An Efficient Output-Sensitive Hidden-Surface Removal Algorithm for Polyhedral Terrains*

John H. Reif† *and Sandeep Sen*‡

† Computer Science Department, Duke University, Durham, N.C. 27706

‡ Department of Computer Science & Engineering, IIT Delhi 110016, India

Abstract

In this paper we present an algorithm for hidden surface removal for a class of polyhedral surfaces which have a property that they can be ordered relatively quickly. For example, our results apply directly to terrain maps. A distinguishing feature of our algorithm is that its running time is sensitive to the actual size of the visible image, rather than the total number of intersections in the image plane which can be much larger than the visible image. The time complexity of this algorithm is $O((k+n)\log^2 n)$ where n and k are respectively the input and the output sizes. Thus, in a significant number of situations this will be faster than the worst case optimal algorithms which have running time of $\Omega(n^2)$ irrespective of the output size.

Key words algorithms, data-structures, computational geometry, graphics, hidden-surface removal

[0]* A preliminary version of the main result appeared as a part of the paper entitled 'An efficient output-sensitive hidden surface removal algorithm and its parallelization' in the Proc of the 4th Annual ACM Symposium on Computational Geometry, 1988.

This research was supported by National Science Foundation Grant CCR-8696134, by a grant from the Office of Naval Research under contract ONR-N00014-87-K0310 and by Air Force contract AFSOR-87-0386.

1 Introduction

1.1 The problem and previous work

The hidden-surface elimination problem (see [9] for an early history) has been a fundamental problem in computer graphics and can be stated in the following manner. Given n polyhedral faces in a three dimensional environment and a projection plane, we wish to determine which portions of the faces are visible when viewed in a direction perpendicular to the projection plane. We are interested in a *object-space* solution (independent of the display device) for this problem. That is, we are interested in producing a combinatorial description of the visible scene which can then be rendered on any display device. Some solutions compute the visibility information at every pixel which makes them device dependent and are called *image space* algorithms. It has been shown that the worst case output size for hidden-surface elimination can be $\Omega(n^2)$ for n segments and hence it is clear that the worst case optimal algorithms for these problems will have a running time of $\Omega(n^2)$. Recently McKenna [12] and Devai [6] proposed algorithms for the general problem that run in $O(n^2)$ time and hence are worst-case optimal.

A slightly different version is the hidden-line elimination problem, where we are concerned only with the visibility of the edges (and not regions). The algorithms for hidden-surface removal can be easily modified for the hidden-line elimination case. There are algorithms for hidden line elimination in literature whose running time is sensitive to the number of intersections, k, (of the projection of the segments) in the image plane, typically of the order of $O((n+k)\log n)$ (for example see Nurmi [14] and Schmitt [20]). Very recently this was improved to $O(n\log n + k + t)$ by Goodrich [7] where t is number of intersecting polygons on the image plane. However, in practice, the size of a displayed image can be far less than the number of intersections in the image plane. By size, we mean the number of edges and vertices of the displayed image as a (planar) graph. This would happen when a large number of these intersections are occluded by visible surfaces and hence do not increase the complexity of the image (see Figure 1). Our objective is to design an algorithm whose running time is sensitive to the final displayed image rather than the number of intersections. In this paper we design output-sensitive algorithms for a restricted class of surfaces like terrains which occur frequently in practice.

The terrain maps are polyhedral surfaces in 3-space (see Figure 2) which can be represented as functions of two variables, for example $z = f(x, y)$. Most geographical features can be represented in this manner. Another characteristic of these surfaces is that the upper boundary of their projection on the $z - y$ plane is monotone with respect to y axis. The term (upper) profile will refer to the piecewise linear function $Z(y)$, which is the point-wise maximum in $+z$ direction of the projected terrain on $y - z$ plane. In fact monotonicity turns out to be a very useful property for making the algorithm somewhat simpler than hidden-surface removal algorithm for general surfaces. In spite of this, it is known that the size of the visible image can be $\Omega(n^2)$ in the worst case, which is the number of intersections in the projection of the line segments into the image plane. Although there has been some previous work on output-sensitive algorithms, they have been restricted [1] to objects that are horizontal axis-parallel rectangles (Bern [1]). We present the first efficient algorithms for

[1] see postscript at the end of the section

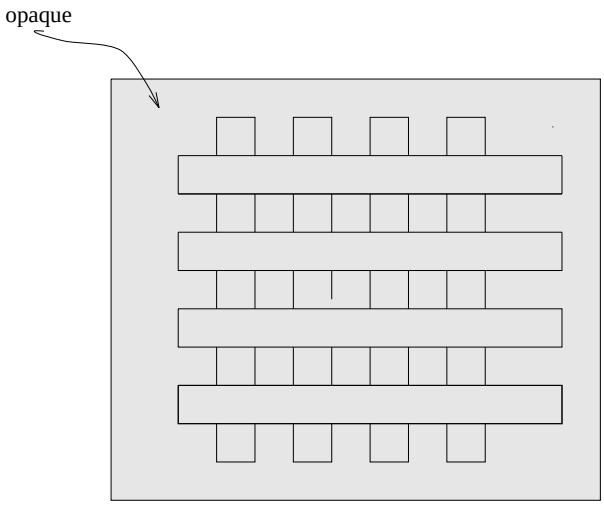


Figure 1: The visible scene has only constant complexity whereas the number of intersections is $\Omega(n^2)$.

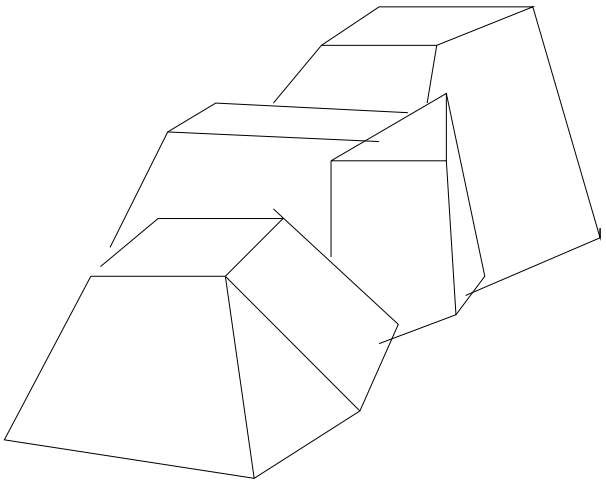


Figure 2: A typical terrain map.

terrain-maps. By efficient we mean within a polylogarithmic [2] factor of the lower-bound.

A commonly used technique is to process the surfaces in increasing distance from the viewer (negative x direction) so that each point needs to be tested only once for visibility, i.e., a point once pronounced as visible is not going to be altered later in the course of the algorithm (the same holds true for the occluded points). The origins of this approach can be found in [21]. However instead of performing the visibility test for each point on the display device so that the complexity of the algorithm is also dependent on the resolution of the display device, we do it in a device-independent manner. The output of our algorithm is a graph of the final image and not a pixel by pixel description of the image. Our techniques apply to terrain maps whose edges can be ordered from 'front to back' very efficiently.

