Computation of the Axial View of a Set of Isothetic Parallelepipeds

FRANCO P. PREPARATA
University of Illinois
JEFFREY SCOTT VITTER
Brown University
and
MARIETTE YVINEC
LIENS, URA CNRS 1327, École Normale Supérieure, Paris

We present a new technique to display a scene of three-dimensional isothetic parallelepipeds (3D-rectangles), viewed from infinity along one of the coordinate axes (axial view). In this situation, there always exists a topological sorting of the 3D-rectangles based on the relation of occlusion (a dominance relation). The arising total order is used to generate the axial view, where the two-dimensional view of each 3D-rectangle is incrementally added, starting from the closest 3D-rectangle. The proposed *scene-sensitive* algorithm runs in time $O(N \log^2 N + d \log N)$, where N is the number of 3D-rectangles and d is the number of edges of the display. This improves over the previously best known technique based on the same approach.

Categories and Subject Descriptors: I.3.5 **[Computer Graphics]**: Computational Geometry and Object Modeling—*curve, surface, solid, and object representations*; I.3.7 **[Computer Graphics]**: Three-Dimensional Graphics and Realism

General Terms: Algorithms, Design

Additional Key Words and Phrases: Amortized analysis, axial view, computational geometry, contracted binary trees, hidden line elimination, scene sensitive, segment trees

1. INTRODUCTION

Generating two-dimensional display of a three-dimensional scene is one of the central problems in computer graphics. It has obvious applications in picture processing, animation, flight simulation, and so on (see [11]). The objective is to

The work of F. P. Preparata was supported by NSF grant CCR-8906469. The work of J. S. Vitter was supported in part by NSF research grant DCR-8403613, NSF Presidential Young Investigator Award CCR-8846714, and a Guggenheim Fellowship.

Authors' current addresses: F. P. Preparata, Coordinated Science Laboratory, University of Illinois, 1101 West Springfield Avenue, Urbana, IL 61801; J. S. Vitter, Department of Computer Science, Brown University, Box 1910, Providence, RI 02912; M. Yvinec, LIENS, URA CNRS 1327, École Normale Supérieure, 45 rue d'Ulm, 75230 Paris, France.

produce a display acceptable to a human user, that is, a (possibly) perspective view of the scene where the hidden portions of the objects have been eliminated.

Hidden-line elimination is thus a fundamental component of any solution to the above-described display task. It is also desirable to produce a solution in terms of the *object space* [20], that is, of the geometric descriptions of the scene objects, which is therefore independent of the particular rendering device. Theoretical investigations of this problem have recently intensified ([2, 4, 5, 7–10, 12, 14, 16, 18–20, 22]), and can be categorized, respectively, according to two eminently distinct approaches, referred to here as "intersection-sensitive" and "scene-sensitive." The two approaches can be contrasted as follows. Let A and B be two objects in three-dimensional space, and let A' and B' be their respective displays. The *intersection-sensitive* approach computes the intersection of these displays A' and B', determines which portions are hidden from the observer and erases them, and finally creates the scene by forming the union of the modified displays. The *scene-sensitive* approach requires the knowledge of the relative order of the two objects with respect to the observer; the scene is created by first displaying the closer object and then intersecting the display of the farther object with it, without visiting the occluded portions.

Notice that the scene, as a planarly-embedded planar graph, is always a subgraph of the union of the displays of the objects; therefore, an efficient scene-sensitive approach could be substantially more efficient than any intersection-sensitive approach, in most cases.

Of course, scene-sensitive techniques are based on an ordering of the objects of the scene according to a relation of occlusion, called *dominance* in [9] and hereafter. If object A occludes (a portion or all of) object B for the observer, then A is said to dominate B. Since objects are processed according to this ordering, such methodology is also known as the *priority* approach to hidden-line elimination. It was pioneered in [7] and analyzed for its mathematical structure in [22].

In general, even the existence of such an ordering for a given set of objects and an arbitrary viewpoint is not known. With the restriction to convex objects, Guibas and Yao [9] have shown that in two dimensions that such an ordering always exists for a given direction (that is, a viewpoint at infinity). On the other hand, this property does not hold in three dimensions; they exhibit a three-object configuration for which the relation of *dominance* is not acyclic. To apply this approach, it may be therefore necessary to split some objects of the collection in order to force acyclicity in the ordering relation.

In this paper, the scene considered consists of N isothetic parallelepipeds (called 3D-rectangles hereafter), and our goal is the construction of their *axial view*, that is, the view from infinity along a coordinate axis. In such a view, the display of each individual 3D-rectangle is an isothetic 2D-rectangle and, therefore, the dominance relation is always acyclic. Thus, our problem is analogous to the problem of displaying a sequence of overlapping "windows," as in a window-based user interface or graphics system, where the windows are two-dimensional isothetic rectangles parallel to the x, y-plane, but with different z-coordinates. In this case the parallelepipeds have zero depth, so that the dominance relation on the set of windows is trivially acyclic.

A scene-sensitive approach to this problem was recently proposed by Güting and Ottmann [7]. They showed that the display of N parallelepipeds can be constructed in time $O((N + d)\log^2 N)$, where d is the complexity of the scene (rather than the complexity of the underlying intersection problem). Güting and Ottmann make a judicious and clever use of data structures, such as segment trees and range trees (see, e.g., [15]). However, these powerful data structures are used in their "general-purpose" ability to handle arbitrary collections of segments and points. By exploiting the particular natures of such sets and of the problem, we show in this paper that the axial view of N isothetic parallelepipeds can be obtained in time $O(N \log^2 N + d \log N)$, thereby significantly improving the Güting–Ottmann result. The algorithm uses a new data structure, called the *contracted binary tree* or CBT for short. The CBT can be thought of as a kind of semidynamic finger tree, designed to maintain a dynamic list of items drawn from a finite totally ordered set known in advance. The whole algorithm is easy to implement and fast in practice; the complexity lies in its analysis, not in its implementation.

Another algorithm solving the same problem was recently proposed independently by M. Bern [1]. This algorithm uses a totally different approach, also based on space sweeps and the use of dynamic data structures. But whereas our method involves a *space* sweep orthogonal to the display plane, Bern's approach uses a *plane* sweep of the display plane itself. In its basic version, Bern's method achieves the same time bound $O(N \log^2 N + d \log N)$ as our solution. Its running time can be reduced to $O(N \log N \log \log N + d \log N)$ using the dynamic fractional cascading technique of Mehlhorn and Näher [6].

2. THE LINE DRAWING ALGORITHM

Let us denote by N the cardinality of a set $\mathscr{R}$ of 3D-rectangles. We wish to display the axial view of this set on the screen, bearing in mind that each parallelepiped is opaque and that hidden parts are not to be displayed. Our algorithm runs in $O(N \log^2 N + d \log N)$ time, where d is the complexity of the final display.

In the axial view, the point of view is the point at infinity of one of the coordinate axes. In this case, each 3D-rectangle is displayed as a 2D-rectangle (isothetic). An example of such type view is shown in Figure 1.

The approach is analogous to that of Güting–Ottmann [7], but it significantly departs from it in the implementation of the data structures. The line-drawing algorithm processes one rectangle at a time, closest rectangle first. Initially, the display is empty and the first rectangle is trivially added. In the general step, the display plane is partitioned into two portions, the *opaque* portion $\mathscr{F}$, consisting of the union of the heretofore processed rectangles, and its complement $\overline{\mathscr{F}}$, the *transparent* portion. If r denotes the 2D-rectangle to be currently processed, we must add to the display the visible parts of r (namely, $r \cap \overline{\mathscr{F}}$) and update the opaque portion to be $r \cup \mathscr{F}$.

A crucial role is played by the boundary of the opaque portion $\mathscr{F}$, referred to as the *silhouette* (of $\mathscr{F}$). The silhouette is represented as a collection of directed polygons, as shown in Figure 2. The orientation on each polygon is chosen so

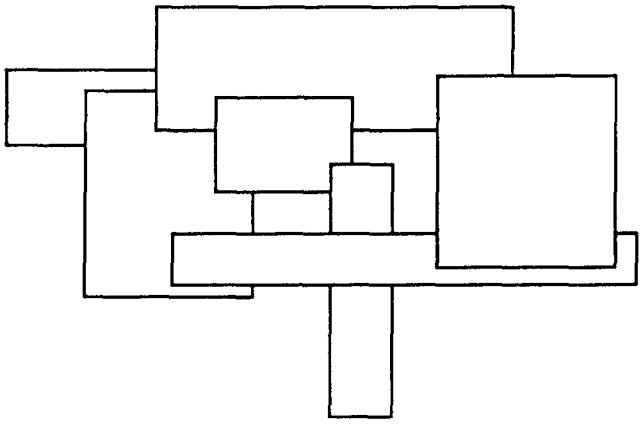

Fig. 1. Axial view of a set of parallelepipeds (a set of rectangles).

