query point.

Now consider one subset Q_i. We know that the projections of all the top sides of the quadrilaterals in Q_i are parallel, that the projections of the bottom sides are parallel and that the projections of the left and right sides are parallel. To simplify the notation, let us apply a transformation such that the viewing plane is the xy–plane, the (projected) bottom edges are parallel to the x–axis and the (projected) left and right sides are parallel to the y–axis. Because the left and right side of a (projected) quadrilateral are parallel to the y–axis, they define an x–interval which we call the *x–segment* of the quadrilateral. The quadrilaterals in Q_i will be stored in a segment tree T (see e.g. [18]) according to their x-segment. Thus we partition the xy–plane into a number of elementary *slabs* by drawing lines parallel to the y–axis through the left and right sides of the (projected) quadrilaterals. These elementary slabs correspond to the leaves of the segment tree. Every node δ of the segment tree corresponds to a slab that is the union of the elementary slabs corresponding to the leaves of the subtree rooted at δ. Each quadrilateral is stored at the nodes δ such that its x–segment is contained in the x–interval of the slab corresponding to δ, but not in the x–interval of the father of δ. Because a quadrilateral can be stored with at most two nodes in every level of the tree, each quadrilateral is stored at most $O(\log n)$ times and the total storage is $O(n \log n)$. Moreover, the quadrilaterals whose x–segment contains q_x are stored exactly once at a node on the search path of q_x in T. Observe that the x–segment of the quadrilateral below q necessarily contains q_x. Hence, if S_δ denotes the set of quadrilaterals stored at a node δ, we only have to find the quadrilateral in S_δ below q for each node δ on the search path of q_x. Then, of the $O(\log n)$ quadrilaterals thus found, we have to select the one with greatest z-coordinate.

Consider S_δ, the set of quadrilaterals stored at node δ (restricted to the slab corresponding to δ). We split each (non-degenerate) quadrilateral into a rectangular part and a triangular part by adding an edge parallel to the bottom side. The rectangular parts and the triangular parts are stored in separate structures that both have to be queried.

Let us first consider the set S_δ^R of rectangles. How do we store S_δ^R so that we can find the rectangle in S_δ^R below q quickly? Here it becomes important that the query point q is visible. This implies that the answer in S_δ^R must be visible at δ. In other words, if $\mathcal{M}(S_\delta^R)$ denotes the visibility map of S_δ^R (restricted to the slab corresponding to δ), then a point location with (q_x, q_y) in $\mathcal{M}(S_\delta^R)$ suffices to find the answer. Recall that S_δ^R consists of axis-parallel rectangles that span the slab corresponding to node δ of the segment tree. Hence, $\mathcal{M}(S_\delta^R)$ is a partitioning of this slab into $O(|S_\delta^R|)$ strips that are parallel to the x–axis. It follows that a point location with (q_x, q_y) in $\mathcal{M}(S_\delta^R)$ is a binary search with q_y in these strips, which takes $O(\log n)$ time. Since we have to do this for every node δ in the segment tree that is on the search path to q_x, the total query time becomes $O(\log^2 n)$. Notice that we always search with the same value q_y at every node on the path. Therefore it is possible to apply a technique of Chazelle and Guibas [3], called fractional cascading, to speed up the query time to $O(\log n)$.

We now turn our attention to the triangular parts. Note that the top sides of the triangles as well as the bottom sides are parallel and that they exactly span the slab corresponding to δ. In other words, the triangles that result from the splitting of the quadrilaterals in S_δ are *translates* of each other. Assume that the top side of the triangles has positive slope; thus each triangle has a unique left vertex. For a query point q, let the triangle $T(q)$ be defined as follows. Reflect any translate in its left vertex and let $T(q)$ be the translate of this mirrored image of the triangles that has $\overline{q}$ as its right vertex. Now observe that $\overline{q}$ is contained in a triangle iff the left vertex of that triangle in contained in $T(q)$ (see Figure 4). Moreover, since all the left vertices lie on a common vertical line l_δ (the left boundary of the slab corresponding to the node δ in the segment tree) it is even true that $\overline{q}$ is contained in a triangle iff the left vertex of that triangle in contained in the intersection $I(q, \delta)$ of $T(q)$ with l_δ. Recall that of all triangles containing $\overline{q}$ we want the highest one. Because the triangles at δ are parallel this corresponds to asking for the highest left triangle vertex whose projection is contained in $I(q, \delta)$.