Postscript

Subsequent to an earlier version of this paper [18], there have been more algorithms for this special case of hidden-surface elimination. The algorithm in [11] improves the running time to $O((n\alpha(n)+k)\log n)$ where as the algorithm in [17] achieves the same running time as reported in this paper. Moreover, even for the general case, there are algorithms which are somewhat output sensitive - for example the algorithms of Berg et al. [5] and Overmars and Sharir [16]. However these are still away from the ideal bound of of $O((n+k)\text{polylog}(n))$.

2 An overview of the algorithm

Recall from the introduction that terrains in this paper refer to piecewise linear surfaces which meets a vertical line in exactly one point. The scene is embedded in a right-hand coordinate system with the surface being a function of the x and y coordinates. The observer is assumed to be located at $x = \text{inf}$, that is, the viewing plane is the $z - y$ plane. We assume that the terrain map is available as a graph G whose vertices are 3-tuples (x, y, z) of coordinates in 3D Euclidean space from these vertices and whose edges correspond to the segments of the polyhedral surface. We also assume that only the top part of the surface is visible, i.e. the faces closest to the observer rise from the ground-level.

The known worst-case optimal algorithms for hidden-surface removal would compute the entire arrangement of the projection of segments on the viewing plane. This makes them unsuitable for output-sensitive as a large number of such intersections could be invisible to the observer. Recently a more clever implementation of this approach in a recent paper by Overmars and Sharir [16] yields output sensitive algorithms which are efficient for output sizes exceeding about $O(n^{4/3})$.

Our algorithm processes segments one by one; it maintains an upper profile of the segments processed until the present stage and tests for the visibility of the current segment by intersecting the segment with this profile. The portion of the segment below the upper profile (which is a simple monotone polygon) is not visible and is hence discarded. The upper profile may have to be updated with the portions of the segment that are visible. An algorithm based on a similar approach was proposed by Guting and Ottman [8] to handle special cases such as aligned rectangular faces and c-oriented polygons. The main procedure in our algorithm centers around detecting the

[2] polylogarithmic in this paper denotes $\log^c n$ for some fixed constant c.

intersection(s) of a line segment with a simple (actually a monotone) polygon. For this purpose, we use an efficient algorithm given by Chazelle and Guibas [3] for ray shooting in a simple polygon. However, we need to modify their algorithm to suit a dynamic environment since the polygon (upper profile) is modified over the course of execution. In a later section, we review their algorithm in the context of making it dynamic.

A key property that allows us to solve the visibility problem efficiently is that the edges can be ordered from 'front' to 'back' using the following observation. We project G on the $x-y$ plane (call this G_{xy}) and now the ordering of segments in the scene in increasing distance from the viewer corresponds to ordering of edges of G_{xy} along x. That is, we define a partial order as follows: edge $e_i \prec e_j$ iff there is a ray in the viewing direction that intersects e_i before e_j, then the projection of the edges on the $x-y$ plane preserves this ordering. To compute the ordering of the projected segments, we make use of a procedure for decomposition of a planar graph into *chains*.

Definition : A chain $\mathcal{C} = (u_1, u_2, ..u_p)$ is a planar straight line graph (PSLG) with vertex set $\{u_1, ..u_p\}$ and edge set $\{(u_i, u_{i+1})\}$ where $i = 1, 2, \cdots p-1$. A chain is called monotone with respect to a straight line l if a line orthogonal to l intersects $\mathcal{C}$ in exactly one point.

To compute the total ordering for the set of edges of G, we decompose G_{xy} into monotone chains (see Figure 3).

Fact 1 ([10]) *An N vertex ($O(N)$ edges) PSLG can be decomposed into a set of monotone chains in $O(N \log N)$ time and $O(N)$ space.*

Alternatively, a procedure given in Cole and Sharir [**?**] can be used for ordering the edges. Cole and Sharir [4] present algorithms for fast processing of query rays emanating from a point above the terrain map. Although the overall approaches are somewhat similar, our algorithm produces a graph of the displayed image without relating to the device-coordinates. It is not clear how the algorithm of Cole and Sharir [4] can be made into an object space algorithm.

The rest of this paper is organized as follows. In section 3, we give a high level description of our algorithm and a brief sketch of the analysis. Following this, we present a detailed description of the central procedure of the algorithm, namely, how to maintain the upper profile and complete the analysis.

3 The main algorithm

In this section we sketch the main phases and then in the following section we focus on the individual steps.

Algorithm **Visible**

(1) We project the terrain map onto the $x-y$ plane and decompose the resultant planar graph into a set of monotone chains (with respect to the x axis). The chains are ordered with respect to the x axis that yields an ordering for painting the surface from front to back.

Comment : During each stage of the algorithm, we maintain an upper profile (of the $y-z$ projection) of the part of the surface processed so far. Notice that any point below this upper profile of edges (in the z direction) would not be visible if it belonged to a surface beyond the part of the image processed until then.

(2) Pick up an segment from the monotone chain (obtained in stage 1) which is the current chain being processed; find the intersections of this segment with the upper profile (which is again a monotone chain). The basic underlying problem is to detect the intersections of a line segment with a monotone chain quickly and update accordingly the upper profile. For this, we use a scheme that described in the next section.

(3) Repeat step 2 until no more segments are left.

Note that the actual displayed image is not the upper-profile itself; rather it is a PSLG with each face (in the PSLG) belonging to a specific surface. This is updated as we detect intersections of segments with the current upper-profile. As soon as a visible region is detected, we traverse its boundary (the bounding segments or vertices) using the ordered list of vertices of the upper profile and charge the cost to the visible face.

From Cole and Sharir refCs:86, the size of the profile is at most $O(n\alpha(n)\log n)$ where $\alpha(n)$ is the inverse Ackermann's function. [3] This does not include the space for the displayed image which is $O(k)$ nor the space for the data-structure associated with the profile for detecting intersections. We now state the main result of this paper:

Theorem 1 *Algorithm* **Visible** *runs in time $O((k+n)\log^2 n)$ and space $O(n\alpha(n)\log n + k)$ where n is related to the input size and k is the output size, i.e. the number of edges and vertices in the planar graph representing the output image.*

The proof of this result follows from the discussion in the next section.

