INFORMATION AND COMPUTATION 107, 1-24 (1993)

Output-Sensitive Methods for Rectilinear Hidden Surface Removal*

MICHAEL T. GOODRICH[†]

Department of Computer Science, Johns Hopkins University, Baltimore, Maryland 21218

MIKHAIL J. ATALLAH[‡]

Department of Computer Science, Purdue University, West Lafayette, Indiana 47907

AND

MARK H. OVERMARS[•]

Department of Computer Science, University of Utrecht, 3508 TB Utrecht, The Netherlands

We present an algorithm for the hidden-surface elimination problem for rectangles, which is also known as window rendering. The time complexity of our algorithm is dependent on both the number of input rectangles, n, and on the size of the output, k. Our algorithm obtains a trade-off between these two components, in that its running time is $O(r(n^{1+1/r}+k))$, where $1 \leqslant r \leqslant \log n$ is a tunable parameter. By using this method while adjusting the parameter r "on the fly" one can achieve a running time that is $O(n \log n + k(\log n/\log(1+k/n)))$. Note that when k is $\Theta(n)$, this achieves an $O(n \log n)$ running time, and when k is $\Theta(n^{1+\varepsilon})$ for any positive constant ε, this achieves an $O(k)$ running time, both of which are optimal. © 1993 Academic Press, Inc.

* A preliminary announcement of this research is to appear at the 17th International Colloquium on Automata, Languages, and Programming. Part of this research was carried out while the authors were visiting Princeton University for the DIMACS Workshop on Geometric Complexity.

[†] This author's research was supported by the National Science Foundation under Grant CCR-8810568 and by the NSF and DARPA under Grant CCR-8908092.

[‡] This author's research was supported by the Office of Naval Research under Grants N00014-84-K-0502 and N00014-86-K-0689, the National Science Foundation under Grant DCR-8451393, and the National Library of Medicine under Grant R01-LM05118. Part of this research was carried out while this author was visiting the Research Institute for Advanced Computer Science, NASA Ames Research Center, Moffett Field, California.

[•] This author's research was partially supported by the ESPRIT II Basic Research Actions Program of the EC, under Contract 3075 (Project ALCOM).

0890-5401/93 $5.00

Copyright © 1993 by Academic Press, Inc.
All rights of reproduction in any form reserved.

1. Introduction

1.1. *The Problem*

The hidden-surface elimination problem is well known in computer graphics and computational geometry. In this problem one is given a set of simple, non-intersecting planar polygons in 3-dimensional space and a projection plane π, and one wishes to determine which portions of the polygons are visible when viewed from infinity along a direction normal to π, assuming all the polygons are opaque. An important special case of this problem occurs when the polygons are all *isothetic* rectangles; i.e., the rectangles are all parallel to the xy-plane and have sides that are parallel to either the x- or the y-axis. This version of the hidden-surface elimination problem is also known as the *window rendering* problem, since it is the problem that must be solved to render the windows that might need to be displayed on the screen of a workstation. (See Fig. 1.)

Using the terminology of [28], we are interested in the *object space* version of this problem. That is, we want a method that produces a device-independent, combinatorial representation of the visible surfaces. Such a solution is not dependent on any specific method for rendering polygons nor on the number of pixels on a display screen. In addition, an object space solution gives us a representation that is easily scaled and rotated.

1.2. *Previous Work*

We briefly review some of the more efficient known algorithms for the window rendering problem. Since this problem is a special case of the general hidden-surface elimination problem, any algorithm for the general case can also be used for this problem. In [16] McKenna shows how to solve the general hidden-surface elimination problem in $O(n^2)$ time, generalizing an algorithm by Dévai [8] for the hidden-line elimination

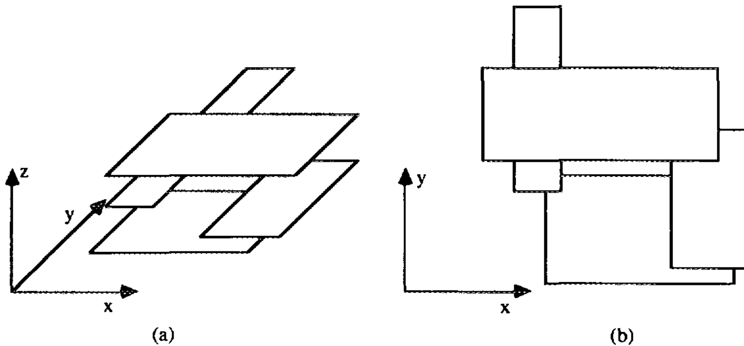

Fig. 1. (a) Isothetic rectangles; (b) their visible portion.

problem that also runs in $O(n^2)$ time (in the hidden-line elimination problem one is only interested in computing the portions of the polygonal boundaries that are visible). These algorithms are worst-case optimal, because there are problem instances that have $\Theta(n^2)$ output size (e.g., see Fig. 2a). Unfortunately, these algorithms always take $\Theta(n^2)$ time [8, 16], even if the size of the output is very small.