So we have the following subproblem. Given a number of points (the left triangle vertices) on a line (l_δ), each with an associated value (the height of the vertex), and given two query points on the line (the endpoints of $I(q, \delta)$), find the point that lies between the two query points with largest associated value. Gabow, Bentley and Tarjan [6] have shown that this query can be answered in $O(1)$ time, provided that we already have located the neighbours of the query points in the set of points. Normally, locating the neighbours takes $O(\log n)$ time but because we have to do this at every node δ on a search path in our 'main' segment tree T we can use

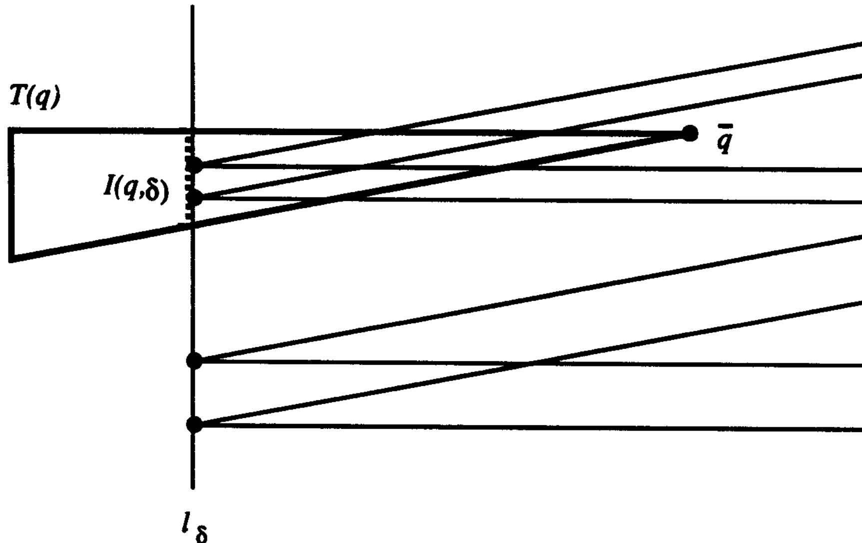

Figure 4: $T(q)$ contains exactly the left vertices of the triangles that contain $\overline{q}$.

fractional cascading to do this in $O(1)$ time. (One word of warning: the intervals $I(q, \delta)$ are not the same at every node δ. But the endpoints of $I(q, \delta)$ are uniquely determined by the endpoints of $I(q, father(\delta))$ and, hence, fractional cascading can still be applied.) Thus the query time at each node δ on the search path in the 'main' segment tree is constant and the total query time for the triangular parts is also $O(\log n)$.

Next it is shown how this structure can be built in $O(n \log n)$ time. First, we construct the segment tree T itself, which takes time $O(n \log n)$ (see [18]). For a node δ in the segment tree, let $[x_\delta : x'_\delta]$ be the x-interval of the slab corresponding to δ.

As for the rectangular parts, observe that the intersection of a rectangle in S^R_δ with the plane $h : x = x_\delta$ is a segment that is parallel to the y-axis. Now $\mathcal{M}(S^R_\delta)$ corresponds to the upper envelope of these segments in the following sense: the part of a rectangle that contributes to $\mathcal{M}(S^R_\delta)$ corresponds the part of the intersection of the rectangle with h that contributes to the upper envelope of the segments. Bern [1] has shown that if the coordinates of the endpoints of a set of m horizontal segments in the plane are integers between 1 and m, then the upper envelope can be computed in $O(m)$ time. Hence, if we have a sorted list of these endpoints available at each node δ in the segment tree, then the total time needed to construct all maps $\mathcal{M}(S^R_\delta)$ is bounded by $\sum_{\delta \in T} |S^R_\delta| = O(n \log n)$. These sorted lists can be obtained in total time $O(n \log n)$ in the following way. The endpoints of the segments are intersections of a top or bottom edge of a quadrilateral with the plane h. These top and bottom edges can be presorted in $O(n \log n)$. By maintaining these lists when

the quadrilaterals are inserted into the segment tree we can obtain at each node δ two sorted lists: one list of the endpoints that correspond to intersections of the bottom edges with h and one list of the endpoints that correspond to intersections of the top edges with h. A simple merge then suffices to obtain the desired sorted list of all endpoints at a node δ.

The structures that store the triangular parts can also be built in $O(n \log n)$ time in total: we can get a sorted list of the left triangle vertices in total time $O(n \log n)$ as described above and then the construction of each associated structure can be done in linear time [6].

Finally we note that the application of fractional cascading does not increase the preprocessing time asymptotically.

We have shown how a visibility query in a set Q_i can be answered in $O(\log n)$ time with a structure that can be built in time $O(n \log n)$. Note that the building time also is a bound on the storage used by the structure. For a query point q, we have to perform a query in each structure and, of the $O(c^2)$ faces thus found, select the one closest to q. Hence, we obtain the following result:

Lemma 3.1 *Visibility queries in a c–oriented set of faces can be answered in time $O(\log n)$ with a structure using $O(n \log n)$ space. This structure can be built in $O(n \log n)$ time.*

4 Shooting queries

We will now present an efficient solution to the ray shooting problem in a c–oriented set E of edges. In a ray shooting query, we are given a query ray ρ^* (in space) and we want to report the first intersection of the projection of ρ^* with the projection of an edge in E that passes above ρ^*. The approach we use resembles the approach used by Cole and Sharir [4] for ray shooting in a polyhedral terrain.