4 Intersecting segments with simple polygons

In this section we focus on a critical procedure of our algorithm. Given a simple polygon $\mathcal{P}$ of n vertices, how does one detect all the intersections with a query segment efficiently ? Chazelle and Guibas [3] provided a near optimal solution to this problem using properties of geometric duality. The dual transform D maps lines to points and vice-versa and and we shall use the following transformation. The point $p : (a, b)$ is mapped to the line $D_p : y = ax + b$ and the line $r : cx + d$ is mapped to the point $D_r : (-c, d)$. In the remaining paper, duality will refer to this transformation D. The result of Chazelle and Guibas (to be referred as CG after this) can be summed up in the following manner:

Fact 2 *There exists an $O(n)$ space data structure representing a simple polygon $\mathcal{P}$ which, given a segment s allows us to find all k intersections between $\mathcal{P}$ and s in $O((k+1)\cdot\log(\frac{n}{(k+1)})$ time. Furthermore, this data structure can be constructed in $O(n\log n)$ time.*

[3] The $\alpha(n)$ has no relation to the α in the BB(α) trees - these are both standard notations in literature.

Although this implies a near optimal (in the sense of an $O(\log n+k)$ query time and linear space data structure) solution. It involves a complex data structure which does not appear to be well-suited for a dynamic environment, that is, one in which we may have to update this data structure periodically to accommodate a change in the polygon itself. Before we describe our dynamic structure, we shall review a simpler version of their algorithm in some detail. We step through their construction so as to highlight the various issues involved in making the data-structure dynamic.

Remark In our case the simple polygon has a special structure. The polygonal chain representing the upper profile is monotone with respect to the y axis. We will exploit this feature suitably.

4.1 Review of the basic data structure

Given a simple polygon, $\mathcal{P}$ we construct a binary tree structure where each node represents a portion of the polygon and the leaves correspond to the triangles corresponding to a triangulation. The size (number of vertices) of the polygons associated with each node decreases geometrically with depth so that the tree has a logarithmic depth. At any level of the tree, the polygons associated with the nodes of the tree are disjoint (except for a shared segment which is a diagonal in the original polygon) and their union is the entire polygon. The polygon can be partitioned into two roughly equal sized polygons using Chazelle's [2] polygon cutting algorithm that finds a diagonal joining two vertices of the polygon in linear time. By repeated applications of this partitioning step, the binary corresponding to the polygon can be constructed in $O(n \log n)$ time. Figure 3 illustrates such a tree. Each node of this tree is labeled by the diagonal edge which partitions the associated polygon into sub-polygons associated with its two children. Following the notation in CG, we denote the associated polygon at any node v of the tree by $P(v)$ and its diagonal by $e(v)$. From our previous remark about the polygon being monotone, the partitioning step becomes much simpler in our case. We drop vertical attachments (in the negative z direction) from each vertex and choose the one from the middle vertex as a separator. Hence, in our case a sorted sequence of vertices yields the required partitioning.

To detect an intersection between a polygon and a line, a locus-based approach is adopted in the dual space which maps lines to points and vice-versa. We look at the locus of points which are duals of lines (in primal space) that intersect a fixed edge of the polygon. We shall review a few results for detecting intersections using geometric duality. The next two facts were observed by CG.

Fact 3 *Let $\mathcal{P}$ be a simple polygon, and e be a fixed edge of $\mathcal{P}$. Consider the duals of all lines that intersect e and categorize them into equivalence classes that correspond to the edges they intersect first (after e) in the primal plane. Then these equivalence classes induce a convex partitioning in the dual-plane.*

See Figure 4 for an illustration. These convex subdivisions will be referred to as the *visibility test polygons* or *test polygons* in short. We shall use the notation $P_{a,b}$ to denote the convex subdivision (test polygon) associated with a line which intersects edges a and b without intersecting any part of the polygon $\mathcal{P}$ (in between). The following fact provides a bound on the total size of all the test polygons:

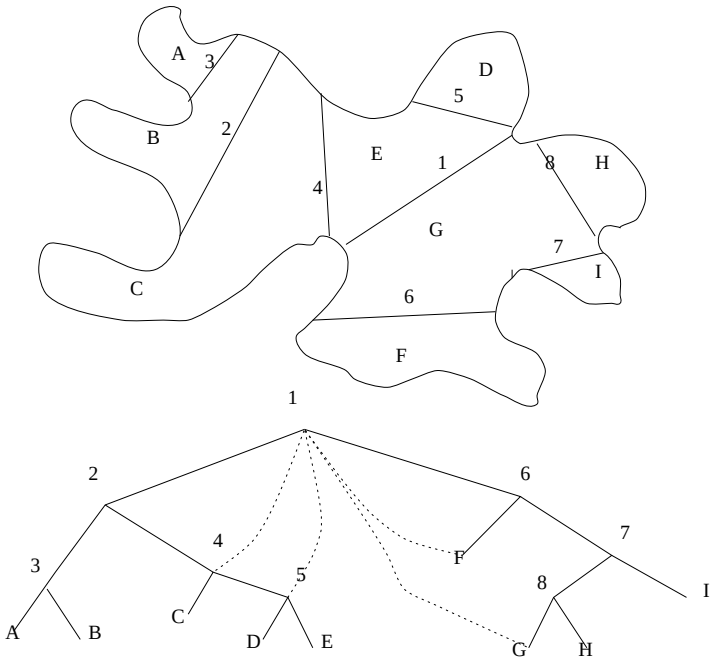


Figure 3: A polygon and its decomposition tree.

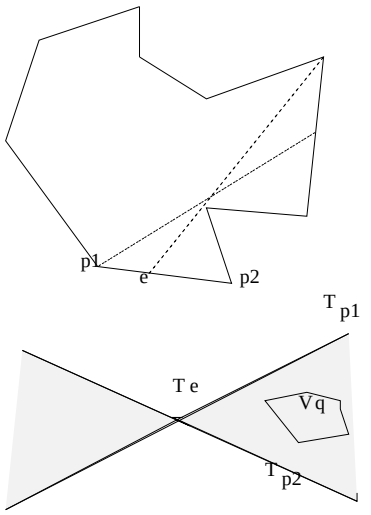


Figure 4: T_{p1} and T_{p2} are duals of points p1 and p2. T_e is the dual of edge e. The shaded region is the dual of all rays intersecting edge e and V_q is the convex region in the dual plane enclosing duals of all lines passing through e and q without intersecting the polygon in between.

Fact 4 *Given a simple polygon $\mathcal{P}$ with N vertices, the total size of all the test polygons (with respect to a fixed edge) is $O(n)$.*

In brief, the algorithm for detecting intersections of a segment s with polygon $\mathcal{P}$ can be viewed as following. Assume that we know a node v such that s intersects $e(v)$ - we shall justify this later. From each node of the binary tree, we try to find the furthest node such that the line containing segment s remains inside $\mathcal{P}$. We look for a node x of least depth (furthest distance-wise) in v's subtrees such that the line does not intersect the boundary of $\mathcal{P}$ between $e(v)$ and $e(x)$. In other words, from a node v, we try to find a test polygon $P_{e(v),e(x)}$ such that the dual of the line lies inside $P_{e(v),e(x)}$ and $e(x)$ is a diagonal associated with a node x that has the least depth among the eligible nodes (see Figure 3). By a slight abuse of notation, we will use $P_{v,x}$ instead of $P_{e(v),e(x)}$. This gives rise to a search structure, resembling a binary tree where each node is augmented by additional pointers. The binary tree in Figure 3 shows the additional (dotted) edges that have been added according to this scheme. A (dotted) edge is added between all pairs of nodes (v, w) such that $e(w)$ is a segment in the boundary of $P(v)$. The following properties can be verified easily:

(P1) For any node v, there can be at most two dotted edges between v and its children at a fixed level.
(P2) In our case (of the monotone-polygon), there can be at most one dotted edge between v and an ancestor of v. This is not true in the general case (see Figure 3) since $P(w)$ can have more than two diagonals in its boundary.
(P3) The size of a test polygon $P_{x,y}$ in our case is proportional to the number of vertices between diagonals $e(x)$ and $e(y)$.

