# Repetitive Hidden Surface Removal for Polyhedra*

Marco Pellegrini†
Dept. of Computer Science, King's College London

March 15, 1995

**Abstract**

The repetitive hidden-surface-removal problem can be rephrased as the problem of finding the most compact representation of all views of a polyhedral scene that allows efficient on-line retrieval of a single view. We assume that a polyhedral scene in 3-space is given in advance and is preprocessed off-line into a data structure. Afterwards, the data structure is accessed repeatedly with view-points given on-line and the portions of the polyhedra visible from each view-point are produced on-line. This mode of operation is close to that of real interactive display systems. The main difficulty is to preprocess the scene without knowing the query view-points. In this paper we present a novel approach to this problem.

Let $n$ be the total number of edges, vertices and faces of the polyhedral objects and let $k$ be the number of vertices and edges of the image. The main result of this paper is that, using an off-line data structure of size $m$ with $n^{1+\epsilon} \leq m \leq n^{2+\epsilon}$, it is possible to answer on-line hidden-surface-removal queries in time $O(k \log n + \min\{n \log n, kn^{1+\epsilon}/m^{1/2}\})$, when the scene is composed of $c$-oriented polyhedra. This data structure allows dynamic insertion and deletion of polyhedral objects. The polyhedra may intersect and may have cycles in the dominance relation. We also improve worst-case time and storage bounds for the repetitive hidden surface removal problem when the polyhedral scene is composed of unrestricted polyhedra.

# 1 Introduction

## 1.1 The problem

The Hidden Surface Removal problem (HSR for short) for polyhedral scenes in 3-space is important for graphical applications [FVFH90] and has attracted much interest in the computational geometry research community in recent years (e.g. [dBHO+91],[Ber89],[RS88],[AS91],

*Preliminary version of this paper appeared in the Proceedings of the 3rd Workshop on Algorithms and Data Structures. Springer Verlag *Lecture Notes in Computer Science* n. 709 pp. 541-551, 1993.

†Dept. of Computer Science, King's College, Strand, London WC2R 2LS U.K. marco@dcs.kcl.ac.uk

[OS89],[dBO92] and [Mul91]). Surveys of old and recent results can be found in Foley et al. [FVFH90], in de Berg's thesis [dB92b], and in the survey article of Sutherland et al. [SSS74].

Given a set $P$ of objects bounded by polygonal faces which we assume to be opaque, a view-point $v$, and a view-plane $V$, the problem consists in computing the visible portion of the scene as viewed from $v$ and projected on $V$. We consider the "object space" variant of this problem where the output of the algorithm is a planar graph representing the subdivision of $V$ into maximal connected regions in each of which a single face of an object (or the background) is seen. This subdivision is called *visibility map* of $P$ from $v$, denoted by $M(v, P)$. Other solutions are classified as "image space" since the aim is to compute for each pixel of the screen the object visible at that pixel [SSS74].

We assume in this paper that the polyhedra are given in advance and are preprocessed into a data structure. This step is critical since we do not know at this time which viewpoints will be used. Therefore, our data structure must be independent from any fixed viewpoint. Afterwards, the data structure is accessed repeatedly with query view-points and the visibility map for each viewpoint is produced on-line (*repetitive mode*). This scenario should be compared with the *single-shot mode*, in which *both* the polyhedral scene *and* a view-point are processed together to produce the output visibility map. No data structure is saved and if a new viewpoint is given the process is repeated from scratch. In the real use of interactive display systems the repetitive mode of operation is more natural than the single-shot one. For example in virtual reality applications the user should have on the screen the impression of a smooth walk through an environment. In most cases it is neither possible nor desirable to determine beforehand the positions of the view-points. On the other hand the images should be produced at an high speed in response to interactive changes of the view-point. These constraints lead us to design a system where we allow massive off-line computations to take place if we can reduce the time to produce the image once the viewpoint is known.

Our first objective in this paper is to achieve optimal output-sensitive query time (up to polylogarithmic factors) and minimize the worst case storage requirement. Then we modify this initial solution by allowing query-time and storage trade-offs. We consider first the case of $c$-oriented polyhedra (i.e. the facets of the polyhedra belong to $c$ sets of parallel facets) for which more effective time bounds can be proved. In the last part of the paper we apply the same approach to the case of general polyhedra, thereby improving worst-case time and storage bounds.

## 1.2 The binary space partition approach

Given a set of non-intersecting polyhedra with a total of $n$ vertices, edges and faces a *binary space partition* for this set has size $\Theta(n^2)$ [FKN80, PY90]. When query view-point is given, this data structure is searched to produce a valid visibility order, which is used to produce an image-space solution (e.g. by using the painter's algorithm [FVFH90]). Since the every node in the data structure is reached during the search, the visibility order is retrieved in time $\Theta(n^2)$, irrespective of the size of the visibility map.

## 1.3 The aspect graph approach

Traditionally, the problem of efficiently computing an object-space representation of the visibility map of a polygonal scene has been tackled using the *aspect graph* approach [KvD76, KvD79, PD86, GCS91, GM90]. The aspect graph of a polyhedral scene is a graph in which each node is associated with a region of $R^3$ where visibility maps with the same combinatorial structure are visible. Provided that we can store efficiently the visibility maps at the nodes of the graph and that, given a view-point we can find efficiently the corresponding node in the graph, the storage depends mainly on the number of nodes and edges in the aspect graph. Gigus, Canny and Seidel [GCS91] give an algorithm for computing the aspect graph of *orthographic views* (i.e with viewpoint at infinity) of a general polyhedral scene. They use $O(n^4 \log n + |G| \log |G|)$ time to produce the aspect graph $G$. This graph is stored in a data structure of size $O(|G|)$ and orthographic HSR-queries can be solved in time $O(\log |G| + k)$, where $k$ is the size of the output visibility map. This method does not seem to generalize immediately to computing also the aspect graph of *perspective views* (i.e. with a proper point as view-point). For perspective views, the aspect graph is computed in time $O(n^5 + n^2|G|)$ and $O(n^2|G|)$ working storage [PD90].

The number of orthographic views generated by $n$ non-intersecting triangles is $\Theta(n^6)$ in the worst case and the number of perspective views is $\Theta(n^9)$ [PD90]. For axis oriented polyhedra with $n$ edges the number of orthographic views is $\Theta(n^6)$ in the worst case [Sno91]. For a polyhedral terrain with $n$ edges there can be $\Omega(n^5\alpha(n))$ orthographic views and $\Omega(n^8\alpha(n))$ perspective views [dBHOvK92].

The aspect graph approach, in a nutshell, consists in computing all possible views and storing them compactly by exploiting coherence between neighbouring views. This approach is only partially satisfactory. On one hand it shows a degree of sensitivity to the "complexity" of the scene measured by the number of possible different combinatorial views. The drawback is that this measure of complexity is quite rough and leads easily to high storage requirements even for fairly simple scenes. No storage/query-time trade off is possible and the method is not sensitive to restrictions in the orientations of the input polyhedra.

In this paper we prove that for $c$-oriented polyhedra $O(n^{2+\epsilon})$ preprocessing time and storage are always sufficient to answer efficiently *both orthographic and perspective HSR-queries* (Theorem 6). In this paper we prove that for general polyhedra $O(n^{4+\epsilon})$ pre-processing time and storage are always sufficient to answer efficiently *both orthographic and perspective HSR-queries* (Theorem 9). Moreover, it is possible to trade off continuously storage and query time. The main conclusion suggested by the new results is that we should be looking for new and more effective measures of the complexity of a polyhedral scene.

