Hidden surface removal for axis-parallel polyhedra

M. de Berg, M.H. Overmars

RUU-CS-90-21
May 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 axis-parallel polyhedra

M. de Berg, M.H. Overmars

Technical Report RUU-CS-90-21
May 1990

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

ISSN:0924-3275

Hidden surface removal for axis-parallel polyhedra*

(extended abstract)

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 axis-parallel polyhedra (i.e., polyhedra with their faces and edges parallel to the coordinate axes) 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 n 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 be extended to c-oriented polyhedra (with faces and edges in c orientations, for some constant c) and to intersecting polyhedra with only a slight increase in the time bound.

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 non-intersecting 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 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, which object is visible at that pixel (see e.g. [20]). Other techniques have an *object-space* flavor. The view of the scene as seen from p_{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. 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.

*This research was partially supported by the ESPRIT II Basic Research Action of the EC under contract 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.).

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 $O(n^2)$ [4, 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 [8, 11, 18]. 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.

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, as seen from the viewing point p_{view}, is known (i.e., there is no cyclic overlap among the objects). The most general output-sensitive solutions have been proposed by Overmars and Sharir [12] (see also [13, 19]). 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 [12] as well. The method though is very complicated and not very practical.) 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 p_{view} increases.

Better solution have been obtained for several special cases. For example, Reif and Sen [17] 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 of horizontal axis-parallel rectangles (also called the *window rendering problem*). See [5, 8, 14] for several solutions. The best result obtained so far is due to Bern [1] and Goodrich, Overmars and Atallah [6] and runs in time $O((n+k)\log n)$. Recently Preparata, Vitter and Yvinec [15] have generalized this to computing 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, 6, 15] and computes the visibility map of a set of axis-parallel polyhedra in time $O((n+k)\log n)$ where n is the number of edges of the polyhedra and k is the size of the visibility map. Here an axis-parallel polyhedron is a polyhedron with its edges and faces parallel to the coordinate axes.

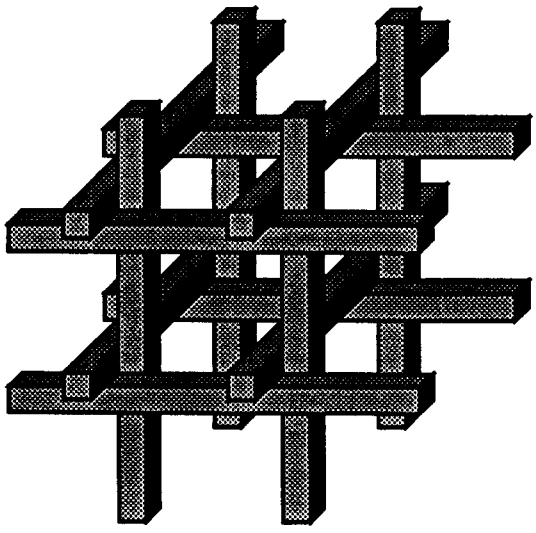

Figure 1: Blocks with cyclic overlap.

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. The only assumption made is that the polyhedra do not intersect. Both parallel and perspective views can be computed.

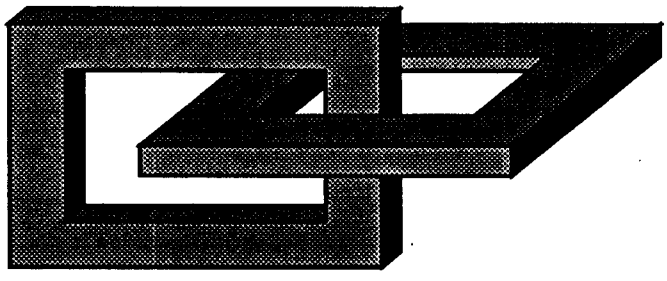

Figure 2: A possible configuration.

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 by shooting queries. This approach has been used before in [13]. Two new data structures are designed, one for answering visibility queries for points and the second for performing 2.5-dimensional shooting queries.

The paper is organized as follows. In section 2 we describe the global strategy of the method that transforms the hidden surface removal problem to two type of queries: visibility queries and shooting queries. We also state the main result of

this paper. In sections 3 and 4 we describe the new data structures for these two query problems. In section 5 we extend the results to perspective views. In section 6 we briefly indicate how to extend the results further to c-oriented polyhedra (a set of polyhedra is c-oriented if the number of different orientations of the faces is c, for some constant c) and to intersecting polyhedra. This will be worked out in more detail in the full version of the paper. Finally, in section 7 we make some some concluding remarks and give some directions for further research.

2 Outline of the method

In the sequel of this paper, let S be a set of axis-parallel polyhedra in 3–space. p_{view} will be the viewpoint and $\mathcal{P}$ the projection plane. We will first treat parallel projections only. Hence, the viewpoint p_{view} lies at infinity. (In section 5 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. We assume that the scene is non-degenerated in the sense that no two vertices in V project onto the same point on $\mathcal{P}$ and no vertex in V projects onto the interior of the projection of any edge in E. The methods can be adapted to degenerate cases but we leave the details to the reader.