Henceforth, we shall refer to the dotted edges of Figure 3 as 'shooting-pointers'. The size of the tree (including the extra pointers) is still $O(n)$; however the size of the test polygons associated with the pointers incident on each level is $O(n)$. This amounts to $O(n \log n)$ space. The data-structure at any node corresponds to test polygons associated with the diagonal edge labeling that node.

To detect a diagonal inside the polygon that the query segment intersect, the algorithm of CG begins with an initial point-location for one of the end-points of the segment. In our case this is very simple because of the monotonicity property. A straightforward binary search (on the diagonals) suffices. In their paper, CG used additional data-structures when the end-point lies outside of polygon. We can dispense with such additional structures because if the point is outside of the polygon, we simply 'walk' to the next intersection and charge the cost to the (deleted) vertices of the profile. In other words this is the easy case and we shall only focus on the case where the end-point is inside.

After this initial point location, we detect the diagonal that the line intersects when we travel along it in a given direction. Then we repeatedly use the procedure described previously to travel to the furthest diagonal by doing repeated point locations with respect to the test polygons associated with the present node. The procedure terminates when we reach a leaf from where we can detect the intersection in constant time (since it is a triangle). Each such traversal from one node to another involves a point location in a test polygon (in the dual plane), and the total number of such searches is bounded by the height of the tree i.e. $O(\log n)$.

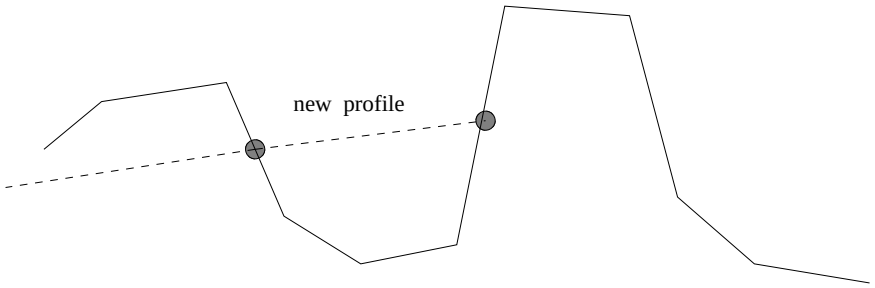


Figure 5: The dotted line indicates the latest segment which caused changes to the profile

Fact 5 *Testing containment of a point in a convex polygon can be done in $O(\log n)$ time given a preprocessing time of $O(n)$.*

The preprocessing can be absorbed in the preprocessing cost for building the data structure and thus we can state the following intermediate result of CG.

Lemma 4.1 *Given a simple polygon $\mathcal{P}$ with n vertices, there exists a $O(n \log n)$ space data structure that can be constructed in $O(n \log n)$ time which supports intersection detection between a line segment and $\mathcal{P}$ in $O(\log^2 n)$ time. The same procedure can be used iteratively to detect all k intersections in $O(k \log^2 n)$ operations.*

4.2 Making data-structure dynamic

Our objective is to modify the procedure described in the previous section to implement Step 2 of Algorithm **Visible**, namely, that of detecting intersections of a segment with the current profile. For that we shall now extend it to a dynamic environment, where not only do we detect the intersections but also modify the polygon by joining the intersections with a straight line (see Figure 5). As mentioned in section 2, the motivation behind this exercise is that the polygon in our case is the upper-profile which is changing during the course of the algorithm. The problem is primarily two-fold :

> (1) We need to quickly update the underlying data structure, (specifically the test polygons) as new segments are inserted.
> (2) We have to keep the underlying tree balanced, so that the depth of the tree remains logarithmic in number of leaves.

A suitable candidate for the underlying balanced tree is a class of weight balanced trees called the BB(α) tree (Mehlhorn [13]). Here α is a constant that determines the 'degree of disbalance' between the size of the left and the right subtree. For this paper, we do not have to explicitly derive the value of α. Appendix 1 gives a description of the general properties of these weight-balanced trees. We outline here some of the more important characteristics of this tree relevant to our needs:

(i) These trees have logarithmic height in the number of nodes.
(ii) The number of rotations for m updates (deletions or insertions) is $O(m)$ amortized over any such sequence of updates. Moreover the number of rotations decreases geometrically as we get closer to the root.

There are two distinct contexts when we refer to updates - one where it pertains to the changes in the actual profile by additions of new segments and the other where it pertains to the changes in the data-structure in terms of insertions, deletions and rotations in the underlying $BB(\alpha)$ tree. In the description that follows, we shall first trace the changes in terms of changes to the profile in terms of insertions, deletions and rotations and subsequently analyze the cost of implementing them. Recall that in the main algorithm, we pick up segments one at a time in a specific order and find out their interaction with the current profile. If it is visible then we make the necessary changes to the profile and update the data-structure.

In terms of modifications to the profile, we distinguish between the following two cases mainly to facilitate presentation.

- *Pure Insertion* Insertion of segments that introduce new diagonals but do not delete existing diagonals (see Figure 6a).
- *Insertions causing Deletions* Insertions of new segments that lead to deletion(s) of existing diagonal(s) (see Figure 7).

The first case turns out slightly simpler to implement. We will discuss the second case when exactly one diagonal is erased - repeated application of this analysis can be applied for the general case.

4.3 Pure insertions

We describe a procedure to insert a segments which does not delete existing diagonals. A vertex corresponding to the projection of one of the end-points of this segment is introduced which creates a diagonal. This is followed by introduction of the other endpoint and its associated diagonal. These are then 'lifted' to their original positions. See Figure 6b for an illustration. Each of these events causes changes in the data-structure - more specifically in the test polygons. The first two events (insertions of diagonals) clearly do not affect any visibility polygons as the profiles are not affected; however the diagonals introduced would cause new nodes to be introduced in the underlying $BB(\alpha)$ trees. This implies introduction of some additional shooting pointers and their associated test polygons (of constant size). However, to keep the tree balanced, the new nodes could cause rotations which turn out to be quite expensive. We shall describe the details of the rotation operation later.

The 'lifting' sub-step could modify all the test polygons that correspond to the regions of the polygon containing the diagonal. There is at most one such affected test polygon at each level of the tree - $O(\log n)$ in all. The maintenance of test polygons will be described in subsection 4.5.