Other approaches to the hidden surface removal problem in repetitive mode are discussed by Mulmuley [Mul91] and Bern et al. [BDEG90].

## 1.4 Previous results on one-shot HSR for $c$-oriented polyhedra

An algorithm proposed by de Berg and Overmars [dBO90, dBO92] for HSR on $c$-oriented polyhedra works in *single-shot mode*, with a time complexity $O((n + k) \log n)$. The view of

$c$-oriented polyhedra from a *fixed point of view* can be maintained efficiently when we insert and delete polyhedra [dB92a].

Since our algorithm borrows some tricks from the algorithm of de Berg and Overmars [dBO92] it is worth to discuss this algorithm in more detail. This method is based on the following primitive operations: (i) ray-shooting queries on "curtains" (see also Bern [Ber89]) and (ii) vertex-visibility queries. Each component of $M(v, P)$ is "traced" using these two primitives. The $O(n \log n)$ overhead term accounts for the construction of the data structures supporting efficient answers to the queries, and for performing a vertex-visibility query for each vertex in $P$.

If we try to use this method for the solution of repetitive HSR we run into the following complications. The data structures for ray-shooting on curtains and for vertex-visibility depend on the viewpoint $v$ and cannot be updated in sublinear time for a new viewpoint $v'$. Moreover, in order to obtain optimal output sensitive query time we cannot afford to consider independently each vertex for visibility. We shall see in this paper how these difficulties can be overcame.

### 1.5 Summary of results

For a set of $c$-oriented polyhedra with a total of $n$ vertices, edges and faces we build off-line data structures of size $m$, where $m$ is a parameter to be chosen between $n^{1+\epsilon}$ and $n^{2+\epsilon}$. Afterwards, the visibility map $M(v, P)$ from a view-point $v$ is produced in time $O(k \log n + \min\{n \log n, kn^{1+\epsilon}/m^{1/2}\})$. If we allow the maximum storage $m = n^{2+\epsilon}$ the query time is $O(k \log^2 n)$, which is almost optimal for all values of $k$. If we allow $m = n^{1+2\epsilon}$ units of storage, the query time is $O(k \log n + \min\{n \log n, kn^{1/2+\epsilon}\})$ (see Figure 1).

We consider also the dynamization of our data structure under insertion and deletion of $c$-oriented polyhedral objects from the scene. The main new idea is that the amortized cost of an update (insertion or deletion) can be split between an *on-line update cost* and an *off-line update cost*. The objective is to minimize the on-line update costs since these are critical for the performance of the algorithm. We suppose that we have a mixed sequence of HSR-queries, insertion and deletions. Let $n$ be the number of edges of polyhedra initially present in the scene and let $n_I$ and $n_D$ be the number of edges of the polyhedra inserted and deleted over the whole sequence of operations. We obtain on-line amortized update time that depends *only* on $n_I$ and $n_D$, *not* on $n$. This results in significant when $n_I, n_D << n$, which is to be expected in many real applications during a session of use of the system (Section 4.2). We place no other restriction on the polygonal scene formed by $c$-oriented polyhedra. In particular, we do not require disjointness of the input polyhedra. As a consequence, the polygonal scene can be specified as the union of polyhedral objects with fewer edges. We do not require acyclicity in the dominance relation.

The same general strategy used to solve the HSR problem for $c$-oriented polyhedra works for general polyhedra with somewhat degraded time and storage performance. For a general set of (intersecting, non-convex) polyhedra with $n$ vertices, edges and faces we can build a data structure of size $m$, with $n^{1+\epsilon} \leq m \leq n^{4+\epsilon}$ such that any HSR-query from a point $v \in R^3$ is answered in time $O(kn^{1+\epsilon}/m^{1/4})$. This result improves over the best one-shot method for small values of $k$. Thus, running this algorithm and the best one-shot algorithm [AM92b] in parallel, we can achieve query time $O(\min\{kn^{1+\epsilon}/m^{1/4}, n^{2/3+\epsilon}k^{2/3} + n^{1+\epsilon} + k\})$ (see Figure 3, for $m = n^2$).

### 1.6 The method

The main underlying theme of this research is to extend some methods used for one-shot HSR in order to solve the repetitive HSR problem, thus improving on the results attained through the aspect graph approach. To this purpose, we mix and modify the approaches of de Berg and Overmars [dBO92, dBHO+91]. We use a sweeping line approach, but we avoid initializing the queue with all the vertices. Instead, we discover the visible vertices on the fly. Vertices that are not visible are never put in the queue. The second challenge is to make our data structure independent of $v$. We attain this objective at a cost of an increase in the storage requirement. Then, we return to linear storage by trading off storage with query time.

In order to achieve these two goals we must support several powerful operations. In particular, beside supporting ray-shooting queries we must be able to count efficiently how many edges of the polyhedral scene meet query *triangles* and query *pyramids*. Moreover, we apply Megiddo's parametric search technique [Meg83] in order to obtain even more powerful operations. Intuitively, we need to detect efficiently the first moment in which a growing triangle or pyramid intersects a feature of $P$. We implement these operations for $c$-oriented polyhedra by reductions to planar point location and half-plane range queries. The algorithm in Section 2 is quite general and works also for general polyhedral scenes when we provide an implementation for the primitive operations. The implementation of the primitive operations for general polyhedra is based on results by the author [Pel94] on collision-free simplices.

We will assume familiarity with standard techniques in computational geometry such as arrangements [Ede87], cuttings [Mat91], multilevel data-structures [Meh84], and parametric search [Meg83]. The paper is organized as follows. In Section 2 we describe a high-level generic algorithm for solving hidden-surface removal queries, which we will denote as the `sweep` algorithm. In Section 3 we discuss the implementation of the primitive operations for a set of $c$-oriented polyhedra. In Section 4 we discuss query-time/storage trade-offs and dynamization of the data structures under insertions and deletions. In Section 5 we discuss the case of general polyhedral scenes.

## 2 The `sweep` Algorithm

In this section we give a skeleton algorithm for solving hidden surface removal queries. This skeleton algorithm uses the off-line data structures as black boxes (i.e asking queries and receiving answers) and is correct independently of any particular implementation of these data structures. The final time bound depends on an efficient implementation of the off-line data structures.

We assume that our set of input polyhedra $P$ is compact, therefore we can find a (finite) box $B$ containing the input set. We assume that all the query points are within the box $B$. None of these conditions is essential; we introduce them as a way of simplifying the arguments. We want to compute the (portions of) facets of polyhedra visible from a query point $v$. More precisely we will compute the projection from $v$ onto the boundary of $B$ of the visible facets. This is equivalent to finding the visible objects in any direction (i.e. we assume a visibility angle of $4\pi$ ste-radiants)[1]. We compute the projection of visible facets independently for each of the six sides of the box $B$. Next we explain the algorithm for one of the faces of $B$, which we call $V$. We also fix a direction on $V$ which we call *vertical*.

