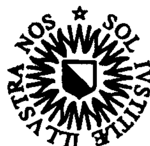
Dynamic Output-Sensitive Hidden Surface Removal for c–Oriented Polyhedra

Mark de Berg

RUU-CS-91-6
February 1991



Utrecht University

Department of Computer Science

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

Dynamic Output-Sensitive Hidden Surface Removal for c–Oriented Polyhedra

Mark de Berg

Technical Report RUU-CS-91-6
February 1991

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

ISSN:0924-3275

Dynamic Output-Sensitive Hidden Surface Removal for c–Oriented Polyhedra

Mark de Berg*

Abstract

In this paper we present an output-sensitive algorithm to maintain the view of a set of c–oriented polyhedra under insertions into or deletions from the set. (A set of polyhedra is c–oriented if the number of different orientations of its edges is bounded by some constant c.) Cyclic overlap in the scene is allowed and the polyhedra may even intersect. The time needed for an update is $O((k+1)\log^3 n)$, where n is the total number of vertices of all polyhedra and k is the number of changes in the visibility map. The solution is based on new dynamic data structures for ray shooting and range searching problems for c–oriented objects.

1 Introduction

One of the most important algorithmic problems in computer graphics is hidden surface removal: Given a set of objects in space, one wants to compute the view of this scene as seen from a given view point. Not surprisingly, this problem has been studied extensively, both from the practical as from the theoretical side. In practice (e.g. animation), however, the scene that one wants to display often changes over time. Therefore we consider the problem of maintaining the view of a set of polyhedra under insertions into or deletions from the set. More precisely, we want to maintain the *visibility map* of the scene. This is the subdivision of the viewing plane into maximal regions, such that in each region one object is seen or no object is seen. Of course one could recompute the whole visibility map from scratch, whenever a polyhedron is added or deleted; but this takes a lot of time, even when there are only few (or no) changes in the map. Hence, it seems natural to try and maintain the visibility map in a more clever way. Strangely enough, this problem has not received much attention. There are only two papers on this problem that we know of. Bern [2] considers the case where the scene consists of n horizontal axis-parallel

*Department of Computer Science, Utrecht University, P.O.Box 80.089, 3508 TB Utrecht, the Netherlands. Supported by the Dutch Organisation for Scientific Research (N.W.O.) and by the ESPRIT Basic Research Action No. 3075 (project ALCOM).

rectangles with the viewing point at $z = \infty$. He obtains a solution that is *output-sensitive*, i.e., the running time depends on k, the number of changes in the visibility map: the time needed to insert or delete a rectangle is $O(\log^2 n \log\log n + k \log^2 n)$. Cheng [4] considers the more general case of horizontal polygons, but his solution is intersection-sensitive instead of output-sensitive; updates take time $O(\sqrt{n} \log^{1.5} n + i \log n + k)$ where i is the total number of intersections between the inserted or deleted polygon and the other polygons in the scene. Note that i can be $\Omega(n)$, even for a polygon that is not visible at all. Moreover, Cheng considers the hidden line problem instead of the hidden surface problem, i.e., he does not maintain the visibility map as a collection of regions, but he only maintains the visible portions of the boundary of each polygon.

We present an output-sensitive algorithm that maintains the visibility map of a set of c–oriented polyhedra. A set of polyhedra is c–oriented if the number of different orientations of the edges is bounded by some constant c [8]. For example, a set of axis-parallel polyhedra is 3–oriented. Thus, unlike in [2, 4], the faces are not necessarily parallel to the viewing plane. The polyhedra are even allowed to have holes and to intersect each other. This imposes many new problems, the most important of which is that cyclic overlap among the faces can occur; in that case a depth ordering on the faces does not exist. See Figure 1.

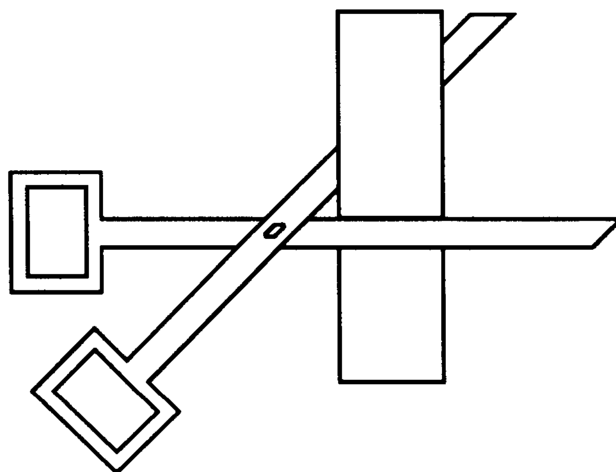


Figure 1: A 4–oriented set of faces with cyclic overlap.

Even for the static hidden surface removal problem, almost all output-sensitive algorithms require that the faces are non-intersecting and that a depth ordering on the objects exists and is known. The only output-sensitive algorithm that can handle cyclic overlap and intersecting faces was presented recently by de Berg and

Overmars [6]. They show how to compute the visibility map of a set of intersecting c-oriented polyhedra in time $O((n+k)\log^2 n)$. (If the polyhedra are non-intersecting then their algorithm runs in time $O((n+k)\log n)$.) We show that the visibility map of a set of c-oriented polyhedra can be maintained in time $O((k+1)\log^3 n)$ per insertion or deletion of a polyhedron of constant size. Thus our solution is only a logarithmic factor worse than the fastest known solution to the static problem.

The solution is based on new dynamic data structures for several basic problems concerning c-oriented objects. In particular, we present dynamic data structures for ray shooting (report the first object, or all objects, hit by a query ray) and range searching (report the leftmost point, or all points, in a query range).

The sequel of this paper is organized as follows. In Section 2 we present efficient dynamic data structures for ray shooting and range searching problems concerning c-oriented objects. These structures are used in Section 3 to obtain the main result of this paper: an output-sensitive solution to the dynamic hidden surface removal problem for c-oriented sets of polyhedra. We conclude the paper in Section 4 with some remarks and directions for further research.

2 Dynamic structures for c-oriented objects

In this section dynamic data structures are presented for several basic problems concerning c-oriented sets of segments and polygons in space and in the plane. We assume that the reader is familiar with standard data structures, such as segment trees and range trees. Recall that a set of segments (polyhedra) is called *c-oriented* if the orientations of the segments (edges of the polyhedra) are taken from a fixed set of c orientations.

2.1 Ray shooting

Let F be a c-oriented set of polygonal faces in space. We want to store F in a dynamic data structure that answers c-oriented *ray shooting queries*: Report the first face (or all faces) in F hit by a c-oriented query ray ρ in space. The solution we give is closely related to the structure for ray shooting queries in [7]. However, some parts of that structure are static and need to be modified. For the reader's convenience, we describe the whole structure anyway.

Because the number of possible directions of the query ray is bounded, we can afford to treat each direction separately. Thus we build a structure for each direction; given a query ray, we simply select the right structure and perform the query in this structure.

