UNIVERSITY OF WATERLOO
COMPUTER SCIENCE DEPARTMENT

Graphics in Flatland: A Case Study

Herbert Edelsbrunner
Mark H. Overmars
Derick Wood

CS-82-25

August, 1982

GRAPHICS IN FLATLAND:

A CASE STUDY[+]

Herbert Edelsbrunner,[(1)] Mark H. Overmars[(2)]
and Derick Wood[(3)]

+ The first author was supported by the Fonds zur Foerderung der wissenschaftlichen Forschung, the second by the Netherlands Organization for the Advancement of Pure Research (ZWO), and the third by the Natural Sciences and Engineering Research Council of Canada, Grant A-7700.

(1) Institutes for Information Processing, Technical University of Graz, Schiesstattgasse 4a, A-8010 Graz, Austria.

(2) Department of Computer Science, University of Utrecht, P.O. Box 80.002, 3508 TA Utrecht, the Netherlands.

(3) Department of Computer Science, University of Waterloo, Waterloo, Ontario, Canada N2L 3G1.

Abstract.

Usually in computer graphics, a two-dimensional view of a set of three-dimensional objects is considered. In the present paper we reduce the dimensionality by one in each case. In other words we study what, for obvious reasons, we call Flatland graphics.

This forms the beginning of a mathematical investigation of computer graphics and, at the same time, provides uniform solutions for a number of computational geometry problems. In particular we study the maintenance of a view during insertion and deletion of objects, and "frame-to-frame" coherence while walking around a set of objects. Both parallel and perspective projections are considered.

Our major concern is convex objects which are simple, in a sense made precise in the paper. However, we close the paper by discussing some possible extensions to non-convex objects and/or to higher dimensions.

The investigation also serves to demonstrate a number of tools that have been developed recently in the context of computational geometry. For example, dynamization and searching.

1. Introduction.

An important and fundamental algorithmic problem in computer graphics is the following: given a set of objects in three-dimensional space, compute the view from some fixed direction or point, that is compute the picture one sees looking from the direction or point. The main issue is to eliminate all parts of the objects that cannot be seen (i.e., lie behind some other object). It is a generalization of the hidden line problem in which objects have straight line edges. The problem has numerous applications in such areas as picture processing, the development of movies, etc. (see e.g. Newman and Sproull [NS]).

Notwithstanding the problem's importance there have been few attempts in the literature to investigate it theoretically (see Sutherland, Sproull and Schumacker [SSS]). It is only recently that such investigations have begun to make their appearance, for example, Beatty, Booth and Matthies [BBM], Fuchs, Kedem and Naylor [FKN], Schmitt [Sc], Wood [W], and Yao [Y].

We simplify the above problem conceptually by only considering graphics in "Flatland". Flatland is a two-dimensional world, described by Edwin A. Abbott in his book "Flatland: a romance of many dimensions" [A] with objects like line segments, convex polygons and circles as inhabitants. We study the generalized hidden line problem in

Flatland for three reasons:

(i) The results and algorithms are interesting in their own right, because of their relation to other planar geometric problems.

(ii) A number of three-dimensional graphics problems are for the main part two-dimensional, for example walking in a maze.

(iii) The results possibly lay a foundation and show directions for further research on the corresponding problems in three and higher-dimensional space (see Section 6).

Some aspects of the hidden line problem in Flatland have already been studied. Shamos [Sh] and Lee and Preparata [LP1] consider the problem of computing the kernel of a simple polygon, that is the region inside the polygon from which the whole polygon is visible. Shamos [Sh] and El-Gindy and Avis [EA] treat the problem of computing the view of a simple polygon from some interior point in $O(n)$ time. Freeman and Loutrel [FL] consider the more general case in which the point of view may also be outside the polygon, but their algorithm appears to run in $O(n^2)$ time. Avis and Toussaint [AT] define the notion of visibility from an edge of a polygon and give and $O(n)$ algorithm for solving the problem. All of these results only deal with the case in which there is a single polygon. In this paper we consider the much more general setting of the "hidden line" problem in which we have a set

of simple objects. We assume that these objects do not intersect, but we allow them to touch. Hence "larger" objects can be decomposed into simple objects (a simple polygon is in general not a simple object).

We consider the two standard types of views in Flatland. The first type is the view from some direction, a parallel view. In this case, the view consists of a line on which the parts of the objects that are visible from the direction are projected. See Figure 1 for an example. (The reader is recommended to look along the paper from the direction of view). For the sake of clarity we number the objects in the set from 1 to n (the total number of objects in the set). The second type of view is the view from a point, a perspective view. In this case the view consists of a circle on which the parts of the objects we can see from the given point are projected. See Figure 2 for an example. Perspective views correspond to our natural way of looking, although we are only able to see part of the perspective projection.

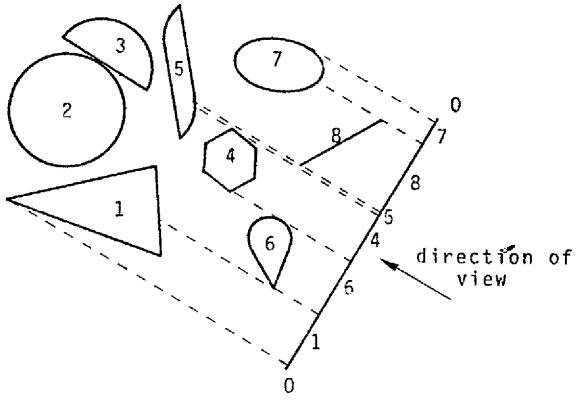
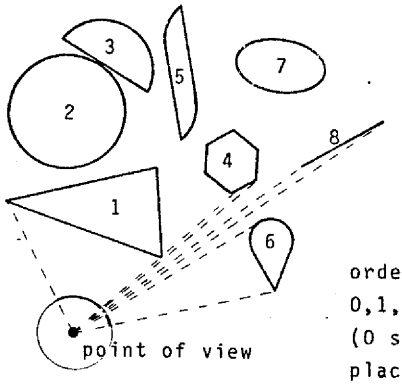


Figure 1



order of view:
0,1,4,0,8,0,6.
(0 stands for an open
place.)

Figure 2

Definition: A simple object is a bounded convex object such that

(i) Any parallel or perspective view of it can be computed in constant time.

(ii) If the view of two simple objects overlap then constant time suffices to decide which one of the two objects can be seen entirely.

(iii) The up to four common tangents of two simple objects can be computed in constant time.