The general structure of the algorithm is reminiscent of Bentley and Ottman's line-sweep algorithm for reporting intersections of segments in the plane [BO79]. We assume that the background screen $V$ is part of the input $P$ whenever it is convenient to do so. The visibility map $M(v, P)$ on $V$ will be constructed by sweeping a vertical line $l(x)$ on $V$, starting from one of the sides of $V$, say the leftmost one, and moving the sweeping line towards the right hand side. As in Bentley and Ottman's algorithm we need to keep two data structures: a dictionary $Line\_status(x)$ which stores the intersection of $M(v, P)$ with the current line $l(x)$, and the priority queue $Event\_queue(x)$ which stores the list of "events" in sorted order along the $x$-axis.

As a matter of convention we will denote with $e$ an edge in 3-space and with $e'$ its projection on $V$. Let $e'$ be an edge stored in $Line\_status(x)$ which is the projection of edge $e$ in 3-space. While $e'$ is traced ahead (left-to-right) we shall detect the *first* of the following events (see Figure 1).

(i) The right end-point of $e$. At this event the edge $e'$ should be deleted from $Line\_status$.

(ii) Edge $e_1$ from face $f_1$ obscures $e$, which is therefore no longer visible. We delete $e'$ from $Line\_status$ and insert $e'_1$ if not already present.

(iii) Edge $e_1$ from face $f_1$ is partially obscured by $e$. We insert $e'_1$ in $Line\_status$ if not already present.

(iv) Edge $e$ intersects face $f_1$. If $f$ is the face containing $e$, then the intersection of $f$ and $f_1$ is a new visible edge, which is inserted in $Line\_status$.

(v) The intersection of three facets of polyhedra in $P$. All new visible edges from this intersection point are inserted in $Line\_status$.

There is a further event that cannot be discovered by tracing a single edge present in $Line\_status(x)$:

(vi) A visible vertex $q$ of a face in $P$ projects on $l(x)$ from $v$. In this case all the visible edges incident to $v$ are inserted in the data structure.

[1] We leave the problem of windowing as an exercise for the reader.

Event (vi) is quite important because it allows us to find on-line all the connected components of the visibility map. These are all and only the interesting events [dBO92, dB92b].

Since each edge is traced until visible and discarded afterwards until it might be visible again, at each instant $Line\_status(x)$ stores a faithful representation of $M(v,P) \cap l(x)$. Since each event corresponds to a feature of the visibility map it can be charged to the output size $k$. Each event is not detected more then a constant number of times. An edge can be inserted and deleted several times but each operation corresponds to a distinct feature of $M(v,P)$. The algorithm proceeds by executing the update corresponding to the next event in $Event\_queue$ and by updating $Event\_queue$ itself with the new events induced by the edges which have been manipulated. All the operations on $Line\_status$ and $Event\_queue$ take $O(\log n)$ time using standard data structures. The primitive procedures needed to discover events are the following:

1. `Ray-shooting` . Given a point $p$ and a ray $\rho$ from $p$, `Ray-shooting` returns the first face of a polyhedron in $P$ intersected by $\rho$, and the intersection point.

2. `Empty-triangle` . Given a triangle $t$ in 3-space, `Empty-triangle` determines whether $t$ intersects any face or edge or vertex in $P$.

3. `Largest-empty-triangle` . Given a family of triangles $t(s)$ depending on a positive real value $s$ such that $s_1 < s_2$ implies $t(s_1) \subset t(s_2)$, `Largest-empty-triangle` reports the minimum value of $s$ for which $t(s)$ intersects a feature of $P$ and the feature of $P$ for which this happens.

4. `Largest-empty-pyramid` . We are given as query a family $\Gamma(s)$ of pyramids with quadrangular base, depending on a positive real value $s$ such that $s_1 < s_2$ implies $\Gamma(s_1) \subset \Gamma(s_2)$. `Largest-empty-pyramid` reports the minimum value of $s$ for which $\Gamma(s)$ intersects a feature (vertex, edge, face) of $P$ and the feature of $P$ for which this happens.

**Setting up Line_status**. Initially we set up $Line\_status(0)$ in the following way. We take the lowest point $a$ of $l(0)$ on $V$, and we *shoot a ray* from $v$ towards $a$. We determine the point $q$ on the first face $f$ hit by the ray. By definition $q$ is visible. Then we take on the face $f$ the line through $q$ which projects from $v$ onto $l(0)$. We parametrize this line with a generic point $p(s)$, where $p(0) = q$ and $s$ is a positive real parameter. Let $\bar{s} > 0$ be the value of $s$ for which $p(s)$ is on the boundary of $f$. We consider the triangle $t(s)$ of vertices $v, q, p(s)$. We call `Empty-triangle` with $t(\bar{s})$, if $t(\bar{s})$ is empty we have discovered the first point on a visible edge of $M(v,P)$. By shooting a ray from $v$ towards $p(\bar{s}+\delta)$, for an infinitesimaly small value $\delta$, we find a point on the first visible face behind $f$ (or the background plane $V$) and we repeat the tracing procedure upwards on this new face (or $V$).

If $t(\bar{s})$ is not empty, we use `Largest-empty-triangle` on the family of triangles $t(s)$ and we find a point on a visible edge of $M(v,P)$. This point belongs to an edge of a face $f'$ different from $f$. We repeat this tracing procedure upwards on $f'$. We stop this procedure when we reach the topmost point of $l(0)$ on $V$. This completes the construction of the initial data structure $Line\_status(0)$.

**Computing events**. Let $e'$ be an edge in $Line\_status$, $e$ the edge in 3-space generating $e'$, let $q$ be a visible point of $e$ projecting from $v$ onto $l(x)$, and let $f$ be a visible face containing $e$ (if both faces incident to $e$ are visible, choose one arbitrarily). We parametrize the edge $e$ by a generic point $p(s)$ where $p(0) = q$ and $s$ is a positive real value. Let $\bar{s} > 0$ be the value of $s$ for which $p(s)$ is the right endpoint of $e$.

1) If the triangle $v, q, p(\bar{s})$ is empty then we have an event (i). Otherwise, by invoking `Largest-empty-triangle` on a family with vertices $v, q, p(s)$ we detect the first event of type (ii), (iv) or (v).

2) By shooting a ray from $v$ towards $q$ we find a point $q'$ on the first visible face $f'$ different from $f$. By projecting $e$ onto $f'$ and repeating the steps at 1) we can detect an event of type (iii). Note that we deal implicitly also with the case when both faces incident to $e$ are visible. In this case $f'$ is one of these faces.

3) In order to detect events of type (vi) we proceed in the following way. We consider a pair of consecutive edges $e'_1$ and $e'_2$ in $Line\_status$. Let $f$ be the face of $P$ visible between $e_1$ and $e_2$ at $l(x)$. We project both $e_1$ and $e_2$ onto this face obtaining $e''_1$ and $e''_2$. We consider a parametrization of $e''_1$ and $e''_2$ with generic points $p_1(s)$ and $p_2(s)$ such that the segment $p_1(s)p_2(s)$ projects on $V$ into a segment parallel to $l(x)$. We call `Largest-empty-pyramid` with a pyramid of vertices $v, p_1(0), p_2(0), p_1(s), p_2(s)$. The value of $s$ returned corresponds to a new visible vertex of $P$ or to some other event (i)-(v) relative to the edges $e'_1$ and $e'_2$. In order to detect vertices and edges lying on the face $f$ we must consider the query pyramid as a closed set.