First E is partitioned into c subsets $E_1, \ldots, E_c$ according to the orientation of the edges: two edges are in the same subset iff they are parallel. Notice that not only the number of directions of the edges is bounded, but also the number of possible directions of the query ray ρ^*: each query ray contains an edge in E or the projection of an edge onto the face below it and therefore the number of possible directions of ρ^* is $O(c^2)$. Hence, we can build a ray shooting structure for each of the $O(c^3)$ combinations (direction of ρ^*, E_i). Note that an edge is stored in only $O(c^2)$ of these structures. Given a ray ρ^*, we then have to perform a query in the c structures corresponding to the direction of ρ^* and select the first of the c answers thus found.

Consider some combination (direction of ρ^*, E_i). Assume w.l.o.g. that ρ^* is parallel to the x–axis and that the edges in E_i are parallel to the y–axis. (This can always be accomplished by applying a suitable transformation.) Let ρ^* be directed in the positive x–direction and let r be the starting point of ρ^*. Because ρ^* is directed in positive direction, an edge can only pass above ρ^* if its projection lies to the right

of $\bar{r}$ (i.e. has larger x–coordinate than $\bar{r}$). Hence, we build a binary search tree T with the x–coordinates of the projections of the edges stored in increasing order in its leaves. For a node $\delta \in T$, let E_δ denote the subset of edges whose x–coordinate is stored at a leaf of the subtree of T rooted at δ. If a search with $\bar{r}_x$ in T ends in leaf γ, then the subset of edges that lie to the right of r is exactly the union of sets E_δ for nodes δ that are right son of a node on the search path to γ but are not on the search path themselves. Let $\delta_1, \ldots, \delta_t$ be an enumeration of these nodes in depth-decreasing order. Thus the nodes are numbered from 'left to right'. More precisely, for two edges $e \in E_{\delta_i}$ and $e' \in E_{\delta_j}$ with $i < j$ we have that e lies to the left of e'. See Figure 5. We are looking for the leftmost intersection of ρ^* with an

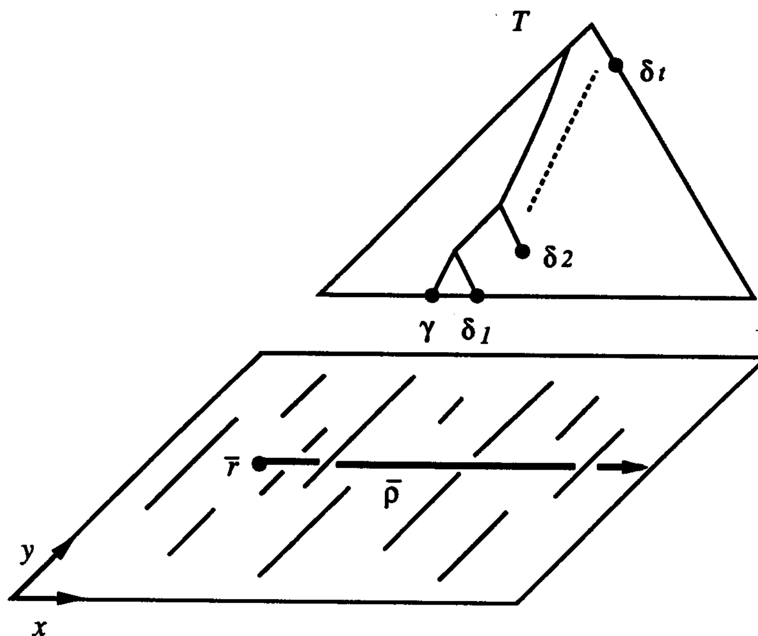

Figure 5: The subtrees rooted at $\delta_1, \ldots, \delta_t$ contain exactly the edges to the right of $\bar{r}$ in a left-to-right order.

edge that passes above ρ^*. Therefore, the first E_{δ_i} that contains at least one edge that passes above ρ^* must contain the answer. So we test if E_{δ_1} contains an edge that passes above ρ^*, if this is not the case we test E_{δ_2}, etc., until we find the first E_{δ_i} that contains an edge that passes above ρ^*. Once we have found the node δ_i such that E_{δ_i} contains the answer we start walking down again: Because the edges in $E_{lson(\delta_i)}$ lie to the left of those in $E_{rson(\delta_i)}$, we turn to the left if $E_{lson(\delta_i)}$ contains an edge that passes above ρ^*. Otherwise we turn to the right. This way we walk down until we reach a leaf. The edge corresponding to this leaf must be the first edge passing above ρ^*.