The overall analysis shows that the total cost of pure insertions includes two insertions and updates of $O(\log n)$ test polygons. Note that there could be alternate (more direct) way of viewing the insertion procedure; we have chosen this to simplify analysis.

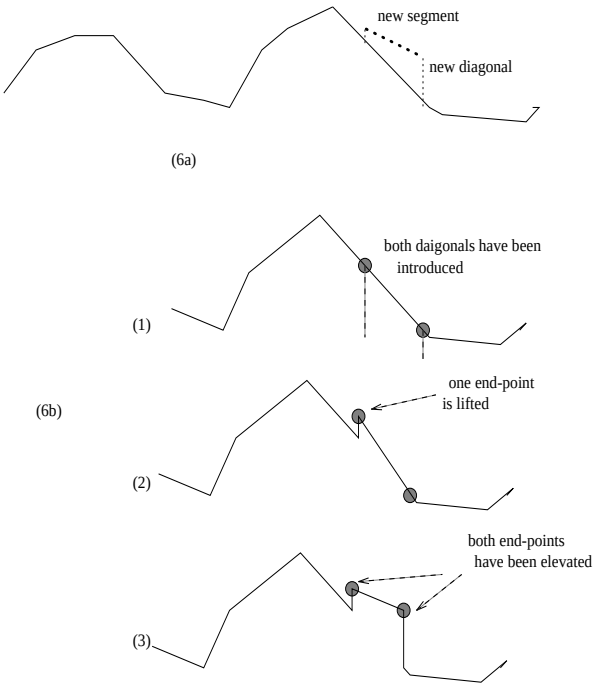


Figure 6: Pure insertion. 6b illustrates the (hypothetical) sequence in which the changes occur in the profile.

Remark An additional technical detail concerns the vertical segments in the profile. These segments can be handled using the following method. For test polygons that correspond to regions spanning the vertical segment, we use the lower end-point of the edge as the upper end-point of the associated diagonal. For test polygons that are incident on the diagonal, we use the upper end-point. When we shoot to that diagonal, a simple additional test can be used to determine if the line intersects the vertical segment in which case an intersection is identified.

4.4 Insertions causing deletions

Again we view this as being comprised of the following sub-steps

(i) Introduce the two diagonals corresponding to the endpoints of the new segment.
(ii) Lift the old diagonal to the appropriate height.
(iii) If necessary, lift the new diagonals to the appropriate levels.
(iv) Delete the old diagonal.

Figure 7b illustrates an example. Note that sub-steps (i) and (iv) may lead to rotations in the primary CG data-structure in order to keep it balanced. Proceeding similarly to that of the previous subsection, we conclude that this operation could involve two insertions, one deletion and updating of $O(\log n)$ test polygons. When more diagonals are deleted we can apply the above sequence repeatedly and charge the cost to the output vertices (that are being deleted in the process).

4.5 Visibility polygons

Before we discuss the implementations of insert, delete and rotation in the context of the underlying $BB(\alpha)$ trees, we need to adopt a representation of the test polygons. These representations must allow fast point-searches (as we traverse shooting pointers) as well as efficient updates. We use the data-structure of Overmars and Leeuwen [15] which supports fast point-searches in convex hulls with efficient deletion and insertion schemes. Recall that the test polygons are convex. Overmars and Van Leeuwen's scheme can be summarised as follows and we shall refer to this as the OL data-structure.

Fact 6 *One can maintain a set S of points in the plane at a cost of $O(\log^2 n)$ time per insertion and deletion, such that queries of the form 'does point p belong to the current convex hull of S' can be answered in $O(\log n)$ time. The same bounds also hold for maintaining intersection of half-planes under insertion and deletion of half-planes and point-location queries relative to the common intersection. If S is ordered, then the data-structure can be constructed in $O(|S|)$ time.*

The underlying data-structure of OL is a balanced tree where the leaves contain the existing points in a sorted order (or the lines in order of their slopes). The internal nodes contain the common tangent to the (disjoint) convex hulls of its left and right sub-trees. If the underlying tree is a weight-balanced tree, then the following additional operations can also be supported.

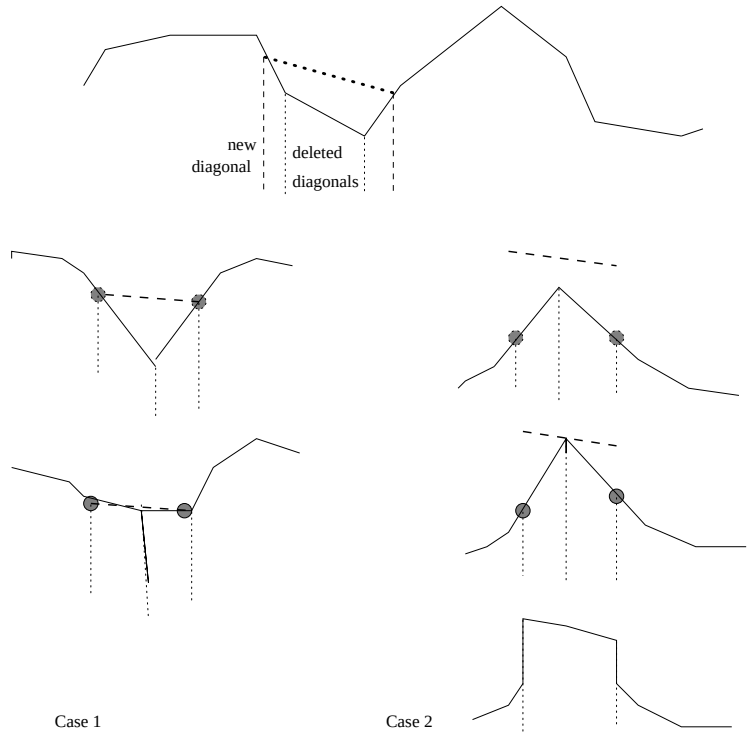


Figure 7: Insertion causing deletions. 7b illustrates the (hypothetical) sequence in which changes occur to the profile.

CONC $(S_1, S_2, S_3) : S_1 \leftarrow S_2 \cup S_3$
SPLIT $(a, S, S_1, S_2) : S_1 \leftarrow \{x : x \leq a \text{ and } x \in S\}$ and
$S_2 \leftarrow \{x : x > a \text{ and } x \in S\}$

In the context of dynamic maintenance of test polygons, these operations will be used for merging and splitting data-structures. In particular, if the vertices of S_1 precede those of S_2, then the **CONC** operation yields the data-structure for the union of the point sets. Similarly for the dual problem (that is then points are transformed via Duality transform), **CONC** yield the data structure for the intersection of the two sets of half-planes. These will be used for the implementation of the rotation operation.