Initially we compute events for the edges and pairs of consecutive edges in $Line\_status(0)$. On the fly, at each update of $Line\_status$, we insert new events in $Event\_queue$ relative to the edges just updated and their neighbours in the linear order stored in $Line\_status$.

When a new visible vertex $q$ is discovered (event (vi)), we insert in $Line\_status$ the new visible edges incident to $q$. For $c$-oriented polyhedra in general position the number of edges incident to a vertex is bounded by a constant, therefore we can afford to check every incident edge for visibility. The case of general polyhedral scenes is more complex since the degree of a vertex is not bounded.

If $q$ projects from $v$ on the sweeping line at $l(\bar{x})$, we consider the perturbed position $l(\bar{x} + \delta)$ for an infinitesimal $\delta > 0$. We choose a point $p$ to the right of $q$ that projects on $l(\bar{x} + \delta)$ and we use twice the procedure for setting up $Line\_status$ in the neighbourhood of $q$ (i.e. at $l(\bar{x} + \delta)$ instead of $l(0)$). Using $p$ as the initial target for ray-shooting. We stop the procedure when we find a visible edge not incident to $q$ above $p$ and one below $p$. Any such edge must be already present in $Line\_status(\bar{x})$, so it is easy to check this halting condition. All the visible edges discovered during this scan are incident to $q$. Conversely, any visible edge incident to $q$ is discovered. The number of primitive queries is proportional to the number of visible edges incident to $q$. We summarize this discussion with the following theorem:

**Theorem 1** *Algorithm* `sweep` *requires $O(k)$ calls to off-line data structures to produce $M(v, P)$. All other operations require total time $O(k \log n)$.*

# 3 Primitive operations for $c$-oriented polyhedra

## 3.1 A computational lemma

Let $\mathcal{A}$ and $\mathcal{B}$ be two classes of geometric objects, let $A$ be a finite subset of $\mathcal{A}$, and let $b$ be an element of $\mathcal{B}$. In our problem the family $\mathcal{A}$ will represent $c$-oriented polyhedra and $c$-oriented facets. The class $\mathcal{B}$ will represent segments, triangles, and pyramids.

Our objective is to compute efficiently $|\{a \in A | F(a, b)\}|$, where $F(a, b) = \bigwedge_{j=1}^{w} C_j(a, b)$ is a conjunction of a *constant* number $w$ of *elementary conditions.* Intuitively, $F(a, b)$ is true when $a$ intersects $b$.

Let $S_a^j(.)$ be a line in $R^2$ whose coefficients depend on $a \in A$ according to a function which is computable in constant time and uses only the parameter $j$ and properties of the classes $\mathcal{A}$ and $\mathcal{B}$. Let $p_b^j$ be a point in $R^2$ whose coordinates depend only on $b \in \mathcal{B}$ according to a function which is computable in constant time and uses only the parameter $j$ and properties of the classes $\mathcal{A}$ and $\mathcal{B}$. The elementary conditions $C_j(a, b)$ must be of the form $S_a^j(p_b^j) \geq 0$. For technical reasons we add an initial extra condition $C_0(a, b) = TRUE$. If $S_a^j(p_b^j) > 0$ we say that the *point* $p_b^j$ is on the *positive side* of the *line* $S_b^j(.) = 0$. From now on we will use the terms *point* and *line* in this context. We define $\mathcal{N}_j(A, b)$ to be $|\{a \in A| \bigwedge_{i=0}^{j} C_i(a, b)\}|$, where $j$ is any integer between 0 and $w$. We are interested thus in computing $\mathcal{N}_w(A, b)$. Let $n$ be the cardinality of $A$.

**Lemma 1** *For every $0 \leq j \leq w$, we can build in time and storage $O(n^{2+\epsilon})$ a data structure depending only on $A$ such that, for every $b \in \mathcal{B}$, $\mathcal{N}_j(A, b)$ is computed on-line in time $O(\log n)$.*

*Proof.* Let $T_j(n)$ be the time needed to build the data structure up to level $j$. We prove the claim by induction on $j$. For $j = 0$, by default $C_0(a, b) = TRUE$, therefore $\mathcal{N}_0$ is the number of elements in $A$. We can determine this number easily in linear time. We have thus $T_0(n) = O(n)$, which satisfies the time bound.

Assume $j > 0$. Applying Matoušek's partitioning technique [Mat91] on the set of lines $\{S_a^j(.) = 0 | a \in A\}$ we compute in time $O(nr)$ a decomposition of the 2-dimensional space into $O(r^2)$ triangles with the property that each triangle is intersected by no more than $n/r$ lines in $\{S_a^j(.) = 0 | a \in A\}$. Also we build in time $O(r)$ a fast planar point location data structure [EGS86, Kir83], which attains $O(\log r)$ query time.

For each triangle $\tau$ we select a point in its interior and we compute the set $S_\tau^+$ of lines not intersecting $\tau$ and in positive position with respect to $\tau$. We associate the cardinality of $S_\tau^+$ with $\tau$ and we recurse this construction on the lines intersecting $\tau$. Also, for each triangle $\tau$ and for the subset $A_\tau$ of $A$ whose corresponding line is in $S_\tau^+$ we build the data structure to compute $\mathcal{N}_{j-1}(A_\tau, b)$, which can be built in time $T_{j-1}(n)$. The total time and space for this construction satisfies the recurrence:

$$T_j(n) \leq \begin{cases} O(1) & \text{if } n = O(1) \\ O(r^2)T_j(n/r) + O(r^2)T_{j-1}(n) + O(nr) + O(r) & \text{otherwise} \end{cases} \quad (1)$$

Assuming by induction hypothesis that $T_{j-1} = O(n^{2+\epsilon_{j-1}})$, the solution to this recurrence is $T_j(n) = O(n^{2+\epsilon_j})$ for an $\epsilon_j > \epsilon_{j-1}$. Note that the result for the time and storage holds also when we choose $r$ to be a small power of $n$.

Given $b$ we compute the query point $p_b^j$ and we locate the triangle $\tau$ containing $p_b^j$ using the fast point-location data structure. Then, we recurse on the data structures associated with $\tau$ using $p_b^j$ in one data structure and the query point $p_b^{j-1}$ in the other. At the last level we collect the counters of the positive regions and we sum them up to form the final result. The query time $Q_j(n)$ satisfies this recurrence:

$$Q_j(n) \leq Q_j(n/r) + Q_{j-1}(n) + O(\log r)$$

If we assume $r = n^\nu$ for a small value $\nu$, and the inductive hypothesis $Q_{j-1} = c_{j-1}\log n$, we obtain the total query time $Q_j(n) = c_j \log n$ for a larger constant $c_j \geq c_{j-1}$. ∎

In the rest of this section we discuss the implementation of the four primitive operations used by algorithm `sweep` when the input is restricted to $c$-oriented polyhedra. Initially we shall see how to answer each query in polylogarithmic time at the expenses of quadratic storage. The general idea is to show that any of the queries used by the `sweep` algorithm is reducible to the computation of $\mathcal{N}_w(A, b)$ for some formula $F$ of the type described above. It is sufficient to show a reduction to any Boolean formula $F$ composed of elementary conditions. Afterwards, $F$ can be put in disjunctive normal form $F_{DNF} = \bigvee_i F_i$, where each $F_i$ is a conjunct. Moreover, we can transform further $F$ by imposing that at most one disjunct is verified for a given pair $(a, b)$ by setting

$$F = \bigvee_i (F_i \wedge (\bigwedge_{j=0}^{i-1} \neg F_j)).$$

If the initial formula has a constant number of elementary conditions each conjunct in the new formula will have constant length.