Two objects that touch each other, that is their boundaries overlap but do not cross, are, by definition, non-intersecting. Typical examples of simple objects are: line segments, discs, convex polygons with a bounded number of edges, etc. In the sequel objects are assumed to be simple and sets of objects are assumed to contain only non-intersecting objects. See Figures 1 and 2 which display a set of 8 non-intersecting simple objects, respectively.

We first consider parallel views. In Section 2 we show that parallel views of a set of n objects can be computed in O(nlogn) time, using a divide-and-conquer technique. It is also shown that known dynamization techniques can be applied for maintaining the view from some fixed direction at the cost of O(n) time per insertion or deletion of an object.

An important related problem is the problem of maintaining the view while "walking around" the set of objects.

So, starting with the view from some starting direction, we want to maintain it while "walking" in a counterclockwise direction, say. Of course, with each new direction the relative size of the parts of the objects one sees changes, but the order in which one sees these parts only changes when a "critical" direction is crossed. In Section 3 we give methods for maintaining this order of objects while walking around the set.

In Section 4 we consider the searching variant of the problem of computing a view: given a set of objects, preprocess them in such a way that views from different directions can be computed efficiently. The problem was recently treated by Fuchs, Kedem and Naylor [FKN]. We will follow a completely different approach to obtain trade-offs between query time, preprocessing time and storage required.

In Section 5 we examine perspective views. It is shown that the method for computing and maintaining a view, as described in Section 2, carry over. Restricted versions of the walking around problem and the searching variant are considered as well.

Finally, in Section 6, we discuss how non-convex objects might be included, and also the problems raised by moving to higher dimensions.

2. Computing and Maintaining a Parallel View.

A parallel view of a set of objects consists of a partition of a line. Each part of the line corresponds to an object in the set of to a place where one can look through the set (from the direction of view). See Figure 3 for some different parallel views of the same set of objects.

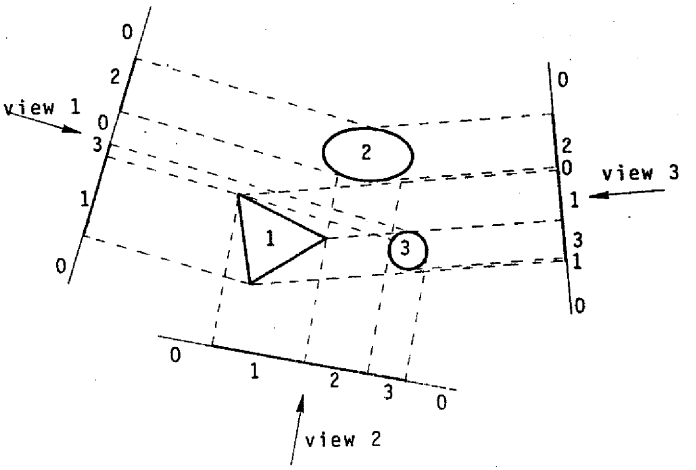


Figure 3

To each part of the line we assign the number of the corresponding point. If the part corresponds to a place where one can look through the set we assign 0 to it. It is possible that different parts of the line correspond to

the same object, and hence, are assigned the same number (see e.g. view 3 of Figure 3). A partition-point corresponds to (i) a leftmost point of an object or (ii) a rightmost point of an object, with respect to the direction of view. See Figure 4 for examples of the two different cases. (A part of the line might consist of one point if it corresponds to a line segment in the direction of view. In this case we treat it as a double partition-point).

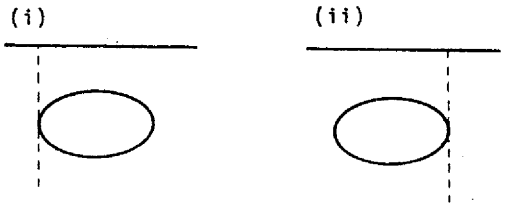


Figure 4

Lemma 2.1: The parallel view of a set of n objects from a fixed direction consists of at most 2n+1 parts, that is has at most 2n partition-points.

Proof: We show that one can assign left and right endpoints of objects to partition-points in a unique way. In case (i) we assign the corresponding left endpoint to the partition-point, in case (ii) we assign the corresponding

right endpoint. As the number of left and right endpoints is bounded by 2n, there are at most 2n partition-points and hence at most 2n+1 parts. This completes the argument.

For computing the view of a set of objects from a fixed direction we will use a divide-and-conquer technique.

Lemma 2.2: Given a fixed direction and a set of objects $V = \{x_1, \ldots x_n\}$. The view of V can be computed in O(n) time from the views of $A = \{x_1, \ldots x_i\}$ and $B = \{x_{i+1}, \ldots x_n\}$, for some $1 \leq i < n$.

Proof: Let the parts of the view of A from left to right be numbered $a_0, a_1, \ldots a_k$ and let the partition-points in between be $p_1, \ldots p_k$. Similarly, let the parts of the view of B be numbered $a'_0, a'_1, \ldots a'_l$, and let the partition-points in between be $p'_1, \ldots p'_l$. One easily verifies that partition-points of $V = A \cup B$ are partion-points of A or B and that if at some location in the view of $A \cup B$ object o is visible, then it is visible at that location in the view of A or in the view of B. See Figure 5 for an example.

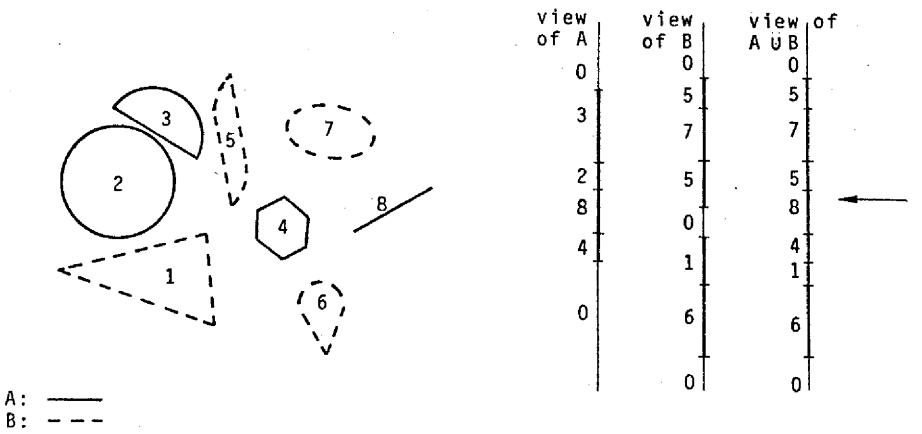


Figure 5

We will compute the view during one simultaneous walk along the lists for the views of A and B. At each partition-point of both views we check whether or not this point is also a partition-point of the total view. Due to the definition of a simple object (condition (ii)) this checking can be done in constant time which immediately implies the assertion. This completes the argument.