Lemma 4.2 *The operations CONC and SPLIT can be implemented on the data-structure for dynamic convex hulls in $O(\log^2 n)$ operations where n is the total number of points in S.*

Proof: From Mehlhorn [13], these operations can be implemented on $BB(\alpha)$ trees using $O(\log n)$ rotations. In context of the OL data-structure, each rotation costs $O(\log n)$ operations, namely, the cost of computing a common tangent. Although the OL data-structure does not store explicitly the full convex hull at every node, one can verify that the necessary convex hulls can be reconstructed in the required bounds. 2

We now discuss the procedures for implementing the changes in the CG data-structure associated with the profile.

4.6 Insertion

When we insert a new node (as a leaf), we create two new shooting pointers, that is construct two constant size test polygon. Figure 8a illustrates that.

4.7 Deletion

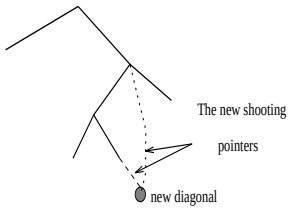
This is a relatively more difficult operation, especially when the deleted node is not a leaf. As we swap the closest leaf with the node being deleted, it may involve modification of $O(\log n)$ test polygons. See Figure 8b for an illustration.

4.8 Rotation

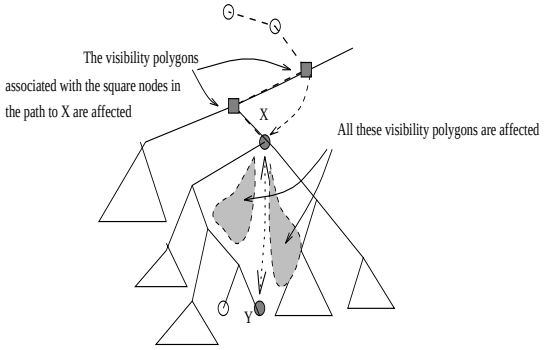
For rotations, a crucial subroutine involves intersection of convex polygons. The following well known result is stated for completeness.

Fact 7 *The intersection of two polygons (also a convex polygon) can be found in $O(m + n)$ time where the polygons have m and n vertices respectively.*

Lemma 4.3 *A rebalancing operation (i.e. a rotation or a double rotation) can be carried out in time $O(sz(v))$ where $sz(v)$ is the number of nodes in the subtree rooted at v (v is the vertex where rebalancing operation is being applied).*



(8a) Inserting a new diagonal



(8b) Deleting diagonal X by first swapping it with Y

Figure 8: Visibility pointers affected due to insertion and deletion of diagonals

Proof We shall prove it for a single rotation - the proof for double rotation is similar (applying it twice). Let us denote the left subtree of x as a and the subtrees of y as b and c. Figure 9 shows single and double rotations. This may have one of the following effects on the test polygons.

(i) We have to recompute the test polygon $P_{parent(x),y}$.
(ii) If there were a pointer from some ancestor of x to x (corresponding to some test polygon), we may have to compute the visibility polygon corresponding to y and this ancestor of x.

An important point to be noted is that the test polygons corresponding to the shooting-pointers incident on nodes in a, b and c are unaffected. Analyzing the effects of (i) more carefully, we have to compute the intersection of two convex polygons, namely that of $P_{parent(x),x}$ and $P_{x,y}$. The size of each polygon is bounded by $O(sz(x))$. From Fact 7, this can be computed in the same asymptotic time bounds. For (ii), we have to compute intersection of $P_{ancestor(x),x}$ and $P_{x,y}$ where the size of $P_{ancestor(x),x}$ is bound by the size of the sub-polygon at x which is $O(sz(x))$

Note that we also need to compute the data-structure for the intersection of convex polygons, that is the OL data-structure. Since the convex hull is already ordered (in terms of slopes of the half-planes), from Fact 6 the data-structure can be constructed in $O(sz(x))$ steps. 2

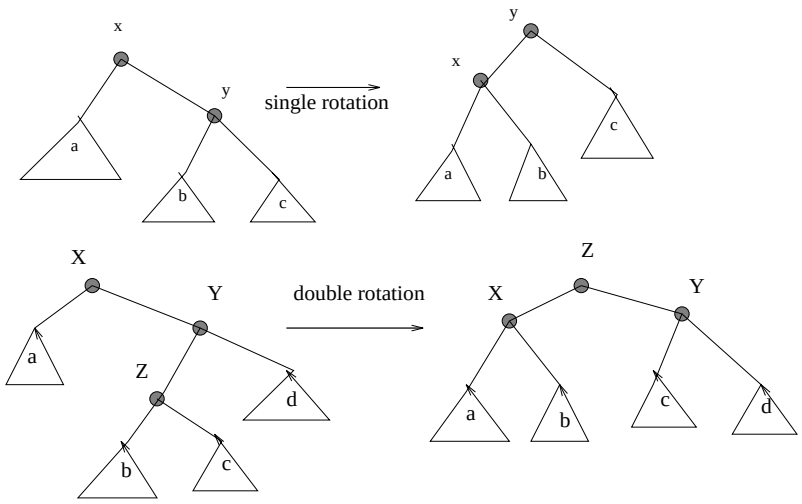


Figure 9: Rotation and double-rotations on BB(α) trees.

Fact 8 *Let* $1/4 \leq \alpha \ 1-\sqrt{2}/2$ *and let* f *be a non-decreasing function, then the total amortized rebalancing cost of* m *insertions and deletions BB(*α*) tree can be bound by* $O(m\sum_{i=0}^{elogn} f((1-\alpha)^{-i})(1-\alpha)^{i})$ *where* $e = 1/log(1/1-\alpha)$

Remark : If $f(m) = O(m(\log^k m))$ then the total cost $= O(m\log^{(k+1)} m)$.

Lemma 4.4 *The total rebalancing cost corresponding to updates of the upper profile is* $O((k+n)\log n)$ *where* k *is the number of insertions and* n *is the input size.*

Proof Since each update can be charged either to the input segment or some output vertex, the lemma follows from Lemma 4.3 and Fact 8 (use $k = 0$ in the previous remark). 2

4.9 The final analysis of Algorithm visible

Stage 1 of the algorithm Visible requires $O(n\log n)$ time (from Fact 1). Stage 2 of the algorithm is dominated by the time to update $O(\log n)$ test polygons for insertions (subsection 4.3 and 4.4). From Fact 6, this can be done in $O(\log^3 n)$ time. Consequently for output size k, the total time (not including rotations) is $O((k+n)\log^3 n)$. Compared to this, the total rebalancing cost is only $O(k\log n)$ from Lemma 4.4. The bound for space follows directly from Lemma 2.1.