## 3.2 Ray shooting on $c$-oriented polyhedra

**Theorem 2** *Given a set of c-oriented polyhedra with $n$ edges there exists a data structure of size $O(n^{2+\epsilon})$ that answers ray shooting queries in $O(\log^2 n)$ time. The data structure can be built using $O(n^{2+\epsilon})$ time and storage.*

*Proof.* Agarwal and Matoušek [AM92b] obtain a method for ray-shooting among hyperplanes by applying parametric search on an algorithm to test whether a segment does not meet any of the input hyperplanes. Using the same general principle we shall obtain a method for ray-shooting on $c$-oriented polyhedra once a method for counting incidences between segments and $c$-oriented facets in 3-space is given.

Counting problems are easily decomposable. If $\mathcal{F} = \mathcal{F}_1 \cup \mathcal{F}_2$ is the set of facets bounding the $c$-oriented polyhedra, the answer for $\mathcal{F}_1$ and the answer for $\mathcal{F}_2$ can be combined in $O(1)$ time to give the answer for $\mathcal{F}$. We partition the facets of the $c$-oriented polyhedra into $c$ sets of parallel facets, and we solve the counting problem on each set separately. After applying a suitable affine transformation we can consider one such set of parallel facets as horizontal. Any such $c$-oriented horizontal facet has at most $2(c-1)$ edges and moreover each such edge is orthogonal to one of $c$ fixed planes $P_1, .., P_c$ *which depend only on the initial choice of the* $c$ *orientations*. We denote with $q^i$ the orthogonal projection of a set $q$ upon the plane $P_i$. Let $l(s)$ be the line spanning the segment $s$ and $\mathit{aff}(f)$ be the plane spanning the facet $f$. We have the following easy lemmas:

**Lemma 2** *For any segment $s$ and facet $f$ in 3-space which are not co-planar:*

$$s \cap f \neq \emptyset \iff s \cap \mathrm{aff}(f) \neq \emptyset \wedge l(s) \cap f \neq \emptyset$$

Note that $s \cap \mathit{aff}(f) \neq \emptyset$ is a comparison of the end-points of $s$ with the plane spanning the facets $f$. Since these planes are all parallel, this is a comparison in a 1-dimensional space.

**Lemma 3** *Given a c-oriented horizontal polygon $f$, and planes $P_1, ..P_c$ as above, a line $l$ intersects $f$ if and only if $l^i$ meets $f^i$, for all $i = 1, .., c$.*

*Proof.* Let $\mathit{aff}(f)$ be the plane spanning $f$ and let $p = l \cap \mathit{aff}(f)$. If $p \in f$ then for every $i$, $p^i \in f^i$. If $p \notin f$ then there is an edge $e$ of $f$ separating $p$ from $f$. If $P_j$ is the projection plane associated with $e$, then $p^j \notin f^j$. ∎

Given a segment $s$ on the plane, let $l(s)$ be the line supporting $s$, $rhp(s)$ the right endpoint and $lhp(s)$ the left endpoint. $Slope(l)$ is the slope of line $l$.

**Lemma 4** *Two segments $s_1$ and $s_2$ on the plane intersect if and only if $l(s_1)$ meets $s_2$ and $l(s_2)$ meets $s_1$.*

**Lemma 5** *A line $l$ and a segment $s$ on the plane meet each other in either of these two cases:*

*A)* $(Slope(l) < Slope(l(s))) \wedge (lhp(s) \textit{ below } l) \wedge (rhp(s) \textit{ above } l)$

*B)* $(Slope(l) > Slope(l(s))) \wedge (lhp(s) \textit{ above } l) \wedge (rhp(s) \textit{ below } l)$

*where the above and below relations refer to the vertical direction on the plane containing $l$ and $s$.*

*Proof.* Follows from elementary geometry. ∎

Lemmas 2, 3, 4 and 5 imply that we can write a formula of type $F$ to denote the fact that a segment meets a c-oriented facet. Moreover, all of the elementary conditions involve testing a point versus a line in some 2-dimensional space or a point versus a point in a 1-dimensional space. Thus, from Lemma 1, we can count efficiently how many $c$-oriented facets meet a query segment, and thus also decide if the query segment is empty of intersections. ∎

As a matter of fact, by extending a slightly different approaches to ray-shooting for axis-oriented polyhedra (a special case of $c$-oriented polyhedra), it is possible to avoid using parametric search and to reduce the query time to $O(\log n)$ [Pel93, dB92b]. We choose to present this weaker result for sake of uniformity with the algorithmic techniques presented in this paper. Using this ray-shooting primitive we can also check whether the view-point $v$ is interior to any polyhedron in $P$. We consider the polyhedra as solid objects, thus being in the interior of one object can be regarded as a non-physical situation. We send a ray from $v$ in an arbitrary direction. If the ray meets a face of $P$ coming from the interior of the polyhedron containing this face we conclude that $v$ is an interior point of $P$. From now on we consider $v$ to be external to $P$.

### 3.3 Empty triangle queries

Given a set of $c$-oriented polyhedra $P$ and a triangle $t$ we want to check whether $t$ meets any polyhedron in $P$. In our application we use triangles $t$ such that at least one vertex in external to any polyhedron in $P$. The following lemmas hold easily:

**Lemma 6** *A triangle $t$ with a vertex disjoint from $P$ intersects $P$ if and only if an edge of $P$ meets $t$ or an edge of $t$ meets a face in $P$.*

**Lemma 7** *For any segment $s$ and triangle $t$ in 3-space which are not co-planar:*

$$s \cap t \neq \emptyset \iff s \cap \mathrm{aff}(t) \neq \emptyset \wedge l(s) \cap t \neq \emptyset.$$

Now we are set to prove the following theorem:

**Theorem 3** *Given a set of c-parallel polyhedra $P$ of complexity $n$, we can implement the calls of* `Empty-triangle` *in* `sweep` *in time $O(\log n)$ using data structures of total size $O(n^{2+\epsilon})$.*

*Proof.* We can easily check whether an edge of $t$ meets a face of $P$ using three ray-shooting queries. In order to check the edges of $P$ against $t$ we partition those edges into $\binom{c}{2}$ sets of parallel edges and we test each set independently.