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 stored 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.

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 intersections between edges in $\overline{E}$, are discovered. (This approach to compute visibility maps was also used by Overmars and Sharir in [13].) 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 data structures 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 $\overline{e_1}$, $\overline{e_2}$ and possibly $\overline{e_3}$ be the 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 v reported in phase 1, these are simply the projections of the edges in E that end in v.) For each such edge $\overline{e}$, if not treated before, we want to determine the other endpoint $\overline{w}$ in $\mathcal{M}(S)$. The process is repeated with $\overline{w}$ and the edges ending there, etc. 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 is reported exactly once.)

To compute $\overline{w}$ we proceed in the following way: Let $\overline{p}$ be the ray starting at $\overline{v}$ along (i.e. in the direction of) $\overline{e}$. Clearly $\overline{w}$ is the intersection of $\overline{p}$ 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 $\overline{\rho}$.

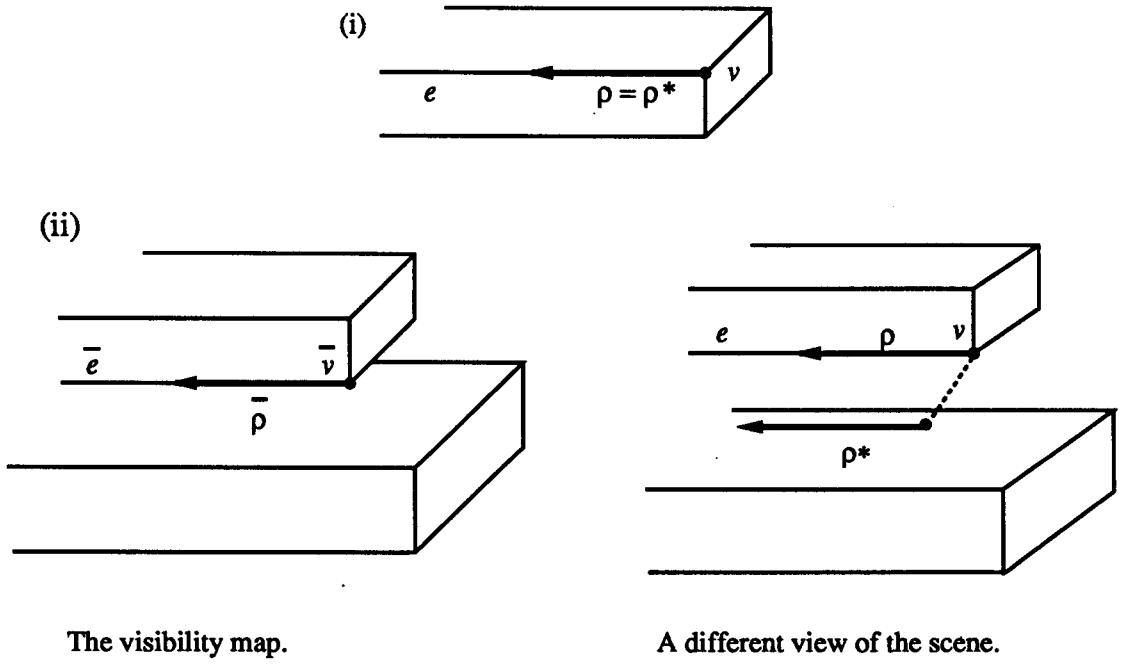

Figure 3: The two possibilities for ρ^*: (i) $\rho^* = \rho$ or (ii) ρ^* is the projection of ρ onto in the face below v.

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' $\overline{e'}$ has to be closer to the viewpoint. We now claim the following:

Lemma 2.1 *$\overline{w}$ is either the projection of an endpoint of e or $\overline{w}$ is the first intersection of $\overline{\rho}$ with the projection of an edge that passes above ρ^*.*

Proof. Suppose $\overline{w}$ is not the projection of an endpoint of e. Then $\overline{w}$ must be closer to $\overline{v}$ than the endpoint of e in the direction of ρ. In the case where $\rho^* = \rho$ (case (i) in Figure 3) this implies that the first visible intersection point must pass above ρ^*. In the case where ρ^* is contained in the face in F below v (case (ii) in Figure 3) $\overline{\rho}$ will intersect the projection of an edge e' of this face and, by definition, this edge passes above ρ^*. Hence, in this case the first visible intersection must lie above or on the face containing ρ^*: otherwise this intersection is hidden by the face containing

ρ^*, or e' is intersected first. Thus in both cases the first visible intersection point must be with an edge that passes above ρ^*.