In [19] Nurmi gives an algorithm for general hidden-line elimination that runs in $O((n+I)\log n)$ time, where I is the number of pairs of line segments whose projections on π intersect (I is $O(n^2)$). Schmitt [25] also achieves this bound. If I is $o(n^2/\log n)$, then these algorithms clearly run faster than $O(n^2)$ time. Their worst-case performance is, however, a suboptimal $O(n^2 \log n)$ time (if I is $\Theta(n^2)$).

In [13] Güting and Ottmann address the window rendering problem, giving an algorithm that runs in $O(n \log^2 n + I)$ time. Using results of Goodrich [11] and Larmore [14] this can be improved to $O(n \log n + I)$ time. Doh [9] also achieves this bound. All of these algorithms are not truly output-sensitive, however. Indeed, there are problem instances where these algorithms run in $O(n^2)$ time even though the output size is constant (e.g., in the case where a large rectangle obscures a collection of rectangles that intersect to form a "grid," as in Fig. 2b).

There are methods whose running time depends on both the input size and output size, however. In [13] Güting and Ottmann also gave an output-sensitive window rendering algorithm that runs in $O(n \log^2 n + k \log^2 n)$ time, where k is the actual size of the output. Bern [4] and Preparata *et al.* [24] have subsequently shown that one can solve the window rendering problem in $O(n \log n \log\log n + k \log n)$ time and $O(n \log^2 n + k \log n)$ time, respectively. In algorithms such as these, the term in the time complexity involving only n is called the *input-size component* and the term involving k (and possibly n as well) is called the *output-size component*.

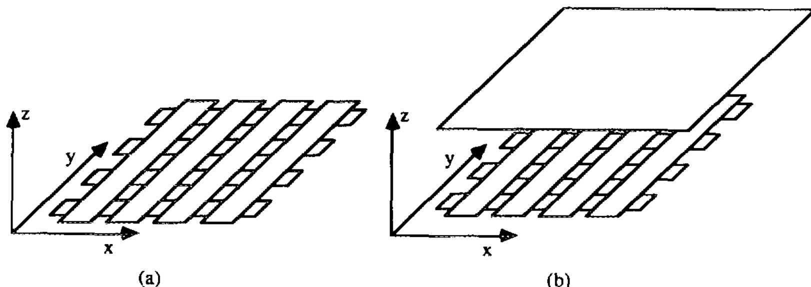

FIG. 2. (a) Quadratic output size; (b) small output size with quadratic I.

1.3. *Our Results*

In this paper we give a new algorithm for the window rendering problem whose running time depends on both the input size and output size. Our algorithm allows one to specify a trade-off between these two components of the running time, in that its running time is $O(r(n^{1+1/r}+k))$, where $1 \geqslant r \geqslant \log n$ is a tunable parameter. Using this method while adjusting the parameter r "on the fly", one can easily achieve $O(n \log n + k(\log n/\log(1+k/n)))$ time, as observed by Paterson [22]. Independently, Bern [5] and Mehlhorn [17] were recently able to achieve $O(n \log n + k \log(2n^2/k))$ time using an elegant method, which is quite different from ours. Note, however, that our time bound is always at least as good as theirs, and is better for quite a large range of k values. For example, if k is $\Theta(n^{1+\varepsilon})$ for any constant ε, $0 < \varepsilon < 1$, then our method achieves an $O(k)$ running time, which is optimal, whereas theirs still has a suboptimal $\Theta(k \log n)$ running time.

We sweep through the collection of rectangles from front to back with a plane parallel to the xy-plane. During this sweep we maintain the *shadow* of all the rectangles already encountered (i.e., the union of their projections on the xy-plane). In encountering a new rectangle R, we determine all the intersections of R with the shadow—each intersection determines a "piece" of a solution to the hidden-surface elimination problem. We complete the processing of R by updating the shadow to include the region obscured by R. The main difficulty is in performing these operations efficiently.