Lemma 2.2 provides the conquer step we need.

Theorem 2.3: Given a set of n objects and a direction, one can compute the parallel view from the given direction in O(nlogn) time and this is optimal in the worst case.

Proof: To compute the view of the set of objects V, we split V into two equal sized subsets, compute the views of the two subsets recursively in the same way, and merge the answers using Lemma 2.2. Besides the recursive calls, this method takes O(n) time. Hence, the total time T(n) needed is given by the following recurrence:

$$T(n) = 2T(n/2) + O(n).$$

It is well-known that the solution of this recurrence is $T(n) = O(n\log n)$.

Since computing the view can also be used to sort n real numbers, the obtained bound is also optimal in the worst case. This completes the argument.

This solves the problem of computing the view of a set of objects from a given direction.

A second problem is the problem of maintaining the view from a fixed direction while objects are inserted into and deleted from the set. We assume that the set will be updated in such a way that objects never intersect. Overmars [O1] considered the following class of problems which are defined for finite sets of objects and hence are called set problems.

Definition: ([O1]) A set problem P is called O(n)-order decomposable if there exists an ordering ORD and a function $\square$ such that for each set $\{a_1,\ldots a_n\}$, ordered according to ORD

$$P(a_1,\ldots a_n) = \square(P(a_1,\ldots a_i),P(a_{i+1},\ldots a_n),$$

for all i with $1 \leq i < n$, where $\square$ takes O(n) time to compute.

Using for ORD any ordering we like and using for $\square$ the merging step described in the proof of Lemma 2.2, it follows from Lemma 2.2 that the problem of computing the view is O(n)-order decomposable. Using the dynamization method known for O(n)-order decomposable set problems ([O1]) we obtain the following result:

Theorem 2.4: Given a fixed direction, one can maintain the view of a set of objects from that direction at the cost of O(n) time per insertion and deletion of an object, where n is the cardinality of the set.

The disadvantage of O(nlogn) space required by this method can be remedied using a technique due to Gowda and Kirkpatrick [GK]. It reduces the space to O(nloglogn) whithout changing the update time bounds.

When we are not interested in the view after each update but only want to maintain the set of objects in such a way that the view can be computed efficiently when needed, we can apply another dynamization technique. This technique was originally described for so-called searching problems, that is a set V of objects is to be stored such that questions dependent on V and on some query object needed for the formulation of the

question can be answered efficiently. For those problems, Overmars [O2] defines the following subclass:

Definition: ([O2]) A searching problem P is called O(n)-decomposable if there exists a function $\square$ such that for all sets A and B with $A \cap B = \emptyset$ and for each query object x

$$P(x, A \cup B) = \square(P(x,A), P(x,B)),$$

where $\square$ takes at most O(n) time to compute when the sets contain n objects.

O(n)-decomposability is an extension of the concept of decomposability for searching problems as defined by Bentley [Be]. As a consequence of Lemma 2.2 the problem of computing the view from a fixed direction is O(n)-decomposable. (The problem at hands can be considered to be a searching problem for which the questions do not depend on some query object). Applying known transformations on structures for O(n)-decomposable searching problems ([O2, Be]) to the structure used in Theorem 2.4 we obtain the following result:

Theorem 2.5: Given some fixed direction, one can maintain a set of objects at the cost of $O(\log^2 n)$ time per insertion and O(n) time per deletion such that at any moment one can compute the view from the given direction in O(n) time.

Using the techniques from [GK] the amount of storage required can be bounded by O(nloglogn).

3. Computing All Parallel Views ("Walking Around the Set").

In computer graphics it is often required to maintain the view of a set of objects while "walking around" the set. So, we start with the view from some starting direction and we want to maintain the view while changing the direction counterclockwise, say, until we again reach the starting direction. Of course, with each new direction, the relative size of the parts of the objects we can see is different and hence it is unlikely that there are truly efficient ways of maintaining the actual view while walking around the set. But the order in which we see parts of the objects only changes at some critical directions. Recently [HZ] have considered frame-to-frame coherence for convex objects in three-dimensional space.

Definition: Scanning the view of a set of objects from some fixed direction from left to right we get a list of at most 2n+1 numbers ranging from 0 to n that gives the order in which we see parts of the objects. This list is called the visibility list of the set from the direction.

Clearly, given a view one can construct the corresponding visibility list in O(k) time, where k is the length of the list. We will show that one can maintain the visibility list of the set efficiently, while walking around. It is important that we are able to compute the view corresponding to a visibility list and an appropriate direction efficiently.

Lemma 3.1: Given a visibility list for some direction, one can compute the view from this direction in O(k) time, where k is the length of the list.

Proof: Let the visibility list be $a_1,\ldots a_k$. Hence, we have to locate the partition-point between each pair a_i, a_{i+1} of parts of the view-line. If a_i=0 then the partition point is the projection of the leftmost point of a_{i+1} on the line. Similarly, if a_{i+1}=0 then the partition-point is the projection of the rightmost point of a_i on the line. (They cannot both be 0). Otherwise we compute the view of a_i and a_{i+1}. This takes constant time by definition. In this view there is exactly one point where a_{i+1} starts replacing a_i. This point is the partition-point we searched for. Hence, each partition-point takes constant time to compute and hence, the total time needed is bounded by O(k). This completes the argument.

Hence, at each moment while walking around the set one can obtain the view in O(k) time. The visibility list of a set of objects only changes when we pass some critical direction.

Definition: Given two objects A and B, a critical direction of A and B is a direction in which the visibility list of A and B changes.

See Figure 6 for an example of the critical directions two simple objects A and B can have. Those directions are given by the up to four common tangents of the two objects. In between the critical directions the visibility lists are given. By condition (iii) of the definition of a simple object, the critical directions for two such objects can be computed in constant time.