It remains to show that the first intersection point $\overline{w}$ of $\overline{\rho}$ with an edge passing above ρ^* must be visible. Define w to be the point on the first edge passing above ρ^* whose projection is $\overline{w}$ and suppose for a contradiction that w is hidden by some face f. Because $\overline{w}$ is the *first* intersection with an edge passing above ρ^*, $\overline{\rho}$ cannot intersect an edge of $\overline{f}$ before $\overline{w}$. Since v is visible, this implies that f lies below v. But this contradicts the definition of ρ^* and the fact that the edge containing w passes above ρ^*. (Here we again use the fact that the polyhedra do not intersect.) □

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^*.

In the next sections we will describe data structures that can answer visibility and ray shooting queries in $O(\log n)$ time. Both structures use $O(n \log n)$ preprocessing time and storage. The structures are described for the parallel view. In section 5 it is shown how the structures can be adapted to perspective views. As we have to perform $O(n)$ visibility queries in phase 1 of the algorithm and $O(k)$ visibility and ray shooting queries in phase 2, we obtain the following result.

Theorem 2.2 *The view of a set of axis-parallel 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.*

3 Visibility queries

In this section we will present a data structure that answers a visibility query efficiently. 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 partition each face (which is a rectilinear polygon) into rectangles. The resulting set of rectangles is partitioned into three subsets: a subset F_1 of rectangles parallel to the xy–plane, a subset F_2 of rectangles parallel to the xz–plane and a subset F_3 of rectangles parallel to the yz–plane.

Now consider the set F_1 (the other two subsets are treated similarly). Using a simple transformation we map the viewing plane P to the xy–plane make the

edges of the rectangles parallel to the x– and y–axis. (This transformation is not really necessary but simplifies the discussion.) The rectangles R_i in F_1, which can be written as $[x_i : x_i'] \times [y_i : y_i'] \times z_i$, will be stored in a segment tree T (see e.g. [16]) 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 points (x_i, y_i) and (x_i', y_i). 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 rectangle R_i is stored at the nodes δ such that $[x_i : x_i']$ is contained in the x–segment of the slab corresponding to δ, but not in the x–segment of the father of δ. Because a rectangle can be stored with at most two nodes in every level of the tree, each rectangle is stored at most $O(\log n)$ times and the total storage is $O(n \log n)$. Moreover, the rectangles 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 rectangle below q necessarily contains q_x. Hence, if S_δ denotes the set of rectangles stored at a node δ, we only have to find the rectangle in S_δ below q for each node δ on the search path of q_x. Then, of the $O(\log n)$ rectangles thus found, we have to select the one with largest z-coordinate.

How do we store S_δ so that we can find the rectangle in S_δ below q quickly? Here it becomes important that the query point q is visible. This implies that the answer in S_δ must be visible at δ. In other words, if $\mathcal{M}(S_\delta)$ denotes the visibility map of S_δ, then a point location with (q_x, q_y) in $\mathcal{M}(S_\delta)$ suffices to find the answer. Recall that S_δ consists of axis-parallel rectangles that span the slab corresponding to node δ of the segment tree. Hence, $\mathcal{M}(S_\delta)$ is a partitioning of this slab into $O(|S_\delta|)$ strips that are parallel to the x–axis. It follows that a point location with (q_x, q_y) in $\mathcal{M}(S_\delta)$ 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 [2], called fractional cascading, to speed up the query time to $O(\log n)$ without increasing the preprocessing time or storage.

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 [16]). For a node δ in the segment tree, let $[x_\delta : x_\delta']$ be the x-interval corresponding to δ. Consider the intersections of the rectangles in S_δ with the plane $h : x = x_\delta$. Now $\mathcal{M}(S_\delta)$ corresponds to the upper envelope of these segments $x_\delta \times [y_i : y_i'] \times z_i$ in the following sense: the part of a rectangle R_i that contributes to $\mathcal{M}(S_\delta)$ corresponds the part of the intersection of R_i 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 the coordinates of the rectangles available at each node δ in the segment tree, then the total time needed to construct all maps $\mathcal{M}(S_\delta)$ is bounded by $\sum_{\delta \in T} |S_\delta| = O(n \log n)$.

These sorted lists can be obtained in total time $O(n \log n)$, by presorting the y– and z–coordinates of the rectangles before they are inserted into the segment tree.

We have shown how a visibility query in the set F_1 can be answered in $O(\log n)$ time with a structure that can be built in time $O(n \log n)$. For the other two subsets F_2 and F_3 similar structures are built. For a query point q, we have to perform a query in each structure and, of the three faces thus found, select the one closest to q. Hence, we obtain the following result:

Lemma 3.1 *Visibility queries 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 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 [3] for ray shooting in a polyhedral terrain.

First E is partitioned into three subsets E_1, E_2 and E_3 of edges that are parallel to the x–, y– and z–axis. 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 six. Hence, we can build a ray shooting structure for each combination (direction of ρ^*, orientation of the edges). Given a ray ρ^*, we then have to perform a query in the three structures (in fact, two queries are enough) corresponding to the direction of ρ^* and select the first of the three answers thus found.

