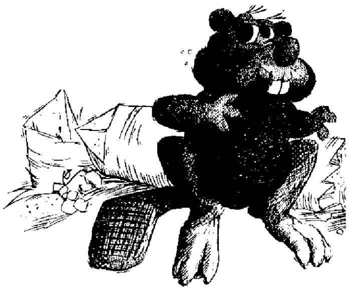
UNIVERSITY OF WATERLOO COMPUTER SCIENCE DEPARTMENT
UNIVERSITY OF WATERLOO COMPUTER SCIENCE DEPARTMENT
UNIVERSITY OF WATERLOO COMPUTER SCIENCE DEPARTMENT
UNIVERSITY OF WATERLOO COMPUTER SCIENCE DEPARTMENT



A Worst-Case Efficient Algorithm for Hidden Line Elimination

Thomas Ottmann
Peter Widmayer
Derick Wood

Data Structuring Group
CS-82-33

October, 1982

A WORST-CASE EFFICIENT ALGORITHM FOR HIDDEN LINE ELIMINATION+

Thomas Ottmann†

Peter Widmayer†

Derick Wood‡

ABSTRACT

Many algorithms for hidden line and surface elimination in a 2-dimensional projection of a 3-dimensional scene have been proposed. However surprisingly little theoretical analysis of the algorithms has been carried out. Indeed no non-trivial lower bounds for the problem are known. In the present, self-contained, paper we present a line-sweep-based hidden line elimination algorithms for 2-dimensional projections of scenes of arbitrary polyhedra. It requires $O(n \log n)$ space and $O((n+k)\log^2 n)$ time, where n is the number of edges in the 3-dimensional scene, and k is the number of edge intersections in the specific projections.

1. INTRODUCTION

In the past two decades, many algorithms for the hidden line elimination and visible surface reporting problems have been proposed in the literature (for an overview see [SSS], [FV] and [NS].) The motivation for the intensive investigation and development of such algorithms stems from their practical importance in the ever expanding field of computer graphics (computer aided design in architecture, layout of VLSI-chips etc.). Consequently, a considerable number of the algorithms have been designed for special applications, and the practitioner's viewpoint has guided nearly all of the previous research. One effect of this is that no attention has been paid to theoretical worst-case time and space bounds for the algorithms; indeed, the authors know of only four (recent) results taking worst-case complexity into account, [BBM], [LP], [S], and [Y]. Using the variable

+ This work was carried out under NATO grant No. RG 155.81; the work of the first and second authors was additionally supported by a grant from the Deutsche Forschungsgemeinschaft DFG, and the work of the third author by a Natural Sciences and Engineering Research Council of Canada Grant No. A-7700.

† Institut für Angewandte Informatik und Formale Beschreibungsverfahren, Universität Karlsruhe, Postfach 6380, 7500 Karlsruhe 1, West Germany.

‡ Data Structuring Group, Department of Computer Science, University of Waterloo, Waterloo, Ontario, Canada N2L 3G1.

grid technique Franklin [F] has obtained a linear expected time algorithm, while [R] considers the problem for one monotone cylinder, and [EOW] for 1-dimensional views of 2-dimensional scenes.

However, with the increasing significance of real-time computer graphics applications, e.g. air traffic control and the like, the need for worst-case efficient algorithms arises. For many other problems, contributions in this direction have already been made within the flourishing field of computational geometry. Paradigms developed there are applicable to the hidden line elimination problem as well.

The purpose of this paper is to give a self-contained exposition of the application of the line-sweep paradigm, see [NP], to the hidden line elimination problem, leading to a worst-case efficient algorithm. Section 2 consists of a short presentation of the problem, following the guideline of definitions and assumptions used in [SSS], [FV], and [NS]. Section 3 gives an outline of the line-sweep algorithm for hidden line elimination, and a detailed presentation follows in Section 4. The worst-case efficiency of the algorithm is derived in Section 5, while some final comments are given in Section 6.

2. THE HIDDEN LINE ELIMINATION PROBLEM

We are dealing with a scene of three-dimensional solids in three-dimensional Euclidean space. We assume that a solid is an opaque object which is bounded by planar polygonal faces, which is not necessarily convex and may have holes. The treatment of curved surfaces and solids is beyond the scope of the present paper. It must be regular in the sense of Tilove [T], that is it must be the closure of its topological interior with respect to the three-dimensional Euclidean topology. Intuitively, no dangling or isolated planes or lines are allowed. Furthermore, no two solids are allowed to intersect, that is, no point of the space belongs to more than one solid. Each face is bounded by edges and a face normal vector is associated with each face, pointing outwards from the object, normal to the face. Looking at a face from outside the object, edges are oriented so that the interior of the face lies to the right of each edge (we follow the convention of polygon edge orientation as used in [L] and [OWW], for example).

The description of the three-dimensional scene is given as a collection of descriptions of the faces. Each face is, in turn, described by a face normal vector and a collection of descriptions of the edges of the face. An edge is given by its starting and terminating point, implicitly giving the orientation, and the face it belongs to.

For a given three-dimensional scene and a given position (viewpoint) and viewing direction of an observer, the problem roughly is to eliminate from an appropriate two-dimensional projection of the scene all (parts of) edges and faces which the observer cannot see. Intuitively, this is the problem of making a realistic image of the scene with respect to the hidden and visible lines and surfaces.

Let us assume that the viewpoint of the observer doesn't lie inside any of the solids. The task of projecting onto a two-dimensional viewing plane (e.g., a screen) exactly what the observer sees, that is producing a perspective view of the scene, involves simple but laborious trigonometric computations. Most representations of three-dimensional scenes on two-dimensional screens take a different view, the orthographic projection, which is computationally easier, but nevertheless still seems to be sufficiently realistic (at least for a distant observer). The orthographic projection uses "viewing rays" which are all parallel (instead of meeting exactly at the observer's eye). [SSS] remark that "there is a perspective projection which transforms a three-dimensional object as viewed in perspective into another three-dimensional object which looks the same when viewed orthographically". Hence, for simplicity we use the orthographic projection in our presentation of the problem.