So let us try to solve the problem for a fixed direction of the query ray ρ, say parallel to the z-axis. Our first step is to decompose 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. This can be done in linear time in total by the recent algorithm of Chazelle [3], but for our purpose any $O(n \log n)$ algorithm will do. 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 slope 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. If the top side of a subset is parallel to the bottom side of that subset, then the subset is divided further such that the quadrilaterals in a subsubset are parallel to a common plane. Note that if the top side of a subset is not parallel to the bottom side, then all quadrilaterals in the subset are already parallel to a common plane. Since the set of faces is c–oriented this results in $O(c^2)$ subsets. We build a separate structure for each subset Q_i. To find the first 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 top sides of the quadrilaterals in Q_i are parallel, that the bottom sides are parallel and that the left and right sides are parallel. To simplify the notation, let us apply a transformation such that the bottom edges are parallel to the x–axis and the left and right sides are parallel to the y–axis. Thus the left and right side of each quadrilateral have constant x–coordinate; these coordinates define the so-called *x–segment* of the quadrilateral. We store the quadrilaterals in a segment tree T (see [16]) according to their x–segments. Let ρ_x be the x–coordinate of ρ. (Recall that we have assumed that ρ is parallel to the z–axis.) A property of the segment tree is that all the quadrilaterals intersected by ρ are stored at nodes on the search path to ρ_x.

Let δ be a node on the search path to ρ_x and consider S_δ, the set of quadrilaterals stored at δ (restricted to the slab corresponding to δ). We want to store S_δ such that we can find the first quadrilateral in S_δ hit by ρ efficiently. Each (non-degenerate) quadrilateral can be split into a rectangular part and a triangular part by adding an edge parallel to the bottom side. We will use separate structures to handle the triangular parts and the rectangular parts.

For the rectangular parts we note that the problem has become 2–dimensional, since we already know that the x–coordinate of ρ lies in the x–segment of each rectangle stored at δ. So we can project the whole scene onto the yz–plane and we have the following subproblem to solve: given a set of horizontal segments in the plane (the projections of the rectangles) report the segment hit by a vertical ray (the projection of ρ). This problem is easily solved with another segment tree giving a query time for the subproblem of $O(\log^2 |S_\delta|)$. See Overmars [12]. The structure uses $O(|S_\delta| \log |S_\delta|)$ storage and can be updated in time $O(\log^2 |S_\delta|)$. Using dynamic fractional cascading [11], both the query and the update time can be reduced to $O(\log |S_\delta| \log\log |S_\delta|)$.

Now consider the triangular parts at node δ. 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 ray ρ, let the triangle $T(\rho)$ be defined as follows. Reflect any translate in its left vertex and let $T(\rho)$ be the translate of this mirrored image of the triangles that has the starting point of ρ as its right vertex. It is easily verified that moving along ρ until a triangle is hit corresponds to moving $T(\rho)$ until a left vertex of a triangle is hit. See Figure 2. Observe that all the left vertices lie on a common vertical plane,

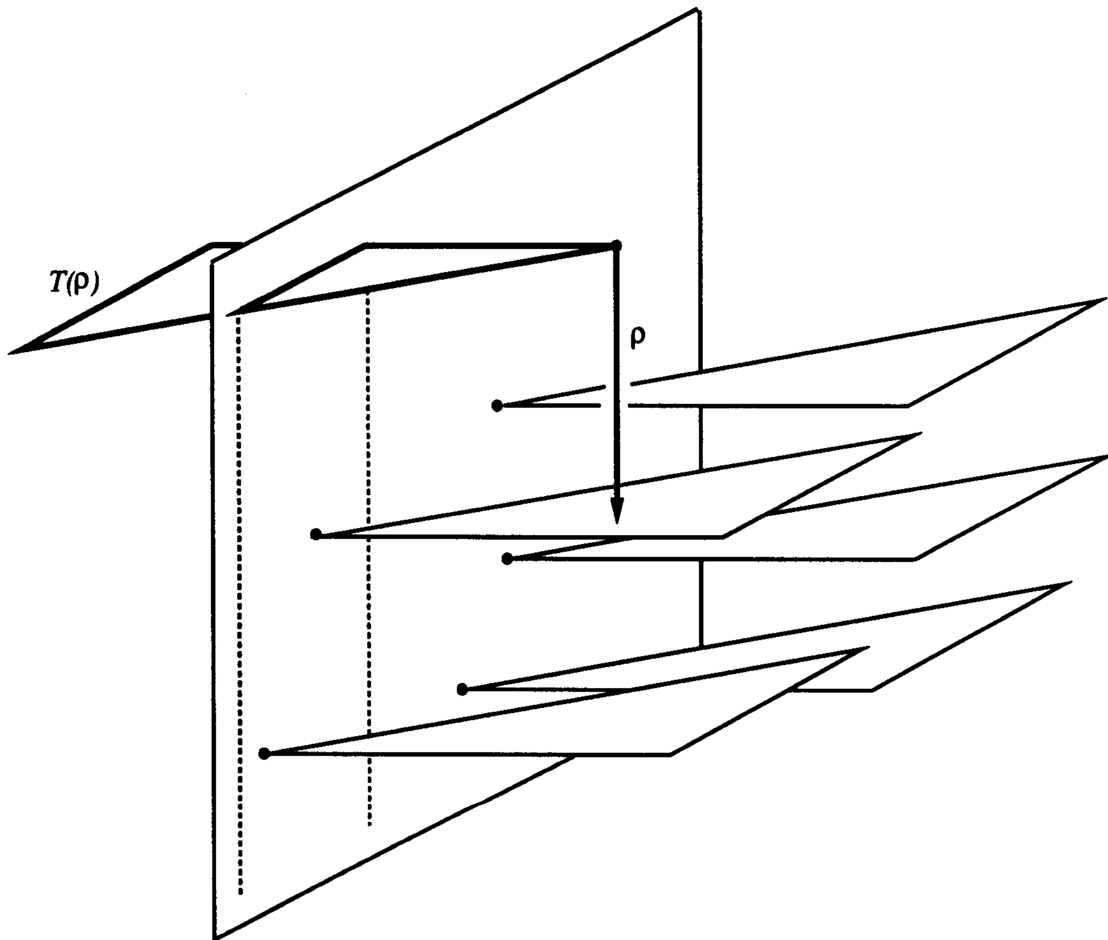


Figure 2: The first triangle hit by ρ is the one whose left vertex is hit first when $T(\rho)$ is moved downward.

namely the plane that bounds the slab corresponding to δ. Thus the only important part of $T(\rho)$ is its intersection with this plane and we are left with the following

subproblem: Given a number of points (the left triangle vertices) in a plane (the left bounding plane of the slab of node δ) and horizontal query segment (the intersection of $T(\rho)$ with the plane), report the first point hit by the segment when it is moved downward. In other words, we are looking for the highest point below a horizontal segment. This problem can be solved using a 2–dimensional range tree (see e.g. [16]). It is well known that such a tree has a query and update time of $O(\log^2 |S_\delta|)$ and that it uses $O(|S_\delta| \log |S_\delta|)$ storage. As before, we can reduce query and update time to $O(\log |S_\delta| \log\log |S_\delta|)$ with dynamic fractional cascading.

Thus the query time at each node δ on the search path in the 'main' segment tree is $O(\log |S_\delta| \log\log |S_\delta|)$, leading to an overall query time of $O(\log^2 n \log\log n)$. The same holds for the update time, and the total amount of space used is $O(n \log^2 n)$. Note that this structure also allows us to report all faces that are intersected by the query ray, in additional time that is linear in the number of reported faces.