Consider some combination (direction of ρ^*, orientation of the edges). Assume w.l.o.g. that ρ^* is parallel to the x–axis (this can always be accomplished by applying a suitable transformation) and consider the edges in E_2, which are parallel to the y–axis. Let ρ^* be directed in the positive x–direction and let $\overline{r}$ be the projection of 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 r (i.e. has larger x–coordinate than r). Hence, we build a binary search tree T with the x–coordinates 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 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, 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'. We are looking for the leftmost intersection of ρ^* with an edge that passes above ρ^*. Therefore, the first

E_{δ_i} that contains an 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^*.

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. To test whether ρ^* passes below this upper envelope we have to perform a binary search on this envelope with the y–coordinate of the starting point 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^*, orientation of the edges). Since the number of combinations is constant, we obtain the following result.

Lemma 4.1 *Ray shooting queries 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 Perspective projections

In the preceding sections, the data structures are described for parallel projections. However, they can be adapted to perspective projections as will be shown next.

Parallel lines in space become lines that intersect in a common point (the vanishing point) when projected perspectively. Hence, 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 can be ordered by their angle θ around a common vanishing point V_y. Now if we write the projections

of points, faces, etc. in φ– en θ–coordinates, then the solutions of the preceding sections can be applied. See the full paper for details.

Theorem 5.1 *The perspective view of a set of axis-parallel 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.*

6 Extensions

In sections 3 and 4 data structures are presented for visibility and ray shooting queries in a set of axis-parallel polyhedra. In this section, the results are extended to c–oriented polyhedra. A set of polyhedra is c–oriented if the number of different orientations of the faces is c, for some constant c. The notion of c–orientedness was introduced by Güting [7]. Later, Güting and Ottmann [8] studied the hidden surface removal problem for c–oriented sets of horizontal polygons. They obtain an $O((n+k)\log^2 n)$ algorithm. Our algorithm, which solves the more general case of c–oriented polyhedra which may have cyclic overlap, improves upon this result.

The basic method remains the same. Thus we need two structures: one for visibility queries and one for ray shooting queries in a c–oriented set of polyhedra. Note that the number of different orientations of the edges bounded as well as the number of possible different directions of the query ray are still bounded. Hence, for the ray shooting we can use the same approach as in the axis-parallel case. So it remains to devise a structure for visibility queries in a c–oriented set of polyhedra. For this problem a structure exists that uses $O(n \log^2 n)$ preprocessing time and storage and in which visibility queries can be performed in $O(\log n)$ time. Due to lack of space, we can only state our result here. Details are given in the full version of the paper. This leads to:

Theorem 6.1 *The view of a c–oriented set of polyhedra with n edges in total can be computed in time $O(n \log^2 n + k \log n)$, where k is the size of the visibility map. The algorithm uses $O(n \log^2 n)$ storage.*

The time bound of the algorithm is quadratic in the number of different orientations of the edges. This dependency can be reduced to linear at the cost of an increase in query time for the visibility queries to $O(\log^2 n)$.

The method can also be extended to intersecting polyhedra: it is still true that each connected component of the visibility map contains the projection of a visible vertex (although it is not as trivial to prove as in the non-intersecting case). However, the vertices of the visibility map are no longer either visible vertices of the polyhedra or visible intersections between edges of the polyhedra. They can also be visible intersections between an edge and a face (a 'penetration point') or a visible intersection between the projection of an edge and the projection of the intersection of two faces. Hence, new structures are needed to detect these new types of vertices. Again we only state the results here and refer to the full paper for details.

Theorem 6.2 *The view of a set of possibly intersecting axis-parallel polyhedra with n edges in total can be computed in time $O(n \log^2 n + k \log n (\log \log n)^2)$, where k is the size of the visibility map. The view of a c–oriented set of possibly intersecting polyhedra with n edges in total can be computed in time $O(n \log^2 n + k \log^2 n)$. The algorithms use $O(n \log^2 n)$ storage.*

7 Conclusions

In this paper, we have presented a first output-sensitive hidden surface algorithm that can deal with cyclic overlap among the objects. For axis-parallel polyhedra it takes time $O((n+k) \log n)$ and for c–oriented polyhedra it runs in time $O(n \log^2 n + k \log n)$. This extends and improves the results in [1, 6, 8, 14, 15]. The method can be xtended 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. 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 [12] can be applied.

References

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

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

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

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

[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, Stabbing c–Oriented Polygons, *Inf. Proc. Lett.* **16** (1983), pp. 35–40.

[8] 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), pp. 188–204.

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

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

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

[14] F.P. Preparata, J.S. Vitter and M. Yvinec, Computation of the axial view of a set of isothetic parallelepipeds, *ACM Trans. on Graphics*, 1990, to appear.

[15] F.P. Preparata, J.S. Vitter and M. Yvinec, Output-sensitive generation of the perspective view of isothetic parallelepipeds, *Proc. Second Scandinavian Workshop on Algorithm theory*, 1990, to appear.