Let $S$ be a set of parallel segments in 3-space and $t$ our query triangle. It is crucial for the analysis of the storage required by this data structure to recall that the planes $\mathit{aff}(t)$ used in by the `sweep` algorithm in Section 2 are of a special kind. Every such plane must contain an edge of $P$, or must be parallel to the sweeping line. We have thus $\binom{c}{2} + 1$ sub-families of planes, where planes in each sub-family must all meet a fixed point at infinity. We project the segments in $S$ from such point at infinity onto the auxiliary plane obtaining a planar set of segments $S'$. We project the plane $\mathit{aff}(t)$ from such point at infinity onto the auxiliary plane obtaining a line $l_t$. The first test in the formula of Lemma 7 is thus reduced to detecting the segments in $S'$ intersected by the line $l_t$, for which we can invoke the formula of Lemma 5. The second test in the formula of Lemma 7 is reduced to the following computation: we project the segments onto a plane orthogonal to the common direction of segments in $S$, obtaining a set of points $S''$. Also, we project triangle $t$ onto this plane obtaining a triangle $t''$. We now require that a point of $S''$ is in the intersection of three half-planes defined by the lines spanning edges of $t''$. All of these conditions are elementary conditions. We can write a formula of type $F$ for counting the number of edges intersecting a query triangle $t$ and also determine if the triangle is empty. The time and storage bounds follow from Lemma 1. ∎

### 3.4 Implementation of `Largest-empty-triangle`

Let us consider a family of planar triangles in 3-space $t(s)$ indexed by a positive real parameter $s$, forming an *inclusion chain* (i.e. $s_1 < s_2 \Rightarrow t(s_1) \subset t(s_2)$). It is easy to see that the property "$t(s)$ intersects $P$" is monotone for such set of triangles (i.e. there is a value $s^*$ such that, for all $s < s^*$, $t(s)$ does not meet $P$ and, for all $s > s^*$, $t(s)$ meets $P$). Therefore we can use Megiddo's parametric search to transform the algorithm of Theorem 3 into an algorithm for finding $s^*$ [Meg83]. The query time of the transformed algorithm is the square of the time of the original one.

**Theorem 4** *Given a set of c-parallel polyhedra $P$ of complexity $n$, we can implement the calls of* `Largest-empty-triangle` *of algorithm* `sweep` *in time $O(\log^2 n)$ using data structures of total size $O(n^{2+\epsilon})$.*

### 3.5 Implementing `Largest-empty-pyramid`

Let us consider a family of pyramids in 3-space $\Gamma(s)$ indexed by a positive real parameter $s$, forming an *inclusion chain* (i.e. $s_1 < s_2 \Rightarrow \Gamma(s_1) \subset \Gamma(s_2)$). Again, the property "$\Gamma(s)$ intersects $P$" is monotone in $s$: there is a value $s^*$ such that, for all $s < s^*$, $\Gamma(s)$ does not meet $P$ and, for all $s > s^*$, $\Gamma(s)$ meets $P$. Therefore we can use Megiddo's parametric search to transform an algorithm for detecting empty pyramids into an algorithm for finding $s^*$.

**Lemma 8** *A pyramid $\Gamma$ with a vertex disjoint from $P$ intersects $P$ of and only if an edge of $P$ meets a face of $\Gamma$ or an edge of $\Gamma$ meets a face in $P$ or a vertex of $P$ is contained in $\Gamma$.*

**Theorem 5** *Given a set of c-parallel polyhedra $P$ of complexity $n$, we can implement the calls of* `Largest-empty-pyramid` *in algorithm* `sweep` *in time $O(\log^2 n)$ using data structures of total size $O(n^{2+\epsilon})$.*

*Proof.* From the above discussion is sufficient to set up a test for pyramid emptiness and apply the parametric search transformation. From Lemma 8 we have three cases to check. We check whether an edge of $\Gamma$ meets a face in $P$ using the ray-shooting data structure. We test whether a face of $\Gamma$ (triangle or quadrangle) meets an edge of $P$ using the result of Theorem 3 (though we need a slight modification to deal with the quadrangular face of $\Gamma$ within the stated storage bound). The last case we need to test is an instance of simplex range searching on the set of vertices of $P$. The pyramid has faces belonging to $\binom{c}{2} + 1$ subfamilies of planes. Each subfamily is constrained to contain a fixed point at infinity. We can thus write a formula for testing whether a point is within a pyramid. We just have to check the position of the vertices of $P$ with respect to each plane spanning a facet of the query pyramid.

One such test is solved by projecting both the plane and the points from the point at infinity constraining the plane. Thus the problem reduces to an half-plane range searching problem in 2-space. Since there is a constant number of different classes of pyramids according to the possible combinations of orientations of the facets we can write a formula $F$ to denote the fact that a point is within the query pyramid. Time and storage bounds follow from Lemma 1. ∎

### 3.6 Putting the pieces together

Theorem 1 and the implementation of the primitive operations in Section 3 prove the following theorem:

**Theorem 6** *Given a scene $P$ composed of c-oriented polyhedra with $n$ vertices, edges and faces, we can build a data structure $\mathcal{D}_1(P)$ of size $O(n^{2+\epsilon})$ such that for a query point $v$ the visibility map $M(v, P)$ can be computed in time $O(k \log^2 n)$, where $k$ is the output size.*

## 4 Trade offs and dynamization

### 4.1 Storage-query time trade offs

The primitive operations of algorithm `sweep` are implemented in Section 3 by a reduction to (several) point location queries on arrangements of lines in 2-spaces. As a matter of fact, we are interested only in the sign of some bilinear forms depending on $P$ and $v$. The same result is obtained by dualizing point and surfaces of Lemma 1, in this case the problem is transformed in a series of halfplane range searching on sets of points in 2-spaces.

Such queries can be solved using techniques for half-space range searching [CSW90, Mat92] in $n^{1+\epsilon} \leq m \leq n^{2+\epsilon}$ storage and $T = O(n^{1+\epsilon}/m^{1/2})$ query time. When we apply the parametric search to superlogarithmic algorithms (as it is the case for the query time after the trade-off) we use the more sophisticated form of parametric search that needs a parallel version of the halfplane range searching algorithm [Meg83]. The data structures of Chazelle et al. [CSW90] and Matoušek [Mat92] are based on a partition-tree approach and the query time depends on the number of nodes in the partition tree visited during the query. We can allocate dynamically processors to the nodes visited during the search, thus we need $p = O(n^{1+\epsilon}/m^{1/2})$ processors. The parallel query time $T'$ is given by the depth of the tree which in $O(\log n)$. After the parametric-search transformation we have a total query time $O(T'p + TT' \log n) = O(n^{1+\epsilon}/m^{1/2})$, for a slightly greater value of $\epsilon$. Also, we can run in parallel the one-shot algorithm [dBO92], which has better performances when $k$ is large. The above discussion leads to the following theorem:

**Theorem 7** *Given set $P$ of c-parallel polyhedra with $n$ vertices, edges and faces, we can build a data structure $\mathcal{D}_2(P,m)$ of size $n^{1+\epsilon} \le m \le n^{2+\epsilon}$ such that for a query point $v$ the visibility map $M(v,P)$ can be computed in time $O(k \log n + \min\{n \log n, kn^{1+\epsilon}/m^{1/2}\})$, where $k$ is the output size.*

**Corollary 1** *Given a set $P$ of c-parallel polyhedra with $n$ vertices, edges and faces, we can build a data structure $\mathcal{D}_3(P)$ of size $O(n)$ such that for a query point $v$ the visibility map $M(v,P)$ can be computed in time $O(k \log n + \min\{n \log n, kn^{1/2+\epsilon}\})$, where $k$ is the output size.*

If we allow only linear storage we obtain a query time roughly order of $k\sqrt{n}$ which is asymptotically better than the best one-shot algorithm [dBO92] for $k \le \sqrt{n}$. The result of Corollary 1 is significant for those scenes in which only a small part of the total number of polyhedra is visible at any given time. A graph of the running time $T(n,k)$ as a function of $k$ in logarithmic scale is in Figure 2, for the case of linear storage.