Lemma 1 *Let F be a c–oriented set of polygonal faces in space with a total number of n vertices. The first face, or all k faces, in F hit by a c–oriented query ray can be found in time $O(\log^2 n \log\log n)$, respectively $O(\log^2 n \log\log n + k)$, with a structure that uses $O(n \log^2 n)$ space. The structure is dynamic and has an update time of $O(\log^2 n \log\log n)$.*

Notice that all the results also apply if the possible directions of the query ray are different from the possible orientations of the edges, as long as the number of directions of the ray and the number of different orientations of the edges are both bounded.

As a corollary of the lemma above, we obtain the following result. For a segment e and a ray ρ, we say that e *passes above* ρ if there exists a ray, parallel to the z–axis and directed downward, that first intersects e and then intersects ρ. In other words, the projections of e and ρ onto the xy–plane intersect and, 'at this intersection point', the z–coordinate of e is greater than the z–coordinate of ρ. Of all segments passing above ρ, we say that the one whose projection is hit first by the projection of ρ, is the first segment passing above ρ. In the next section, we will need to be able to find the first segment passing above a query ray, as well as to report all such segments.

Corollary 1 *Let E be a c–oriented set of n segments in space. The first segment, or all k segments, in E passing above a c–oriented query ray can be found in $O(\log^2 n \log\log n)$ time, respectively in $O(\log^2 n \log\log n + k)$ time, with a structure that uses $O(n \log^2 n)$ space. The structure is dynamic and has an update time of $O(\log^2 n \log\log n)$.*

Proof: Define a *curtain* to be an unbounded polygon with three edges, two of which are parallel to the z–axis and extend downward to minus infinity. Thus the polygon can be seen as an infinitely long curtain hanging from the third, bounded,

edge. Now observe that a ray passes below a segment if and only if the ray intersects the curtain hanging from that segment. Hence, the first segment in E passing above a query ray ρ can be found by ray shooting in the set of curtains hanging from the segments in E. Since E is c-oriented, the curtains will be $c+1$-oriented, and Lemma 1 applies. As we already noted, the structure of Lemma 1 also enables us to report all curtains intersected by a ray, or, in other words, all segments passing above the ray. □

Remark: For this problem better bounds are in fact possible. Using a combination of segment trees and priority search trees, we are able to obtain a structure using $O(n \log n)$ space in which queries and updates take $O(\log^2 n)$ time. For the application in the hidden surface removal algorithm, however, the other time bounds suffice.

A final result that we need concerns ray shooting in the plane.

Lemma 2 *Let E be a c-oriented set of n segments in the plane. The first segment in E hit by a c-oriented query ray can be found in $O(\log^2 n)$ time with a structure that uses $O(n)$ space. The structure is dynamic and has an update time of $O(\log n)$.*

Proof: Treat all c possible directions separately. For one fixed direction, use the dynamic ray shooting structure of [5]. □

2.2 Range queries

Let us start by considering the following planar *range searching* problem: preprocess a set V of points in the plane such that the leftmost point, or all points, of V inside a c-oriented query polygon P (of constant size) can be reported efficiently. (The leftmost point is the point with minimum y-coordinate.) This problem has also been studied by Güting [9]. He obtains a data structure of size $O(n \log^2 n)$ such that queries and updates take time $O(\log^2 n)$. We reduce the space to $O(n \log n)$ without affecting the query and update time.

We note that any query polygon P can be decomposed into a constant number of quadrilaterals, in the same way as in the section on ray shooting queries: add edges parallel to the y-axis from every vertex to its opposite edge. Since the query polygons are c-oriented, only c^2 different types are possible for the resulting quadrilaterals. (As before, two quadrilaterals have the same type if their top sides are parallel and their bottom sides are parallel). We build a separate structure for each type. With each quadrilateral resulting from the decomposition of P, we search in the structure of the corresponding type; the answer to the query is easily computed from the constant number of subanswers that we find.

Consider a quadrilateral q of a fixed type. This quadrilateral has its left and right side parallel to the y-axis, its top side has a fixed direction and its bottom

side has a fixed direction. If we disregard the top side, then we are searching for the leftmost point in a half-infinite vertical slab and the problem is easily solved in $O(n)$ space with $O(\log n)$ query and update time using a priority search tree [10]. Note that the priority search tree can also be used to report all points in the slab. Taking the top side into account means adding a range restriction to the searching problem. This can be done at the cost of an extra factor of $O(\log n)$ to the space and the query and update time, using the general techniques of Willard and Lueker [17].

Since we have used only standard structures and techniques, we leave the details to the reader. We thus obtain:

Lemma 3 *Let V be a set of n points in the plane. The leftmost point, or all k points, of V inside a c–oriented polygon P can be found in $O(\log^2 n)$ time, respectively in $O(\log^2 n + k)$ time, with a structure that uses $O(n \log n)$ space. The structure is dynamic and has an update time of $O(\log^2 n)$.*

This structure can be used to solve a similar query in the 3–dimensional case. Define a point to be *below* a face (*above* a face) if there is a vertically downward directed (vertically upward directed) ray that first intersects the face and then intersects the point. In other words a point is below (above) a face if it is hidden by the face when we look from $z = \infty$ ($z = -\infty$).

Corollary 2 *Let V be a set of n points in space. The leftmost point, or all k points, of V below (or above) a c–oriented face f can be found in $O(\log^3 n)$ time, respectively in $O(\log^3 n + k)$ time, with a structure that uses $O(n \log^2 n)$ space. The structure is dynamic and has an update time of $O(\log^3 n)$.*

Proof: Build a separate structure for each possible orientation of the face. For a fixed orientation of the face, the structure consist of the 2–dimensional structure of Lemma 3 with another range restriction added to select the points that satisfy the restriction in the third dimension. □

Finally, we need to be able to find the leftmost intersection of a given set of segments with a query face.

Corollary 3 *Let E be a c–oriented set of n segments in space. The leftmost intersection of a c–oriented face f with a segment in E can be found in time $O(\log^3 n)$ with a structure that uses $O(n \log^2 n)$ space. The structure is dynamic and has an update time of $O(\log^3 n)$.*

Proof: Split the problem into a constant number of subproblems where the orientation of the segments is fixed. Consider a fixed orientation and assume w.l.o.g. that the segments are parallel to the z–axis. The structure for this orientation consists of a segment tree on the z–ranges of the segments; the associated structure that stores the segments at a node in the segment tree is the structure of Lemma 3. Thus the segment tree serves to filter out the segments that intersect the plane containing the

query face; once we know this, the problem has become 2–dimensional and Lemma 3 can be applied. □

3 Dynamic hidden surface removal

In this section it is shown how the data structures developed in the previous section can be used to obtain an output-sensitive solution to the dynamic hidden surface removal problem. We are given a (fixed) viewpoint and we want to maintain the visibility map, denoted by $\mathcal{M}(S)$, of a c–oriented set S of polyhedra. To simplify the notation, we map the viewing plane to the xy–plane and we assume that the viewpoint is at $z = \infty$. The solution can be extended to perspective views by using the same techniques as in [6, 15].