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

[17] 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.

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

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

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

Hidden surface removal for axis-parallel polyhedra

M. de Berg, M.H. Overmars

RUU-CS-90-21
May 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 axis-parallel polyhedra

M. de Berg, M.H. Overmars

Technical Report RUU-CS-90-21
May 1990

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

ISSN:0924-3275

Hidden surface removal for axis-parallel polyhedra*

(extended abstract)

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 axis-parallel polyhedra (i.e., polyhedra with their faces and edges parallel to the coordinate axes) 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 n 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 be extended to c-oriented polyhedra (with faces and edges in c orientations, for some constant c) and to intersecting polyhedra with only a slight increase in the time bound.

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 non-intersecting 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 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, which object is visible at that pixel (see e.g. [20]). Other techniques have an *object-space* flavor. The view of the scene as seen from p_{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. 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.

*This research was partially supported by the ESPRIT II Basic Research Action of the EC under contract 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.).

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 $O(n^2)$ [4, 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 [8, 11, 18]. 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.

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, as seen from the viewing point p_{view}, is known (i.e., there is no cyclic overlap among the objects). The most general output-sensitive solutions have been proposed by Overmars and Sharir [12] (see also [13, 19]). 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 [12] as well. The method though is very complicated and not very practical.) 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 p_{view} increases.

Better solution have been obtained for several special cases. For example, Reif and Sen [17] 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 of horizontal axis-parallel rectangles (also called the *window rendering problem*). See [5, 8, 14] for several solutions. The best result obtained so far is due to Bern [1] and Goodrich, Overmars and Atallah [6] and runs in time $O((n+k)\log n)$. Recently Preparata, Vitter and Yvinec [15] have generalized this to computing 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, 6, 15] and computes the visibility map of a set of axis-parallel polyhedra in time $O((n+k)\log n)$ where n is the number of edges of the polyhedra and k is the size of the visibility map. Here an axis-parallel polyhedron is a polyhedron with its edges and faces parallel to the coordinate axes.

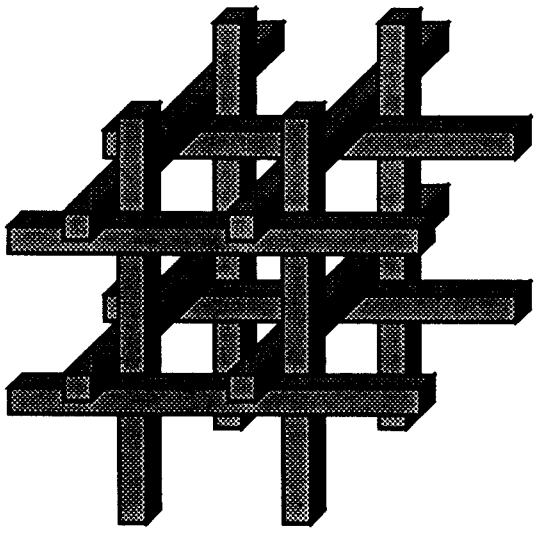

Figure 1: Blocks with cyclic overlap.

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. The only assumption made is that the polyhedra do not intersect. Both parallel and perspective views can be computed.

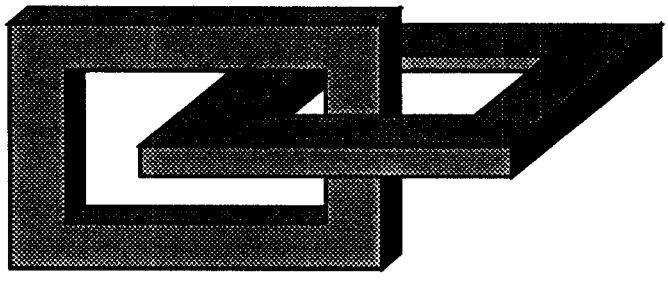

Figure 2: A possible configuration.

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 by shooting queries. This approach has been used before in [13]. Two new data structures are designed, one for answering visibility queries for points and the second for performing 2.5-dimensional shooting queries.

The paper is organized as follows. In section 2 we describe the global strategy of the method that transforms the hidden surface removal problem to two type of queries: visibility queries and shooting queries. We also state the main result of

this paper. In sections 3 and 4 we describe the new data structures for these two query problems. In section 5 we extend the results to perspective views. In section 6 we briefly indicate how to extend the results further to c-oriented polyhedra (a set of polyhedra is c-oriented if the number of different orientations of the faces is c, for some constant c) and to intersecting polyhedra. This will be worked out in more detail in the full version of the paper. Finally, in section 7 we make some some concluding remarks and give some directions for further research.

2 Outline of the method

In the sequel of this paper, let S be a set of axis-parallel polyhedra in 3–space. p_{view} will be the viewpoint and $\mathcal{P}$ the projection plane. We will first treat parallel projections only. Hence, the viewpoint p_{view} lies at infinity. (In section 5 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. We assume that the scene is non-degenerated in the sense that no two vertices in V project onto the same point on $\mathcal{P}$ and no vertex in V projects onto the interior of the projection of any edge in E. The methods can be adapted to degenerate cases but we leave the details to the reader.