This gives us a result slightly inferior to that of Theorem 1, namely, by a factor of $O(\log n)$. To overcome this, we make use of a simple observation. The $O(\log n)$ test (convex) polygons that we have to update have common portions - in fact, those portions correspond to the hierarchy of the convex polygons that get modified in the OL data-structure. More precisely, let the root of the OL data-structure represent the test polygon $P_{l,r}$ where l and r are the extreme diagonals. Then these $O(\log n)$ test polygons are the ones that are affected due to a single update in the OL data-structure, namely, the polygons in the path taken by the point the OL data structure. We do

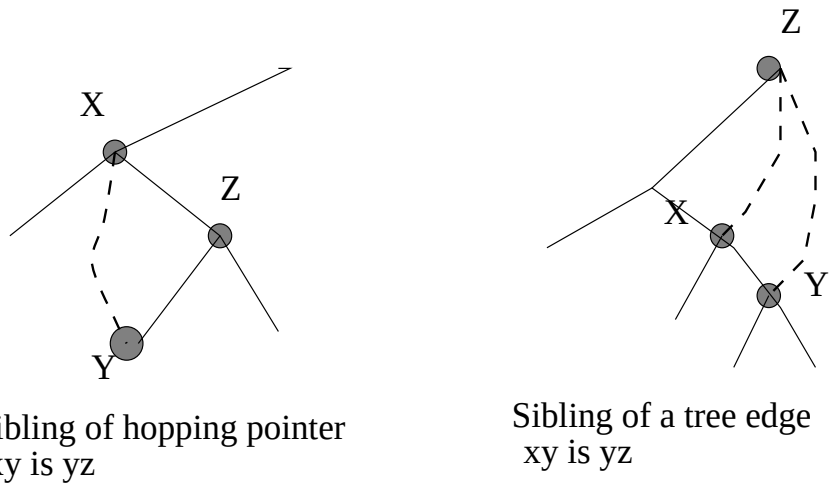


Figure 10: Mapping of OL data-structure into CG primary tree structure.

not require the entire hierarchical structure of OL with every edge in the primary data-structure; rather, we need only the search-tree corresponding to the test polygon for point-location. Recall that in OL data-structure a node stores only the part of the hull that is not common with its parent and we can reconstruct the required hull from its ancestor at $O(\log n)$ extra steps. This way, during updates, only a total of $O(\log n)$ common-tangents have to be recomputed.

To take advantage of this, we embed the OL data-structure in the shooting pointers of the CG primary data-structure. Each shooting-pointer is associated with a convex-hull of the OL data-structure; more specifically a node of the primary data-structure of OL. In the OL data-structure only a part of the hull (of the subtree) is stored at a node and the entire hull at the node can be assembled by walking from the root to that node and performing SPLIT/CONC operations. Hence, in order to do intersection queries with the CG data-structure (storing the profile), we also need to reconstruct the full hull associated with the shooting pointer. The two children of a visibility pointer (x, y) in CG primary structure are $y, rchild(y)$ and $x, rchild(y)$ when y is the left child of x. For the case y is a right child of x use $lchild(y)$ instead.

Let us now define the sibling of a shooting-pointer (when it exists). If a shooting pointer connects node x to node y, then its sibling is the tree-edge connecting y to y's parent. The sibling of a tree-edge (of CG primary data-structure) connecting x to y is the shooting pointer connecting y to its ancestor (see Figure 10). In the description that follows we will not distinguish between tree-edges and shooting pointers and uniformly refer to them as *visibility-pointers*.

Assume that we already have the convex hull of the pointer we we are currently trying to traverse (that is do a point location) and denote as *sib* its sibling. Let us denote the shooting pointers in the present node v as $h_v(i)$, $i = 1, 2...k$ where $h_v(i)$ goes to a node which is at a level $level(v) + i$. In the corresponding OL data-structure, $h_v(i)$ is the parent of $h_v(i+1)$. So, as we try to find the next node that we can shoot, we can easily construct the corresponding hulls at $h_v(i)$ and update *sib*. Let j be the smallest i, such that $h_v(j)$ (that is the hull associated) contains the

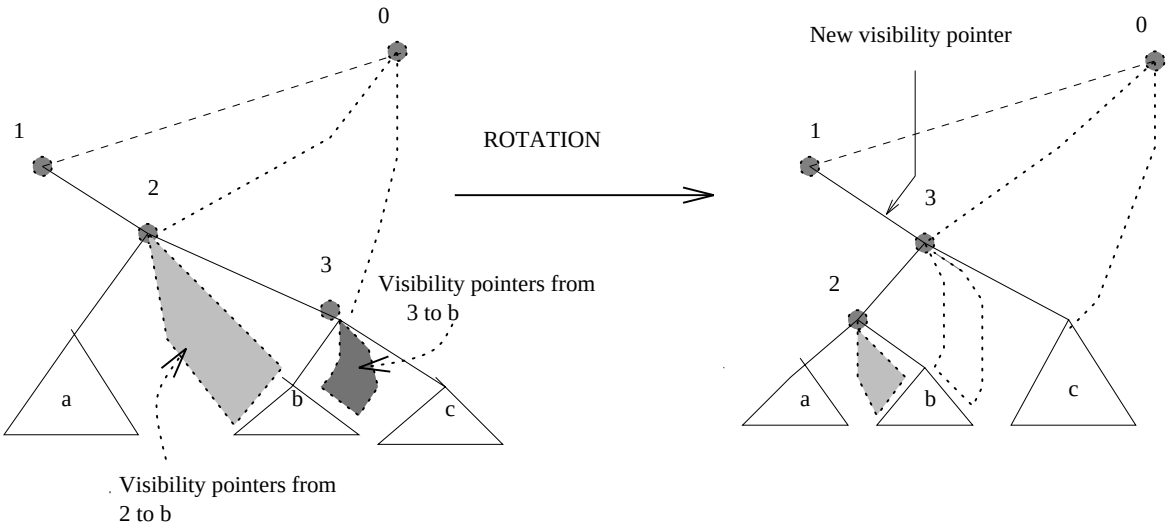


Figure 11: How single rotations affect visibility pointers.

point (dual of the line) and takes us to a node w. Then at $h_w(1)$, the next shooting pointer in our search, the associated hull can be constructed from $h_v(j)$'s sibling. Geometrically the (the test polygon associated with) sibling of a pointer bounds the diagonal in the profile before which the intersection occurs. The entire procedure can now be repeated from w. Thus the search for the next intersection takes $O(\log^2 n)$ time.

During insertion and deletion of diagonals, a total of $O(\log n)$ test polygons are modified - one in each level of the CG data-structure. This is similar to updating the OL data-structure where tangents (intersections) are recomputed at each level of the tree. By the above scheme of associating pointers with test polygons, the data-structure can be modified in one sweep from leaf to root of the CG structure and would take $O(\log^2 n)$ time.

A rotation in the primary structure of CG affects one pointer (see Figure 11 for one case; the remaining being similar). So this modification can be made in one sweep down the primary tree which takes $O(\log^2 n)$ steps. A double rotation costs the same in asymptotic terms.