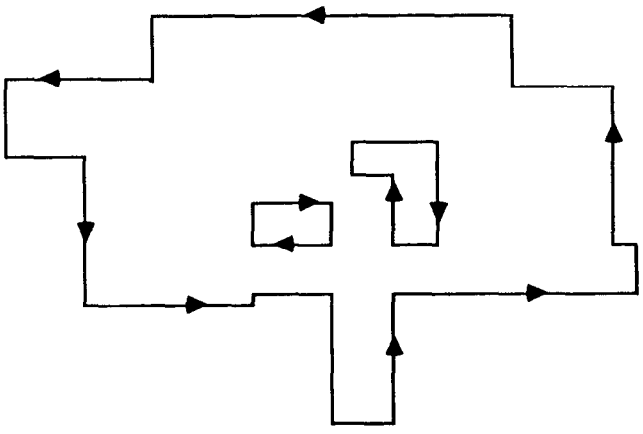

Fig. 2. An oriented silhouette.

that the interior of $\mathcal{F}$ lies to the left of each edge. The silhouette is conveniently decomposed into the *external* silhouette, consisting of the set of the anticlockwise directed polygons of the silhouette, and the *internal* silhouette, the relative complement of the external silhouette.

2.1 The Data Structure

To obtain an efficient implementation of the construction of $r \cap \overline{\mathcal{F}}$ and of $r \cup \mathcal{F}$, special attention must be given to the data structure. Since we are dealing with isothetic rectangles, the use of segment trees—and related constructs—is natural.[1] Our main data structure, which is designed to store the edges of the

[1] Point trees and segment trees are used throughout this paper, and a brief view of their structure and mechanisms is given in Appendix A.

current silhouette, is still referred to as a *segment* tree for simplicity, although it is moderately more complex than the conventional one; we use two such structures, $\mathcal{T}_x$ for the horizontal segments and $\mathcal{T}_y$ for the vertical segments. Since the utilizations of $\mathcal{T}_x$ and $\mathcal{T}_y$ are identical, we consider here just $\mathcal{T}_x$. Here and in the following, each rectangle r of the set $\mathcal{R}$ to be displayed is specified by the coordinates $(x_l(r), y_b(r))$ of its lower-left corner and the coordinates $(x_r(r), y_t(r))$ of its upper-right corner. Furthermore, let X be the set $\{x_l(r), x_r(r): r \in \mathcal{R}\}$ and Y be the set $\{y_b(r), y_t(r): r \in \mathcal{R}\}$. For the sake of simplicity of presentation and with no loss of generality, we assume that the rectangles are in general position, which means that the coordinates of sets X and Y are all distinct. The degenerate cases can be handled easily.

Segment tree $\mathcal{T}_x$ has as its primary skeletal structure a segment tree built on the set X. All (horizontal) segments recorded at a given node V in $\mathcal{T}_x$ are stored—ordered by increasing ordinate—in a secondary data structure $\mathcal{L}(V)$, which is a dictionary realized as a *contracted binary tree*, described below. (The use of the contracted binary tree instead of a height-balanced tree is the essential difference between our scheme and that of Overmars [13]. Indeed, whereas the conventional height-balanced tree has the capability of handling an arbitrary set of real-valued coordinates, the contracted binary tree capitalizes on the fact that the set of coordinates is entirely known beforehand and can be standardized.)

A *contracted binary tree* (CBT for short) is a tree structure specially designed to maintain a dynamic list of elements that are known to belong to a finite totally ordered set, called the *universe*, which is given a priori. Anticipating the particular use we shall make of such a data structure, we let Y denote the universe and y its generic element. We also assume without loss of generality that the cardinality of Y is $2N$, where N is a power of 2. In a simple $O(N \log N)$-time pass, we can map each element in Y to its rank (an integer between 0 and $2N - 1$) in the totally ordered set Y; so, from now on, we assume that Y is the set $\{0, 1, \ldots, 2N - 1\}$. The following lemma summarizes the properties of the CBT:

LEMMA 1. *The* CBT *uses space proportional to the size n of the list it stores and does not require any rebalancing. It can support the operations of* $MEMBER(y)$ *and* $INSERT(y)$ *in* $O(\log N)$ *time, where N is the size of the universe, and if we are given a pointer to y's leaf node we can perform* $DELETE(y)$ *and can find y's predecessor and successor in the* CBT *in constant time. In addition,* $INSERT(y)$ *can be done in constant time if we are given a neighboring element of y in the* CBT *or if we are given y's companion nodes, which we define below.*

The CBT is best described in terms of its underlying *shadow structure*, which is not actually implemented, but is useful to consider for the purposes of exposition (see Figure 3). The shadow structure is a complete (balanced) binary tree with $2N$ nodes. The leaves, ordered from left to right in the usual way, are mapped to the key elements $\{0, 1, \ldots, 2N - 1\}$ of the universe Y. Each leaf has a label which is the binary representation of the corresponding key. Each internal node is labelled with the common prefix of all its descendants (the root being labelled with the empty word). Each leaf is in one of two states: *occupied* if the corresponding key belongs to the list implemented by the CBT; *unoccupied*

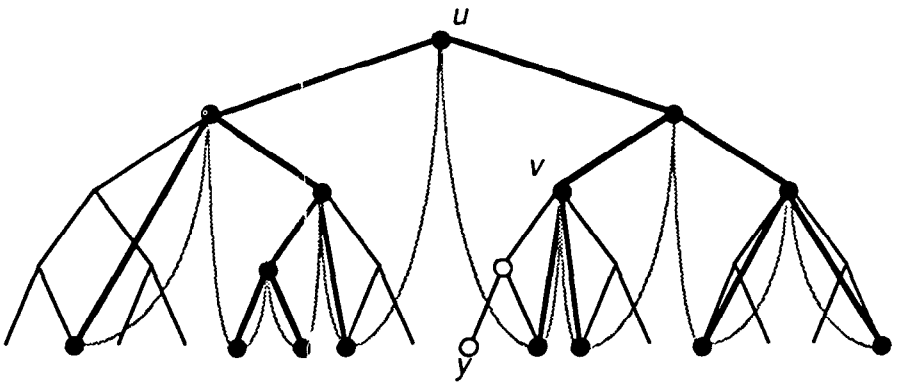

Fig. 3. A contracted binary tree and its underlying shadow structure. The active nodes are represented by black circles. The *PARENT*, *RIGHT*, and *LEFT* links are shown as heavy lines, and the *NEXT* and *PREV* pointers are shown as dashed lines. The companion nodes v and u for the insertion of the new leaf y are also shown.

otherwise. The nodes of the shadow structure present in the CBT are called *active*. Active nodes are defined recursively as follows:

(1) The root is (always) active.
(2) A leaf is active if occupied.
(3) An internal node is active if both of its subtrees contain active nodes.

Let us denote by $|v|$ the label of node v. The extension $|v|^*$ of a label $|v|$ is the integer obtained by appending a 0 and a run of 1s to the right of $|v|$ to obtain a binary string of the same length as the labels of the leaves; we also use the notation $v_1 \leq^* v_2$ to denote $|v_1|^* \leq |v_2|^*$.

Now we describe the CBT formed on the set of active nodes. If there is only one active leaf, we are done, so let us consider the case when there are at least two active leaves. The parent of a node v in the CBT is its lowest active ancestor in the shadow structure. For each node, child-to-parent pointers are bidirectional and are denoted by LEFT, RIGHT, and PARENT. Note that if S_1 and S_2 are two subsets of the universe with $S_1 \subseteq S_2$, then each node of the CBT for S_1 appears in the CBT for S_2. Each node also has two additional pointers, called *NEXT* and *PREV*, which point to its successor and predecessor in an in-order traversal of the CBT. For the first and last nodes encountered in the in-order traversal, *PREV* and *NEXT* are, respectively, set to **nil**. (Some space can, however, be saved: the *NEXT* pointer of a node is not needed when the node is an active leaf and a left child in the CBT, and similarly *PREV* is not needed when the node is an active leaf and a right child.) It is immediate that, using the pointers *NEXT* and *PREV*, an ordered traversal of the leaves takes constant time per leaf visited. For reasons that will become clear soon, we must also keep track of the smallest and largest members of Y currently in the CBT, which we denote by *smallest* and *largest*; *smallest* and *largest* are both set to **nil** when no leaf is active.