To obtain an efficient running time we develop a new data structure that we call the *hive tree*. This structure is a combination of the hive graph structure of Chazelle [6] and the segment tree structure of Bentley and Wood [3], augmented with a number of supporting auxiliary structures. Each supporting structure is implemented with the simplest data structures—arrays and linked lists—hence, our method should be fairly easy to program.

The paper is organized as follows. In the next section we describe the hive tree data structure and give a method for its construction. In Section 3 we show how to use the hive tree to derive a simple, efficient method for rectilinear hidden-line elimination. We show how to extend this method to the hidden-surface elimination problem in Section 4. Both of these methods run in time that is $O(r(n^{1+1/r}+k))$, except for a preprocessing step that requires $O(r(n^{1+1/r} \log n + k))$ time. In Section 5 we show how to derive the claimed time bound by eliminating this preprocessing bottleneck (at the expense of introducing some sophisticated data structuring techniques).

2. The Hive Tree

Suppose we are given a collection S of n non-intersecting isothetic rectangles in $\mathfrak{R}^3$, i.e., a collection of rectangles parallel to the xy-plane such that all edges are parallel to either the x- or the y-axis. The problem is to compute all the portions of each rectangle that are visible from $z = \infty$ with light rays that are parallel to the z-axis (i.e., the projection plane is the xy-plane).

Specifically, each rectangle R is given by a triple $((x_1, y_1), (x_2, y_2), z)$, where (x_1, y_1) is the lower-left corner of R, (x_2, y_2) is the upper-right corner of R, and z is the z-coordinate of the plane to which R belongs. For the remainder of this paper we assume that the relationships "to the right of" and "to the left of" are with respect to x-coordinates, that the relationships "above" and "below" are with respect to y-coordinates, and that the relationships "in front of" and "behind" are with respect to z-coordinates. Given an isothetic rectangle R in $\mathfrak{R}^3$ we let $z(R)$ denote the z-coordinate of the plane to which R belongs. Similarly, for any point p in $\mathfrak{R}^3$, we use $x(p)$, $y(p)$, and $z(p)$ to denote the x-, y-, and z-coordinate of p, respectively.

Let Hid(S) be the planar subdivision determined by a solution to the hidden-line elimination problem. That is, Hid(S) is an embedded planar graph whose edges correspond to the visible segments. In order to better motivate our hidden-surface method, let us examine the structure of Hid(S) more closely. For each vertex v of Hid(S) either v corresponds to a (visible) *corner point* of a rectangle in S or v corresponds to an intersection of two visible edges (where one of them becomes occluded by the other, i.e., an intersection of the form $\top$, $\bot$, $\vdash$, or $\dashv$). We call such intersections *dead ends*, and classify them into two types: *vertical dead ends*, where the terminating segment is vertical (i.e., $\top$ or $\bot$), and *horizontal dead ends*, where the terminating segment is horizontal (i.e., $\vdash$ or $\dashv$).

Before we give our hidden-line elimination method, we describe the primary data structure we use in our algorithm, namely, the *hive tree*. This data structure is defined for a given collection of rectangles in the plane. In our case we use the projections of the rectangles in S on the xy-plane. To construct a hive tree we project the vertical rectangle boundaries on the x-axis and place a vertical line between each consecutive pair of projection points (any such vertical line will do). This partitions the plane into at most $2n+1$ "slabs." Note that none of the dividing vertical lines contains the vertical boundary of a rectangle in S. We then build a complete $n^{1/t}$-ary tree T (i.e., a rooted tree such that each internal node has $n^{1/t}$ children) on these slabs in the natural way, so that each leaf is associated with a slab, where $2 \leqslant t \leqslant \log n$ is a tunable parameter. We will use t to denote this "branching factor" throughout the remainder of this paper, and use the

relationship $r = t/2$ to derive the bounds claimed in the introduction (which involve the parameter r). To simplify computations that we perform for leaf nodes, we augment T by giving each leaf v a parent w, such that v is the only child of w (so that the parent of w has $n^{1/t}$ children). Thus, T has height $\lceil t \rceil + 1$, since each leaf node has no siblings.

We use Π_v to denote the slab associated with the leaf v. To each internal node v in T we associate a slab Π_v, which is the union of all the slabs associated with the children of v. Let $\mathscr{L}(\Pi_v)$ (resp., $\mathscr{R}(\Pi_v)$) denote the left (resp., right) vertical line that is the boundary of Π_v. Note that by projecting back to three dimensions $\mathscr{L}(\Pi_v)$ (resp., $\mathscr{R}(\Pi_v)$) can also be viewed as a plane parallel to the yz-plane such that any rectangle $R \in S$ intersects this plane in a line segment parallel to the y-axis. (This alternate view will be useful for our window-rendering methods.)