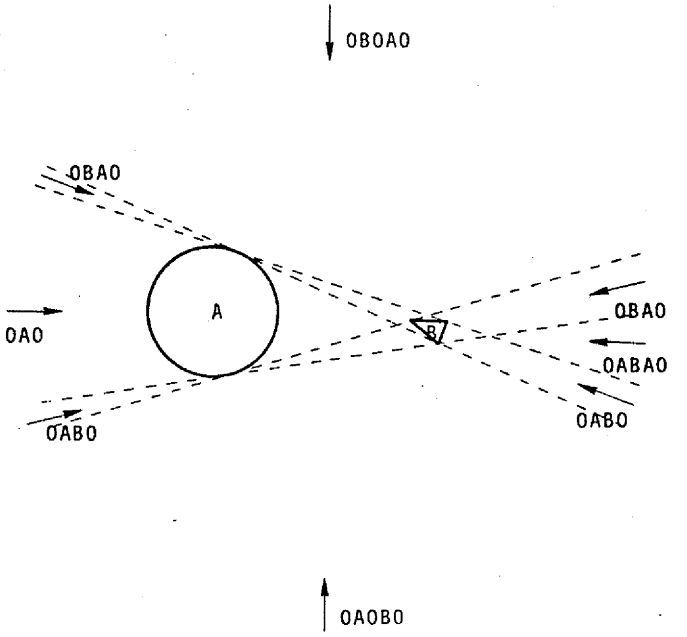


Figure 6

Lemma 3.2: Two simple objects have at most 8 critical directions.

The proof is obvious since the at most four common tangents of two convex objects define at most 8 directions.

Lemma 3.3: Given a set of n objects, there are at most $4n(n-1)$ critical directions.

Proof: There are $n(n-1)/2$ pairs of objects. Each pair of objects gives rise to at most 8 critical directions which completes the argument.

See Figure 7 for an example of all critical directions of a set of three objects.

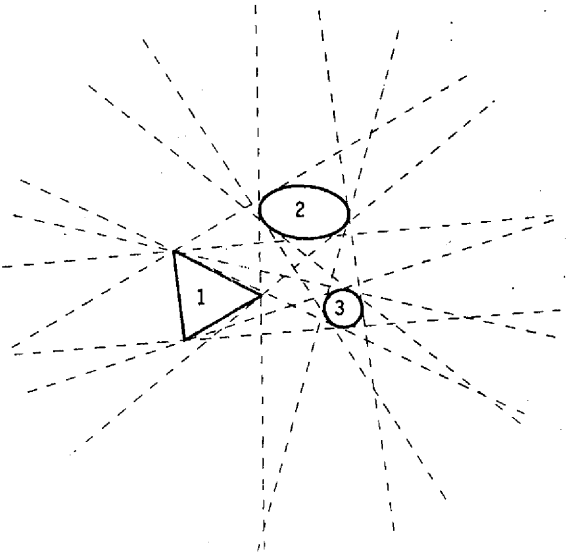


Figure 7

The following lemma shows that the visibility list of the total set only changes when passing a critical direction of some pair of objects in the set.

Lemma 3.4: Given two directions of view α and β. If there is no critical direction in between α and β, then the visibility lists from both directions are the same.

Proof: If the visibility list from α is not equal to the visibility list from β, then there are two objects A and B such that the visibility list from α restricted to A and B is not equal to the visibility list from β restricted to A and B. Hence, there must be a critical direction of A and B in between α and β. This contradicts the assumption and completes the argument.

To be able to maintain the visibility list while walking around the set of objects, we have to represent the list in such a way that the "update" necessary when passing critical directions can be made efficiently.

Lemma 3.5: One can represent a visibility list in such a way that the updates needed when passing a critical direction can be made in O(logk) time, where k is the size of the list.

Proof: We store the k elements of the visibility list in a dictionary D which allows us to locate in O(logk) time the element C of the visibility list with the following properties: Given a direction for which the visibility list is correct and a point p in the view from this direction then p lies in an interval of the view which corresponds to C. In addition to this, the dictionary D accommodates in O(logk) time changes which are necessary when a critical direction is passed. (D can be implemented as an AVL-tree storing the elements of the visibility list in its inner nodes. The nodes are also maintained in a doubly linked list which allows us to determine in constant time the predecessor or successor in the symmetric order.)

For convenience, we will interchangably use D to denote the dictionary which stores a visibility list as well as the visibility list itself. We assume that with each critical direction we are given the two objects A and B of which it is a critical direction. Examining the view of A and B before and after the critical direction we find out how their view changes and where the new or disappearing partition-point p lies in the view from the critical direction. We will locate the object C in the visibility list stored in the dictionary D such that p lies in an interval corresponding to C. Examining the view of A, B and C (C is possibly A or B) we can find out in constant time what change in the visibility list needs to be made (if necessary). This change consists of

the insertion or deletion of one object. This clearly takes at most O(logk) time which completes the argument.

(If two of more critical directions are the same we treat them separately.) The structure clearly requires O(k) storage.

Lemma 3.6: All critical directions of a set of n objects, in order of occurrence, can be computed in $O(n^2 \log n)$ time.

Proof: We start with object 1 and compute all its critical directions with the other objects. This can be done in O(n) time. Next, we compute all critical directions between object 2 and the objects 3, ... n, between object 3 and objects 4, ... n, etc. This takes a total of $O(n^2)$ time. Ordering the $O(n^2)$ critical directions takes $O(n^2 \log n)$ time which completes the argument.

Lemma 3.5 and 3.6 lead to the following solution for the problem of walking around a set of objects:

Theorem 3.7: Given a set of n objects, one can maintain the visibility list while walking around the objects in a total of $O(n^2 \log n)$ time using $O(n^2)$ space.

Proof: We first compute the view from the starting direction and represent the corresponding visibility list in a dictionary D as described in the proof of Lemma 3.5. This takes a total of O(nlogn) time. Next we compute all critical directions ordered counterclockwise from the starting direction. This takes $O(n^2 \log n)$ time according to Lemma 3.6. Passing each next critical line takes O(logk) = O(logn) time for updating the visibility list, according to Lemma 3.5. As there are $O(n^2)$ critical lines to pass when walking around, this takes a total of $O(n^2 \log n)$ time. The asserted time bound follows. Because we have to store all $O(n^2)$ critical directions, we need $O(n^2)$ storage. This completes the argument.

Although the method presented is fast, the required amount of storage is rather large. We will now describe a method that takes more time but uses only linear storage. (The same amount of storage has also been achieved for a special case of our problem by van Leeuwen and Mehlhorn [vLM]. They considered the walking around problem for a single polygon and obtained an O(nlogn) time and O(n) space algorithm where n denotes the number of vertices of the polygon.)