Before we describe the algorithms for inserting and deleting polyhedra, we must be a bit more specific about the way $\mathcal{M}(S)$ is represented and about what the output of our algorithms should be. Let F_S, E_S and V_S be the sets of faces, edges resp. vertices of the polyhedra in S. To simplify the discussion, we augment F_S with a large face $f_{-\infty}$ that is below everything in the scene and always remains present. (Recall that the viewpoint is at $z = \infty$. Thus 'below' means 'further away from the viewpoint'.) Hence, the visibility map is a collection $R_{\mathcal{M}} = \{r_1, \ldots, r_s\}$ of regions that form a subdivision of the viewing plane such that in each region one face of a polyhedron is visible or $f_{-\infty}$ is visible. The face that is visible in a region r_i is called its *background face*; it is denoted by $BF(r_i)$. See Figure 3 for an example. We define $E_{\mathcal{M}}$ and $V_{\mathcal{M}}$ to be the set of edges resp. vertices of the regions of $\mathcal{M}(S)$. However, these edges (vertices) are considered to be located on the background face of the corresponding region and not on the viewing plane. Observe that since every vertex of the visibility map is incident to at least two regions, there are at least two corresponding points in $V_{\mathcal{M}}$. Similarly, for each edge of the visibility map there are two regions that have that edge as a part of their boundary. The set $R_{\mathcal{M}}$ is kept in a list $\mathcal{R}$. With each region r_i we store a pointer to its background face and vice versa, so that we know for each face the regions in which it is visible. We also store with r_i the edges of ∂r_i, the boundary of r_i. This set of boundary edges is organized as a concatenable queue (implemented e.g. as a 2-3 tree [1]) with the order corresponding to a clockwise traversal of ∂r_i. Hence, we can insert and delete edges into resp. from ∂r_i in time $O(\log n)$. We can also split ∂r_i at two given points into two subchains in logarithmic time. (Because we have to choose a point on ∂r_i as a starting point one of the subchains will be delivered in two pieces. This minor problem is easy to deal with and we ignore it from now on.) Finally, we have cross pointers between adjacent regions.

Let P denote the c–oriented polyhedron of constant size that we want to insert into or delete from S. Given $\mathcal{M}(S)$, we want to compute $\mathcal{M}(S \cup \{P\})$, respectively $\mathcal{M}(S - \{P\})$, and we have to report the edges and background face of the old regions

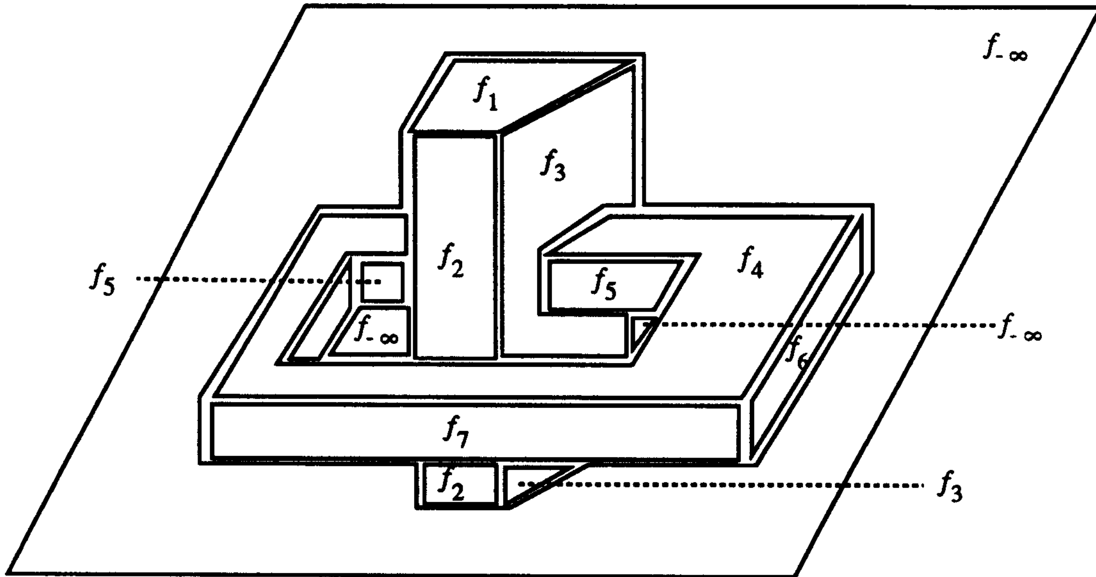


Figure 3: The regions of the visibility map of two intersecting polyhedra. The regions are labeled with their background faces.

that disappear and the edges and background face of the new regions that appear. Thus k, the number of changes in the view, is equal to the number of changes in $E_{\mathcal{M}}$ and $V_{\mathcal{M}}$. Note that an old region disappears and a new one appears if the background face of the region changes. In the remainder of this section we prove our main theorem that states that these changes in the view can be reported in an output-sensitive manner.

Theorem 1 *The visibility map of a c–oriented set of polyhedra (of constant size) can be maintained at the cost of $O((k+1)\log^3 n)$ per insertion or deletion, where k is the number of changes in the view. The method uses $O((n+K)\log^2 n)$ space, where K is the size of the visibility map.*

The data structures described in the previous section are used to support the insertions and deletions. More precisely, the algorithm needs:

- A structure for ray shooting queries as in Lemma 1 on the set $F_{\mathcal{S}}$.
- A structure for 'ray shooting' as in Corollary 1 on the set $E_{\mathcal{S}}$.
- A structure for 'ray shooting' as in Corollary 1 on the set $E_{\mathcal{M}}$.
- A structure for range searching queries as in Corollary 2 on the set $V_{\mathcal{M}}$.
- A structure for range searching queries as in Corollary 2 on the set $V_{\mathcal{S}}$.

- A structure for range searching queries as in Corollary 3 on the set E_S.

The rays for which the first three structures are built are either parallel to an edge in E_S, or parallel to the projection of an edge in E_S onto a face in F_S, or parallel to the viewing direction. The latter three structure are built for searching with polygons that have edges in the c orientations that are allowed for the edges of the polyhedra in S. From this list of structures that are used and the results of the previous section the space bound of Theorem 1 immediately follows. In the remainder of this section we show how to perform insertions and deletions in the stated time bound.

3.1 Insertions

A polyhedron P is inserted by subsequently inserting its frontfaces[1]. (Clearly a backface, i.e. a face with its normal pointing away from the viewpoint, need not be considered.) A face f is inserted according to the following algorithm.

1. Find all the regions that are (partially) hidden by f. These regions are called the *affected regions.*
2. Split each affected region r_i into the parts where $BF(r_i)$ is still visible and the parts where f is the new background face.
3. Merge adjacent regions that have the same background face.
4. Update all the data structures.

The correctness of the algorithm is obvious. Next a more detailed description of the steps of the algorithm is given.