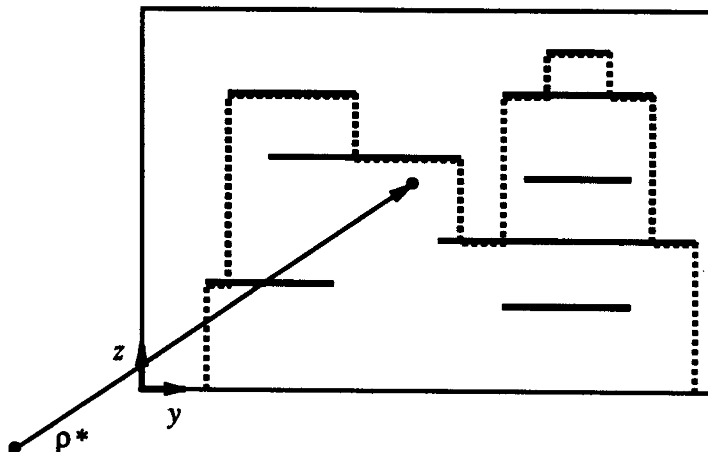

Figure 6: The dashed segments form the upper envelope of the (projected) edges. ρ^* passes below this upper envelope.

It remains to show how to test efficiently whether some subset E_{δ_i} contains an edge passing above ρ^*. We know that the edges in E_{δ_i} lie to the right of r. Hence, it suffices to store the 'upper rim' of the edges as seen from r and test whether ρ^* passes below this upper rim. More precisely, we store the upper envelope of the orthogonal projections of the edges onto the yz–plane. See Figure 6. (Recall that ρ^* is parallel to the x–axis.) To test whether ρ^* passes below this upper envelope we have to perform a binary search on this envelope with the y–coordinate of ρ^*. This way we find a segment of the upper envelope whose z–coordinate then has to be tested against the z–coordinate of ρ^*. If the z–coordinate of ρ^* is greater, then ρ^* passes above the upper envelope which means that there is no edge passing above ρ^*. Otherwise ρ^* passes below the upper envelope which means that there is at least one edge passing above ρ^*. Thus the test takes $O(\log n)$ time, to perform the binary search. Since this test has to be done $O(\log n)$ times, the total query time is $O(\log^2 n)$. Again this can be reduced to $O(\log n)$ by using fractional cascading.

The preprocessing time and the storage of the total structure is $O(n \log n)$: each edge is contained in $O(\log n)$ subsets E_δ (namely at nodes δ on the search path to the x–coordinate of the segment) and, as before, each upper envelope can be constructed in linear time.

A structure as described above has to be built for every combination (direction of ρ^*, E_i). Since the number of combinations is constant, we obtain the following result.

Lemma 4.1 *Ray shooting queries in a c–oriented set of edges can be answered in time $O(\log n)$ with a structure using $O(n \log n)$ space. This structure can be built in $O(n \log n)$ time.*

5 Intersecting polyhedra

In this section we will extend our algorithm so that it also can handle intersecting polyhedra. We will show that intersecting faces can be handled if we have a data structure available that can answer so-called penetration queries, and we present such a structure.

First, let us take a closer look at the visibility map of a set S of intersecting polyhedra. Let F, E and V be defined as before. When the polyhedra do not intersect then the edges of $\mathcal{M}(S)$ are (part of) the projection of edges in E. When the polyhedra intersect, however, edges can also be (part of) the projection of the intersection of two faces in F. Similarly, instead of having two different types of vertices in $\mathcal{M}(S)$ (projections of vertices in V and intersections between projections of edges in E), we now have five different types of vertices. This is summarized in the following observation.

Observation 5.1 *An edge $\overline{e} \in \mathcal{M}(S)$ is of one of the following two types*

(I) *$\overline{e}$ is (a part of) the projection of an edge in E*

(II) *$\overline{e}$ is (a part of) $\overline{f_1 \cap f_2}$ (the projection of the intersection of two faces)*

A vertex $\overline{v} \in \mathcal{M}(S)$ is of one of the following five types

(i) *$\overline{v}$ is the projection of a vertex in V*

(ii) $\overline{v} = \overline{e_1} \cap \overline{e_2}$

(iii) $\overline{v} = \overline{e \cap f}$

(iv) $\overline{v} = \overline{e} \cap \overline{f_1 \cap f_2}$

(v) $\overline{v} = \overline{f_1 \cap f_2 \cap f_3}$

where $e, e_1, e_2 \in E$ and $f, f_1, f_2, f_3 \in F$.

Here $\overline{e_1} \cap \overline{e_2}$ denotes the intersection of the projection of e_1 and e_2, $\overline{e \cap f}$ denotes the projection of the intersection of e and f, etc. We say that a vertex of $\mathcal{M}(S)$ *is defined by* the edges and/or faces involved in its definition. See Figure 7 for an illustration of the various types of edges and vertices.