Again we start with computing the view from the starting direction and representing the corresponding visibility list in a dictionary D. Next, for each object i we compute the critical direction d_i of i with some other object such that (i) i lies behind the other object where the change in view from d_i occurs (this is needed to prevent a direction from being

reported more than once), and (ii) d_i lies nearest to the starting direction. This takes $O(n^2)$ time. See Figure 8 for an example. We know that the first critical direction we have to pass must be among the n directions $d_1,\ldots d_n$. We store these directions in order of occurrence in a priority queue Q, such that the first critical direction is the first element of Q. Let this direction be d_i.

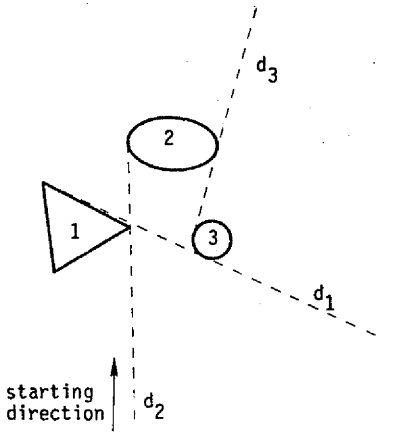


Figure 8

To pass this direction we (i) delete d_i from Q, (ii) perform the appropriate update in the visibility list D, (iii) compute the new d_i, and (iv) insert this new d_i into Q. (i), (ii) and (iv) take O(logn) time, see e.g. [AHU] and (iii) takes

$O(n)$ time. Hence, passing a critical direction takes $O(n)$ time. One easily verifies that in this way the next critical direction to be passed is always the first element in Q.

Theorem 3.8: Given a set of n objects, one can maintain the visibility list while walking around the set in a total of $O(n^3)$ time using $O(n)$ storage.

Proof: Both Q and D have size $O(n)$ and hence, the amount of storage required is $O(n)$. The preprocessing takes $O(n^2)$ time. The crossing of a critical direction takes $O(n)$ and, as there are $O(n^2)$ critical directions, the total amount of time needed is $O(n^3)$. This completes the argument.

The method described above is only one example of a whole class of methods for walking around the set of objects, yielding different trade-offs between the runtime and the amount of storage required.

Theorem 3.9: Given a set of n objects, for each function $f(n)$, with $1 \leq f(n) \leq n/\log n$, there exists a method for maintaining the visibility list while walking around the set in a total of $O(n^3/f(n))$ time using $O(nf(n))$ storage.

Proof: After computing the visibility list D for the starting direction we compute for each object i the first f(n) critical directions $d_i^1,\ldots d_i^{f(n)}$ between i and other objects after the starting direction, where i lies behind the other object at the place of the change. This takes O(n) time per object and hence a total of $O(n^2)$ time. Storing the nf(n) critical directions in a priority queue Q takes O(nf(n)logn) time. We know that the first direction in Q is the first critical direction we have to pass. Passing this direction consists of deleting it from Q and updating the visibility list D. When we pass a $d_i^{f(n)}$ then we also compute the new $d_i^1,\ldots d_i^{f(n)}$ and insert them into Q. In this way Q clearly always contains the first critical direction. For each object i we pass at most O(n/f(n)) times a $d_i^{f(n)}$. Hence, the total amount of time needed for passing all critical directions is bounded by $O(n^2\log n)$ time for updating D and deleting and inserting directions in Q and $O(n^3/f(n))$ time for computing critical directions. As $1 \leq f(n) \leq n/\log n$, we can estimate these costs by $O(n^3/f(n))$. The amount of storage required for D and Q is clearly bounded by O(nf(n)). This completes the argument.

For example, one can obtain a total runtime of $O(n^2\log n)$ using only $O(n^2/\log n)$ storage (compare Theorem 3.7).

There are a number of problems that can be solved using the "walking around" algorithm described in Theorem 3.9. For example the shadow problem due to Preparata [P].

Corollary 3.10: Given a set of n objects, for each function $f(n)$ with $1 \leq f(n) \leq n/\log n$, there exists a method to compute all directions in which the shadows of the objects do not overlap, i.e., in which all objects can be seen separately, in $O(n^3/f(n))$ time using $O(nf(n))$ storage.

Proof: We use the method of Theorem 3.9 for walking around the set. With the dictionary D which contains the visibility list we maintain the information whether or not the objects can be seen separately. By a proper choice of D, this information can be maintained at a cost of $O(\log n)$ time per crossing a critical direction. The assertion follows.

A second example is the stabbing problem, see [EMPRWW]. In [E], the general topic of transversals is discussed.

Corollary 3.11: Given a set of n objects, for each function $f(n)$ with $1 \leq f(n) \leq n/\log n$, there exists a method for computing a line that intersects all objects (when it exists) in $O(n^3/f(n))$ time using $O(nf(n))$ storage.

Proof: One easily verifies that if there is such a line with direction d and $d_1 \leq d \leq d_2$ are the two nearest critical directions then there is such a line with any direction between d_1 and d_2. There is such a line with direction d

if and only if the views of all object from d overlap at some point. Whether there is such a point can be maintained in D at the cost of O(logn) time per crossing a critical direction. Details are left to the interested reader.

Other problems which can be solved by these methods are, for example (i) give all directions in which the view contains holes (places where you can look through the set), (ii) give all directions in which the view is connected, (iii) give all directions in which you can see only one object, etc.

4. Searching for the Parallel View.

In this section we treat the searching variant of the problem of computing a view: Preprocess the set of objects in such a way that views from different directions can be computed efficiently. Hence, the directions are query objects of the searching problem.

Theorem 4.1: Given a set of n objects, for each function $f(n)$, with $1 \leq f(n) \leq n$, there exists a method of preprocessing the set in time $O(n^2 \log n + n^3/f(n))$ using $O(n^3/f(n))$ storage, such that the views from different directions can be computed in $O(f(n) \log n + k)$ time, where k is the number of objects in the view.

Proof: We will show how to preprocess the set such that one can obtain the visibility list from any direction efficiently. Computing the view afterwards takes $O(k)$ time according to Lemma 3.1.