In step 1 the affected regions have to be found. See Figure 4. If a region r_i is affected then at least one of the following cases occurs: (i) a vertex of r_i is below f (ii) an edge of r_i passes below an edge of f (iii) r_i lies below a vertex of f. The first type of affected regions can be found by a range searching query with f to find all vertices in $V_{\mathcal{M}}$ below f, and the second type by ray shooting queries along each edge e of f to find all edges in $E_{\mathcal{M}}$ passing below e. For the third type of affected regions we perform a point location in $\mathcal{M}(S)$ with the projection of each vertex of f. Because our subdivision is not connected, we cannot use any of the known dynamic point location structures. Instead, we perform the point location as follows. First we perform a ray shooting query in F_S with a ray from the viewing point in the direction of the vertex. This gives us the background face of the region (but not the region itself). If the vertex lies below this face then there is no affected region

[1]Because P need not be convex, the insertion of one face may yield regions that are hidden by other faces of P that will be inserted later. However, P has constant size and therefore the extra work involved in this does not increase the time complexity asymptotically. An alternative approach would be to first compute the visibility map of P and then only insert those parts of the faces that are not hidden by other faces of P.

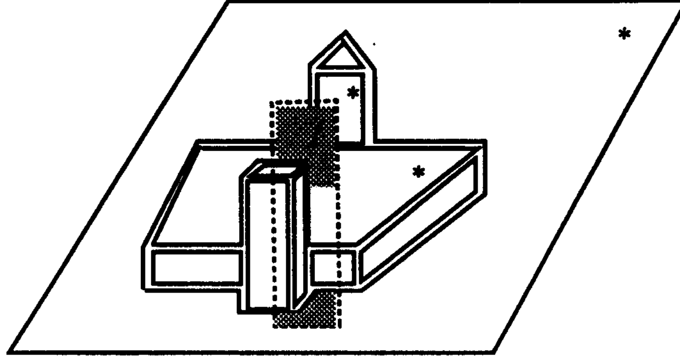


Figure 4: The regions affected by the insertion of face f are indicated with an asterisk (*). Notice that the background face of one of the regions is intersected by f. The part of f that will be visible is shaded.

of type (iii) for this vertex. Otherwise, we shoot with a new ray from the point where the ray hits the face, along the face in any direction to obtain the first edge of $E_{\mathcal{M}}$ passing above this ray. (Of course the direction must be one of the predefined directions for which the ray shooting structure is built.) The edge that is found is an edge on the boundary of an affected region of type (iii).

By Lemma 1 and Corollaries 1, 2 and 3, all the affected regions can thus be found in time $(\log^3 n + k')$, where k' is the number of reported answers. Notice that all reported vertices and edges disappear and, hence, $k' \leq k$.

In step 2 we have to split each affected region into (maximal) subregions where either the 'old' background face is still visible or f is visible. Consider an affected region r_i and let $h(r_i)$ be the plane containing $BF(r_i)$. Intersect $h(r_i)$ with f. This partitions f into a number of parts. The parts below $h(r_i)$ clearly do not affect r_i, so these parts can be discarded. Observe that the edges on the boundary of the remaining parts are either parts of boundary edges of f or they are parts of the intersection of $h(r_i)$ with f. Recall that we already determined the visible intersections of the projections of the boundaries of the remaining parts with the projection of ∂r_i, when we determined the affected regions of type (ii). Hence, we can split the boundaries of these parts into chains 'inside' r_i and chains 'outside' r_i. The latter chains can be discarded. Moreover, we can split ∂r_i (which is stored in a concatenable queue) at the intersection points into a number of subchains in time $O(k_i \log n)$, where k_i is the number of (visible) intersections of ∂r_i with ∂f. With the chains on ∂r_i and the chains of the parts of f above r_i available, it is easy to assemble the resulting regions in time $O((k_i + 1) \log n)$ by concatenating the chains that form the boundary of each region. Step 2 is illustrated in Figure 5. Note that the pieces on the boundary of (the parts of) f have to be duplicated, since they are bounding two new regions. The copy that bounds the region in which $BF(r_i)$ is still visible has to be relocated,

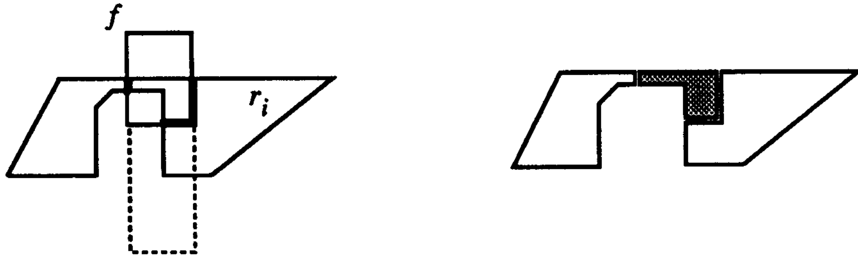


Figure 5: The part of f below $h(r_i)$ (bounded by the dotted lines) and the part of f outside r_i (the solid lines) are discarded, giving two chains inside r_i (the fat lines). Next r_i is split into regions where f is visible (shaded) and regions where $BF(r_i)$ is still visible.

i.e., projected onto $BF(r_i)$. Also note that the new cross pointers between regions and their backfaces and between adjacent regions are readily available. Thus step 2 takes time $O(k \log n)$ in total.

Once step 2 has been performed properly, step 3 is relatively simple. Adjacent regions that have the same background face can only be regions with f as background face. (The other new regions that appear are subregions of a region with the same background face that already existed.) After step 2 we know all the regions where f is visible and their neighboring regions. Furthermore, adjacent regions can be merged in time $O(\log n)$ plus linear time in the number of common boundary edges that have to be removed. Notice that these edges are edges of $E_{\mathcal{M}}$ that are hidden by f and, hence, we are allowed to spend time to remove them. Thus step 3 also takes $O(k \log n)$ time. See Figure 6 for an illustration.

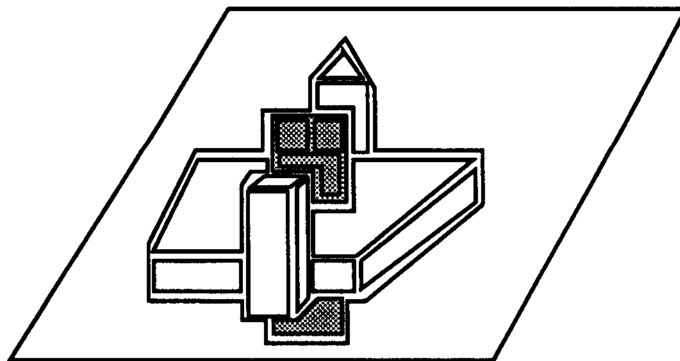


Figure 6: f is visible in the two shaded regions. One of them consists of three subregions that have been merged.

The updating the structures in step 4 can be divided into two tasks: the structures that store F_S, E_S and V_S have to be updated, and the structures that store the sets $R_{\mathcal{M}}$, $E_{\mathcal{M}}$ and $V_{\mathcal{M}}$ must be updated. From the results of Section 2 it follows that the maximum time needed to update any of the structures storing F_S, E_S or V_S is $O(\log^3 n)$. Since P has constant size the total time for the first task is also bounded by $O(\log^3 n)$. The worst update time of any of the structure storing the visibility map is also $O(\log^3 n)$ and, as there are k changes in the map, the second task takes time $O(k \log^3 n)$.

We thus arrive at the following lemma, which summarizes the results of this section.