## 4.2 Dynamization

Since the data structures we have described are multilayer data structures based on partition-tree schemes [CSW90, AS91] [Mat92] we can use known dynamization results [AS91]:

**Theorem 8 (Theorem 3.1 in [AS91])** *Given a set of $n$ points in $R^d$ and a parameter $n^{1+\epsilon} \le m \le n^{d+\epsilon}$ one can maintain the CSW-partitioning structure in $O(m/n^{1-\epsilon})$ amortized time as we insert or delete a point, and can answer half-space range queries in time $O(n^{1+\epsilon}/m^{1/d})$.*

Thus the off-line amortized update time for $\mathcal{D}_2(P,m)$ is $O(m/n^{1-\epsilon})$. On-line we can do better under a reasonable assumption. Notice that all of the data structures for answering primitive queries of the `sweep` algorithm *count* the number of features (vertices, edges, faces) intersecting the query objects. Let us suppose that we have a mixed sequence of update operations and queries. Let $P$ be the initial set of $c$-oriented polyhedra, $P_I$ the set of $c$-oriented polyhedra inserted up to the present moment and $P_D$ the set of $c$ oriented polyhedra deleted form $P \cup P_I$ up to the present moment. Let $N_g(S,q)$ be the number of features of type $g$ (vertex, edge, face) of a set of polyhedra $S$ intersecting the query object $q$ (ray, triangle, pyramid). Clearly:

$$N_g((P \cup P_I) \setminus P_D, q) = N_g(P,q) + N_g(P_I,q) - N_g(P_D,q) \tag{2}$$

We build three separate dynamic data structures for $P$, $P_I$ and $P_D$. The three data structures storing the sets $P$, $P_I$ and $P_D$ will use three parameters of storage $m, m_I$ and $m_D$ with $n^{1+\epsilon} \le m \le n^{2+\epsilon}$, $n_I^{1+\epsilon} \le m_I \le n_I^{2+\epsilon}$ and $n_D^{1+\epsilon} \le m_D \le n_D^{2+\epsilon}$. Parameters $m, m_I$ and $m_D$ can be chosen so to tune the performance of the system. The on-line amortized time for an insertion is now only $O(m_I/n_I^{1-\epsilon})$. Since in real applications we expect that over a session of use of the system $n_I << n$, we obtain a very small slowdown of the system due to reconstructions. A similar argument holds for $P_D$. The query time is changed only by a constant factor. Extensive reconstruction of the data structure in order to maintain the overall performance can be done off-line between sessions using the method of Theorem 8.

# 5 Primitive operations for general polyhedra

In this section we discuss the implementation for the primitive operations used in the `sweep` algorithm in the case of general polyhedral scenes.

1. We can answer ray-shooting queries in time $O(\log n)$ using $O(n^{4+\epsilon})$ storage [Pel93, Theorem 8].

2. We can implement `Empty-triangle` in time $O(\log n)$ using $O(n^{4+\epsilon})$ storage [Pel94, Theorem 3].

3. We use a data structure developed by the author [Pel94, Theorem 3] to determine whether a query triangle intersects a set of polyhedral obstacles. We modify the query method by applying Megiddo's parametric search [Meg83], thus obtaining a method to determine the smallest empty triangle in a family of triangles forming a chain of inclusions. The query time becomes $O(\log^2 n)$.

4. We use a method developed by the author [Pel94, Theorem 4] to determine whether a query simplex intersects a set of polyhedral obstacles in time $O(\log n)$ using $O(n^{4+\epsilon})$ storage. This method can be easily adapted to answer emptiness queries with any convex polyhedron with a constant number of facets (in particular, a pyramid with quadrangular base). Similarly to the previous case we modify the query algorithm by applying Megiddo's parametric search [Meg83]. `Largest-empty-pyramid` is thus implemented in time $O(\log^2 n)$ using $O(n^{4+\epsilon})$ storage.

The `sweep` Algorithm together with the above primitives leads to the following result:

**Theorem 9** *Given a set $P$ of polyhedra with $n$ vertices, edges and faces, we can build a data structure $\mathcal{D}_4(P)$ of size $O(n^{4+\epsilon})$ such that for a query point $v$ the visibility map $M(v,P)$ can be computed in time $O(k\log^2 n)$, where $k$ is the output size.*

Using results of Agarwal and Matoušek [AM92a], it is possible to trade off storage and query time.

**Corollary 2** *Given a set $P$ of polyhedra with $n$ vertices, edges and faces, we can build a data structure $\mathcal{D}_5(P,m)$ of size $n^{1+\epsilon} \le m \le n^{4+\epsilon}$ such that for a query point $v$ the visibility map $M(v,P)$ can be computed in time $O(kn^{1+\epsilon}/m^{1/4})$, where $k$ is the output size.*

The method of Corollary 2 compares favourably with the one-shot methods only for quite small values of $k$, unless a large amount of storage is allowed. Thus, the best bound on the time needed to produce a view is given by combining Corollary 2 with the best one-shot method. We obtain a bound $O(\min\{kn^{1+\epsilon}/m^{1/4}, n^{2/3+\epsilon}k^{2/3} + n^{1+\epsilon} + k\})$. In Figure 3 it is shown a graph of the bound in logarithmic scale for $m = n^2$.

# 6 Conclusions

In this paper we have shown several results on repetitive hidden surface removal for polyhedral scenes. We considered both $c$-oriented polyhedra, for which better bounds are attained, and general polyhedral scenes. We reduce the HSR problem to combinations of point-location and half-plane range searching, in conjunction with a general sweeping line algorithmic skeleton. For several visibility problems in 3-space we improve on previously known best bounds.

# 7 Acknowledgments

I wish to thank the anonymous referee whose comments have greatly improved the style of the presentation.

# References

[AM92a] P.K. Agarwal and J. Matoušek. Range searching with semialgebraic sets. In *Proc. of the 17th Symp. on Mathematical Foundations of Computer Science*, number 629 in Lecture Notes in Computer Science, pages 1–13, 1992.

[AM92b] P.K. Agarwal and J. Matoušek. Ray shooting and parametric search. In *Proceedings of the 24th Annual ACM Symposium on Theory of Computing*, pages 517–526, 1992.

[AS91] P. K. Agarwal and M. Sharir. Applications of a new space partitioning technique. In *Proceedings of the 1991 Workshop on Algorithms and Data Structures*, number 519 in Lecture Notes in Computer Science, pages 379–391. Springer Verlag, 1991.

[BDEG90] M. Bern, D. Dobkin, D. Eppstein, and R. Grossman. Visibility with a moving point. In *Proceedings of the 1st ACM-SIAM Symposium on Discrete Algorithms*, pages 107–117, 1990.

[Ber89] M. Bern. Hidden surface removal for rectangles. *J. Comput. Syst. Sci.*, 40:49–59, 1989.

[BO79] J.L. Bentley and T. Ottman. Algorithms for reporting and counting geometric intersections. *IEEE Trans. on Computers*, C-28:643–647, 1979.

[CSW90] B. Chazelle, M. Sharir, and E. Welzl. Quasi-optimal upper bounds for simplex range searching and new zone theorems. In *Proceedings of the 6th ACM Symposium on Computational Geometry*, pages 23–33, 1990.