Since the nodes in a root-to-leaf path in the CBT are a subset of the nodes in an analogous path in the shadow binary tree, it follows that the depth of the CBT is at most $\log N + 1$. Let y denote an integer in the range $[0, 2N)$. The operation *MEMBER*(y) consists of tracing a path in the CBT from the root to a leaf. The labelling at each node along the path is used to guide the search, at a constant time per traversed node.

We now consider the problem of inserting an element y into the CBT. By tracing a path from the root we identify the *companion nodes* of y, which we call v and u. Node v is defined as the CBT node corresponding to the lowest ancestor of y in the shadow structure that is active in the CBT before the insertion of y. If y is to the left of v, then u is defined to be v's lowest ancestor in the CBT such that v is in u's right subtree; similarly, if y is to the right of v, then u is defined to be v's lowest ancestor in the CBT such that v is in u's left subtree. The important point is that, given v and u, we can determine in constant time the smallest and largest leaves in v's subtree each time they are needed to update the *NEXT* and *PREV* fields. Note that, since the root is always active, v is always defined, while u may not exist; in such case u is set to **nil**. The algorithm *INSERT* given in Figure 4 inserts y in constant time into a nonempty CBT, once its companion nodes v and u have been located. (We make the reasonable assumption that the common prefix of the labels of two nodes can be computed in constant time; formal justification of this appears in [3].)

We can also insert y in constant time if we are given an element y' in Y that immediately precedes or follows y. We can use the companion-node-based insertion algorithm as a subroutine. Given y', we can find y's companion v in constant time by following a *NEXT* or *PREV* pointer. In the cases in the above algorithm where the second companion node u is also needed, we can also find u easily in constant time. (Note that it is not always possible to find u in constant time, but u is only needed in those cases where it is easy to compute.) The deletion of a leaf y in the CBT is an even simpler operation, also executable in constant time if a pointer to y's leaf is given. The algorithms for these two operations are given in Appendix B.

We now return to the data structure describing the edges of the current silhouette. To each node V of the primary segment tree $\mathcal{T}_x$ we append a CBT, devoted to store, as an ordered list, the set $Y(V)$ of ordinates of horizontal segments recorded at node V. We denote the CBT appended to node V by $\mathcal{L}(V)$. Clearly, set $Y(V)$ is a subset of the set of coordinates $Y = \{y_b(r), y_t(r): r \in \mathcal{R}\}$ and $|Y(V)| \leq 2N$.

A (horizontal) segment $s = [x_1, y; x_2, y)$[2] is inserted into the segment tree $\mathcal{T}_x$ by creating an entry for s in the CBT $\mathcal{L}(V)$ of each node V of $\mathcal{T}_x$ allocated for s. In each of those CBTs, the entry for s is a leaf labelled by the binary representation $|y|$ of y. The leaf also includes three pointers: two of the pointers are used to link the successive fragments of s in the segment tree $\mathcal{T}_x$ into a doubly-linked list, and the third one points to an entry corresponding to the

[2] Consistent with our definition of the proper intervals of segment trees, we use $[p_1; p_2)$ or, more explicitly, $[x_1, y; x_2, y)$ to denote the segment s whose left and right endpoints are, respectively, the points p_1 with coordinates (x_1, y) and p_2 with coordinates (x_2, y).

```
algorithm INSERT(|y|, v, u, root);
{ This procedure inserts a new element into a CBT, given its companion nodes.
The variable root is a pointer to the root of the CBT,
|y| is the binary label of the new leaf to be inserted,
u and v are pointers to the companion nodes,
y is a pointer to the new leaf,
w is a pointer to the new internal node created by the insertion, and
PREFIX is a function that returns the common prefix of the labels of its arguments.
The "linear order" alluded to below refers the order induced by the pointer NEXT. }
begin
create a leaf y;
LABEL(y) := |y|;
if y ≤* v then
   if LEFT(v) = nil then begin { v coincides with the root }
      make y the left child v;
      establish the linear order nil, y, v;
      smallest := y;
      if RIGHT(v) = nil then largest := y { y is the only active leaf }
      end
   else begin
      create an internal node w;
      LABEL(w) := PREFIX(y, LEFT(v));
      if y ≤* w then begin
         y becomes the left child of w;
         the left child of v becomes the right child of w;
         w becomes the left child of v;
         if u = nil then begin
            establish the linear order y, w, smallest;
            smallest := y
            end
         else insert y and w in the linear order u, y, w, NEXT(u)
         end
      else symmetrically
      end
else symmetrically
end
```

Fig. 4. Algorithm *INSERT*.

segment s in an edge list $\mathscr{B}$. The edge list $\mathscr{B}$ is a list of edges of the current silhouette in which we maintain, for each segment s, the coordinates (x_1, y) and (x_2, y) of its endpoints. In the *conventional insertion* operation, each of the log N fragments of a segment s is inserted into the appropriate CBT in $O(\log N)$ time using the standard algorithm, which leads to a total cost of $O(\log^2 N)$ time per segment insertion.

We now introduce an alternative way of inserting a segment into the segment tree $\mathscr{T}_x$, called *guided insertion*. Specifically, we say that *s is inserted into $\mathscr{T}_x$ using s' as a guide*, when $\mathscr{T}_x$ already contains a segment s', so that (horizontal) segments s and s' span exactly the same horizontal interval and their ordinates are adjacent in all their allocation nodes. Since all fragments of s' are linked in $\mathscr{T}_x$, we can insert s in $O(\log N)$ time as follows. First, we locate the leftmost fragment of s' in $\mathscr{T}_x$ (at a cost of $O(\log N)$) and insert next to it in constant time

(using the CBT insertion algorithm described earlier) the corresponding fragment of s. We then proceed in constant time to the next fragment of s' and insert next to it in constant time the corresponding fragment of s. This is repeated until the $O(\log N)$ fragments of s are inserted, for a total of $O(\log N)$ time.

Besides the segment trees $\mathscr{T}_x$ and $\mathscr{T}_y$ designed to store the edges of the current silhouette, we need an additional structure to store the vertices of the silhouette. This additional structure, called a *point tree* and denoted $\mathscr{P}_x$, is in some sense dual to the segment tree $\mathscr{T}_x$. Structure $\mathscr{P}_x$ is a binary tree built on the set $\{x_l(r), x_r(r): r \in \mathscr{R}\}$. Rather than storing segments like $\mathscr{T}_x$, the point tree $\mathscr{P}_x$ stores points. Specifically, for each vertex $p = (x, y)$ of the current silhouette, an entry is stored in the leaf of $\mathscr{P}_x$ that corresponds to abscissa x, as well as in each ancestor of this leaf in $\mathscr{P}_x$. We say that p is *recorded at* these nodes, and that these nodes are *allocated for* p. At each node V of $\mathscr{P}_x$, we attach a CBT $\mathscr{K}(V)$ to store the set of the vertices recorded at V, ordered according to increasing ordinate y. More precisely, as the silhouette may have several vertices with the same ordinate, a leaf v of the CBT $\mathscr{K}(V)$, labelled with the ordinate y, points to a simple list of points $p = (x, y)$ whose ordinate is y and whose abscissa x belongs to the interval $[B(V), E(V))$ of primary node V. Notice that each CBT associated with a leaf of the primary structure is the usual CBT with only one point per leaf v.

Each horizontal edge $e = [x', y; x'', y)$ in the silhouette has two endpoints $p' = (x', y)$ and $p'' = (x'', y)$ stored in $\mathscr{P}_x$. Each point $p = (x, y)$ appears in $\log N$ secondary structures $\mathscr{K}(V)$, specifically, those pertaining to the primary nodes V allocated for p. Clearly, for d segments there are $2d$ endpoints and $2d \log N$ entries in $\mathscr{P}_x$. Each entry for a point $p = (x, y)$ includes a pointer to the corresponding edge e in the list $\mathscr{E}$ and a pair of pointers *PL1* and *PL2* that link (bidirectionally) the CBT entries for p in the nodes allocated for p.

In addition, the structure includes a collection of pointers (*RLINK*, *LLINK*, and *PLINK*) that link (in a tree-like fashion) equally labelled nodes in different CBTs. Since the set of points recorded at a primary node W of $\mathscr{P}_x$ is a subset of those recorded at the parent W' of W, then for each node w in the CBT $\mathscr{K}(W)$, there is a node w' with the same label in the CBT $\mathscr{K}(W')$, and we establish the following pointers:

$$\begin{aligned} PLINK(w) &= w'; \\ RLINK(w') &= w, \text{ if } W \text{ is the right child of } W'; \\ LLINK(w') &= w, \text{ if } W \text{ is the left child of } W'. \end{aligned}$$

Undefined pointers are set to the value **nil** as usual.