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 stored 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.

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 intersections between edges in $\overline{E}$, are discovered. (This approach to compute visibility maps was also used by Overmars and Sharir in [13].) 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 data structures 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 $\overline{e_1}$, $\overline{e_2}$ and possibly $\overline{e_3}$ be the 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 v reported in phase 1, these are simply the projections of the edges in E that end in v.) For each such edge $\overline{e}$, if not treated before, we want to determine the other endpoint $\overline{w}$ in $\mathcal{M}(S)$. The process is repeated with $\overline{w}$ and the edges ending there, etc. 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 is reported exactly once.)

To compute $\overline{w}$ we proceed in the following way: Let $\overline{p}$ be the ray starting at $\overline{v}$ along (i.e. in the direction of) $\overline{e}$. Clearly $\overline{w}$ is the intersection of $\overline{p}$ 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 $\overline{\rho}$.

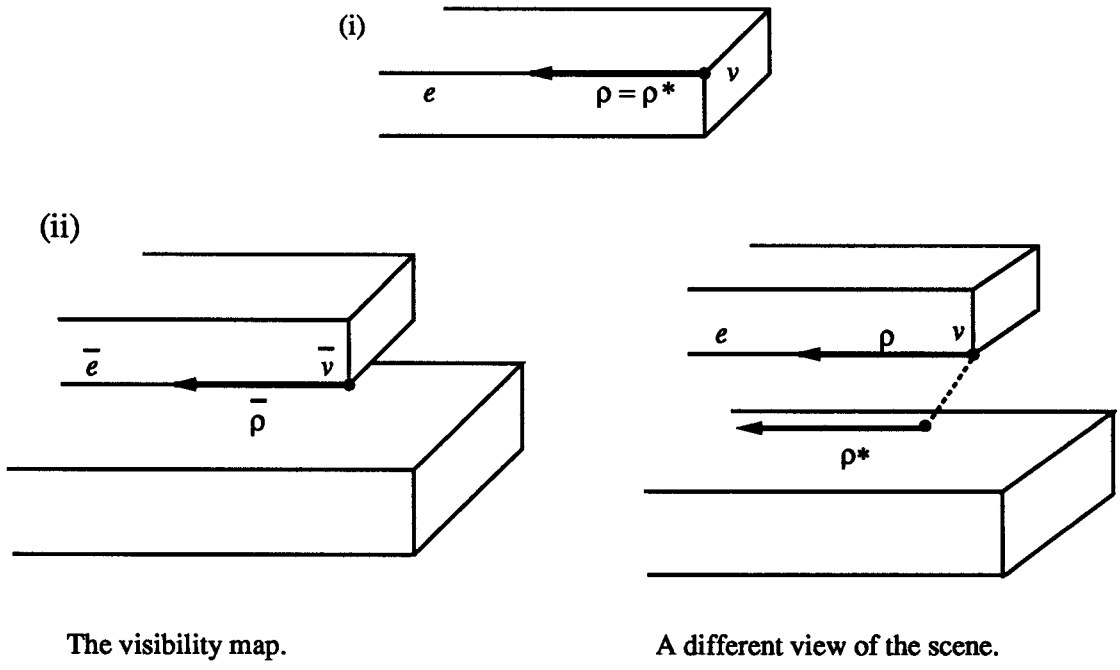

Figure 3: The two possibilities for ρ^*: (i) $\rho^* = \rho$ or (ii) ρ^* is the projection of ρ onto in the face below v.

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' $\overline{e'}$ has to be closer to the viewpoint. We now claim the following:

Lemma 2.1 *$\overline{w}$ is either the projection of an endpoint of e or $\overline{w}$ is the first intersection of $\overline{\rho}$ with the projection of an edge that passes above ρ^*.*

Proof. Suppose $\overline{w}$ is not the projection of an endpoint of e. Then $\overline{w}$ must be closer to $\overline{v}$ than the endpoint of e in the direction of ρ. In the case where $\rho^* = \rho$ (case (i) in Figure 3) this implies that the first visible intersection point must pass above ρ^*. In the case where ρ^* is contained in the face in F below v (case (ii) in Figure 3) $\overline{\rho}$ will intersect the projection of an edge e' of this face and, by definition, this edge passes above ρ^*. Hence, in this case the first visible intersection must lie above or on the face containing ρ^*: otherwise this intersection is hidden by the face containing

ρ^*, or e' is intersected first. Thus in both cases the first visible intersection point must be with an edge that passes above ρ^*.