The output of the algorithm is a description of the collection of all visible parts of edges. This description must be precise, that is the coordinates of the end points of the visible parts of edges are not necessarily restricted to some finite set of values, depending, for example, on the resolution of some graphic output device, but vary over the whole range of real numbers. Of course the precision of computations is restricted to the computer's range of values. We are dealing with "object space" rather than "image space". Note that usually computations in image space refer to each "atomic resolution element", for example pixel on a screen, separately in an order which is achieved naturally by an application of the line-sweep paradigm. Indeed, output devices such as a graphics screen with a line scanning the image have been a major motivation for the development of the line-sweep paradigm.

3. AN OUTLINE OF THE ALGORITHM

It is convenient to distinguish between the object coordinate system, that is the coordinate system in which the objects's coordinates are given as input, and the eye coordinate system which is defined to be the coordinate system such that the observer's eye is at the origin and the observer looks in the direction of the positive z-axis. We will assume that the well-known linear transformation (see [NS]) has been applied to all objects' coordinates and henceforth refer to the eye coordinate system. Let us look at the orthographic projection of all objects onto the (x, y)-plane. We'll use the arrangement of coordinate axes as is usual for the two-dimensional case; the direction of the positive z-axis is away from the viewer. This convention means that we deal with a left-handed coordinate system which, however, is more intuitive than the usual right-handed one.

We assume that the usual "clipping" has been done: there are no objects "behind" the observer, that is with negative z-coordinate, and there is no restriction of the range of visibility in the (x, y)-plane, as for example the boundary of a screen. Observe that the situation described so far can be achieved by a number of operations which are only linear in the number of edges involved in the input. Let us simplify our further considerations by an additional preprocessing step: we remove all back faces. They are clearly invisible, as they are obscured by the solid to which they belong. This can be done in linear time by just looking at the z-coordinates of the faces's normal vectors, and removing the faces with this coordinate greater than or equal to zero. Hence, we are left with a set of polygonal planar faces, where (a part of) a face is visible if and only if it is not obscured by (a part of) some other face.

From now on, we deal with the projections of the remaining faces on the (x, y)-plane. With each polygon in the (x, y)-plane, a distance from the observer is associated in the form of the equation of the plane from which the polygon stems. As all solids are nonintersecting, polygons with common inner points are unambiguously ordered with respect to their distance. Note that in general polygons touching at the boundary need not be ordered, because they may be adjacent faces of one solid. Then, a (part of a) polygon edge is invisible (hidden) if and only if it lies in the intersection of the polygon to which it belongs with a less distant polygon.

Schmitt ([S]) uses the line-sweep paradigm to build up a "connection graph" containing all starting, terminating and intersection points of polygon edges on all (parts of) polygon edges, augmented with some additional information. In order to detect all visible lines, a traversal of the connection graph is made as a second step of the algorithm. Although this algorithm has a worst-case runtime of only $O((n+k)\log n)$, where n is the number of input polygon edges and k is the number of pairs of intersecting edges, the space requirement for the connection graph can be as high as $(n+k)$. See Section 6 for further comments on [S]. In contrast, we'll present an algorithm having a runtime of $O((n+k)\log^2 n)$, but requiring only $O(n \log n)$ space. The expected space requirements are indeed much smaller, as at each step of the algorithm only a small subset of all polygon edges needs to be stored. As our algorithm works with only one sweep, each polygon edge is accessed as input, for example from a secondary storage device, only once, and is stored only for a short while during the sweep. This feature allows even large amounts of data to be processed with little actual space being used.

In detail, our application of the line-sweep paradigm to solve the given problem works as follows: Sweep a horizontal line from top to bottom through the plane, halting at each point where an edge starts or terminates or two edges intersect; the *sweeping points*. Detect the intersection points during the sweep as described in [BO], with the trivial change of sweeping "from top to bottom" instead of "from left to right". Maintain a line-sweep data structure L representing the arrangement of polygons and edges currently cut by the sweeping line. At each sweeping point, update L appropriately, depending on the type of sweeping point: at an upper end point of an edge, insert the edge into L ; at a lower end point of an edge, delete the edge from L ; at an intersection point of two edges, do whatever is necessary to maintain L .

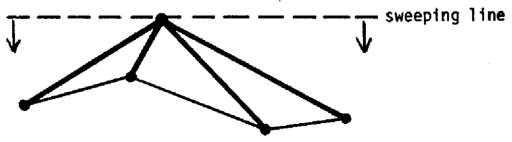
Maintain a buffer B[e] for each edge e . Whenever an edge is inserted, check whether it is visible. If it is, store the coordinates of its upper end point in its buffer. Whenever an edge is deleted, check whether it has been visible. If it was visible, output its buffer entry, together with its lower end point. Whenever two edges intersect, check each of them for previous visibility and for changes in visibility status, updating and outputting the corresponding buffers as appropriate.

The output buffering technique is used in order to avoid splitting visible lines into small pieces. The output of the algorithm is the desired description of the set of visible parts of edges. The runtime of the algorithm depends on the number of sweeping points and on the complexity of the operations carried out at each sweeping point. As the number of sweeping points is determined by the given scene, and the buffering operations are simple it is crucial to find a good data structure L supporting the desired operations.

4. THE HIDDEN LINE ELIMINATION ALGORITHM

The sweeping line stops at upper end points, lower end points, and intersection points of edges. We assume that these sweeping points are determined analogous to the description in [BO], and that the associated computations take place. We will not refer to this part of the algorithm again.