Lemma 4 *The visibility map of a c–oriented set of polyhedra can be maintained with $O((k+1)\log^3 n)$ time per insertion.*

3.2 Deletions

We now turn our attention to the deletion of a polyhedron P. Because the new regions that appear after the deletion of P have nothing to do with P itself (they were just hidden by P), deletions are harder than insertions. Indeed, one can obtain an output-sensitive solution to the static hidden surface problem by first inserting a large face that hides the whole scene, then insert the polyhedra of the scene, and finally remove the large face. Now the deletion algorithm has to discover the whole visibility map (which possibly has size $\Omega(n^2)$). Observe that we cannot obtain an output-sensitive algorithm by just inserting the polyhedra: since we do not have a depth order, we might create many intersections that are visible in some intermediate stadium of the algorithm, but not in the end.

The basic strategy for the deletion of a polyhedron is as follows. The term *component* in this algorithm refers to a connected component of the graph whose nodes are the vertices of the visibility map and whose arcs are the edges of the map.

1. Find all the regions that are affected by the deletion of P, i.e., all the regions where a face of P is visible.

2. Delete the faces, edges and vertices of P from the data structures that store F_S, E_S and V_S.

3. For each affected region r_i compute the new part of the visibility map 'inside' r_i as follows:

    (i) Compute the new components of $\mathcal{M}(S)$ that are attached to ∂r_i.

    (ii) Compute the new components of $\mathcal{M}(S)$ that 'float' inside r_i.

4. Merge adjacent regions that have the same background face.

5. Update the data structures that store $R_{\mathcal{M}}$, $E_{\mathcal{M}}$ and $V_{\mathcal{M}}$.

In step 1 of the algorithm we find all the affected regions. Because we have pointers from each face of P to the regions where the face is visible, this is a trivial task.

In step 2 we already update the data structures that store F_S, E_S and V_S. This is necessary because these structures will be used in step 3 to compute the new parts of the visibility map. (If we would not do this at this point, then the old visibility map would be rediscovered in step 3.) This takes time $O(\log^3 n)$, by Lemma 1 and Corollaries 1, 2 and 3.

Step 3 is the heart of the deletion algorithm. To compute the part of the visibility map hidden by P in an output-sensitive manner, we use the following method (see also [6, 7, 14]). As noted before, the visibility map can be seen as a graph. In general, this graph consists of several connected components. The method first finds one vertex on each component and then traces along the edges of the component to discover the new vertices and edges. During the second phase (called the *ray shooting phase*) the basic operation is the following: given a vertex $\overline{v}$ of the visibility map and the direction of an edge $\overline{e}$ incident to it, find the other vertex $\overline{w}$ in that direction. In [7], this other vertex is characterized as follows. Let $\overline{\rho}$ be the ray (in the projection plane) starting at $\overline{v}$ in the direction of $\overline{e}$. We will lift $\overline{\rho}$ to obtain two rays ρ and ρ^* in space as follows. Consider the background faces of the two regions that are incident to $\overline{e}$. We define ρ to be the ray whose projection is $\overline{\rho}$ lifted to the higher of these faces and ρ^* to be $\overline{\rho}$ lifted to the lower of the two faces. Notice that if $\overline{e}$ is a part of the intersection of the two faces of different polyhedra or the two faces are incident faces of the same polyhedron then $\rho = \rho^*$. Now the other vertex $\overline{w}$ of $\overline{e}$ satisfies:

Lemma 5 [7] *$\overline{w}$ is the point closest to $\overline{v}$ of the following 'event points':*

- *the projection of an endpoint of the edge in E_S, or the intersection of two faces in F_S, containing ρ*
- *the first intersection of $\overline{\rho}$ with the projection of an edge in E_S passing above ρ^**
- *the projection of the first intersection of ρ with a face in F_S*
- *the projection of the first intersection of ρ^* with a face in F_S*

Thus we need to be able to find the first of each of the four different event points. The first event point is just the projection of an endpoint of an edge or of the intersection of two faces that we know, and therefore we can find it in constant time. For the other three event point we have the structures for ray shooting queries in E_S and F_S.

However, we need one more query. Up to now, we have ignored the fact that, if $\rho \neq \rho^*$, it is not always trivial to compute ρ^*. Since we are computing new regions it can happen that we do not know their background faces yet, and these background faces are needed to compute ρ^*. In Figure 7, for example, we do not know the

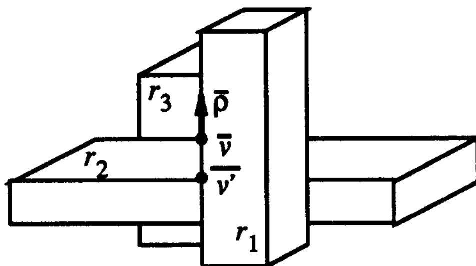


Figure 7: ρ^* is $\overline{\rho}$ lifted to $BF(r_3)$, but the background face of region r_3 is not known when we arrive at $\overline{v}$ from $\overline{v'}$.

background face of r_3 when we arrive from $\overline{v'}$. To compute this background face, we again use the structure for ray shooting queries on F_S. When $\rho^* \neq \rho$, then $\overline{v}$ is the intersection of the projection of two edges. Let v be the point on the lowest of these edges whose projection is $\overline{v}$. When we arrive at $\overline{v}$, the face immediately below v can be found by shooting a ray from v in the viewing direction. This is the 'lower' background face that is needed to compute ρ^*.

Since at most four ray shooting queries are performed that take $O(\log^2 n \log\log n)$ time each, we can compute the other vertex $\overline{w}$ of $\overline{e}$ in $O(\log^2 n \log\log n)$ time.

Now that we have this strategy for computing visibility maps, let us go back to step 3 of our deletion algorithm.

In step 3(i) we compute the new components that are attached to ∂r_i. In fact these components form one new component with the component to which ∂r_i belongs. See Figure 8. This is done as in the shooting phase of the algorithm of [6], as described above. Note that we already have a point to start from: any vertex on ∂r_i will do. Observe that some parts of ∂r_i may consist of edges of P. Still, because we do not want to go outside r_i, we must trace along these parts as well. These parts of ∂r_i will be removed when we merge adjacent regions with the same background face in step 4. The fact that we must stay inside r_i thus imposes a small problem: there is a new type of event point, besides the four types mentioned in Lemma 5. This event point is the first intersection of $\overline{\rho}$ with ∂r_i. Hence, we have to build a data structure for planar ray shooting as in Corollary 2 on the edges of ∂r_i. Since r_i is a region that disappears, we are allowed to spend time the $O(|\partial r_i| \log^2 |\partial r_i|)$ time to build this structure. Moreover, the planar ray shooting takes time $O(\log^2 |\partial r_i|)$, so every new vertex is still discovered in $O(\log^2 n \log\log n)$ time. In all, step 3(i) takes time $O(k \log^2 n \log\log n)$, where k is the sum of the number of disappearing edges (the edges on ∂r_i) and the number of new edges inside r_i that are discovered.