It remains to show that the first intersection point $\overline{w}$ of $\overline{\rho}$ with an edge passing above ρ^* must be visible. Define w to be the point on the first edge passing above ρ^* whose projection is $\overline{w}$ and suppose for a contradiction that w is hidden by some face f. Because $\overline{w}$ is the *first* intersection with an edge passing above ρ^*, $\overline{\rho}$ cannot intersect an edge of $\overline{f}$ before $\overline{w}$. Since v is visible, this implies that f lies below v. But this contradicts the definition of ρ^* and the fact that the edge containing w passes above ρ^*. (Here we again use the fact that the polyhedra do not intersect.) □

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^*.

In the next sections we will describe data structures that can answer visibility and ray shooting queries in $O(\log n)$ time. Both structures use $O(n \log n)$ preprocessing time and storage. The structures are described for the parallel view. In section 5 it is shown how the structures can be adapted to perspective views. As we have to perform $O(n)$ visibility queries in phase 1 of the algorithm and $O(k)$ visibility and ray shooting queries in phase 2, we obtain the following result.

Theorem 2.2 *The view of a set of axis-parallel 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.*

3 Visibility queries

In this section we will present a data structure that answers a visibility query efficiently. 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 partition each face (which is a rectilinear polygon) into rectangles. The resulting set of rectangles is partitioned into three subsets: a subset F_1 of rectangles parallel to the xy–plane, a subset F_2 of rectangles parallel to the xz–plane and a subset F_3 of rectangles parallel to the yz–plane.

Now consider the set F_1 (the other two subsets are treated similarly). Using a simple transformation we map the viewing plane P to the xy–plane make the

edges of the rectangles parallel to the x– and y–axis. (This transformation is not really necessary but simplifies the discussion.) The rectangles R_i in F_1, which can be written as $[x_i : x_i'] \times [y_i : y_i'] \times z_i$, will be stored in a segment tree T (see e.g. [16]) 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 points (x_i, y_i) and (x_i', y_i). 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 rectangle R_i is stored at the nodes δ such that $[x_i : x_i']$ is contained in the x–segment of the slab corresponding to δ, but not in the x–segment of the father of δ. Because a rectangle can be stored with at most two nodes in every level of the tree, each rectangle is stored at most $O(\log n)$ times and the total storage is $O(n \log n)$. Moreover, the rectangles 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 rectangle below q necessarily contains q_x. Hence, if S_δ denotes the set of rectangles stored at a node δ, we only have to find the rectangle in S_δ below q for each node δ on the search path of q_x. Then, of the $O(\log n)$ rectangles thus found, we have to select the one with largest z-coordinate.

How do we store S_δ so that we can find the rectangle in S_δ below q quickly? Here it becomes important that the query point q is visible. This implies that the answer in S_δ must be visible at δ. In other words, if $\mathcal{M}(S_\delta)$ denotes the visibility map of S_δ, then a point location with (q_x, q_y) in $\mathcal{M}(S_\delta)$ suffices to find the answer. Recall that S_δ consists of axis-parallel rectangles that span the slab corresponding to node δ of the segment tree. Hence, $\mathcal{M}(S_\delta)$ is a partitioning of this slab into $O(|S_\delta|)$ strips that are parallel to the x–axis. It follows that a point location with (q_x, q_y) in $\mathcal{M}(S_\delta)$ 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 [2], called fractional cascading, to speed up the query time to $O(\log n)$ without increasing the preprocessing time or storage.

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 [16]). For a node δ in the segment tree, let $[x_\delta : x_\delta']$ be the x-interval corresponding to δ. Consider the intersections of the rectangles in S_δ with the plane $h : x = x_\delta$. Now $\mathcal{M}(S_\delta)$ corresponds to the upper envelope of these segments $x_\delta \times [y_i : y_i'] \times z_i$ in the following sense: the part of a rectangle R_i that contributes to $\mathcal{M}(S_\delta)$ corresponds the part of the intersection of R_i 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 the coordinates of the rectangles available at each node δ in the segment tree, then the total time needed to construct all maps $\mathcal{M}(S_\delta)$ is bounded by $\sum_{\delta \in T} |S_\delta| = O(n \log n)$.

These sorted lists can be obtained in total time $O(n \log n)$, by presorting the y– and z–coordinates of the rectangles before they are inserted into the segment tree.

We have shown how a visibility query in the set F_1 can be answered in $O(\log n)$ time with a structure that can be built in time $O(n \log n)$. For the other two subsets F_2 and F_3 similar structures are built. For a query point q, we have to perform a query in each structure and, of the three faces thus found, select the one closest to q. Hence, we obtain the following result:

Lemma 3.1 *Visibility queries 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 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 [3] for ray shooting in a polyhedral terrain.

First E is partitioned into three subsets E_1, E_2 and E_3 of edges that are parallel to the x–, y– and z–axis. 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 six. Hence, we can build a ray shooting structure for each combination (direction of ρ^*, orientation of the edges). Given a ray ρ^*, we then have to perform a query in the three structures (in fact, two queries are enough) corresponding to the direction of ρ^* and select the first of the three answers thus found.