During the execution of the algorithm, vertices are in turn inserted into the structure $\mathscr{P}_x$ and deleted from it, and the following lemma is used in the analysis of the algorithm:

LEMMA 2. *The insertion of a point $p = (x, y)$ into the point tree $\mathscr{P}_x$, and its deletion from it, can each be performed in $O(\log N)$ time, where N is the size of the basis of $\mathscr{P}_x$.*

PROOF. We first consider the simpler operation of deleting a point $p = (x, y)$. In $O(\log N)$ time, we identify the leaf V of the primary structure corresponding

to the abscissa x of p, and we locate p in the CBT $\mathscr{X}(V)$, since there is only one point per leaf in that CBT. Subsequently, we use the pointers *PL1* to locate all the occurrences of p in the point tree $\mathscr{P}_x$; each of them can be deleted in constant time.

The insertion of a point $p = (x, y)$ is more subtle. We begin with the root of the point tree as the current node W, and we insert p into $\mathscr{X}(W)$ in time $O(\log N)$. As a byproduct of this operation, we also determine the companion nodes v and u of y in $\mathscr{X}(W)$. Let W' be the child of the current node W that is to be allocated for p. Assume that W' is the left child (respectively, the right child) of W. The following procedure is used to find the two companions of y in $\mathscr{X}(W')$: We walk in $\mathscr{X}(W)$ upward from v toward the root until we find a node w such that $LLINK(w) \neq$ **nil** (respectively, $RLINK(w) \neq$ **nil**). Node $LLINK(w)$ (respectively, $RLINK(w)$) is then the first companion node v' of y in $\mathscr{X}(W')$. The other companion node u' can be found by an upward walk from u.

Given the companion nodes v' and u' of y in $\mathscr{L}(W')$, we can insert p into $\mathscr{X}(W)$ with a constant amount of additional work using the CBT insertion algorithm described earlier. We then reset $W := W'$, $v := v'$, and $u := u'$, and we repeat the above process until p is inserted into all its allocation nodes in the point tree. The total time used can be seen to be $O(\log N)$ by the following argument. Each time we traverse a CBT, we walk upward. In terms of the shadow structure, the nodes traversed follow an upward path. The height of shadow structure is $\log N + 1$, thus at most $O(\log N)$ time is expended in the traversals. The actual insertions take constant time each, once the companion nodes are found, so the total time is $O(\log N)$. □

2.2 The Line Drawing Algorithm

We now describe the line drawing algorithm. The silhouette is initialized as empty and the rectangles are processed in the order closest-rectangle-first. For each rectangle r, the five steps described below are performed in sequence. The first two steps generate the contribution of the rectangle r to the display, while the next steps update the silhouette. Each of the first four steps is applied uniformly to the four edges s_t, s_r, s_b, and s_l of rectangle r. For simplicity, we confine our description to how the horizontal top edge s_t is processed.

Step 1. *Intersect the rectangle r with the current silhouette.* When processing the current rectangle r, we must first determine the intersection of r with the silhouette of $\mathscr{T}$. We do that as follows. Let $s_t = [x_l, y_t; x_r, y_t)$ be the top horizontal side of the rectangle. In the segment tree $\mathscr{T}_y$, we visit each node that contains the ordinate y_t. For each such node V, the list $\mathscr{L}(V)$ contains a sequence of vertical segments that the segment s_t may intersect. In each CBT $\mathscr{L}(V)$, we locate the first vertical segment whose abscissa is $\geq x_l$, and extract the sublist of segments whose abscissa is $\leq x_r$.

For each of the $O(\log N)$ visited nodes, we obtain one such list. These lists are then merged according to a straightforward binary tournament. This yields a single left-to-right ordered list L_t of vertical edges of the silhouette intersected by s_t.

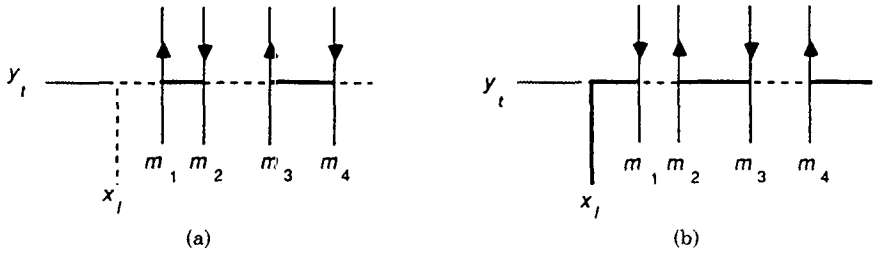

Fig. 5. The contribution of the top horizontal side s_t of the rectangle r to the display: (a) the left extreme of s_t is internal to the current silhouette; (b) the left extreme of s_t is external to the current silhouette.

Step 2. *Generate the contribution of rectangle r to the display.* To correctly draw the visible portions of edge s_t of r, we have to determine if a fixed extreme of segment s_t (say, its left extreme $p = (x_l, y_t)$) is internal or external to the silhouette. This is readily accomplished with the aid of the list L_t obtained in Step 1. If L_t is the empty set or if the leftmost term of L_t is downward-directed, then p is external; otherwise p is internal. (Note that the orientation of a segment is given by the order of its two endpoints in the oriented silhouette $\mathscr{B}$.)

We can now generate the visible portions of s_t. Let $m_1, m_2, \ldots, m_k$ be the sequence of the abscissae of the intersections of s_t with the members of L_t. If p is internal to the silhouette, we add to the display the set of segments $F(s_t) = [m_1, y_t; m_2, y_t), [m_3, y_t; m_4, y_t), \ldots$; otherwise we generate $F(s_t) = [x_l, y_t; m_1, y_t), [m_2, y_t; m_3, y_t), \ldots$ (refer to Figure 5).

Step 3. *Insert, into the segment trees of the silhouette, the segments contributed by rectangle r.* Each segment of the set $F(s_t)$ must be inserted into $\mathscr{B}$, which is straightforward to do, and into the segment tree $\mathscr{T}_x$. This latter part is done as follows:

Case 1. If segment s of $F(s_t)$ to be inserted contains a vertex of the rectangle r, then we perform a conventional insertion of s. Otherwise, segment s is delimited by two vertical edges of the silhouette successive in the list L_t. Let $e_1 = [p_1; p_1')$ and $e_2 = [p_2; p_2')$ be these two edges. We then distinguish the following remaining cases (refer to Figure 6).

Case 2. If at least one of the two segments $[p_1; p_2)$ or $[p_1'; p_2')$ is horizontal and belongs to the current silhouette (which can be tested in $O(\log N)$ time by searching these segments in the CBT of one of their allocation nodes), then we insert s using this edge of $\mathscr{B}$ as a guide (see Figures 6a and 6b).

Case 3. If neither $[p_1; p_2)$ nor $[p_1'; p_2')$ can be used as a guide, we insert s conventionally (see Figure 6c).

Remark. When inserting the segments of set $F(s_b)$ (respectively, $F(s_l)$) contributed by the third edge s_b (respectively, the fourth edge s_l) of the rectangle r, before testing for cases 2 and 3, we first check to see if the four endpoints p_1, p_1', p_2, p_2' of e_1 and e_2 are outside the rectangle r (see Figure 6d). If that is the case,

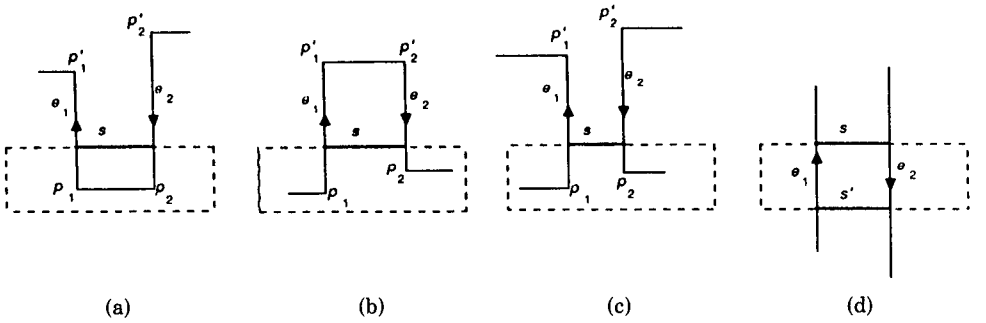

Fig. 6. Insertion of the segments contributed by the top horizontal side of a rectangle.