However, in contrast to many other line-sweep algorithms, we may not assume that at each sweeping point only one of the halting conditions is fulfilled. For example at the upper end of the projection of a solid many polygons may start:



In such a case, we first change the situation immediately above the sweeping line to the situation immediately below it, that is apply all changes to the line sweep data structure L , and then check for visibility all edges that have been affected by the operations.

We may, however, impose the usual restriction that no horizontal edges occur. If they do we transform the coordinate system into a new one such that this constraint is satisfied, then apply our algorithm and transform the resulting description back to the original coordinate system. Both transformations need only time linear in the number of edges involved.

An edge is visible if and only if all polygons in which it lies are more distant than the polygon to which the edge belongs. The visibility of (a part of) an edge can change only at its own starting and terminating points and at intersection points with other edges. Hence, to detect the visible parts of edges during a line-sweep it is sufficient to stop at those points.

At each step of the algorithm, the sweep-line data structure L keeps track of only those edges and associated polygons that are cut by the actual sweep-line position. This is sufficient in order to determine the visibility of edges at the crucial points. The parts of each polygon actually cut by the sweep-line are represented as intervals, such that the two boundaries of each interval are the two corresponding polygon edges. The ordering of interval boundaries does not change between any two adjacent sweeping points. With each interval is associated its distance in the form of the equation of the plane in which the corresponding polygon lies.

In order to maintain L during the sweep, the following operations have to be performed: whenever a new polygon (part) is encountered, that is, the sweep-line stops at the starting point of two adjacent edges, the corresponding interval has to be inserted into L, together with its distance; whenever the sweep-line stops at a terminating point of two adjacent edges, the corresponding interval has to be deleted from L; whenever the sweep-line stops at the terminating point of an edge which is the starting point of another edge, the interval boundary has to be replaced by the new one (the ordering doesn't change, however).

For example, immediately above the sweeping point depicted in Figure 1, L stores the three intervals $I_1 = [e_1, e_2]$, $I_2 = [e_6, e_1]$, and $I_3 = [e_2, e_7]$, representing the "active" parts of polygons p_1, p_2 and p_3 respectively. At the sweeping point, I_1 has to be deleted; the boundary e_1 of I_2 has to be replaced by e_3; the boundary e_2 of I_3 has to be replaced by e_5;

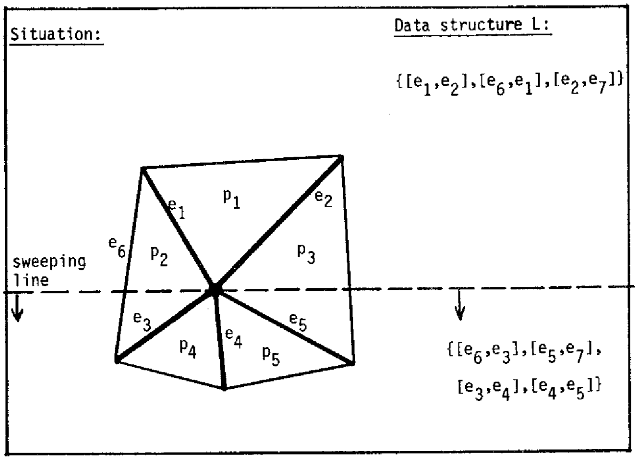


Figure 1

two intervals $I_4 = [e_3, e_4]$ and $I_5 = [e_4, e_5]$ have to be inserted into L. Hence, immediately below the sweeping point, L stores four intervals $I_2 = [e_6, e_3]$, $I_3 = [e_5, e_7]$, $I_4 = [e_3, e_4]$ and $I_5 = [e_4, e_5]$, as desired.

In addition to the maintenance of L, visibility checks for edges have to be performed. The active part of an edge e is visible if and only if each interval in L within which e lies has distance greater than the polygons to which e belongs. Hence, it is sufficient to consider the closest of all intervals in L enclosing e.

Observe that queries for the closest enclosing interval take place whenever the visibility of an edge might change, that is, at each sweeping point. On the other hand, the structure L needs to be updated only at starting and terminating points of edges. As we shall see later, it is useful to maintain the total ordering of the interval boundaries; if we do so, it is of course necessary to update this ordering at all sweeping points, too.

The possibly large number of sweeping points naturally leads to a splitting of visible edges into many small successive visible parts. We avoid this by the standard buffering technique already described in Section 3.

Let us summarize the operations that must be supported by the scan line data structure L :

(i) *insertion* of an interval with associated distance;
(ii) *deletion* of an interval with associated distance;
(iii) *query* with a point to determine the closest enclosing interval.

An interval, having two boundaries, together with the distance, can be viewed as a 3-dimensional point. Formulating the enclosure query as a dominance query ([EO] show how this is possible), we can express the above operations alternatively as:

(i') *insertion* of a 3-dimensional point;
(ii') *deletion* of a 3-dimensional point;
(iii') *query* with a 2-dimensional point to determine the 3-dimensional point with minimal first coordinate among those 3-dimensional points that are dominated by the query point in both other coordinates.

By this reformulation we have a special case of a 3-dimensional dominance query. [EO] show that each 2d-dimensional dominance searching problem is equivalent to a d-dimensional rectangle containment searching problem. For a special type of d-dimensional rectangle containment query, namely the orthogonal range query, a lower bound of $\Omega(n(\log n)^d)$ is known from Fredman [Fr]. This lower bound also carries over to the 2d-dimensional dominance searching problem. Though this does not really provide a lower bound for our special 3-dimensional dominance query, it leads us to suppose that a data structure L requires $O(\log^2 n)$ time in the worst case for at least one of the operations.