Consider some combination (direction of ρ^*, orientation of the edges). Assume w.l.o.g. that ρ^* is parallel to the x–axis (this can always be accomplished by applying a suitable transformation) and consider the edges in E_2, which are parallel to the y–axis. Let ρ^* be directed in the positive x–direction and let $\overline{r}$ be the projection of 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 r (i.e. has larger x–coordinate than r). Hence, we build a binary search tree T with the x–coordinates 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 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, 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'. We are looking for the leftmost intersection of ρ^* with an edge that passes above ρ^*. Therefore, the first

E_{δ_i} that contains an 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^*.

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. To test whether ρ^* passes below this upper envelope we have to perform a binary search on this envelope with the y–coordinate of the starting point 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^*, orientation of the edges). Since the number of combinations is constant, we obtain the following result.

Lemma 4.1 *Ray shooting queries 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 Perspective projections

In the preceding sections, the data structures are described for parallel projections. However, they can be adapted to perspective projections as will be shown next.

Parallel lines in space become lines that intersect in a common point (the vanishing point) when projected perspectively. Hence, 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 can be ordered by their angle θ around a common vanishing point V_y. Now if we write the projections

of points, faces, etc. in φ– en θ–coordinates, then the solutions of the preceding sections can be applied. See the full paper for details.

Theorem 5.1 *The perspective view of a set of axis-parallel 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.*

6 Extensions

In sections 3 and 4 data structures are presented for visibility and ray shooting queries in a set of axis-parallel polyhedra. In this section, the results are extended to c–oriented polyhedra. A set of polyhedra is c–oriented if the number of different orientations of the faces is c, for some constant c. The notion of c–orientedness was introduced by Güting [7]. Later, Güting and Ottmann [8] studied the hidden surface removal problem for c–oriented sets of horizontal polygons. They obtain an $O((n+k)\log^2 n)$ algorithm. Our algorithm, which solves the more general case of c–oriented polyhedra which may have cyclic overlap, improves upon this result.

The basic method remains the same. Thus we need two structures: one for visibility queries and one for ray shooting queries in a c–oriented set of polyhedra. Note that the number of different orientations of the edges bounded as well as the number of possible different directions of the query ray are still bounded. Hence, for the ray shooting we can use the same approach as in the axis-parallel case. So it remains to devise a structure for visibility queries in a c–oriented set of polyhedra. For this problem a structure exists that uses $O(n \log^2 n)$ preprocessing time and storage and in which visibility queries can be performed in $O(\log n)$ time. Due to lack of space, we can only state our result here. Details are given in the full version of the paper. This leads to:

Theorem 6.1 *The view of a c–oriented set of polyhedra with n edges in total can be computed in time $O(n \log^2 n + k \log n)$, where k is the size of the visibility map. The algorithm uses $O(n \log^2 n)$ storage.*

The time bound of the algorithm is quadratic in the number of different orientations of the edges. This dependency can be reduced to linear at the cost of an increase in query time for the visibility queries to $O(\log^2 n)$.

The method can also be extended to intersecting polyhedra: it is still true that each connected component of the visibility map contains the projection of a visible vertex (although it is not as trivial to prove as in the non-intersecting case). However, the vertices of the visibility map are no longer either visible vertices of the polyhedra or visible intersections between edges of the polyhedra. They can also be visible intersections between an edge and a face (a 'penetration point') or a visible intersection between the projection of an edge and the projection of the intersection of two faces. Hence, new structures are needed to detect these new types of vertices. Again we only state the results here and refer to the full paper for details.

Theorem 6.2 *The view of a set of possibly intersecting axis-parallel polyhedra with n edges in total can be computed in time $O(n \log^2 n + k \log n (\log \log n)^2)$, where k is the size of the visibility map. The view of a c–oriented set of possibly intersecting polyhedra with n edges in total can be computed in time $O(n \log^2 n + k \log^2 n)$. The algorithms use $O(n \log^2 n)$ storage.*

7 Conclusions

In this paper, we have presented a first output-sensitive hidden surface algorithm that can deal with cyclic overlap among the objects. For axis-parallel polyhedra it takes time $O((n+k) \log n)$ and for c–oriented polyhedra it runs in time $O(n \log^2 n + k \log n)$. This extends and improves the results in [1, 6, 8, 14, 15]. The method can be xtended 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. 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 [12] can be applied.

References

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

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

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

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

[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, Stabbing c–Oriented Polygons, *Inf. Proc. Lett.* **16** (1983), pp. 35–40.

[8] 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), pp. 188–204.

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

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

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

[14] F.P. Preparata, J.S. Vitter and M. Yvinec, Computation of the axial view of a set of isothetic parallelepipeds, *ACM Trans. on Graphics*, 1990, to appear.

[15] F.P. Preparata, J.S. Vitter and M. Yvinec, Output-sensitive generation of the perspective view of isothetic parallelepipeds, *Proc. Second Scandinavian Workshop on Algorithm theory*, 1990, to appear.

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

[17] 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.

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

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

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