To compute $\mathcal{M}(S)$ we use the same basic method as before. Thus we first compute which of the vertices in V are visible. Then, starting from these vertices, the rest of $\mathcal{M}(S)$ is computed. This approach is still valid because every connected component of $\mathcal{M}(S)$ still contains a visible vertex as the following lemma shows.

Lemma 5.2 *Every component of $\mathcal{M}(S)$ contains at least one visible vertex.*

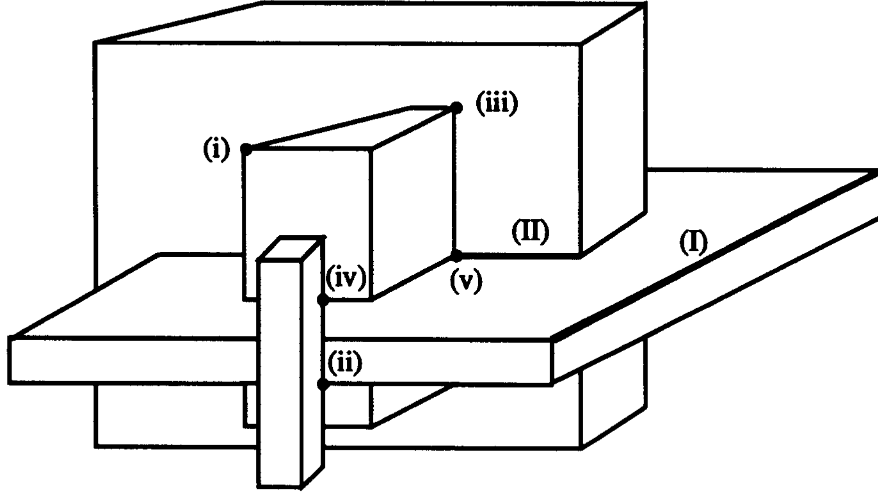

Figure 7: The different types of edges and vertices in the visibility map of a set of intersecting polyhedra.

Proof. Consider any component C of $\mathcal{M}(S)$. Observe that C is enclosed by a unique region of $\mathcal{M}(S)$. Define the *background face* of C, denoted BF_C, to be the face that is visible in this region. (If no face is visible in this region then clearly the leftmost vertex of C must be a visible vertex and we are done.) Consider the (parts of) edges in E or intersections of faces in F whose projections form C. These are called the *lifted* edges of C. Let v be the point on these lifted edges whose distance to BF_C in the viewing direction is maximal. If there are more points having the same distance to BF_C then v is the point that is greatest in the lexicographical order. We claim that v is a visible vertex.

First note that the definition of v implies that v cannot lie on an edge that is visible on two sides of v. Hence $\overline{v}$ is a vertex of C and, moreover, this vertex cannot be of type (ii) or (iv). Now suppose $\overline{v}$ is of type (iii), i.e., $\overline{v} = \overline{e \cap f}$. Imagine moving along e towards v. By definition of v the distance to BF_C must increase while travelling, from which it follows that $f \neq BF_C$. When we pass v, f hides e and thus the distance (in the viewing direction) to BF_C still increases when we keep on moving on f into the same direction. Because $f \neq BF_C$ we must eventually encounter an edge of C that lies above or on f and, hence, has greater distance to BF_C than v. (Note that we can also encounter edges that belong to other components 'floating' inside f. When this happens we just keep moving, knowing that we will always return to f.) See Figure 8. But this contradicts the definition of v so we conclude that v cannot be of type (iii). Finally, v cannot be of type (v) because in that case one easily verifies that it is impossible that along all three lifted edges that are incident to $\overline{v}$ the distance to BF_C decreases. Hence, $\overline{v}$ is of type (i), i.e., a visible vertex. □

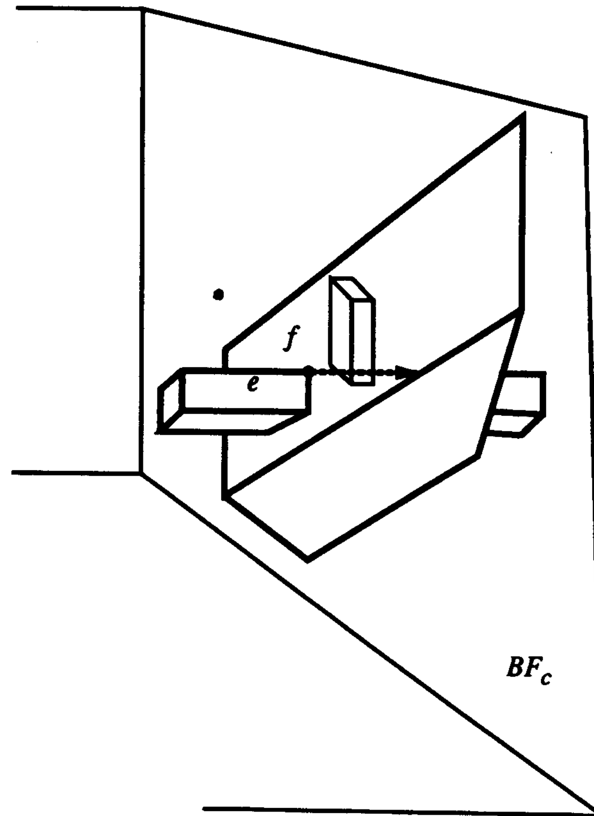

Figure 8: Illustration of the proof of Lemma 5.2. The fat edges are the edges of C.