We define some relationships similar to those defined for the segment tree data structure of Bentley and Wood [3]. We say that a rectangle R *spans* a slab Π_v if R intersects Π_v, but neither of R's vertical boundaries lie inside Π_v. A rectangle R *covers* a node v in T if it spans Π_v but does not span Π_z, where z is the parent of v. A rectangle R *ends in* a slab Π_v if R does not span Π_v and has a vertical boundary inside Π_v. For each v in T we define two lists, Cover(v) and End(v), such that Cover(v) stores all the rectangles that cover v and End(v) stores all the rectangles that end in Π_v. Note that any rectangle in S can belong to at most $2\lceil t \rceil + 2$ of the End(v) lists and no more than $(2\lceil t \rceil + 2)\, n^{1/t}$ of the Cover(v) lists.

We partition each Π_v slab into horizontal *strips*, whose vertical boundaries are delimited by $\mathscr{L}(\Pi_v)$ and $\mathscr{R}(\Pi_v)$, respectively, and whose horizontal boundaries are delimited by horizontal lines passing through two consecutive y-coordinates in a y-sorted listing of the horizontal boundaries of the rectangles in Cover(v) $\cup$ End(v). We let Strip(v) denote the list of horizontal strips so-constructed for Π_v.

We also define two lists, Up(h) and Down(h), for each horizontal strip h in Strip(v), as follows:

- Up(h) is the set of horizontal strips h' such that h' is in Strip(z) and h' intersects h, where z is the parent of v in T;
- Down(h) is the set of horizontal strips h' such that h' is in Strip(w) for some child w of v and h' intersects h.

Note that $a \in$ Down(b) if and only if $b \in$ Up(a). Let $Y(h)$ denote the (interval) projection of a horizontal strip h onto the y-axis. The following lemma establishes an important relationship between $Y(h)$ and $Y(h')$, where $h \in$ Down(h'), respectively.

LEMMA 2.1. *Let h be a strip such that $h \in$ Down(h') and $h \cap h' \neq \varnothing$. Then $Y(h') \subseteq Y(h)$.*

Proof. Let v and z be the nodes in T such that $h \in \text{Strip}(v)$ and $h' \in \text{Strip}(z)$. Thus, z is the parent of v. Since the strips in Strip(z) are built on consecutive y-coordinates in a y-sorting of the horizontal boundaries of the rectangles in $\text{Cover}(z) \cup \text{End}(z)$ (and similarly for v), it suffices to show that $\text{Cover}(v) \cup \text{End}(v) \subseteq \text{Cover}(z) \cup \text{End}(z)$. By the definition of Π_z, $\text{End}(v) \subseteq \text{End}(z)$, since $\Pi_v \subset \Pi_z$. By the definition of Cover(v), each rectangle in Cover(v) does not span Π_z; hence, each rectangle in Cover(v) has a vertical boundary in Π_z. Thus, $\text{Cover}(v) \subseteq \text{End}(z)$. ∎

COROLLARY 2.2. *Let h be a strip such that $h \in \text{Up}(h')$ and $h \cap h' \neq \emptyset$. Then $Y(h) \subseteq Y(h')$.*

We call the property defined by Lemma 2.1, and its corollary, the *enclosure* property of the strips in the hive tree. (See Fig. 3.) Viewed another way, if v is a child of z, then constructing Strip(z) involves extending the horizontal boundaries of strips in Strip(v) to be horizontal boundaries in Strip(z) as well. This extending of boundaries is reminiscent of segment extensions used by Chazelle [6] in his hive graph structure, and motivates the name, *hive tree*, for our structure.

Algorithmically, Lemma 2.1 implies that constructing the Up(h) and Down(h) lists will increase the space complexity of the data structure by at most a factor of $n^{1/t}$. We assume that Up and Down lists are represented as doubly linked lists, and are augmented with extra pointers so that for