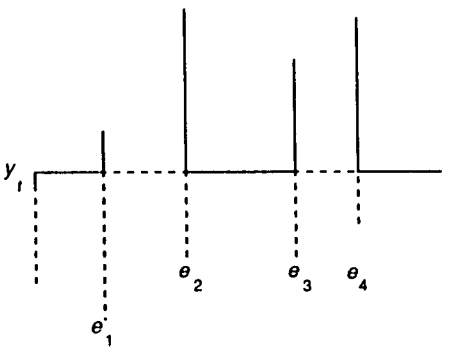

Fig. 7. The sequences of silhouette edges intersected by the top horizontal edge s_t of rectangle r.

there is a segment s' of $F(s_t)$ (respectively, $F(s_r)$) with the same span as s that can be used as a guide for the insertion of s.

Step 4. *Update the silhouette segments cut by rectangle r.* We consider the sequence of vertical silhouette segments $e_1, e_2, \ldots, e_m$ from list L_t that are intersected by the top side s_t of current rectangle r (see Figure 7). For each such segment we must remove from $\mathcal{T}_y$ the portion occluded by r and keep the visible portion. For each $e_i = [x_i, y_i'; x_i, y_i'')$, $1 \leq i \leq m$, the allocation nodes in $\mathcal{T}_y$ for e_i are of three types:

(1) The initial nodes that are allocated for the occluded portions of e_i. The entries for e_i in these nodes will be removed.

(2) Possibly one node V whose associated interval properly contains y_t (that is, $B(V) < y_t < E(V)$, where $[B(V), E(V))$ denotes the interval associated with V). Let e_i' denote the visible portion $[x_i, y_t; x_i, E(V))$ of e_i corresponding to node V. The entry for e_i in $\mathcal{L}(V)$ will be removed, and entries inserted into all the allocation nodes of $[y_t, E(V))$. These nodes are referred to as the *splinter nodes for* e_i'. With a slight abuse of notation, we refer to the allocation nodes of the interval $[y_t, 2N)$ as the *splinter nodes for* y_t.

(3) The remaining nodes, which correspond to visible portions of e_i. The entries of e_i in those nodes remain untouched.

```
algorithm UPDATE;
begin
for each splinter node W do {Initialize the pointer guide(W)}
   guide(W) := pointer to the leaf j of L(W) such that
                  LABEL(j) < x_l < LABEL(NEXT(j)) ;
for i = 1 to m do begin { Process each intersected segment }
   { Insert the appropriate fragments of e'_i into the splinter nodes for e'_i }
   for each splinter node W of e'_i do begin
      if guide(W) is the predecessor of e'_i in L(W) then
         Insert e'_i in L(W) using guide(W) as a subguide
      else insert e'_i in L(W) conventionally ;
      guide(W) := pointer to the entry for e'_i in L(W)
      end;
   { Update the guide pointer in the remaining splinter nodes for e_i }
   for each node W allocated to the visible portion of e_i do
      if W is a splinter node for y then
         guide(W) := pointer to the entry for e_i in L(W)
   end
end
```

Fig. 8. Algorithm *UPDATE*.

Since the successive fragments of each segment stored in the segment tree are linked, the deletion of the entries in subcases 1 and 2 is easy and can be done in time $O(\log N)$ for each intersected segment e_i.

More difficult is the insertion of the new entries in subcase 2, above. The algorithm below uses a modification of the idea of guided insertion, which we introduced in the last section. We say that *fragment t is inserted into a CBT using t′ as a subguide* when we are given a pointer to a node t' that is a neighbor fragment of t in the CBT. Since we are given a pointer to t' and do not have to search for it, the total process takes constant time using the CBT insertion algorithm described in the last section. The algorithm *UPDATE* given in Figure 8 initializes and maintains a pointer called *guide*(W) in the CBT for each splinter node W, and tries to use it whenever possible as a subguide for the insertion of the splinter fragments of the e'_i.

Step 5. *Remove the edges of the silhouette internal to rectangle r.* To complete the update of the silhouette, we have to remove from the data structure the edges that have become internal to r. To identify those edges we use the *point tree* $\mathscr{P}_x$, described in Section 2.1.

Tree $\mathscr{P}_x$ is searched as follows. Given the current rectangle $r = [x_l, x_r) \times [y_b, y_t)$, we visit the nodes in $\mathscr{P}_x$ that would be allocated for $[x_l, x_r)$ if $\mathscr{P}_x$ were a segment tree. In time $O(\log^2 N + k)$, we determine the k vertices of the silhouette contained in the interior of r. Any such vertex $p' = (x', y)$ is the extreme of an edge e whose other extreme is denoted p''. Vertex p' is processed as follows:

—If both points p' and p'' are inside rectangle r, then edge e is deleted from the appropriate segment tree $\mathscr{T}_x$ or $\mathscr{T}_y$, and both points p' and p'' are deleted from point tree $\mathscr{P}_x$.

—If only p' is inside r, then edge e has already been updated at Step 4, and we simply update the point tree $\mathscr{P}_x$ by deleting point p' and inserting the new point p, the intersection point of e with the boundary of r.

Over the course of execution of the algorithm, there are at most d edges deleted from $\mathscr{T}_x$ and at most d insertions or deletions in $\mathscr{P}_x$. A deletion of an edge from $\mathscr{T}_x$ can be performed in time $O(\log N)$, since the successive fragments of the edge are linked in the data structure. From Lemma 2, the insertion or deletion of a point $p = (x, y)$ in the point tree can be performed in $O(\log N)$ time, thus giving a $O(d \log N)$ time bound for Step 5.

3. AMORTIZED ANALYSIS

In this section we show that the algorithm given in the last section constructs the axial view of a scene of N isothetic 3D-rectangles in $O(N \log^2 N + d \log N)$ time, where d is the complexity of the display (that is, the number of edges of the display considered as an isothetic planarly embedded planar graph).

In the previous section we have already shown that the contribution of Step 5 to the overall running time is $O(d \log N)$. The cost of Step 2 is obviously $O(N + d)$ and the analysis of Step 1 is straightforward. Indeed, the determination of the intersection of the top horizontal side s_t of the rectangle r with the current silhouette is performed in time $O(\log^2 N + f)$, where f is the number of intersected segments, and the merging of the $O(\log N)$ lists of intersected segments is performed in time $O(f \log \log N)$. Since f is the number of segments contributed by s_t to the display, the global cost of Step 1 in the execution of the algorithm is $O(N \log^2 N + d \log \log N)$.

Let us now concentrate on the analysis of Step 3. We still assume that the segments to be inserted are horizontal segments; a quite symmetrical argument will work for the insertions of vertical segments. Each conventional insertion of a segment is performed in time $O(\log^2 N)$, while a guided insertion is performed in time $O(\log N)$. There are at most $4N$ conventional insertions of segments having as endpoint a vertex of the current rectangle; thus, the total cost for Case 1 in the whole computation is $O(N \log^2 N)$. There are at most d guided insertions corresponding to Case 2, so that the total cost for these cases is $O(d \log N)$.

We now show that conventional insertions of Case 3 occur $O(N)$ times. Let $e_1 = [p_1; p_1')$ and $e_2 = [p_2; p_2')$ be the edges of the silhouette bounding the segment s that is to be inserted. First, we dispose of the subcase where e_1 and e_2 belong to different connected components of the current silhouette. In this case the two components will be spliced by the insertion of rectangle r. This reduces the total number of components by 1, and this can happen at most $N - 1$ times (see Figure 9a).

Let us consider next the more typical case, where e_1 and e_2 belong to the same connected component of the silhouette. Our proof is based upon a charging argument. For this purpose we need to make a distinction between the *true vertices* of the silhouette (which are also vertices of the original input rectangles) and the *pseudovertices* (which arise from the intersection of rectangle edges).