In the preprocessing phase we compute all critical directions between each pair of objects in the set, order them and store them in a dictionary D. We also assume that constant time suffices to determine the predecessor and the successor of a given critical direction in D. Let d_0 be the first direction in D. We compute the view from d_0 and store the corresponding visibility list in a structure D_0 which we connect with d_0. Our goal is to build a visibility list after each $f(n)$ critical directions. To build the second visibility

list D_1 connected to $d_{f(n)}$, we first copy the list D_0 and perform the appropriate updates on it while we walk from d_0 to $d_{f(n)}$. In general, to build the visibility list D_i that has to be associated with $d_{if(n)}$ we first copy D_{i-1} and then update the copy while walking from $d_{(i-1)f(n)}$ to $d_{if(n)}$. The building of D_i in this way takes $O(n)$ time for copying plus $O(f(n)\log n)$ time for performing the updates. Hence, the building of all $O(n^2/f(n))$ visibility lists takes $O(n^3/f(n)+n^2\log n)$ time. As the computation of the critical directions and the building of D take $O(n^2\log n)$ time, the bound on the preprocessing time follows. As D takes $O(n^2)$ storage and each of the $O(n^2/f(n))$ visibility lists takes $O(n)$ storage, the total amount of storage needed is $O(n^2+n^3/f(n)) = O(n^3/f(n))$ because $f(n) \leq n$.

To perform a query with direction d to compute the visibility list of the set from d we search with d in D for the direction d_i nearest to and before d. Next we walk along the list of critical directions until we find a direction with and associated visibility list D_j. On D_j we perform the necessary updates while walking back to d_i. D_j now contains the visibility list from d. We copy it and afterwards we make all updates to restore it to its proper form at $d_{jf(n)}$. The searching with d in D takes $O(\log n)$ time. D_j can be at most $f(n)$ directions away from d_i and hence the locating and updating of D_j takes $O(f(n)\log n)$ time. Copying D_j takes $O(k)$ time. Undoing the actions we performed on D_j also takes $O(f(n)\log n)$ time. The bound for the query time follows which completes the argument.

Since $k=O(n)$, we consider $f(n)=n/\log n$ as a reasonable choice. It yields a data structure which takes $O(n^2\log n)$ time to construct and uses $O(n^2\log n)$ storage such that the views from different directions can be computed in $O(n)$ time. If we only want to ask some questions about the view or visibility list, there is in general neither any need to compute the view itself nor to copy the structure D_j. We can answer the question using the updated D_j after which we undo the actions. In this case the $O(k)$ in the query time need not to be paid.

According to Lemma 2.2 the problem of searching for the view is an $O(n)$-decomposable searching problem. Using known results for $O(n)$-decomposable searching problems we get the following result.

Theorem 4.2: For a set of n objects in the plane there exists a data structure which requires $O(n^2\log n)$ storage, $O(n\log n)$ time per insertion, and $O(n)$ time for computing the view from a specified direction.

5. Perspective Views.

The perspective view of a set of objects, that is the view from some fixed point, consists of a partition of a circle. Each circle segment corresponds to a part of an object that one can see from the fixed point or to a place where one can look through the set. See Figure 9 for some examples of perspective views of the same set of objects from different viewpoints. One easily verifies that a perspective view

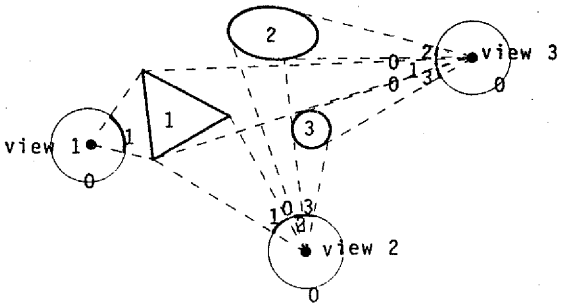


Figure 9

contains at most 2n partition points and hence at most 2n parts. In what now follows, we list a few results for computing perspective views. Proofs and details are omitted since the methods are very similar to those described in Sections 3 and 4 for parallel views.

Lemma 5.1: Given a fixed point and a set of objects $V = \{x_1,...x_n\}$. The view of V from that point can be computed in O(n) time from the views of $A = \{x_1,...x_i\}$ and $B = \{x_{i+1},...x_n\}$, for all $1 \leq i < n$.

Theorem 5.2: Given a set of n objects and a fixed point, one can compute the view from that point in O(nlogn) time.

Theorem 5.3: Given a fixed point, one can maintain the view of a set of objects from that point at a cost of O(n) time per insertion and deletion of an object, where n is the cardinality of the set.

Theorem 5.4: Given a fixed point, one can maintain a set of objects at a cost of $O(\log^2 n)$ time per insertion and O(n) time per deletion such that an any moment one can compute the view from the given point in O(n) time.

To adopt the ideas of walking around the set of objects to the case of perspective views, we need an analog of the notion of visibility list.

Definition: Given a perspective view, the corresponding visibility cycle is the cyclic list of the numbers of the objects one can see (0 for a place where one can look through the set) in order of occurrence.

Clearly, the visibility cycle of a circular view can be computed in O(k) time from the view, where k is the length of the list, and given the visibility cycle, the corresponding view from a point for which the cycle is correct can also be computed in O(k) time.

We generalize the notion of walking around in the following way: Given some connected curve, called tour, through the plane (see Figure 10), we want to maintain the visibility cycle while the point of view walks along the tour.

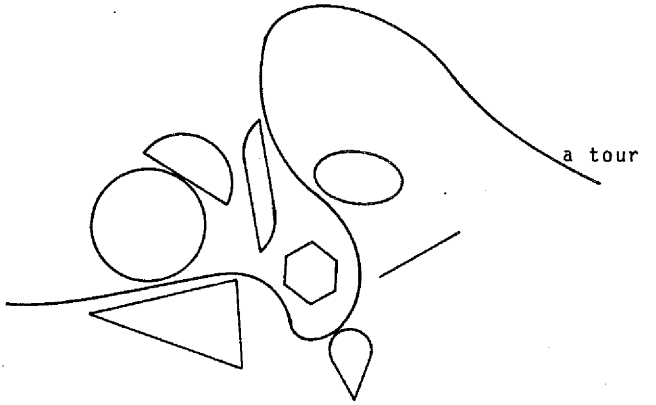


Figure 10

We assume that a tour is simple in the sense that it crosses any line only a constant number of times and that the intersections with a given line can be computed in constant time. A tour is also assumed to intersect no object in the

set. For example, a tour might be a cirle, a line, or the boundary of some object in the set.

Definition: Given two objects A and B, a critical line of A and B is a line, the crossing of which somewhere causes a change in the visibility cycle.

The critical lines of two simple objects are exactly their common tangents. There are at most four critical lines for any two simple objects and these lines can be computed in constant time. One easily verfies that, while walking along a tour, the visibility cycle does not change as long as we do not cross a critical line of some pair of objects.