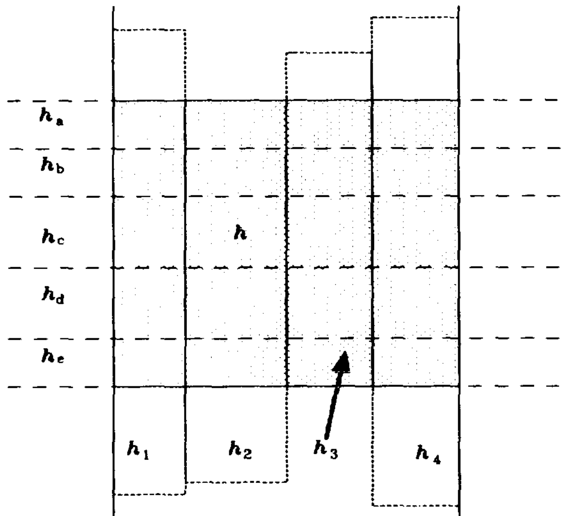

FIG. 3. Illustrating the enclosure property: h is the shaded region, $\text{Up}(h) = h_a, h_b, h_c, h_d, h_e$, and $\text{Down}(h) = h_1, h_2, h_3, h_4$.

each (h, h') pair with $h \in \mathrm{Up}(h')$ we have symmetric pointers between the copy of h in $\mathrm{Up}(h')$ and the copy of h' in $\mathrm{Down}(h)$.

Before we show how we use the hive tree for hidden-line and hidden-surface elimination, let us briefly outline how to efficiently construct a hive tree. As shown in [3] it is fairly straightforward to determine for each rectangle R all the nodes in T that R covers or ends in. This takes $O(tn^{1/t})$ time for each R, or $O(tn^{1+1/t})$ time overall (since we are using an $n^{1/t}$-ary tree instead of a binary tree). Thus we can construct all the $\mathrm{Cover}(v)$ and $\mathrm{End}(v)$ lists in $O(tn^{1+1/t})$ time. As for the Strip lists (and the associated Up and Down lists), note that, by the enclosure property, the y-coordinates of the boundaries of the strips in $\mathrm{Strip}(v)$ are a subset of the y-coordinates of the boundaries of the strips in $\mathrm{Strip}(z)$, where z is v's parent. Our method, then, is to construct the $\mathrm{Strip}(r)$ list for the root node, r. This takes $O(n \log n)$ time (to sort all the y-coordinates). Then, we copy out (in order) the boundaries that are also in each of $\mathrm{Strip}(v_1)$, $\mathrm{Strip}(v_2)$, ..., $\mathrm{Strip}(v_{n^{1/t}})$, in turn, where $v_1, v_2, ..., v_{n^{1/t}}$ are the children of r. Given the lists $\mathrm{Cover}(v_i)$ and $\mathrm{End}(v_i)$ already constructed for each v_i, this is easy to do in $O(|\mathrm{Strip}(r)|)$ time for each v_i. Repeating this recursively, for $v_1, ..., v_{n^{1/t}}$, constructs all the Strip lists in D. While we are copying out the strips from the Strip lists for a node, v, to one of its children, v_i, it is a straightforward addition to also be constructing the Up lists for the strips in $\mathrm{Strip}(v_i)$ and adding the $\mathrm{Strip}(v_i)$ strips to the Down lists for the strips in $\mathrm{Strip}(v)$. In addition, while we are building these lists we also build a list $\mathrm{CoverStrips}(R)$ for each rectangle R that contains a pointer to each strip h in D such that h is in $\mathrm{Strip}(v)$, R is in $\mathrm{Cover}(v)$, and a horizontal boundary of R contains a horizontal boundary of h. Since each recursive call takes $O(|\mathrm{Strip}(v)|\, n^{1/t})$ time plus the time for the smaller recursive calls, the total time for this construction is $O(n \log n + tn^{1+2/t})$. Thus, we have the following lemma:

LEMMA 2.3. *Given a collection S of n isothetic rectangles in the plane, one can construct a hive tree for S in $O(tn^{1+2/t})$ time, where $2 \leq t \leq \log n$ is a tunable parameter.*

Proof. $n \log n$ is $O(tn^{1+2/t})$ for $2 \leq t \leq \log n$. ∎

In the next section we show how to use the hive tree to solve the hidden-line elimination problem for isothetic rectangles.

3. RECTILINEAR HIDDEN-LINE ELIMINATION

Suppose we are given a collection, S, of n isothetic rectangles in $\Re^3$. In this section we show how to construct $\mathrm{Hid}(S)$. For simplicity of expression

in the description that follows we assume that no two horizontal (resp., vertical) boundaries have the same y-coordinate (resp., x-coordinate). It is straightforward to modify our algorithm for the more general case, as this only adds a number of trivial special cases to various steps in our method.