From the above discussion the total cost of the (modified) algorithm is $O((n+k)\log^2 n)$ steps. Since the size of the data-structure is no more than the current size of the profile in the modified scheme, the main theorem follows. 2

Chazelle and Guibas [3] improve on the running time of their algorithm from the bounds stated in Lemma 4.1, by a clever application of *fractional cascading*, which improve their running time by a factor of $O(\log n)$. The reason for this being the ability to search the test polygons in various levels given its position at some level in constant additional time. The cost for augmenting the data-structure to facilitate *fractional cascading* can be absorbed in the preprocessing cost. Our problem is more formidable since we have a dynamic environment and it is not clear how fractional cascading can be applied in our case.

5 Concluding remarks

In this paper, we presented an efficient sequential algorithm for hidden surface elimination for terrain maps. The running time of the sequential algorithm is proportional to the size of the output image, thus achieving our primary objective. However, the performance of our algorithm is not optimal in the worst case; in fact it is not clear what is the optimal running time for such a class of algorithms (which depends on the output size). As mentioned in the postscript of the introduction, the algorithm of Katz et al. [11] attains a superior running time which is closer to the best possible complexity of $\Omega(n \log n + k)$.

A natural direction for further work is to generalize the algorithm for hidden-surface elimination for any collection of objects. For that, we need efficient algorithms for partitioning the scene into disjoint parts such that an ordering the edges is feasible within each part. Note that such a partitioning scheme is also required to be output-sensitive.

In a companion paper [19], we will present a parallelization of our algorithm which runs in polylogarithmic parallel time using an output sensitive number of processors.

References

[1] M. Bern. Hidden surface removal for rectangles. *Proc. 4th ACM Symp. on Computational Geometry*, 183 – 192, 1988.

[2] B. Chazelle. A theorem on polygon cutting with applications. *Proc. of IEEE FOCS*, 339 – 349, 1982.

[3] B. Chazelle and L. Guibas. Visibility and intersection problems in plane geometry. *Proc. ACM Symp.on Computational geometry*, 135 – 146, 1985.

[4] R. Cole and M. Sharir. Visibility problems for polyhedral terrains. *Tech. Rept. No. 92, Courant Institute of Math. Sc.*, 1986.

[5] M. de Berg D. Halpern M. Overmars J. Snoeyink and M. Krevald. Efficient ray-shooting and hidden surface removal. *Proc. 7th ACM Symp. on Computational Geometry*, 21–30, 1991.

[6] F. Devai. Quadratic bounds for hidden line elimination. *Proc. 2nd Annual Symp. on Computational Geometry*, 269–275, 1986.

[7] M.T. Goodrich. A polygonal approach to hidden-line elimination. *GVGIP:graphical models and image processing*, 54(1), 1–12, 1992.

[8] R.H. Guting and T. Ottmann. New algorithms for special cases of the hidden-line elimination problem. *Unversitat Karlruhe, Institut fur angewandte informatik und formale beschreibunsverfahren, Report 184*, 1984.

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

[10] D.T. Lee and F.P. Preparata. Location of a point in a planar subdivision and its applications. *SIAM Journal of Comput.*, 6(3):594–606, 1977.

[11] M. Overmars M. Katz and M. Sharir. Efficient hidden surface removal for objects with small union size. *Computational Geometry: Theory and Applications*, 2:223–234, 1992.

[12] M. McKenna. Worst-case optimal hidden-surface removal. *ACM Transactions on Graphics*, 19 – 28, 1987.

[13] K. Mehlhorn. *Data Structures and Algorithms.* Volume Vol. 1: Sorting and Searching, Vol. 3: Multidimensional Searching and Computational Geometry, Springer Publishing Company, 1984.

[14] O. Nurmi. A fast line-sweep algorithm for hidden line elimination. *BIT*, 25:466 – 472, 1985.

[15] M. Overmars and J. Leeuwen. Maintainence of configurations in space. *Journal of Computer and System Sciences*, 166–204, 1981.

[16] M. Overmars and M. Sharir. Output-sensitive hidden surface removal. *Proc. 30th IEEE Symp. on Foundations of Computer Science*, 598 – 603, 1989.

[17] F. Preparata and J. Vitter. A simplified technique for hidden-line elimination in terrains. *Proc. of STACS*, 1992.

[18] J. Reif and S. Sen. An efficient output-sensitive hidden-surface algorithm and its parallelization. *Proc. of the 4th ACM Symp. on Computational Geometry*, 193–200, 1988.

[19] J. Reif and S. Sen. An efficient output-sensitive parallel algorithm for hidden-surface removal. *unpublished manuscript.*

[20] A. Schmitt. Time and space bounds for hidden line and hidden surface elimination algorithms. *EUROGRAPHICS*, 43 – 56, 1981.

[21] T.J. Wright. A two-space solution to the hidden line problem for plotting functions of two variables. *IEEE Trans. on Comput.*, vol. c-32, no. 1:28 – 33, 1972.

6 Appendix

BB(α) trees are a class of weight-balanced trees i.e the number of nodes in the subtrees are balanced. If T is a binary tree on n nodes with left subtree T_l and right subtree T_r and α a fixed real in the range [1/4, 1- $\sqrt{2}/2$], then T is of bounded balance α if for every subtree $\bar{T}$ of T
$\alpha \leq \rho(\bar{T}) \quad \leq \quad 1 \; - \; \alpha$ where $\rho(T) = \mid T_l \mid = 1$ - $|T_r|/|\mathrm{T}|$.
BB(α) trees have the following properties:
(i) they have logarithmic depth: Height(T) $\leq 1 + (\log(n+1) - 1)/\log(1/1-\alpha))$
(ii) they have logarithmic average path length

Fact 9 (Mehlhorn [13]) *: For all $\alpha \in$ (1/4, 1 - $\sqrt{2}$/2] there are constants $d \in [\alpha, 1 - \alpha]$ and $\delta \geq 0$ such that for T, a binary tree with subtrees T_l and T_r and*
(1) T_l and T_r are in $BB(\alpha)$
(2) $|T_l|/|T| < \alpha$ and either

2.1 $|T_l|/(|T| - 1) \geq \alpha$ implying that an insertion into T_r occurred or
2.2 $(|T_l|+1)/(|T|+ 1) \geq \alpha$ implying that a deletion from left subtree had taken place.
2.3 ρ_2 is the root balance of T_r

then (i) if $\rho_2 \leq d$ then a rotation rebalances the tree
(ii) if $\rho_2 > d$ then a double rotation rebalances the tree.

Figure 9 shows the rotation and double rotation operations and the corresponding changes in the root balances. Note that d and δ are functions of α.

Fact 10 (Mehlhorn [13]) *There is a constant c such that the total number of rotations and double rotations required to process an arbitrary sequence of m insertions and deletions into an initially empty BB(α) tree is $< cm$ and the total number of rebalancing operations over all the vertices of level i $BO_i(v) = O(m\ (1-\alpha)^i)$.*

(Note that i increases as we go up the tree which implies that rebalancing operations are very rare as we get closer to the root.)