This establishes the correctness of our approach, but how do we compute which vertices are visible? In the non-intersecting case this is done by performing a visibility query with each vertex. Obviously, this method also works for intersecting polyhedra. Thus we need a structure for visibility queries in a set of intersecting faces. Recall that the structure for visibility queries described in section 3 in fact consists of c^2 substructures. Each substructure stores a set of quadrilaterals whose top sides are parallel and whose bottom sides are parallel. Hence, if we use the same structure the fact that the faces in F can intersect each other does not bother us in the least: the quadrilaterals in one substructure are parallel and therefore they do not intersect. We can conclude that a structure exists for visibility queries in a c–oriented set of intersecting faces with a query time of $O(\log n)$ that can be built in $O(n \log n)$ time. As a consequence, the visible vertices in V can be found in $O(n \log n)$ time.

We now turn our attention to the second phase of the algorithm, the ray shooting phase. In this phase the basic operation is the following: Given a vertex $\overline{v}$ of $\mathcal{M}(S)$ and an initial portion of an edge $\overline{e}$ incident to it, compute the other vertex $\overline{w}$ of

$\mathcal{M}(S)$. Lemma 2.1 gives a characterization of $\overline{w}$ for the non-intersecting case. But we already saw that there are more types of vertices in the intersecting case and Lemma 2.1 has to be changed accordingly. To this end we extend the definition of the rays ρ and ρ^* given in section 2 to intersecting faces. Let $\overline{\rho}$ be defined as before, i.e., $\overline{\rho}$ is the ray in the viewing plane starting at $\overline{v}$ and containing $\overline{e}$. If $\overline{e}$ is (a part of) the projection of an edge $e \in E$ (type (I) in Observation 5.1) then ρ and ρ^* are defined as before. If $\overline{e}$ is (a part of) $\overline{f_1 \cap f_2}$ for two faces $f_1, f_2 \in F$ (type (II)) then ρ is the ray along $f_1 \cap f_2$ whose projection is $\overline{\rho}$ and $\rho^* = \rho$.

Lemma 5.3 *$\overline{w}$ is the point closest to $\overline{v}$ of the following event points:*

- *the projection of an endpoint of e (if $\overline{e}$ is of type (I))*
- *the first intersection of $\overline{\rho}$ with the projection of an edge in E passing above ρ^**
- *the projection of an endpoint of $f_1 \cap f_2$ (if $\overline{e}$ is of type (II))*
- *the projection of the first intersection of ρ with a face in F*
- *the projection of the first intersection of ρ^* with a face in F*

Proof. First we show by a simple case study that all different types of vertices that are mentioned in Observation 5.1 are included in the five event points above. Then we show that the event point closest to $\overline{v}$ must be visible, thus proving the claim.

If $\overline{w}$ is of type (i) then obviously $\overline{w}$ is an endpoint of the edge along which we are shooting, which is the first event point. If $\overline{w}$ is of type (ii) then $\overline{w}$ is the intersection of $\overline{\rho}$ with the projection of an edge passing above ρ^*, which is the second event point. (This follows in exactly the same way as in the non-intersecting case.) If $\overline{w}$ is of type (iii) then there are two cases: we are shooting along e which is handled by the fourth event point, or we are shooting along the intersection of two faces (namely f and the face containing e) which is handled by the third event point. If $\overline{w}$ is of type (iv) then again we have two cases: we are shooting along $f_1 \cap f_2$ which corresponds to the second event point, or we are shooting along e which is handled by the last event point. Finally, if $\overline{w}$ is of type (v) then $\overline{w}$ is a 'double' event point of the fourth (or fifth, since in this case $\rho = \rho^*$) type: ρ intersects two faces simultaneously and $\overline{w}$ is this common intersection point.