We now give a description of the hidden line elimination algorithm in terms of the primitive operations:

```
Algorithm HLE
    { The operations of Bentley-Ottmann's line segment intersection algorithm
    [BO] are carried out as well, without further notice. They determine the
    sweeping points.}

    procedure try output ( e : edge);
    begin   if output buffer B[e] is marked
                    then  begin  output (e, B[e]);
                                 output current point of e;
                                 unmark B[e]
                          end
    end;
    function    visible (e : edge) : boolean;
    begin       query L with e determining the distance d
                of the enclosing interval;
                visible := (d ≥ distance of interval of e)
    end.

begin { HLE }
    Start with initially empty sweep-line data structure L
    and unmarked output buffer B.
    for all sweeping points y from top to bottom do
    begin {updates of L }
        for all intervals I starting at y do insert I into L ;
        for all intervals I ending at y do delete I from L ;
        for all intervals I for which at y a boundary edge
                is replaced by another do
            begin    delete old I from L ;
                     insert new I into L
            end;
        {visibility checks and output}
        for all edges e ceasing to be active at y do try output (e);
        for all edges e becoming active at y do
            if visible (e) then mark B[e] with y;
        for all edges e intersecting some other edge at y do
            if visible (e)     then if B[e] is not marked
                                    then mark B[e] with y
                                    else
                               else try output (e)
    end
    {at the end of the operations of the algorithm, L and B are again empty,
    and all visible parts of edges have been output}
end { HLE }.
```

In order to show that Algorithm HLE does indeed have a worst-case complexity of $O(\log^2 n)$ per insert, delete and query operation, we present a data structure L supporting the required operations within this time bound. Let us recall that, for a specific sweep-line position, the visibility query is the question of determining the closest interval (of a given set of intervals) in which a query point lies. In order to be able to answer such a question efficiently, we chose an augmented dynamic segment tree to be the sweep-line data structure. In this structure, storing a set of intervals in one dimension involves maintaining the values of the interval's end points in sorted order. Recall that L also requires updates at intersection points of edges, because the total ordering of interval boundaries changes. As for the usual segment trees, the sweep-line is divided into elementary intervals, so-called fragments, defined by all pairs of adjacent interval end points. Hence, each (original) interval consists of a sequence of adjacent fragments. For a detailed discussion of the segment tree see [BW], for example.

The dynamic segment tree was first described in [E]. Instead of an optimal binary tree as for the static segment tree version (see [B], [BW]) a tree of bounded balance (see [NR], [W1] and [W2]) is used. As in the usual static segment tree, each leaf of the tree represents a fragment, and each inner node represents an interval which is the union of the intervals represented by its sons. For each node p, we denote by $I(p)$ the interval represented by p. With each node p, a secondary structure is associated, storing all segments s for which $(I(p) \subseteq s)$ $(I(\text{father}(p)) \not\subseteq s)$ The weight-balance property of the dynamic segment tree provides the basis for a counting argument showing that the worst-case cost for rebalancing and the necessary creation of new secondary structures is $O(n \log n)$ for a series of n insertions and deletions of segments in the initially empty tree, provided that the creation of a new secondary structure from old ones takes time at most linear in the number of segments stored in the secondary structure. For a detailed presentation of this argument see [E].

The dynamic segment tree, which we will call the primary structure, supports the operations

$$\left.\begin{array}{l}\text{INSERT LEAF}\\ \text{DELETE LEAF}\\ \text{SEARCH LEAF}\end{array}\right\} \text{ structural changes including rebalancing}$$

with an amortized worst-case cost per operation of $O(\log n)$, when n polygon edges are given as input, providing an upper bound for the number of fragments ever stored in the primary structure. The secondary structure is chosen in order to support the operations

SEARCH interval
INSERT interval
DELETE interval
MIN DISTANCE

in at most $O(\log m)$ time, when m intervals are currently stored in the secondary structure, and the operations

UNION
INTERSECTION
DIFFERENCE

of two secondary structures yielding one new secondary structure with at most $O(m)$ cost per operation. The latter bound is sufficient to guarantee an $O(n \log n)$ total cost for rebalancing operations.

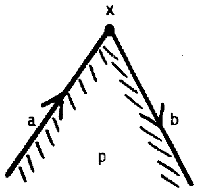
The time needed for the secondary SEARCH, INSERT, DELETE, and MIN DISTANCE operations turns out to be crucial for the total worst-case time bound of the algorithm. We have not been able to find a structure that supports all of these operations faster than $O(\log n)$ time. It is, however, possible to account for SEARCH, DELETE and MIN DISTANCE or for SEARCH, DELETE and INSERT with only $O(1)$ worst-case cost by using additional pointers and an additional dictionary (for the secondary structures altogether) or by using a layered tree scheme for all secondary structures together (see [VW]), respectively. As in both cases one of the operations keeps its $O(\log m)$ complexity, this doesn't improve the overall worst-case behaviour of the secondary structures. Hence, we use the simplest of the structures suitable for our purposes, a height-balanced binary search tree (AVL-tree, see [AVL]). Each node of an AVL-tree stores a polygon, associating with it the equation of the plane in which the polygon lies. The entries are sorted with respect to the distances of the polygons. Note that within the secondary structures we are not concerned with polygon boundaries or changes of the order in which the polygons are stored; the algorithm will ensure, on the primary structure's level, that the entries of the secondary structures are correct by appropriately inserting and deleting polygons.

Let us recall the overall data structure: The primary structure PRIM is a dynamic weight-balanced segment tree, in which each node represents an interval. The interval boundaries are represented by the equation of the corresponding edges. A secondary structure $\mathrm{SEC}(p)$ is associated with each node p of PRIM. $\mathrm{SEC}(p)$ is an AVL-tree storing all polygons that is, intervals for any given sweep-line position, which make use of the interval $I(p)$ in their canonical segment tree covering. The polygons in $\mathrm{SEC}(p)$ are sorted with respect to their distances.