Suppose at first that segments e_1 and e_2 have endpoints inside r (at most one endpoint per edge). Say that p_1 is inside r. If p_1 is a true vertex, then we charge the $O(\log^2 N)$ cost of conventional insertion of s to p_1. If p_1 is a pseudovertex, then either there is at least one true vertex p inside r (and we charge the $O(\log^2 N)$ cost of conventional insertion of s to p) or else p_2 is also a pseudovertex and $[p_1; p_2)$ is a silhouette edge. The former case is illustrated in Figure 9b. The

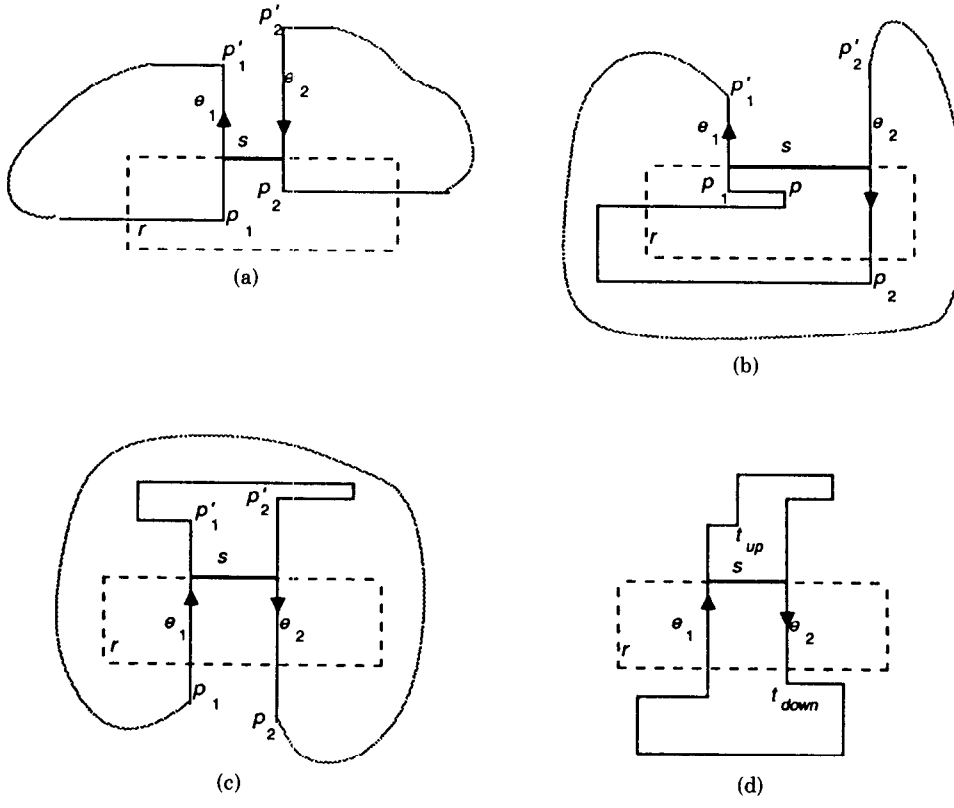

Fig. 9. The different cases for conventional insertion of a segment s into the data structure.

latter case has already been handled by Case 2: we perform a guided insertion of s in $O(\log N)$ time. Note that each time we charge a true vertex, the vertex is covered by rectangle r and is removed from the silhouette; thus, each true vertex is charged at most once for this subcase.

In the remaining subcase, where the four endpoints of e_1 and e_2 are outside r, both the top and bottom horizontal sides of r intersect e_1 and e_2, and we are in one of the following situations (see Figures 9c and 9d):

(1) The rectangle r delimits a hole within the opaque portion $\mathscr{F}$ of the plane. We distinguish two subcases:

  (1.1) The hole is a rectangle. This case is actually part of Case 2, in which we perform a guided insertion using the existing parallel edge $[p_1'; p_2')$ of the silhouette as a guide for the insertion of s.

  (1.2) The hole is not a rectangle (see Figure 9c). In this case, the boundary of the hole contains at least one true vertex, and we charge the cost of the conventional insertion to any such true vertex. Each true vertex can be charged at most once for this subcase.

(2) The rectangle r splits an existing hole into two parts and creates two new holes. We again distinguish two subcases:

(2.1) One of the new holes is a rectangle and, analogously to (1.1) above, we are in fact in an instance of Case 2, where a guided insertion is performed.

(2.2) Each new hole contains at least one true vertex. This is the interesting case. (Note that neither edge $[p_1; p_2)$ nor $[p_1'; p_2')$ can be part of the silhouette, since otherwise we would be in Case 2.) Let us denote the endpoints of segment s to be inserted by (x_1, y_t) and (x_2, y_t), with $x_1 < x_2$. We define the two true vertices t_{up} and t_{down} for s as follows (see Figure 9d): t_{up} is the true vertex along the silhouette above s whose abscissa belongs to $[x_1, x_2]$ and that has the smallest ordinate value $> y_t$; t_{down} is the true vertex along the silhouette below s whose abscissa belongs to $[x_1, x_2]$ and that has the largest ordinate value $< y_t$.

LEMMA 3. *In situation* 2.2 *above, true vertices* t_{up} *and* t_{down} *exist and are well defined.*

PROOF. There can be no ties (vertices with the same ordinate value) in the definitions of t_{up} and t_{down}, since the rectangles are in general position. (And even if there were a tie, a winner could be chosen arbitrarily.) The existence of t_{up} and t_{down} follows from a simple case analysis. □

We charge the cost of the conventional insertion to t_{up} (and we say that t_{up} is "upward charged") if t_{up} has not already been "upward charged" for the insertion of another horizontal segment. Otherwise, we charge the cost to t_{down} (and we say that t_{down} is "downward charged") if t_{down} has not already been "downward charged" for the insertion of another horizontal segment. The following lemma shows that either t_{up} or t_{down} will always be available for charging, thus completing the analysis of Step 3.

LEMMA 4. *In the subcase discussed above, either* t_{up} *has not yet been upward charged or* t_{down} *has not yet been downward charged.*

PROOF. Suppose that true vertex p is t_{down} and is downward charged for the insertion of segment s at a given stage of the execution of the algorithm. Suppose also that p has the role of t_{down} for the later insertion of another segment s'. We show that the vertex p' designated as t_{up} for s' has not yet been upward charged (see Figure 10). From the definition of t_{down}, s' must have a lesser ordinate value than s. It follows that p' lies along the silhouette (immediately prior to the insertion of s') between s and s'. (Note that if p' did not exist, s could have been used as a guide for the insertion of s'.) We claim that p' has not yet been upward charged. By the definition of t_{up}, there can be no other segments inserted along the silhouette between p' and s'. Thus, the only way in which p' could have been upward charged previously is during the prior insertion of some segment s'' below s'. But s'' must also be below p, or else p would not be t_{down} for the insertions of s and s'. This contradicts the fact that p' is t_{up} for s'', since p has lower ordinate value than p'. The proof for the other case, showing that t_{down} is

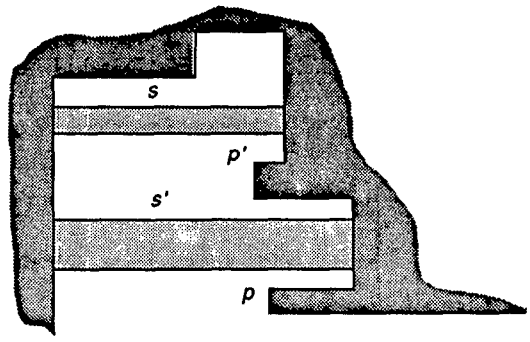

Fig. 10. The true vertices charged for unguided insertions.

available for being downward charged when t_{up} has already been upward charged, is symmetrical. □

Thus, during the execution of the algorithm, there are $O(N)$ conventional insertions and $O(d)$ guided insertions performed in Step 3, and the total cost of this step is $O(N \log^2 N + d \log N)$.

The rest of this section is devoted to the analysis of Step 4. Step 4 updates the segments intersected by the current rectangle r. It involves two processes. The first process deletes from the data structure the occluded fragments of each intersected segment e_i. For each intersected segment, the deletion of the first occluded fragment is done in $O(\log N)$ time. Deletion of the succeeding ones takes only constant time per deleted fragment, since the successive fragments of a given segment are linked together in the data structure. Thus, the total cost for deletions in Step 4 is $O(d \log N)$.

The second process updates in the data structure the visible portion of each intersected segment e_i, $i = 1, \ldots, m$, and is confined to nodes contained in the subtree of a *single* allocation node V for e_i. The visible portion of e_i corresponding to V is called e_i'. For brevity, we confine our analysis to the update of vertical segments intersected by the top horizontal side of the current rectangle. The analyses for the other cases are similar.

It is appropriate at this point to review the relevant properties of splinter nodes. The *splinter nodes for* e_i' (whose lower ordinate is y_t) are the allocation nodes of the segment $[y_t, E(V))$, where V is the allocation node of e_i such that $B(V) \leq y_t < E(V)$. Let U be the leaf whose interval is $[y_t, y_t + 1)$. By the nature of the segment tree, these splinter nodes are right children of nodes on the path from U to V that are not on the path themselves. It follows then that the union of the splinter nodes for e_i', $i = 1, \ldots, m$, is a subset of the *splinter nodes for (ordinate)* y_t, which are defined to be all analogous right children for the path from U to the root of $\mathcal{T}_y$. There can be at most $\log N$ splinter nodes for y_t.

The subroutine *UPDATE* has an initialization step in which a pointer *guide*(W) is set at a cost of $O(\log N)$ for each of the $O(\log N)$ splinter nodes W for y_t, where y_t is the ordinate of the top horizontal side of the current rectangle. This induces a total cost of $O(N \log^2 N)$ over the whole execution of the algorithm.