As mentioned above, the main idea of our algorithm is to sweep through the collection of rectangles from front to back with a plane parallel to the xy-plane, maintaining the shadow of all the rectangles encountered as we go. (The shadow of a collection of rectangles is the union of their projections of the xy-plane.) We use a hive tree, constructed on the projection of the rectangles in S, to maintain the shadow of the rectangles in the subset $S' \subseteq S$ of rectangles encountered so far by the sweep. In particular, there are two operations that we support:

- **v-query**(R): Given a rectangle $R \in S - S'$, determine all the intersections R has with vertical edges in the shadow of the rectangles in S'. This operation also identifies which corner points of R (in any) are not obscured by the shadow. (See Fig. 4a.)
- **add**(R), update D so as to represent the shadow of $S' \cup \{R\}$, and assign $S' := S' \cup \{R\}$. (See Fig. 4b.)

We sort the rectangles in S by decreasing z-coordinates and **add** the rectangles in S to S', one by one, in this order. Just before adding a rectangle R to S' we perform a **v-query** for R. Since we add the rectangles to S' in order by their z-coordinates, any intersections a rectangle R has with the shadow of the rectangles in S' (at that time) must all be part of the hidden-surface map for S. In fact, these are all the horizontal dead ends in Hid(S) that are determined by R. In addition, a **v-query** for a rectangle R tells us whether each corner point p of R is visible or not. Thus, this space-sweep gives us all the corner points and horizontal dead ends (i.e., points of the form $\vdash$ or $\dashv$) in Hid(S). We then repeat this same space-sweep one more

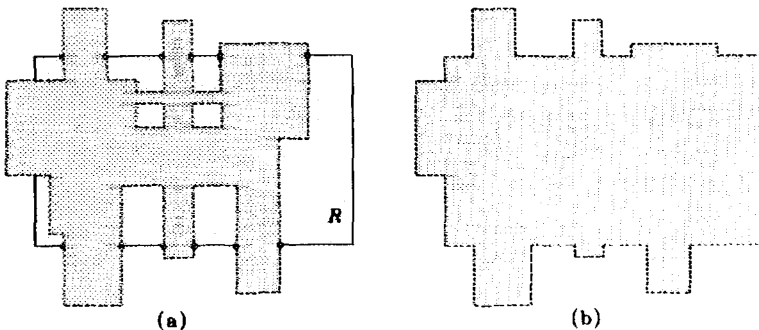

FIG. 4. The two shadow operations: (a) **v-query**(R); (b) **add**(R).

time, with the roles of the x- and y-axes interchanged (that is, with the hive tree determined by the vertical segments in S), giving us all the vertical dead ends in Hid(S) (i.e., points of the form $\top$ or $\perp$). We focus on the first space-sweep, the second one being similar.

We complete the algorithm by constructing a representation of the Hid(S) (minus edge–face adjacency information) from the corner and intersection points, which are the vertices of Hid(S). This can easily be done by sorting the corner points lexicographically twice—once with the x-coordinate being most significant and once with the y-coordinate being most significant. This allows us to determine for any point p the points immediately adjacent to p in each of the four possible directions. To implement this post-processing step, we can normalize all the x- and y-coordinates to be integers in the range $[1, n]$ and use radix sort to perform the sorting (see [1]). This step takes $O(n \log n + k)$ time.

The remainder of this section, then, is devoted to explaining how to augment the hive tree for shadow maintenance and also how to use this augmented hive tree to perform the operations **v-query**(R) and **add**(R), given S. Given a parameter, t, we show that the running time of our pre-processing step is $O(tn^{1+1/t} \log n + tn^{1+2/t})$, that the running time of any **v-query**(R) operation is $O(t(n^{2/t} + k_R))$, where k_R is the number of answers, and that the amortized running time of any **add**(R) operation is $O(tn^{2/t})$. This shows that the total running time of our method is $O(t(n^{1+1/t} \log n + n^{1+2/t} + k))$, where k is the size of the output. We show in Section 5 how to eliminate the $\log n$ factor in the running time of the pre-processing step.

3.1. *Using the Hive Tree for Shadow Maintenance*

So let T be a hive tree constructed on the projections of the rectangles in S on the xy-plane. In order to use the hive tree for shadow maintenance, we define three states for any strip h in Strip(v) for some node v in T as follows:

- *full*: h is *full* if it is completely obscured by the shadow of the rectangles in S'.
- *open*: h is *open* if it is not full and is not intersected by a vertical boundary of the shadow of the rectangles in S'.
- *touched*: h is *touched* if it is not full but is intersected by a vertical boundary of the shadow of the rectangles in S'.