It remains to show that the first point $\overline{w}$ of the five event points must be visible. Assume, for example, that $\overline{w}$ is the first intersection of ρ^* with a face in F (the cases where $\overline{w}$ is one of the other event points are similar) and suppose for a contradiction that $\overline{w}$ is hidden by some face f. Observe that no projection of an edge of f can be intersected by $\overline{\rho}$: either this edge passes above ρ^* in which case we would have an event point of the second type before $\overline{w}$, or the edge passes below ρ^* in which case ρ^* must have intersected f and we would have an event point of the fifth type before $\overline{w}$. Because v is visible, this implies that v lies above f. But then either f

lies immediately below v in which case f contains ρ^* and thus cannot hide w, or f is intersected by ρ^* contradicting the definition of $\overline{w}$. □

Because we always know the edge in E or the two faces in F that define the edge of $\mathcal{M}(S)$ along which we are shooting, we have the first or third event point always available. To find the second event point we must perform a ray shooting query as defined in section 2. Since only edges are involved in such a query, the fact that faces can intersect is of no importance. Thus we can use the data structure developed in section 4 in order to find this point in $O(\log n)$ time after $O(n \log n)$ preprocessing. To detect the last two event points we need a structure that can answer so-called *penetration queries*:

Given a query ray, report the first face in F that is hit by by this ray.

Below we present a structure that answers such a query. Notice that, like in ray shooting queries, the number of different directions of the query ray is bounded: the ray that we shoot with either contains an edge in E or it contains the projection of an edge in E onto a face in F or it contains the intersection of two faces. Thus we build a separate structure for every possible direction of the query ray. Given a fixed direction of the query ray, the first step is similar to the first step for visibility queries: the faces in F are partitioned into quadrilaterals and the resulting set of quadrilaterals is partitioned into subsets $Q_1, \ldots, Q_{c^2}$ such that two quadrilaterals are in the same subset iff their top sides are parallel and their bottom sides are parallel. For each subset we build a separate structure; a query is performed in all three structures and the first of the c^2 faces thus found is the answer to the query.

So let us fix a subset Q_i, and a direction for the query ray, say the negative z-direction. Notice that the query that we have to answer is very similar to a visibility query. The only difference is that we no longer shoot from a point that is visible, but from a point that can also be somewhere 'inside' the scene. Recall that the faces in Q_i are parallel to each other and assume for the sake of exposition that they are parallel to the xy-plane. Then a penetration query in Q_i corresponds to a visibility query in the subset of faces in Q_i whose z-coordinate is smaller than the z-coordinate of the starting point of the query ray. In other words, penetration queries are visibility queries with an extra range restriction added. Using the general technique of Willard and Lueker [23] a range restriction can be added at the cost of an extra factor of $O(\log n)$ in both query time and preprocessing time and space. Summarizing, we have:

Lemma 5.4 *Penetration queries in a c-oriented set of faces can be answered in $O(\log^2 n)$ time with a structure using $O(n \log^2 n)$ space. This structure can be built in time $O(n \log^2 n)$.*

One final issue remains before we can state our result: the order in which the other vertices of $\mathcal{M}(S)$ are computed during the ray shooting phase. In the non-intersecting case the ray shooting was performed more or less from left to right. This

way the size of the queue Q (storing all the vertices that have been computed but from which we still have to shoot) was guaranteed to be $O(n)$. Since the leftmost point of a component no longer necessarily is a visible vertex, this approach does not work in the intersecting case. Therefore we do not organize Q as a priority queue on x–coordinate but as an ordinary first-in first-out queue. When we discover a vertex of the map we only insert it into Q if it has not been discovered before. To find out if a vertex is discovered for the first time we keep the already discovered vertices in a search tree. Furthermore, for every computed vertex we remember which of the incident edges already have been computed. Thus each edge is computed exactly once. Notice that the algorithm now uses $O(n \log^2 + k)$ space. This leads to the following theorem:

Theorem 5.5 *The view of a set of possibly intersecting c–oriented polyhedra with n edges in total can be computed in time $((n+k)\log^2 n)$, where k is the size of the visibility map. The algorithm uses $O(n \log^2 n + k)$ space.*

Remark: If we want to keep the space requirements low we have to devise a data structure that can detect the leftmost vertex of a component; then the vertices can be treated from left to right as in the non-intersecting case. Such a structure exists: it has a query time of $O(\log^3 n)$ and it uses $O(n \log^3 n)$ space. Hence, the space requirements can be reduced to $O(n \log^3 n)$ at the cost of an extra $O(\log n)$ factor in the time bound.

6 Perspective projections

In the preceding sections, the data structures are described for parallel projections. However, they can be adapted to perspective projections. This is done in the same way as in [17]. For completeness we give a short description of the method.

Parallel lines in space become lines that intersect in a common point (the vanishing point) when projected perspectively. Consider for example the 3–oriented case where all edges are parallel to one of the coordinate axes. The projections of lines parallel to the x–axis all intersect some vanishing point V_x and they can be ordered by angle φ around V_x. Similarly, the projections of lines parallel to the y–axis (z–axis) can be ordered by their angle θ (ψ) around a common vanishing point V_y (V_z). Now if we write the projections of points, faces, etc. in φ–, θ– and ψ–coordinates, then the solutions of the preceding sections can be applied. The same technique applies with more than three orientations.