Each intersected segment e_i is then processed, in turn, as follows. The splinter fragments of e_i' are inserted into the data structure using the pointers *guide*(W)

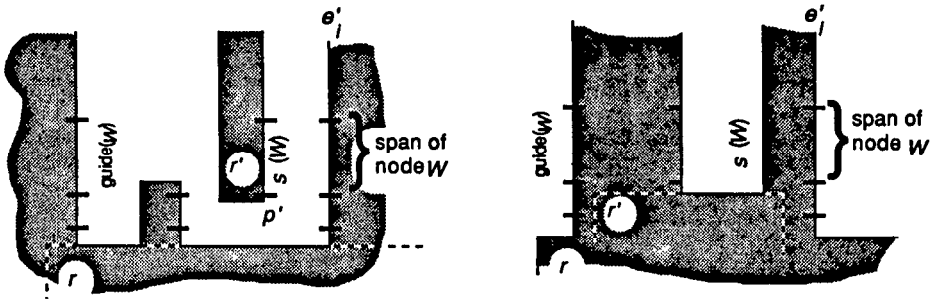

Fig. 11. Divider fragments.

as subguides whenever possible. The $O(\log N)$ nodes W allocated to the visible portion of e_i are visited, and whenever W is a splinter node, $guide(W)$ is updated to point to the entry for e_i in W's CBT. The subguided fragment insertion, as well as the update of the pointer's $guide(W)$, is performed in constant time for each visited node and contributes $O(d \log N)$ to the overall cost of the algorithm.

Insertion of a splinter fragment without a subguide is performed conventionally at a cost of $O(\log N)$ time. We are left with the problem of counting the number of nonsubguided insertions of splinter fragments during the algorithm. To do that, we use the following charging argument.

The pointer $guide(W)$ is not a valid subguide for the insertion of e_i' into the splinter node W if the CBT $\mathscr{L}(W)$ contains a vertical fragment whose abscissa is between the abscissa of $guide(W)$ and e_i'. We call such a vertical fragment a *divider*. Divider fragments belong to segments recorded at node W that do not intersect the current rectangle r and whose abscissae are greater than x_l, the abscissa of the leftmost edge of r.

Let $s(W)$ be the *rightmost* divider fragment between $guide(W)$ and $e_i'(W)$ in the CBT $\mathscr{L}(W)$. If the bottom endpoint of the edge containing $s(W)$ is a true vertex p' (as in Figure 11a), then the $O(\log N)$ cost for the insertion of fragment $e_i'(W)$ is charged to p'. Otherwise, the edge containing $s(W)$ is cut by the top side of some rectangle r' (as in Figure 11b), and we charge the $O(\log N)$ cost to r'.

The lemma below completes the analysis. It shows that there are $O(N \log N)$ charges for nonsubguided insertions, thus giving us the desired $O(N \log^2 N)$ bound.

LEMMA 5. *Each true vertex p' and each rectangle r' can be charged for the nonsubguided insertion of a splinter fragment at most* log N *times over the course of the algorithm, that is, at most once for each of the splinter nodes for y', where y' is, respectively, the ordinate of p' or the ordinate of the top horizontal side of r'.*

PROOF. For brevity, we restrict ourselves to the case where the bottommost endpoint of the segment containing $s(W)$ is a true vertex $p' = (x', y')$. (The proof of the other case, in which rectangle r' is charged, is similar.)

Let vertex p' be charged at one of the splinter nodes for y' when inserting e_i', produced by the intersection of e_i with the topmost side of rectangle r (whose ordinate is y_t). Let W be the splinter node in question, and let $s(W)$ be the rightmost divider fragment (note that x' is the abscissa of $s(W)$). Since W is a right child in $\mathcal{T}_y$, y_t belongs to the proper range of the parent of W.

Suppose now, for a contradiction, that later p' is charged, with the same divider $s(W)$, for the insertion of splinter fragment $e'(W)$, produced by the intersection of some edge e with the topmost side of some rectangle r'' with ordinate y''. For this to happen, e must intersect r'' *but not* r; otherwise, e would already be inserted in $\mathcal{L}(W)$. However, we note the following facts:

(1) The abscissa $x(e)$ of e is contained in the x-range of r. This follows because $x_l < x(s(W))$ (by the definition of *guide*(W)), $x(s(W)) < x(e)$, trivially, and $x(e) < x(e_i)$, since $s(W)$ is the divider.

(2) Since $e'(W)$ is a splinter fragment of e', it follows that e spans W's range $[B(W), E(W))$, but is not recorded at W. Thus, e spans the range of the parent of W in $\mathcal{T}_y$.

We noted earlier that y_t belongs to the proper range of W's parent, and hence by Fact 2, y_t is contained in the vertical span of e. Combining this with Fact 1, we obtain that e intersects r, a contradiction. □

The description of the data structure in Section 2, and the preceding analysis, leads to the following theorem:

THEOREM 1. *Let d be the number of segments in the axial view of a scene of N isothetic rectangles. The above algorithm constructs this axial view in time $O(N \log^2 N + d \log N)$ using storage $O(N \log N)$.*

4. CONCLUSION AND OPEN QUESTIONS

We have shown that the axial view of a scene of N isothetic 3D-rectangles can be computed in time $O(N \log^2 N + d \log N)$, thereby improving, in a particular case, the result of Güting and Ottmann [7]. As it is presented, our algorithm actually solves the hidden line problem for such a scene, but it can be extended to solve the hidden surface problem without much difficulty. One of our future goals will be to explore the extensibility of the proposed technique to more general views, that is, to *perspective views* from an arbitrary viewpoint, finite or at infinity. This objective, however, involves resolving the difficult question of the cyclicity of the dominance relation in three dimensions. In fact, since our scene-sensitive approach to hidden-line elimination is of the "priority" type, it is essential to order the objects consistently with the dominance relation. As noted in the Introduction, this relation is in general not acyclic when the scene is viewed from an arbitrary viewpoint. In such a general case, in order to attain the desired acyclicity, it may be necessary to appropriately split some parallelepipeds into two or more parts. This approach would raise, among others, the following interesting questions:

—Is there an $o(N^2)$-time algorithm to determine whether the dominance relation of a set of N 3D-rectangles is acyclic?

—Cycles can be eliminated by splitting selected 3D-rectangles. Is the problem of finding a minimum number of splits to achieve acyclicity $\mathcal{NP}$-complete?

It must be observed, however, that in the practically very important case where the projections of the parallelepipeds on a plane are disjoint, the dominance relation becomes two-dimensional and is therefore acyclic. This is the case, for example, for a set of parallelepipeds resting on a ground plane.

If we implement the segment and point tree algorithms of [7] using the dynamic fractional cascading technique of [6], we get an algorithm whose running time is $O((N + d)\log n \log\log n)$. The same running time can also be achieved using the data structure of van Emde Boas [21]. Recently, an algorithm was independently proposed with the same $O(N \log^2 N + d \log N)$ time bound as our solution, but using a totally different approach, involving plane-sweep and balanced tree techniques [1]. Its running time can be reduced to $O(N \log N \log\log N + d \log N)$ by using dynamic fractional cascading. For small d, the running times for the two uses of dynamic fractional cascading given above are asymptotically less than the running time of the algorithm we have presented, although, for practical purposes, dynamic fractional cascading has a very high overhead. For many cases of interest, the value of d is superlinear (perhaps $\Theta(N^2)$), and the term involving d dominates. In such cases, our algorithm and the ones in [1] have comparable asymptotic behavior.

APPENDIX A. Segment and Point Trees[3]

The segment tree is a semidynamic data structure designed to handle intervals on the real line whose extremes belong to a fixed set of N abscissae. Since the set of abscissae is fixed, the segment tree is a static structure with respect to the abscissae (that is, one that does not support insertions or deletions of abscissae); in addition, the abscissae can be normalized by replacing each of them by its rank in their left-to-right order. Without loss of generality, we may consider these abscissae as integers in the range $[O, N - 1]$.

The segment tree is rooted binary tree. Given integers l and r with $l < r$, the segment tree $T(l, r)$ is recursively built as follows: it consists of a root v with two parameters $B(v) = l$ and $E(v) = r$, and, if $r - l > 1$, of a left subtree $T(l, \lfloor(B(v) + E(v))/2\rfloor)$ and a right subtree $T(\lfloor(B(v) + E(v))/2\rfloor, r)$. The parameters $B(v)$ and $E(v)$ define the interval $[B(v), E(v)) \subset [l, r)$ associated with the node v.