When we have computed the components that are attached to ∂r_i we are not yet ready. There can be a lot of other components 'floating' somewhere in the newly discovered regions $r_{i,j}$ (see Figure 8). These other components are found in step

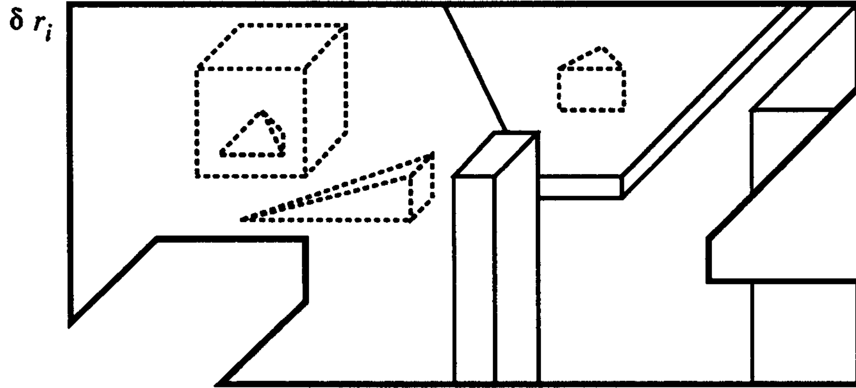


Figure 8: The the non-bold components (which are attached to ∂r_i) are discovered in step 3(i). The dotted components (which are floating inside r_i) are discovered in step 3(ii).

3(ii). Again we use the basic method described above. So we have to find one vertex on each component and then we discover the rest of the component by ray shooting along the edges. Above we have seen how the ray shooting phase can be performed at the cost of $O(\log^2 n \log\log n)$ per edge. We next describe how to find a vertex on each component. To this end we prove a lemma that characterizes the leftmost vertex of all components floating in a new region $r_{i,j}$. The idea is then to find this leftmost vertex and compute the component of this vertex, find the leftmost vertex of the remaining components and compute its component, etcetera, until all components have been computed.

Lemma 6 *The leftmost vertex of all components floating inside a region $r_{i,j}$ is the leftmost of:*
(i) the leftmost vertex in V_S that is above $r_{i,j}$
(ii) the leftmost intersection of an edge in E_S with $r_{i,j}$.

Proof: The vertices of a visibility map are of five different types: they can be the projection of a vertex of a polyhedron, they can be the projection of the intersection of an edge with a face, they can be the intersection of the projection of two edges of a polyhedron, they can be the projection of the intersection of three faces of (different) polyhedra, and they can be the intersection of the projection of an edge with the projection of the intersection of two faces (of different polyhedra). First we observe that the latter three types can never be the leftmost vertex of a component. This follows immediately from the fact that for those three types it is not possible that all three incident edges lie completely to the right of the vertex.

It remains to show that the leftmost of the two points as defined in the lemma

(which are of the first two types) must be visible. Assume for a contradiction that this point, let's call it v, is hidden by some face f.

If the projection of ∂f does not intersect $\partial r_{i,j}$ to the left of v then f completely lies within $r_{i,j}$ (to the left of v). But this contradicts the definition of v: either one of f's vertices must lie to the left of v and above $BF(r_{i,j})$, or one of f's edges must intersect $BF(r_{i,j})$ to the left of v.

On the other hand, if ∂f does intersect $\partial r_{i,j}$ to the left of v then we also get a contradiction: either the intersection is visible in which case v cannot lie inside $r_{i,j}$, or the intersection is not visible in which case f must intersect $BF(r_{i,j})$ to the left of v. $\square$

So we have to search for the leftmost point of each of these two types of points. Since $r_{i,j}$ can be a very complex polygonal region, we have to decompose it into smaller polygons and search with them. More precisely, we decompose $r_{i,j}$ into a number of quadrilaterals by drawing vertical (i.e. parallel to the y–axis) edges from every vertex on $\partial r_{i,j}$ to the opposite edge on $\partial r_{i,j}$. Now we first search with the leftmost of these quadrilaterals, then (if we do not find a point) with the next one, etc. This is repeated until we get an answer, or there are no quadrilaterals left (in which case all components inside $r_{i,j}$ have been computed). To find the leftmost vertex above the quadrilateral we use the range search searching structure of Corollary 2 on V_S, and to find the leftmost intersection of the quadrilateral with an edge we use the range search searching structure of Corollary 3 on E_S. Thus the leftmost point of a component inside the quadrilateral can be found in time $O(\log^3 n)$.

One last issue remains. How do we compute the quadrilaterals? It is not hard to compute these in $O(|\partial r_{i,j}| \log |\partial r_{i,j}|)$ time. Moreover, we are allowed to spend this amount of time since $\partial r_{i,j}$ consists of new edges. However, we cannot do this over and over again, every time we have to compute the leftmost vertex of the remaining components. On the other hand, we cannot use the old quadrilaterals because they also cover the already computed components and we have to restrict ourselves to $r_{i,j}$. Therefore we do not compute all the quadrilaterals before we start querying. Instead, we compute the leftmost quadrilateral first, then perform a query with it and we compute the next quadrilateral only if we do not find an answer. Notice that in that case we know that there is no vertex in this quadrilateral and, hence, we will never need it again.

Let $\overline{v}$ be a vertex of $\partial r_{i,j}$ and suppose we want to compute the quadrilateral $q(\overline{v})$ with $\overline{v}$ on its left side. This can be done in $O(\log^2 n)$ time as follows. Shoot from $\overline{v}$ in y–direction to obtain the edges $\overline{e}$ and $\overline{e'}$ of $\partial r_{i,j}$ that bound $q(\overline{v})$ from above and below. To this end we build a planar ray shooting structure on $\partial r_{i,j}$ as in Corollary 2. (Note that if $\overline{v}$ has no incident edges to its right then we have to perform two queries, and if one the incident edges is to its right then we have to perform one query. If both incident edges are to its right then there are two quadrilaterals with $\overline{v}$ on their left side and we compute both.) Now we have found the left side of $q(\overline{v})$ and we know the two edges $\overline{e}$ and $\overline{e'}$ such that its top and bottom sides are (parts of)

$\overline{e}$ and $\overline{e'}$. The right side is determined by the vertex of $\partial r_{i,j}$ closest to the left side. More precisely, consider the (possibly unbounded) triangle with the same left side as $q(\overline{v})$ and with as its top and bottom side the rays containing the top and bottom side of $q(\overline{v})$. Then the leftmost vertex of $\partial r_{i,j}$ inside this quadrilateral (including the right vertices of $\overline{e}$ and $\overline{e'}$) determines the x-coordinate of the right side $q(\overline{v})$. See Figure 9. This vertex can be found in time $O(\log^2 n)$ by the range searching structure of Lemma 3 on the vertices of $\partial r_{i,j}$ and, hence, $q(\overline{v})$ can be computed in time $O(\log^2 n)$.