Theorem 5.5: Given a set of n objects and a tour. For each function f(n), with $1 \leq f(n) \leq n/\log n$, there exists a method of maintaining the visibility cycle while walking with the viewpoint along the tour in a total of $O(n^2+k_{cr}n/f(n))$ time, using $O(nf(n))$ storage, where k_{cr} is the total number of crossings between the tour and the critical lines of objects in the set.

Proof: The method is quite similar to the method described in the proof of Theorem 3.9. We first compute the visibility cycle from the starting point. One can represent this cycle in a dictionary D such that the updates needed when crossing a critical line can be performed in $O(\log n)$ time. Next, for

each object i we compute the first f(n) points of intersection $p_i^1,\ldots p_i^{f(n)}$ between the tour and critical lines of i and other objects where i lies behind the other object at the place the view changes. Those nf(n) intersection points are stored in a priority queue Q. The topmost point in Q is the first point on the tour where we cross a critical line. Crossing such a line consists of deleting the point from Q and performing the necessary updates on the visibility cycle in D. This takes O(logn) time. When we cross a $p_i^{f(n)}$ (for some i) we have to compute the next $p_i^1,\ldots p_i^{f(n)}$ and insert them into Q. This takes a total of O(n+f(n)logn) time. Clearly, Q always contains the first intersection with a critical line. While walking along the tour we cross k_{cr} critical lines which takes $O(k_{cr}\log n)$ time. Moreover we have to compute $O(k_{cr}/f(n))$ times new intersection points (when passing a $p_i^{f(n)}$) which takes a total of $O(k_{cr}n/f(n)+k_{cr}\log n)$ time. Hence, the total amount of time needed is bounded by $O(k_{cr}n/f(n)+n^2)$ because $1 \leq f(n) \leq n/\log n$ which completes the argument.

Preprocessing the set of objects in such a way that views from different points can be computed efficiently gives rise to tremendous problems. As stated above, the visibility cycle might change when the point of view crosses a critical line. There are $O(n^2)$ critical lines and hence there are $O(n^4)$ regions with possibly different visibility cycles. Preprocessing the set in a similar way to that for

parallel views as described in Section 4 would hence take $\Omega(n^4)$ storage and $\Omega(n^4 \log n)$ preprocessing time in the worst case.

We will consider a more restricted version of the searching problem. Given a set of objects and a tour, preprocess the set in such a way that the view from any point on the tour can be computed efficiently. The method for solving this problem is very similar to the method described in the proof of Theorem 4.1. We first compute all points of intersection of the tour with critical lines. We order those points and store them in a dictionary D. Next, we compute the visibility cycles from the starting point and after each f(n) intersection points in the same way as described in the proof of Theorem 4.1. To search for the view from some point on the tour we search for the point in D, walk along the list of intersection points to the nearest visibility cycle, walk back while updating the visibility cycle, compute the view from the point using the updated visibility cycle and restore afterwards the visibility cycle in its original form. This leads to:

Theorem 5.6: Given a set of n objects and a tour, for each function f(n), with $1 \leq f(n) \leq n$, there exists a method of preprocessing the set of objects in time $O(n^2+k_{cr}\log n+k_{cr}n/f(n))$ using $O(k_{cr}n/f(n)+n)$ space such that the view from any point on the tour can be computed in $O(f(n)\log n+k)$ time, where k_{cr} is the number of crossings between critical lines and the tour and k is the length of the view.

Using the fact that the searching problem is $O(n)$ decomposable we get the following dynamic solution (choosing $f(n)=n/\log n$):

Theorem 5.7: For a set of n objects and a tour, there exists a data structure which requires $O(k_{cr}\log n+n)$ space, $O(n+k_{cr}\log n/n)$ time per insertion, and $O(n)$ time for computing the view from a specified point on the tour.

6. Concluding Remarks.

In this paper we have provided a general setting for solving a number of computational geometry problems in the plane. In particular we have considered the frame-to-frame coherence problem for one-dimensional views of two-dimensional scenes, viz in Flatland graphics. In this setting the objects are assumed to be convex, bounded, non-intersecting and stationary. Either the whole scene is rotated in the plane while the viewpoint is fixed, or a walk-around or prespecified tour of the scene is taken, when the scene is fixed. For these equivalent scenarios we are able to demonstrate a time-space trade-off for computing and updating the visibility status of the objects. These results hold for the parallel and perspective projections.

Our results,while setting a number of questions, raise even more. These can be split, somewhat arbitrarily, into "pure" computational geometry questions and computer graphics questions. We will discuss them in this order.

In Corollary 3.10 we have provided an $O(n^2 \log n)$ time algorithm for the shadow problem. This problem arises in [LP2] who consider the shortest path between two points in the presence of line segment obstacles. If there is a "shadow direction" then they have an $O(n \log n)$ time algorithm, whereas when no such direction is known they have an $O(n^2 \log n)$ time algorithm. Thus, solving the shadow problem in substantually less than $O(n^2 \log n)$ time ensures a better algorithm for the

shortest path problem when a shadow direction exists. Can the shadow problem be solved in O(nlogn) or even $O(n^2)$ time? Is the shadow problem simpler than the obstacle proble, that is computing the shortest path?

There are also questions of optimality with respect to Theorem 3.7. Is n^2logn also a lower bound for the time needed to compute a walk-around? It seems that this should be reducible to the sorting of n^2 elements.

Turning to computer graphically-oriented questions, perhaps the first obvious question is: can general two-dimensional objects be dealt with, rather than only convex ones? If we relax the convexity condition, then Lemma 2.1 no longer holds as stated. For example the objects given in Figure 11 exhibit the difficulties which now arise.

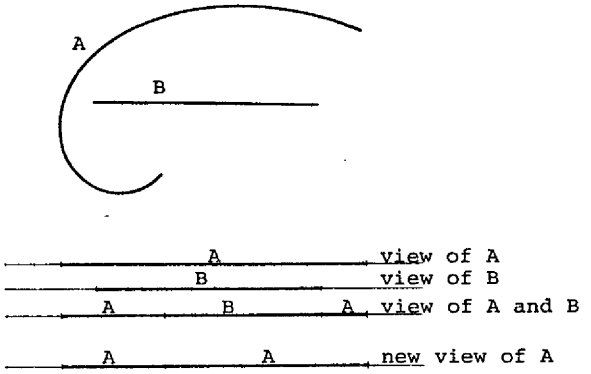


Figure 11

Intuitivly, the new difficulties stem from the fact that A may hide part of B while B again may hide part of A. However, these difficulties can be overcome as follows: The view of an object O is no longer a single interval but a sequence of intervals whose endpoints correspond to extreme points of O with respect to the given direction, see the new view of A in Figure 11.