It remains to be shown how the updates of the line-sweep data structure can be performed, and how the visibility check for an edge is done. Observe that an edge is visible if and only if it belongs to the nearest face present in one of the adjacent elementary intervals in PRIM. All faces f present at any given elementary interval $I(l)$ of a leaf l of PRIM have exactly one entry of f in $\mathrm{SEC}(p)$ for one of the nodes p on the path from the root of PRIM to leaf l. No other entries occur in $\mathrm{SEC}(p)$ for these nodes p. Hence, the nearest of the faces present in $I(l)$ is the nearest of the faces with MIN DISTANCE in $\mathrm{SEC}(p)$. This means that we can find the nearest face present in $I(l)$ in $O(\log^2 n)$ time by finding the minimum in $O(\log n)$ structures $\mathrm{SEC}(p)$. Hence, each visibility test for an edge in PRIM can be answered in $O(\log^2 n)$ time by looking at both adjacent elementary intervals.

In order to show how the updates of the line-sweep data structure can be performed, we distinguish between the possible cases occurring at a sweeping point x :

(1)



{ x is a sweeping point }

{a, b are edges }

{ p is a polygon }

- PRIM INSERT LEAF (a,b)
  by splitting the leaf into which (a, b) falls (see Figure 2) { note that SEC(*leaf* (a, b)) is empty };
- SEC INSERT p INTO *leaf*(a, b) .

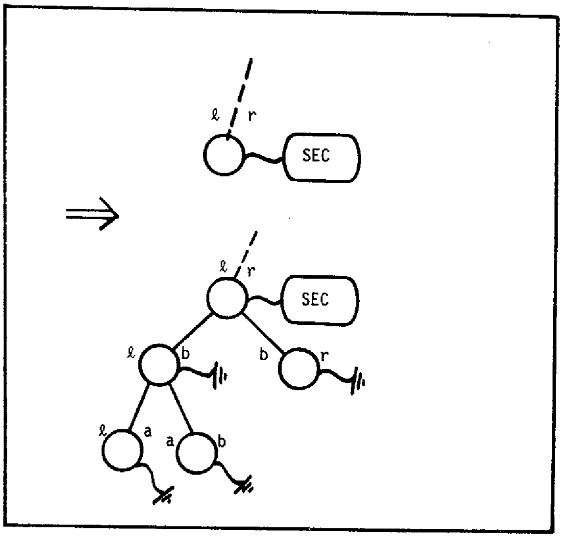
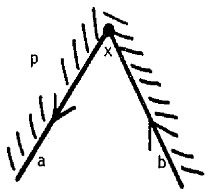


Figure 2 : split of a leaf in PRIM

(2)



- PRIM INSERT LEAF (a, b)
  by splitting the leaf into which (a, b) falls (see Figure 2);
- let u be the (unique) node on the path from the former $leaf(l, r)$ to the root such that $p \in$ SEC(u) ; u can be found by a series of $O(\log n)$ SEC SEARCH operations;
  SEC DELETE p FROM u ;
- DISTRIBUTE p DOWNWARDS FROM u (see Figure 3)
  { distributes the entry p to those nodes in the subtree below u which p uses for the canonical segment covering }.

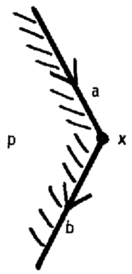
```
DISTRIBUTE p DOWNWARDS FROM u :

begin if interval ( u ) IN p
            then SEC INSERT p INTO u
            else  begin
                        DISTRIBUTE p DOWNWARDS FROM leftson(u) ;
                        DISTRIBUTE p DOWNWARDS FROM rightson(u)
                  end
end;

{ entry p is distributed to O(log n) nodes and inserted in their SECs,
according to the canonical segment covering }
```

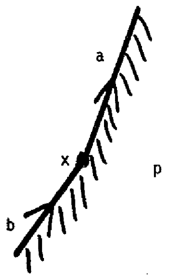
Figure 3: The DISTRIBUTE procedure

(3)



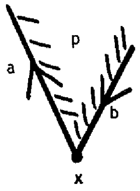
- PRIM SEARCH LEAF $(?, a)$ and
  PRIM SEARCH LEAF $(a, ?)$;
- replace a by b everywhere on the path from the root to the two found (adjacent) leaves in PRIM.

(4)



- same as (3)

(5)



```
- PRIM SEARCH LEAF $(a, b)$
  { $(a, b)$ really is a leaf and $p$ is the only entry in its SEC, because $a$ and
  $b$ are both boundaries of $p$ ; the situation is as depicted in Figure 4 or
  symmetric to it }
- observe (see Figure 4):
      all faces in SECs of nodes on the path from $u_l$ to $v_l$ and from $u_r$ to
      $v_r$ have right ends $\geq r$ and left ends $\leq l$ , respectively; edge a
      separates left and right subtree of $u$ ;
  choose $l$ to be the new separating edge:
  replace $a$ and $b$ by $l$ in the PRIM interval descriptions of all nodes from
  $u_l$ to $v_l$ and from $u_r$ to $v_r$ ;
```