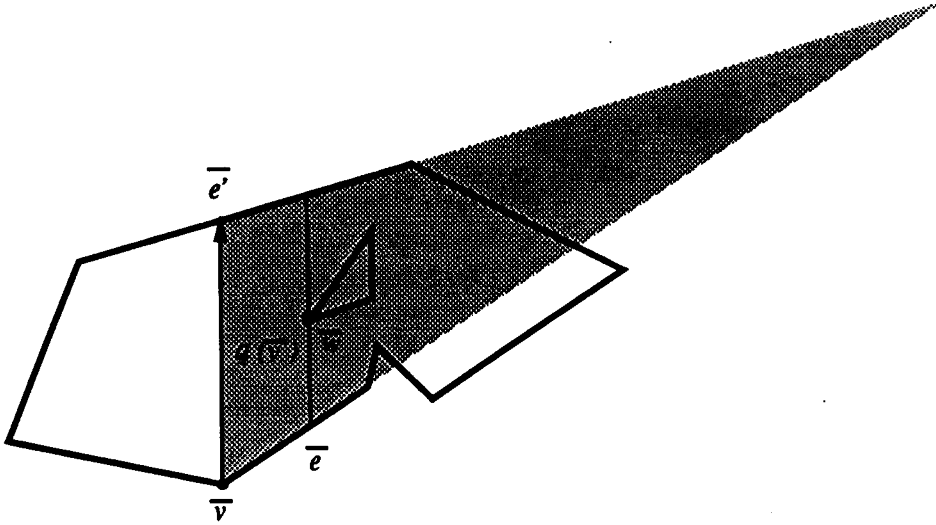


Figure 9: $\overline{w}$ is the leftmost point in the lightly shaded unbounded triangle and, hence, determines the right side of $q(\overline{v})$. Note that $q(\overline{w})$ would consist of two quadrilaterals.

Summarizing, the floating components are discovered as follows:

3(ii) Find the floating components inside each new subregion $r_{i,j}$.

- Store the vertices of $\partial r_{i,j}$ in a priority queue $\mathcal{Q}$ on x–coordinate, build a range searching structure as in Lemma 3 on the vertices of $\partial r_{i,j}$ and build a ray shooting structure as in Corollary 2 on the edges of $\partial r_{i,j}$.

  While $\mathcal{Q}$ is not empty do the following. Remove the vertex $\overline{v}$ with the smallest x–coordinate from $\mathcal{Q}$. Compute the quadrilateral(s) with $\overline{v}$ on its (their) left side as described above. Compute the leftmost vertex (vertices) in this quadrilateral (these two quadrilaterals) by performing two (four) queries, one to compute the leftmost vertex of V_S above the quadrilateral and the other to compute the leftmost intersection point of an edge in E_S with the quadrilateral. If a vertex is found, then compute

the rest of its component by ray shooting along the edges, insert the new vertices of $\partial r_{i,j}$ into Q and the new edges into the ray shooting structure.

- Recurse in the regions of the newly discovered components.

How much time does all this take? We implement the priority queue Q as a heap, so it can be built in time $O(|\partial r_{i,j}|)$; the two other structures in time $O(|\partial r_{i,j}| \log^2 |\partial r_{i,j}|)$ (Lemma 3 and Corollary 2). We have seen above that each quadrilateral can be computed in time $O(\log^2 n)$; furthermore, the two queries with $q(\overline{v})$ take time $O(\log^3 n)$. These costs can be charged to $\overline{v}$. Since $\overline{v}$ is deleted from Q it is charged only once. Hence, the time that we spend computing the leftmost vertices of the new components inside $r_{i,j}$ is bounded by $O(|\partial r_{i,j}| \log^3 n)$. Once we have a vertex on a component we can compute the rest of the component in $O(\log^2 n \log\log n)$ per edge. Summing up over all the newly discovered regions gives a total time of $O(k \log^3 n)$ for step 3(ii).

In Figure 10 an illustration is given of step 3(ii) for one of the subregions of Figure 8. The components that not yet have been found and the quadrilaterals with which we search are dotted and the current boundary of $r_{i,j}$ (i.e. the part that already has been discovered) is bold. First queries are performed with the leftmost quadrilateral but no vertex is found. Vertex $\overline{v_1}$ is found when searching with the next quadrilateral and its component C_1 is computed. Then we discover $\overline{v_2}$ and C_2. All the other searches fail, so we recurse in the regions of the new components. This way C_3 is found. We recurse in the regions of C_3 but we do not find any components and so we are finally ready.

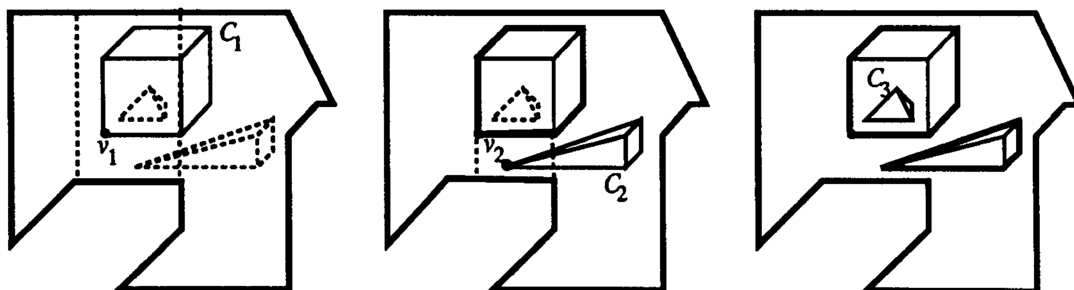


Figure 10: Computing the floating components.

For step 4 of the deletion algorithm we need to know those regions that are adjacent but have the same background face. These can only be regions whose common edge is a part of the projection of an edge of P. Thus all candidate regions can be found using the cross pointers. The merging is then easily performed in time $O(k \log n)$. Finally, step 5 takes time $O(k \log^3 n)$.

This leads to:

Lemma 7 *The visibility map of a c–oriented set of polyhedra can be maintained with* $O((k+1)\log^3 n)$ *time per deletion.*

This completes the proof of Theorem 1.

4 Concluding Remarks

In this paper we have presented an efficient output-sensitive algorithm for maintaining the visibility map of a c–oriented set of polyhedra. Inserting or deleting a polyhedron P (of constant size) takes time $O((k+1)\log^3 n)$, where n is the total number of vertices of the polyhedra and k is the number of changes in the visibility map. The most important feature of the algorithm is that it is the first output-sensitive dynamic hidden surface removal algorithm that can handle cyclic overlap and intersecting polyhedra. The algorithm is based on new dynamic data structures for ray shooting and range searching queries in c–oriented faces and segments in the plane and in space.

The most important open problem is of course (besides improving the time bound of our solution) to extend the results to arbitrary polyhedra. Here it is worthwhile noting that the algorithms presented in section 3 still work in the general case; the fact that the polyhedra are c–oriented is only used in Section 2 to obtain efficient implementations of the data structures that are used by the algorithms. Such efficient data structures are currently not available for the general case.

References

[1] A.V. Aho, J.E. Hopcroft and J.D. Ullman, *The Design and Analysis of Computer Algorithms*, Addison-Wesley, Reading, Mass., 1974.

[2] M. Bern, Hidden Surface Removal for Rectangles, *J. of Comp. and Syst. Sciences* **40**, 1990, pp. 49–69.

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

[4] S.W. Cheng, Dynamic Hidden Line Elimination, 1990, manuscript.

[5] S.W. Cheng and R. Janardan, New Results on Dynamic Planar Point Location, *Proc. 31rst IEEE Symp. on Foundations of Computer Science*, 1990, pp. 96–105.

[6] M. de Berg and M.H. Overmars, Hidden Surface Removal for Axis-Parallel Polyhedra, *Proc. 31rst IEEE Symp. on Foundations of Computer Science*, 1990, pp. 252–261.