[dB92a] M. de Berg. Dynamic output-sensitive hidden surface removal for c-oriented polyhedra. *Computational Geometry: Theory and Applications*, 2:119–140, 1992.

[dB92b] M. de Berg. *Efficient Algorithms for ray-shooting and hidden surface removal.* PhD thesis, Utrecht University, Dept. of Comp. Sci., 1992.

[dBHO+91] M. de Berg, D. Halperin, M. Overmars, J. Snoeyink, and M. van Kreveld. Efficient ray-shooting and hidden surface removal. In *Proceedings of the 7th ACM Symposium on Computational Geometry*, pages 21–30, 1991.

[dBHOvK92] M. de Berg, D. Halperin, M. Overmars, and M. van Kreveld. Sparse arrangements and the number of views of polyhedral scenes. Manuscript, June 1992.

[dBO90] M. de Berg and M. Overmars. Hidden surface removal for axis-parallel polyhedra. In *Proceedings of the 31st IEEE Symposium on Foundations of Computer Science*, pages 252–261, 1990.

[dBO92] M. de Berg and M. Overmars. Hidden surface removal for $c$-oriented polyhedra. *Comput. Geom. Theory Appl.*, 1(5):247–268, 1992.

[Ede87] H. Edelsbrunner. *Algorithms in Combinatorial Geometry.* Springer Verlag, 1987.

[EGS86] H. Edelsbrunner, L. Guibas, and J. Stolfi. Optimal point location in a monotone subdivision. *SIAM Journal of Computing*, (15):317–339, 1986.

[FKN80] H. Fuchs, Z. M. Kedem, and B. Naylor. On visible surface generation by a priori tree structures. *Comput. Graph.*, 14(3):124–133, 1980.

[FVFH90] J. D. Foley, A. Van Dam, S. K. Feiner, and J. F. Hughes. *Computer Graphics: Principles and Practice.* Addison-Wesley, Reading, MA, 1990.

[GCS91] Z. Gigus, J. Canny, and R. Seidel. Efficiently computing and representing aspect graphs of polyhedral objects. *IEEE Transactions on pattern Analysis and Machine Intelligence*, 13:542–551, 1991.

[GM90] Z. Gigus and J. Malik. Computing the aspect graphs for line drawings of polyhedral objects. *IEEE Transactions on pattern Analysis and Machine Intelligence*, 12:113–122, 1990.

[Kir83] D. Kirkpatrick. Optimal search in planar subdivision. *SIAM Journal of Computing*, (12):28–35, 1983.

[KvD76] J.J. Koenderink and J. van Doorn. The singularities of visual mapping. *Biological Cybernetics*, 24:51–59, 1976.

[KvD79] J.J. Koenderink and J. van Doorn. The internal representation of solid shape with respect to vision. *Biological Cybernetics*, 32:211–216, 1979.

[Mat91] J. Matoušek. Cutting hyperplane arrangements. *Discrete Comput. Geom.*, 6:385–406, 1991.

[Mat92] J. Matoušek. Efficient partition trees. *Discrete & Computational Geometry*, 8:315–334, 1992.

[Meg83] N. Megiddo. Applying parallel computation algorithms in the design of sequential algorithms. *J. of ACM*, 30:852–865, 1983.

[Meh84] K. Mehlhorn. *Multidimensional Searching and Computational Geometry*, chapter 2. Springer Verlag, 1984.

[Mul91] K. Mulmuley. Hidden surface removal with respect to a moving view point. In *Proceedings of the 23rd Annual ACM Symposium on Theory of Computing*, pages 512–522, 1991.

[OS89] M. Overmars and M. Sharir. Output-sensitive hidden surface removal. In *Proceedings of the 30th IEEE Symposium on Foundations of Computer Science*, pages 598–603, 1989.

[PD86] W.H. Plantiga and C.R. Dyer. An algorithm for constructing the aspect graph. In *Proceedings of the 27th IEEE Symposium on Foundations of Computer Science*, pages 123–131, 1986.

[PD90] W.H. Plantiga and C.R. Dyer. Visibility, occlusion and the aspect graph. *Int. J. of Computer Vision*, 5:137–160, 1990.

[Pel93] M. Pellegrini. Ray shooting on triangles in 3-space. *Algorithmica*, 9:471–494, 1993.

[Pel94] M. Pellegrini. On collision-free placements of simplices and the closest pair of lines in 3-space. *SIAM J. on Computing*, 23(1):133–153, 1994.

[PY90] M. S. Paterson and F. F. Yao. Efficient binary space partitions for hidden-surface removal and solid modeling. *Discrete Comput. Geom.*, 5:485–503, 1990.

[RS88] J. H. Reif and S. Sen. An efficient output-sensitive hidden-surface removal algorithms and its parallelization. In *Proc. 4th Annu. ACM Sympos. Comput. Geom.*, pages 193–200, 1988.

[Sno91] J. Snoeyink. The number of views of axis-parallel objects. *Algorithmic Review*, 2:27–32, 1991.

[SSS74] I.E. Sutherland, R.F. Sproul, and R.A. Schumaker. A characterization of ten hidden-surface algorithms. *Computing Surveys*, 6(1):1–55, 1974.

Figure 1: Example of events i..vi.

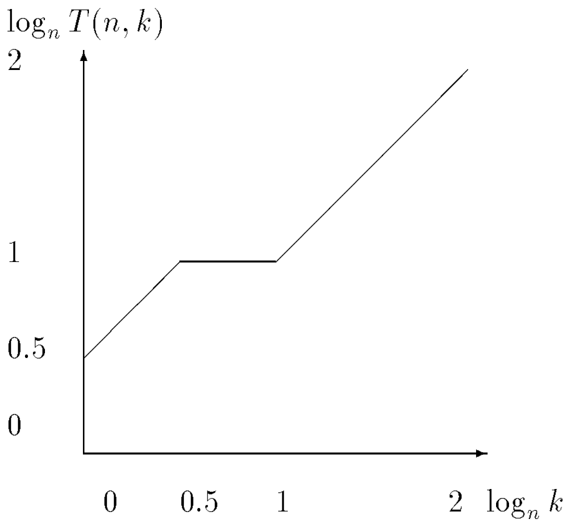


Figure 2: Time in logarithmic scale as function of $k$ for c-oriented polyhedra and linear storage.

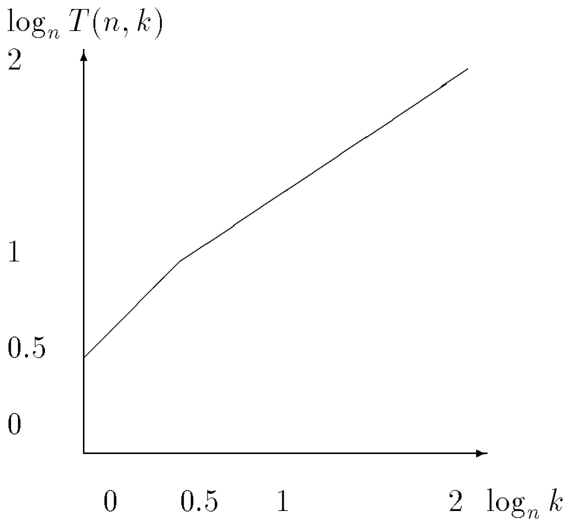


Figure 3: Time in logarithmic scale as function of $k$ for general polyhedra and quadratic storage.