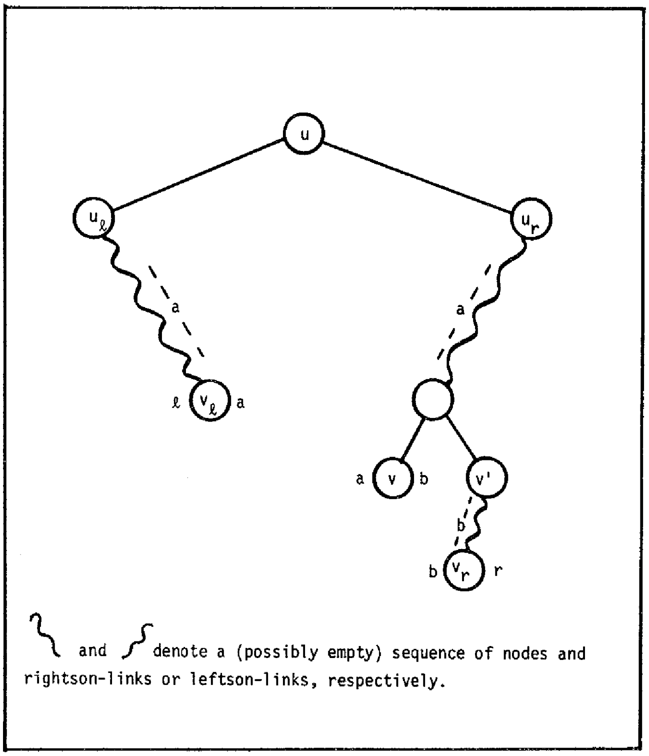

u
u_ℓ
u_r
a
a
ℓ v_ℓ a
a v b
v'
b
b v_r r
and denote a (possibly empty) sequence of nodes and rightson-links or leftson-links, respectively.

Figure 4: PRIM before sweep-line halt

- PRIM DELETE LEAF v ;
- PRIM DELETE LEAF v_l ;
- merge v' with its father: take SEC from father(v'), because SEC(v') is empty;
- **IF** former father $(v_l) = u$
  then merge u with *rightson*(u): build a disjoint union of their SECs of constant length
  else merge former father(v_l) with former brother(v_l): build disjoint union of SECs of constant length.

The situation changes as depicted in Figure 5.

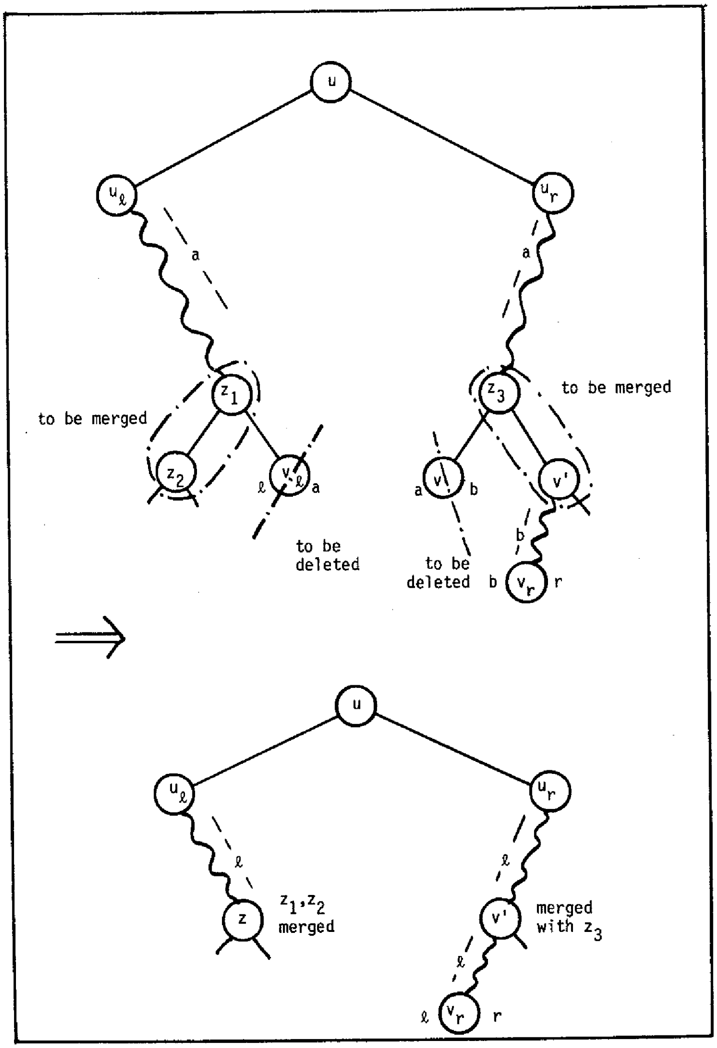
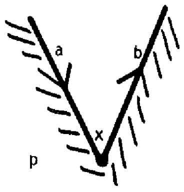


Figure 5: Change of situation in cases (5), (6); example for $u_\ell \neq z_1$

(6)



- the situation for PRIM is as depicted in Figure 4; p NOT IN SEC(v); exactly one node from u_l to v_l and exactly one from v' to v_r contains p IN SEC; restructure PRIM and merge SECs as in case (5), yielding the situation depicted in Figure 5;
- because the region occupied by p has increased, it may be necessary to collect p's entries in SECs and make an entry of p at some higher node; the only place for this change can be v' : the former uncle of v' has become its brother; therefore

```
if ( $p$ in SEC( $v'$ )) and ( $p$ in SEC (brother( $v'$ ))) then
            begin
                  SEC DELETE $p$ FROM $v'$ ;
                  COLLECT $p$ UPWARDS FROM $v'$
            end;
{ see Figure 6 }.
```

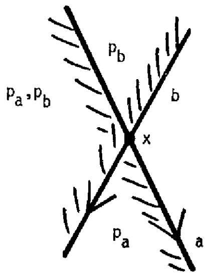
COLLECT p UPWARDS FROM v :

```
begin if $p$ in SEC (brother( $v$ ))
            then begin
                        SEC DELETE $p$ FROM brother( $v$ );
                        COLLECT $p$ UPWARDS FROM father( $v$ )
                  end
            else SEC INSERT $p$ INTO $v$
end;
```

{ entry p is collected and deleted from $O(\log n)$ SECs of brothers of path nodes and inserted in one node's SEC }

Figure 6: The COLLECT procedure

(7)