Obviously, we have to reformulate the definition of a simple object such that the intervals which are the view of a simple object can be computed in constant time. By the same method also the perspective view can now be handled efficiently.

Similarly, the walking around algorithms can be adopted to non-convex objects. Note that now two simple objects may give rise to more than four critical lines, resp. directions.

Second, we have shown that addition and removal of objects can be dealt with using dynamization methods. However, can moving objects be integrated into our framework? This appears to be difficult to achieve, since even moving points in the plane are non-trivial, see [OW].

Third, since we have claimed that this forms the beginning of a mathematical theory of computer graphics, how can these methods and techniques be extended into three and more dimensions? Recently, [HZ] have considered the frame-to-frame coherence problem for three-dimensional scenes consisting of stationary, bounded, convex, non-intersecting polyhedra. They make use of critical planes

as part of their characterization of how the view changes during rotation. From their five characterizing criteria, they derive an algorithm to generate successive frames of an "animation" sequence. Although they present no complexity analysis at all, they demonstrate that the techniques we have used should be extendable to three and more dimensions, albeit with greatly increasing time and space requirements. One would expect this problem to be simpler for wire frames, rather than for solid polyhedra, but we have to results at this time.

Of course, the hidden line problem of Flatland becomes the hidden surface problem for three-dimensional scenes. In the spirit of this paper we are primarily interested in knowing which polyhedra are visible to some extend, rather than the precise specification of visible surfaces. There are two possible approaches, at least. On the one hand, since polyhedra have polygonal faces, this reduces to knowing which faces are visible to some extend. From this point of view the three-dimensional scene becomes a collection of stationary, bounded, convex, and non-intersecting two-dimensional objects arbitrarily oriented in three-dimensional space. On the other hand, if the three-dimensional objects are arbitrary, as our two-dimensional ones are, then such a decomposition cannot, in general, be achieved. Following the development in the present paper we need to consider "simple" objects once more.

Which of these two approaches is the more reasonable remains to be seen, but it should be clear that in both cases many interesting questions are raised.

Let us finally state how we see the contents of this paper related to the research that is done on computer graphics questions. The present research there is mainly experimental and its concerns are:

how to deal adequately with arbitrary shapes;

how to tint, texture, color and shade such objects;

how to represent scenes composed of such objects;

how to animate, rotate, and update such scenes.

The results in this present paper can only be considered as the first step towards answering such questions. Whether or not an adequate mathematical theory of present day computer graphics can be constructed remains to be seen.

References.

[A] Abbott,E.A. Flatland. A Romance of Many Dimensions. 2nd ed., Dover Publications, 1884.

[AHU] Aho,A.V., Hopcroft,J.E. and Ullman,J.D. The Design and Analysis of Computer Algorithms. Addison-Wesley Publishing Company, 1974.

[AT] Avis,D. and Toussaint,G.T. An optimal algorithm for determining the visibility of a polygon from an edge. IEEE Tr. on Comp. C-30 (1981), 910-1014.

[BBM] Beatty,J.C., Booth,K.S. and Matthies,L.H. Complexity of scan line algorithms. Unpublished manuscript, 1981.

[Be] Bentley,J.L. Decomposable searching problems. Inf. Proc. Lett. 8 (1979), 244-251.

[E] Edelsbrunner,H. Finding Transversals for Sets of Simple Geometric Figures. In preparation, 1982.

[EMPRWW] Edelsbrunner,H., Maurer,H.A., Preparata,F.P., Rosenberg,A.L., Welzl,E. and Wood,D. Stabbing Line Segments. Report F84, Institute fuer Informationsverarb., Techn. Univ. of Graz, Austria, 1982.

[EA] El-Gindy,H. and Avis,D. A linear algorithm for determining the visibility polygon from a point. J. of Algorithms 2 (1981), 186-197.

[FL] Freeman,H. and Loutrel,P.P. An algorithm for the two dimensional "hidden line" problem. IEEE Tr. on Electr. Comp. EC-16 (1967), 784-790.

[FKN] Fuchs,H., Kedem,Z.M. and Naylor,B.F. On visible surface generation by a priori tree structures. Computer Graphics 14 (1980), 124-133.

[GK] Gowda,I.G. and Kirkpatrick,D.G. Exploiting linear merging and extra storage in the maintenance of fully dynamic geometric data structures. Proc. 18th Ann. Allerton Conf. on Comm., Control, and Comp. (1980), 1-10.

[HZ] Hubschman,H. and Zucker,S.W. Frame-to-Frame Coherence and the Hidden Surface Computation: Constraints for a Convex World. Computer Graphics (Conf. Proc.) 15 (1981), 45-54.

[LP1] Lee,D.T. and Preparata,F.P. An optimal algorithm for finding the kernel of a polygon. J. of the ACM 26 (1979), 415-421.

[LP2] Lee,D.T. and Preparata,F.P. Euclidean shortest paths in the presence of rectiliear barriers. Submitted for publication.

[NS] Newman,W.M. and Sproull,R.E. Principles of Interactive Computer Graphics. 2nd ed., McGraw-Hill Book Co., New York, 1979.

[OW] Ottmann,Th. and Wood,D. Dynamical sets of points. In preparation, 1982.

[O1] Overmars,M.H. Dynamization of order decomposable set problems. J. of Algorithms 2 (1981), 245-260.

[O2] Overmars,M.H. Searching in the past II: General transforms. Report RUU-CS-81-9, Dep. of Comp. Sci., Univ. of Utrecht, the Netherlands, 1981.

[P] Preparata,F.P. Private communication, 1981.

[Sc] Schmitt,A. Time and space bounds for hidden line and hidden surface algorithms. Proc. Eurographics-81, North-Holland Publ. Comp., 1981.

[Sh] Shamos,M.I. Problems in computational geometry. Report, Dep. of Comp. Sci., Carnegie-Mellon Univ., Pittsburgh, Penn., 1977.

[SSS] Sutherland,I.E., Sproull,R.F. and Shumacker,R.A. A characterization of ten hidden surface algorithms. Computing Surveys 6 (1979), 1-55.

[vLM] van Leeuwen,J. and Mehlhorn,K. Private communication, 1981.

[W] Wood,D. On wire frames and haloes. Unpublished manuscript, 1982.