It should be clear that any strip h will always be in exactly one of these states. Also note that, by the enclosure property, if a strip $h \in$ Strip(v) is open, then any full strip h' that intersects h must span Π_v and $h' \cap \Pi_v$ must be completely contained inside h. Similarly, if a strip $h \in$ Strip(v) is touched, then any full strip h' that intersects h must either span Π_v or intersect both

of the horizontal boundaries of h. Moreover, if such an h' spans Π_v, then $h' \cap \Pi_v$ is completely contained inside h.

To facilitate the searching and updating of the shadow of S', we maintain the following auxiliary structures for quickly differentiating between strips in different states:

- *NFU*(h): for each h in Strip(v) we maintain a doubly linked list, *NFU*(h), which stores all the strips in Up(h) that are not full.
- *TD*(h): for each non-full h in Strip(v) we maintain a doubly linked list, *TD*(h), which stores all the strips in Down(h) that are touched.
- *OD*(h): for each non-full h in Strip(v) we maintain a doubly linked list, *OD*(h), which stores all the strips in Down(h) that are open.

Initially, $NFU(h) = \text{Up}(h)$, $TD(h) = \emptyset$, and $OD(h) = \text{Down}(h)$ for all strips h in T. Thus, each of these lists can easily be constructed prior to the space sweep in the same bounds as all the Up(h) and Down(h) lists.

3.2. *Principal Rectangles*

There is one more auxiliary structure that we add to T to help implement our space sweeping procedure. Its definition is a little more involved that the previous auxiliary structures, however. It is based on the following notion.

DEFINITION. Given a strip h in Strip(v), the rectangle with largest z-coordinate (i.e., the first one to be added), over all rectangles that are in Cover(v) and completely obscure h, is called the *principal rectangle* for h.

Note that a strip h can have at most one principal rectangle, and that it is possible that h has no principal rectangle. The final auxiliary structure we add to T is a list, $P(R)$, for each rectangle R, which is defined as follows:

- $P(R)$: for each rectangle R in S, $P(R)$ stores each strip h such that R is the principal rectangle for h.

We can construct all the $P(R)$ lists as follows.

1. For each v construct a representation, Vis_v, of a solution to the hidden-surface elimination problem for the rectangles in Cover(v), restricted to Π_v. Since all the rectangles in Cover(v) span Π_v, this is essentially equivalent to the problem of computing the upper envelope, in the $\mathscr{L}(\Pi_v)$ plane, of a collection of line segments parallel to the y-axis (the so-called "skyline problem" [15]). This step can easily be implemented, for each v in T, by a mergesort-like divide-and-conquer scheme, where the "merge" step amounts to combining two lists of y-parallel segments in the

yz-plane ordered by y-coordinates while maintaining the segment (piece) with largest z-coordinate. Since each merge can be done in linear time, this computation requires $O(n_v \log n_v)$ time for each v in T, where $n_v = |\text{Cover}(v)|$. Thus, the total time for this step is $O(tn^{1+1/t} \log n)$.

2. For each v merge Vis_v and Strip(v) (as in the mergesort procedure [1]), to assign to each $h \in \text{Strip}(v)$ the rectangle associated with the face in Vis_v that contains h. This is the principal rectangle for h, so add h to the $P(R)$ list for this rectangle. This takes an additional $O(n_v + |\text{Strip}(v)|)$ time for each v; hence, a total of $O(tn^{1+2/t})$ time.

The correctness of the above method follows immediately from the fact that each horizontal boundary of a rectangle in Cover(v) (restricted to Π_v) is also a horizontal boundary of a strip in Strip(v), by definition. Thus, in Step 2 there can be at most one face in Vis_v that contains any h and the rectangle corresponding to this face must be the principal rectangle for h (unless of course this face is assigned the "rectangle at $+\infty$," in which case this h has no principal rectangle).

This completes the description of the data structure, which we call the *augmented hive tree* and denote by D, for maintaining the shadow of S'. We have the following lemma:

LEMMA 3.1. *Given a collection S of n isothetic rectangles in $\Re^3$, one can construct an augmented hive tree, D, for the rectangles in S in $O(tn^{1+1/t} \log n + tn^{1+2/t})$ time, where t is a tunable parameter.*

Proof. The proof follows immediately from the above discussion and Lemma 2.3. ∎

Having described our method for constructing D, let us turn to our method for performing each of the operations **v-query** and **add**. We begin with **v-query**.