- don't change the structure of PRIM; on the paths from the root to the (adjacent) leaves (l, a), (a, b) and (b, r), exchange a with b ;
- one of the following two cases can occur:

(i) as depicted in Figure 4:
exactly one node from u_l to v_l contains p_a and one contains p_b in SEC; also, p_b IN SEC(v); so

- SEC DELETE p_b FROM v ;
- SEC INSERT p_a INTO v ;

(ii) as depicted in Figure 7:
let v_2 be the unique node between u_l and z_1 such that p_b IN SEC(v_2); there is exactly one node between z_2 and v_l with p_a IN SEC; p_b NOT IN SEC(v); so

- SEC DELETE p_b FROM v_2 ;
- DISTRIBUTE p_b DOWNWARDS FROM v_2 ;
- COLLECT P_a UPWARDS FROM v ;

{ note that no restructuring takes place in PRIM }

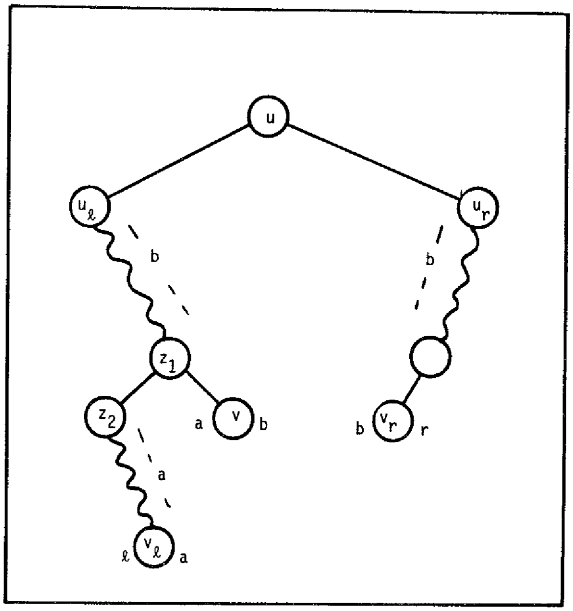
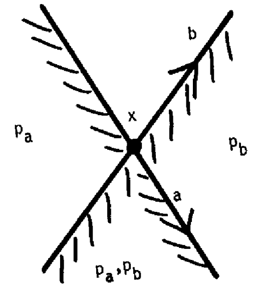


Figure 7: PRIM before sweep-line halt

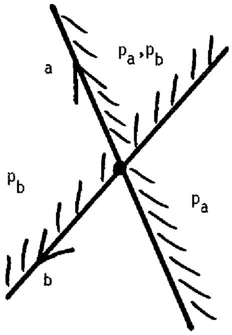
(8)



- see the situation as depicted in Figure 4; the treatment of the symmetric situation is, as opposite to case (7), symmetric; let v_1 be the unique node between u_l and v_l such that p_a IN SEC(v_1); let v_2 be the unique node between v_r and v' such that p_b IN SEC(v_2);

  - COLLECT p_1 UPWARDS FROM v ;
  - COLLECT p_2 UPWARDS FROM v ;
  - exchange a with b at the interval boundaries in PRIM.

(9)



- one of the following two cases occurs:

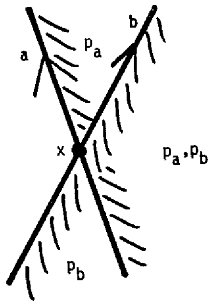
  (i) as depicted in Figure 4:

  p_b IN SEC(v); let v_2 be the unique node between u_l and v_l such that p_b IN SEC(v_2); let v_1 be the unique node between u_r and v such that p_a IN SEC(v_1);

  - SEC DELETE p_b FROM v ;
  - SEC DELETE p_a FROM v_1 ;
  - DISTRIBUTE p_a DOWNWARDS FROM v_1 ;
  - exchange a with b at the interval boundaries in PRIM:

  (ii) as depicted in Figure 7: symmetric to (i): exchange indices l with r , a with b , and 1 with 2.

(10)



This case is symmetric to case (7). See case (7) for details.

5. A WORST-CASE UPPER BOUND FOR THE COMPLEXITY OF THE ALGORITHM

From the previous observations, we know that at each sweeping point $O(\log^2 n)$ operations are sufficient, if n denotes the number of edges given in the input to the algorithm. Let k be the number of intersection points of polygon edges. Then the Bentley-Ottmann algorithm finds all intersections in time $O((n+k)\log n)$. Hence, the time for our hidden line elimination algorithm is $O((n+k)\log^2 n)$, taking into account the detection of the sweeping points, the updates of the sweep-line data structure, the visibility checks for edges and the output of the visible lines.

The space requirements are bounded from above by the space requirements for the sweep-line data structure, because the output buffer as well as the structure needed for detection of the intersections of edges can be kept linear (see [Br]). The primary structure PRIM needs at most linear space, too. However, each polygon may have an entry in $O(\log n)$ secondary structures SEC at any given time. Hence, the overall space complexity is $O(n \log n)$.

We state the essence of our investigation in the following:

Theorem: The hidden line elimination problem can be solved by a line-sweep algorithm in $O((n+k)\log^2 n)$ time using $O(n \log n)$ space, where n is the number of edges given in the input and k is the number of edge intersections.

Bearing in mind that the fastest known algorithm for detecting all edge intersections takes $O((n+k)\log n)$ time, it would be interesting to find a line-sweep algorithm for hidden line elimination with the same complexity. Such an algorithm could be said to be quasi-optimal with respect to the state-of-the-art in line-sweeping.

Note that our hidden line elimination algorithm can be easily adapted to be a visible surface reporting algorithm, by just changing the visibility checks so that they refer to (parts of) polygons instead of (parts of) edges. The dynamic segment tree structure augmented with AVL-trees is indeed powerful enough to support a number of modifications to the original problem very easily.