Theorem 6.1 *The perspective view of a c–oriented set of non-intersecting polyhedra with n edges in total can be computed in time $O((n+k)\log n)$, where k is the size of the visibility map. The algorithm uses $O(n \log n)$ space. If the polyhedra are allowed to intersect then the algorithm runs in time $O((n+k)\log^2 n)$ and uses $(n \log^2 n + k)$ space.*

7 Conclusions

In this paper, we have presented a first output-sensitive hidden surface removal algorithm that can deal with cyclic overlap among the objects. Our method works for axis-parallel polyhedra or, in general, for any set of polyhedra whose faces have a constant number of different orientations, and it takes time $O((n+k)\log n)$ where n is the total number of edges of the polyhedra and k is the complexity of the visibility map. This extends and improves the results in [1, 8, 10, 16, 17]. The method can be extended to intersecting polyhedra with only a small increase in time.

The most challenging open problem is to give output-sensitive algorithms that can handle cyclic overlap for general scenes, where the number of different orientations of the faces is not bounded. One possible approach is to develop structures that efficiently answer visibility and ray shooting queries in general scenes and then use the basic algorithm described in this paper. Another interesting problem is to determine efficiently if there is cyclic overlap in a scene and, if not, to compute a depth ordering on the faces so that the algorithms of Overmars and Sharir [14] can be applied.

References

[1] M. Bern, Hidden Surface Removal for Rectangles, *J. Comp. Syst. Sciences* **40** (1990), pp. 49–69.

[2] B. Chazelle, Triangulating a Simple Polygon in Linear Time, *Proc. 31rst IEEE Symp. on Foundations of Computer Science*, 1990, pp. 220–230.

[3] B. Chazelle and L.J. Guibas, Fractional Cascading I: A Data Structuring Technique, *Algorithmica* **1** (1986), pp. 133–162.

[4] R. Cole and M. Sharir, Visibility Problems for Polyhedral Terrains, *J. Symbolic Computation* **7** (1989), pp. 11–30.

[5] F. Dévai, Quadratic Bounds for Hidden Line Elimination, *Proc. 2nd ACM Symp. on Computational Geometry*, 1986, pp. 269–275.

[6] H.N. Gabow, J.L. Bentley and R.E. Tarjan, Scaling and Related Techniques for Geometry Problems, *Proc. 16th ACM Symp. on Theory of Computing*, 1984, pp. 135–143.

[7] M.T. Goodrich, A Polygonal Approach to Hidden Line Elimination, *Proc. 25th Allerton Conf. on Communication, Control and Computing*, 1987, pp. 849–858.

[8] 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. 17th Int. Coll. on Automata, Languages and Programming*, Lecture Notes in Computer Science 443, 1990, pp. 689–702.

[9] R.H. Güting, Stabbing c–Oriented Polygons, *Inf. Proc. Lett.* **16** (1983), pp. 35–40.

[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), pp. 188–204.

[11] M. McKenna, Worst-Case Optimal Hidden Surface Removal, *ACM Trans. Graphics* **6** (1987) pp. 19–28.

[12] K. Mulmuley, An Efficient Algorithm for Hidden Surface Removal, I, *Computer Graphics* **23** (1989), pp. 379–388.

[13] O. Nurmi, A Fast Line-Sweep Algorithm for Hidden Line Elimination, *BIT* **25** (1985) pp. 466–472.

[14] M.H. Overmars and M. Sharir, Output-Sensitive Hidden Surface Removal, *Proc. 30th IEEE Symp. on Foundations of Computer Science*, 1989, pp. 598–603.

[15] M.H. Overmars and M. Sharir, An Improved Technique for Output-Sensitive Hidden Surface Removal, Tech. Rept. RUU-CS-89-32, Dept. of Comp. Science, Utrecht University, 1989.

[16] F.P. Preparata, J.S. Vitter and M. Yvinec, Computation of the Axial View of a Set of Isothetic Parallelepipeds, *ACM Trans. on Graphics* **9** (1990), pp. 278–300.

[17] F.P. Preparata, J.S. Vitter and M. Yvinec, Output-Sensitive Generation of the Perspective View of Isothetic Parallelepipeds, *Proc. 2nd Scandinavian Workshop on Algorithm Theory*, 1990, Lecture Notes in Computer Science 447, 1990, pp. 71–84.

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

[19] 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.

[20] A. Schmitt, Time and Space Bounds for Hidden Line and Hidden Surface Algorithms, *Eurographics '81*, pp. 43–56.

[21] M. Sharir and M.H. Overmars, A Simple Method for Output-Sensitive Hidden Surface Removal, *ACM Trans. on Graphics*, 1990, to appear.

[22] I.E. Sutherland, R.F. Sproull and R.A. Schumacker, A Characterization of Ten Hidden-Surface Algorithms, *Computing Surveys* **6** (1974) pp. 1–25.

[23] D.E. Willard and G.S. Lueker, Adding Range Restriction Capability to Dynamic Data Structures, *J. ACM* **32** (1985) pp. 597–617.