We can establish that $T(l, r)$ is balanced and has depth $\lceil \log_2(l - r)\rceil$. The segment tree $T(l, r)$ is designed to store intervals whose extremes belong to $\{l, \ldots, r\}$, in a dynamic fashion (that is, supporting insertions and deletions). Specifically, for $r - l > 3$, an arbitrary interval $[b, e)$, with integers $b < e$, will be partitioned into a collection of at most $\lceil \log_2(l - r)\rceil + \lfloor \log_2(l - r)\rfloor - 2$ standard intervals of $T(l, r)$. The segmentation of interval $[b, e)$ is completely specified by the operation that inserts $[b, e)$ into T, that is, by a call of the procedure $INSERT(b, e; root(T))$ given in Figure 12.

[3] Part of this section is closely patterned after the presentation given in [15].

```
procedure INSERT(b, e, v)
  begin if (b ≤ B(v)) and (E(v) ≤ e) then allocate [b, e] to v
    else begin
      if b < ⌈(B(v) + E(v))/2⌉ then INSERT(b, e, LSON(v));
      if ⌈(B(v) + E(v))/2⌉ < e then INSERT(b, e, RSON(v))
    end
  end
```

Fig. 12. Procedure *INSERT*.

```
procedure INSERTAFTER(lab, y', root)
{ lab is the label of the new element to be inserted,
y' is a pointer to the preceding element,
root is a pointer to the root of the CBT.
This procedure finds the companion node v of lab, and the companion node u when it is needed,
and then calls the procedure INSERT(lab, v, u, root). }
begin
    v := NEXT(y');
    if (v = nil) then INSERT(lab, PREV(y'),nil, root)
    else if (y ≤* v) then
        if (y' is a left child) then INSERT(lab, v, nil, root)
        else INSERT(lab, PREV(y'), NEXT(y'), root)
    else INSERTBEFORE(lab, NEXT(v), root)
end
```

Fig. 13. Procedure *INSERTAFTER*.

The nodes of T to which the fragments segmentation of $[b, e)$ have been assigned are said to be *allocated to* $[b, e)$, and the segment $[b, e)$ is said to be *recorded at* those nodes. Perfectly symmetrical to *INSERT* is the *DELETE* operation.

Normally, a segment tree is used to store a collection of segments whose extremes are among the given N abscissae. In this case each node v of $T(l, r)$ will have a secondary structure storing the fragments of segments recorded at v. A typical search operation applied to such trees is the determination of all segments intersected by a given vertical line $x = c$. This corresponds to specifying a root-to-leaf path in $T(l, r)$ and retrieving all segments in the secondary structures of the node of this path.

In this paper, we reserve the denomination of *segment tree* to a segment tree designed to store segments and queried along a path (a "point"). Conversely, we call *point tree* a segment tree designed to store points and searched according to a segment. In a point tree each node v stores all points that have the x-coordinate, for instance, within the interval $[B(v), E(v))$. The search for the points that have the x-coordinate within a given range is then performed by inspecting all the allocation nodes for the query range.

APPENDIX B. Procedures for CBT Insertion and Deletion

The interacting procedures *INSERTAFTER* and *INSERTBEFORE* insert, in constant time, a new element y into a CBT. Procedure *INSERTAFTER*(*lab*, y', *root*), given in Figure 13, inserts a new element labelled *lab* just after a given

```
procedure DELETE(y, root)
  begin
  if y = smallest then smallest := NEXT(NEXT(y));
  if y = largest then largest := PREV(PREV(y));
  if y ≤* PARENT(y) then {y is a left child }
     begin
     v := PARENT(y);
     if v = root then LEFT(v) := nil
     else begin
        if v ≤* PARENT(v) then LEFT(PARENT(v)) := RIGHT(v)
        else RIGHT(PARENT(v)) := RIGHT(v);
        establish the linear order PREV(y), NEXT(v)
        end
     end
  else { y is a right child } symmetrically
  end
```

Fig. 14. Procedure *DELETE*.

element pointed to by y' in the CBT rooted at *root*. The procedure *INSERTBEFORE*(*lab*, y', *root*) which inserts a leaf labelled *lab* just before the leaf pointed to by y' is symmetrical to *INSERTAFTER*, and its description is therefore omitted. The procedure *DELETE*, given in Figure 14, deletes in constant time the leaf pointed to by y from the CBT rooted at *root*.

REFERENCES

1. BERN, M. Hidden surface removal for rectangles. In *Proceedings of the 4th ACM Symposium on Computational Geometry* (Urbana, Ill., June 1988), ACM, New York, 1988, 183–195.
2. DÉVAI, F. Quadratic bounds for hidden line elimination. In *Proceedings of the 2nd ACM Symposium on Computational Geometry* (Yorktown Heights, N.Y., 1986). ACM, New York, 1986, 269–275.
3. EDELSBRUNNER, H., GUIBAS, L. J., AND STOLFI, J. Optimal point location in a monotone subdivision. *SIAM J. Comput. 15*, 2 (1986), 317–340.
4. EDELSBRUNNER, H., OVERMARS, M. H., AND WOOD, D. Graphics in flatland. In *Advances in Computing Research. Vol. 1: Computational Geometry*, F. P. Preparata, Ed., JAI Press, Greenwich, Conn., 1983, 35–39.
5. FUCHS, H., KEDEM, M., AND NAYLOR, B. F. On visible surface generation by a priori tree structures. *Comput. Gr. 14* (1980), 124–133.
6. FRIES, O., MEHLHORN, K., AND NÄHER, S. Dynamisation of geometric data structures. In *Proceedings of the 1st ACM Symposium on Computational Geometry* (Baltimore, Md., 1985). ACM, New York, 1985, 168–176.
7. GÜTING, R. H., AND OTTMANN, T. H. New algorithms for special cases of the hidden line elimination problem. *Comput. Vision, Gr. Image Process. 40* (1987), 188–204.
8. GOODRICH, M. T. A polygonal approach to hidden-line elimination. In *Proceedings of the 25th Allerton Conference on Communication, Control, and Computing* (Allerton, Ill., Oct. 1987). University of Illinois Press, 1987.
9. GUIBAS, L. J., AND YAO, F. F. Translating a set of rectangles. In *Advances in Computing Research, Vol. 1: Computational Geometry*, F. P. Preparata, Ed., JAI Press, Greenwich, Conn., 1983, 61–77.
10. MCKENNA, M. Worst-case optimal hidden-surface removal. *ACM Trans. Gr. 6*, 1 (Jan. 1987), 19–28.
11. NEWMAN, W. M., AND SPROULL, R. E. *Principles of Interactive Computer Graphics*, 2nd ed. McGraw-Hill, New York, 1979.

12. NURMI, O. A fast line sweep algorithm for hidden line elimination. *BIT 25* (1985), 466–472.
13. OVERMARS, M. H. Range searching in a set of line segments. In *Proceedings of the 1st ACM Symposium on Computational Geometry* (Baltimore, Md., 1985). ACM, New York, 1985, 177–185.
14. OTTMANN, T., AND WIDMAYER, P. Solving visibility problems by using skeleton structures. In *Proceedings of the 11th Symposium on Mathematical Foundations of Computer Science* (Prague, Czechoslovakia). Lecture Notes in Computer Science, 176, Springer-Verlag. 1984, 459–470.
15. PREPARATA, F. P., AND SHAMOS, M. I. *Computational Geometry*. Springer, New York, 1985.
16. REIF, J. H., AND SEN, S. An efficient output-sensitive hidden surface removal algorithm and its parallelization. In *Proceedings of the 4th ACM Symposium on Computational Geometry* (Urbana, Ill., June 1988), ACM, New York, 1988, 193–200.
17. SCHUMACKER, R. A., BRAND, B., GIGILLAND, M., AND SHARP, M. Study for applying computer-generated images to visual simulation. Tech. Rep. TR AFHL-TR-69-14, USAF Human Resources Lab., 1969.
18. SCHMITT, A. Time and space bounds for hidden line and hidden surface computation. In *Proceedings of Eurographics '81*. North-Holland, Amsterdam, 1981, 43–56.
19. SECHREST, S., AND GREENBERG, D. P. A visibility polygon reconstruction algorithm. *ACM Trans. Gr. 1*, 1 (1982), 25–42.
20. SUTHERLAND, I. E., SPROULL, R. F., AND SCHUMACKER, R. A. A characterization of ten hidden-surface algorithms. *ACM Comput. Surv. 6*, 1 (1974), 1–25.
21. VAN EMDE BOAS, P., KAAS, R., AND ZILJSTRA, E. Design and implementation of an efficient priority queue. *Math. Syst. Theor. 10* (1977), 99–127.
22. YAO, F. F. On the priority approach to hidden surface algorithm. In *Proceedings of the 21st IEEE Symposium on Foundations of Computer Science* (Syracuse, N.Y., 1980). IEEE, New York, 1980, 301–307.

Received May 1988; revised April 1989; accepted April 1989

Editor: Leo J. Guibas