6. FINAL REMARKS

The result reported here can be considered to be a further step in developing a mathematical theory for computer graphics, cf. [EOW]. However, having said this, it still leaves much to be desired. [S] has produced an $O((n+k)\log n)$ time algorithm for hidden line elimination, which unfortunately requires $O(n+k)$ space. Thus, although the algorithm is quasi-optimal, it is so at the expense of high space requirements, and even the author of the algorithm [S1] has found this to be unbearable when implemented, since the connection graph must be available throughout the computation.

The first question is, simply: Is there a hidden line elimination algorithm requiring $O((n+k)\log n)$ time but only $O(n \log n)$ space?

Second, is there a quasi-optimal algorithm requiring only $O(n)$ space?

Third, and more interestingly, edge intersections may play little part in a particular projection. For example, consider a scene of polyhedra which is blocked by a large wall. A projection may only show the wall, yet all edge intersections are considered by our algorithm. Is there an algorithm whose complexity is expressed in terms of both the number of edges in the scene and the number of visible edges in the projection?

Fourth, and finally, what non-trivial lower bounds can be obtained?

REFERENCES

[AVL] Adelson-Velskii, G.M. and Landis, Y.M.: An Algorithm for the Organization of Information; *Dokl. Akad. Nauk SSSR* 146, (1962), 263-266.

[BBM] Beatty, J.C., Booth, K.S. and Matthies, L.H.: Revisiting Watkins' Algorithm, *Seventh Canadian Man-Computer Communications Society Conference* (June 1981), 359-370.

[B] Bentley, J.L.: Solutions to Klee's Rectangle Problems; unpublished manuscript, 1977.

[BO] Bentley, J.L. and Ottmann, Th.: Algorithms for Reporting and Counting Geometric Intersections; *IEEE Transactions on Computers* C-28, (1979), 643-647.

[BW] Bentley, J.L. and Wood, D.: An Optimal Worst-Case Algorithm for Reporting Intersections of Rectangles; *IEEE Transactions on Computers* C-29, (1980), 571-577.

[Br] Brown, K.Q.: Comments on 'Algorithms for Reporting and Counting Geometric Intersections'; *IEEE Transactions on Computers* C-30 (1981), 147-148.

[E] Edelsbrunner, H.: Dynamic Data Structures for Orthogonal Intersection Queries; Institut für Informationsverarbeitung, Technische Universität Graz, Report No. 59, 1980.

[EO] Edelsbrunner, H. and Overmars, M.H.: On the Equivalence of Some Rectangle Problems; *Information Processing Letters* 14 (1982), 124-127.

[EOW] Edelsbrunner, H., Overmars, M.H. and Wood, D.: Graphics in Flatland: A Case Study. Technical Report CS-82-25, Department of Computer Science, University of Waterloo 1982.

[FV] Foley, J.D. and Van Dam, A.: *Fundamentals of Interactive Computer Graphics*; Addison-Wesley Publishing Co., Reading, Mass. 1982.

[F] Franklin, W.R.: A Linear Time Exact Hidden Surface Algorithm. *SIGGRAPH '80 Conference Proceedings, Computer Graphics* 14 (1980), 117-123.

[Fr] Fredman, M.L.: A Lower Bound on the Complexity of Orthogonal Range Queries; *Journal of the ACM* Vol. 28, (1981), 696-705.

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

[L] Lauther, U.: An $O(N \log N)$ Algorithm for Boolean Mask Operations; *Proceedings 18th Design Automation Conference,* Nashville, (1981), 555-562.

[LP] Lee, D.T. and Preparata, F.P.: Private communication, 1982.

[NS] Newman, W. and Sproull, R.: *Principles of Interactive Computer Graphics*; 2nd edition, McGraw-Hill, New York, 1979.

[NP] Nievergelt, J. and Preparata, F.P.: Plane-Sweep Algorithms for Intersecting Geometric Figures; *Communications of the ACM* (1982), to appear.

[NR] Nievergelt, J. and Reingold, E.M.: Binary Search Trees of Bounded Balance; *SIAM Journal on Computing* 2, (1973), 33-43.

[OWW] Ottmann, Th., Widmayer, P. and Wood, D.: A Fast Algorithm for Boolean Mask Operations; Technical Report CS-82-37, Department of Computer Science, University of Waterloo, 1982.

[R] Rappaport, D.: Eliminating Hidden Lines from Monotone Slabs; *Proceedings of the 20th Allerton Conference on Communication, Control, and Computing* (1982), to appear.

[S] Schmitt, A.: On the Time and Space Complexity of Certain Exact Hidden Line Algorithms; Universität Karlsruhe, Fakultät für Informatik, Report No. 24/81, 1981.

[S1] Schmitt, A.: Personal communication, 1982.

[SSS] Sutherland, I.E., Sproull, R.F. and Schumacker, R.A.: A Characterization of Ten Hidden-Surface Algorithms; *Computing Surveys* 6, (1974), 1-55.

[T] Tilove, B.: Set Membership Classification: A Unified Approach to Geometric Intersection problems; *IEEE Transactions on Computers* C-29, (1980), 874-883.

[VW] Vaishnavi, V.K. and Wood, D.: Rectilinear Line Segment Intersection, Layered Segment Trees and Dynamization; *Journal of Algorithms*, 3 (1982), 160-176.

[W1] Willard, D.E.: Predicate Oriented Database Search Algorithms; Harvard University, Aiken Computer Laboratory, Report TR-20-78, 1978.

[W2] Willard, D.E.: An Introduction to Super-B-Trees; University of Iowa, Department of Computer Science, 1979.

[Y] Yao, F.: On the Priority Approach to Hidden-Surface Algorithms. *Proceedings of the 21st IEEE Annual Symposium on Foundations of Computer Science* (1980), 301-307.