3.3. *Performing a Query on the Shadow*

Recall that in the **v-query**(R) operation we wish to determine all the intersections between R's horizontal boundaries and the vertical edges of the shadow, as well as to determine which corner points of R (if any) are not obscured by the shadow. So let s be one of R's horizontal boundaries, say the top one. For each node v that s covers (in the segment tree sense) we locate the horizontal strip h in Strip(v) whose bottom boundary coincides with s (note that h is not obscured by R, since s is the top boundary of R). Since R is in Cover(v) for any such node v, s corresponds to a horizontal boundary between two strips in Strip(v); hence, each such s can be derived by searching through the CoverStrips(R) list for R. Thus, searching through all such h's can be done in $O(tn^{1/t})$ time. If an individual

h from this group is not marked "touched," then s intersects no vertical edges of the shadow boundary in h. Thus, after examining such a strip, we need not perform any more work for it. If, on the other hand, an h is marked "touched," then we must determine all the visible vertical edges of the shadow that are in h—they must all intersect s. We do this by calling the following recursive procedure, passing it s and h.

Search(s, h):
 If h is a bottom-level strip **then**
  Return the (single) vertical boundary cutting through h.
 Else
  Combine all the vertical boundaries returned by calling
  Search(s, h') for each $h' \in TD(h)$.
 End-if
End Search(s, h).

By collecting the answers from all calls of Search(s, h) (i.e., for all h's such that s intersects $h \in \text{Strip}(v)$ and s covers v), we get all the intersections of s with vertical edges of the shadow. Let us analyze how long this takes. There are $O(tn^{1/t})$ nodes v such that s covers v. For each such node we only call Search(s, h) if we know there is an answer in h, i.e., if h is touched. Moreover, we only call Search(s, h') recursively if we know there is an answer in h'. Therefore, since there can be at most t levels of recursion, and we perform the same computation for R's lower horizontal boundary, the total time spent in calls to the Touch procedure is $O(t(n^{1/t} + k_R))$, where k_R is the number of $\vdash$ or $\dashv$ intersection points determined by R in the hidden-surface map.

It is an easy matter to determine also if the four corner points of R are visible or not, within these same time bounds. In particular, we can determine if a corner point p is visible or not as follows. First, locate the leaf v with strip $h \in \text{Strip}(v)$ such that h contains p. Note that h must be the leaf strip associated with one of R's vertical boundaries. If h is full, then p is not visible. If h is not full, then we "march up" the tree from v to the root, testing for each w on this path if the strip $h \in \text{Strip}(w)$ that contains p is full or not. If none of these strips are full, then p is visible. Since this can easily be done in $O(t(n^{1/t}))$ time for each corner point of R, the total time for performing a **v-query**(R) is $O(t(n^{1/t} + k_R))$.

3.4. *Updating the Shadow*

So, having described how to perform a **v-query**(R) operation, let us now describe how to perform an **add**(R) operation. Recall that in this operation we must update D to reflect the adding of R to the subset S', i.e., so that D represents the shadow of the rectangles in $S' \cup \{R\}$. Our method consists of two steps. In the first step we process all the "open" strips in T

that become "touched" by the addition of R, and in the second step we process all the "open" and "touched" strips in T that become "full" by the addition of R.

In the first step we must correctly mark all the "open" strips in T that become "touched" because of the addition of R (i.e., because they are intersected by one of the vertical boundaries of R). We begin by locating in D the two leaves that contain the vertical boundaries of R. Because of our convention of making the parent of each leaf node in T have only one child, there are three strips in the slab for such a leaf (i.e., $|\text{Strip}(v)| = 3$). Moreover, it is the middle strip, h, that contains the vertical boundary of R. If h is marked "full," then we need not update anything for h, for adding R does not change how the shadow intersects h. If, on the other hand, h is "open" (h cannot be "touched" prior to adding R), then we mark h as "touched." This is because the vertical boundary of R can only partially obscure this strip, by our convention of not allowing the dividing lines to contain vertical boundaries. Doing this for each of the two vertical boundaries of R can easily be done in $O(t)$ time.

This is clearly not enough, however, for we must update *all* the strips in D that become "touched" by the addition of R to the subset S'. We perform all of these updates by "climbing" up D, incorporating the effect of adding R. Since we can ignore any strips that are marked "full," for any strip h' we mark as "touched," we need only examine the non-full strips in $\text{Up}(h')$ (i.e., the strips in $NFU(h')$), and mark any that were "open" as "touched." This observation immediately gives us the following recursive procedure. **Touch**(h), for updating all the strips in D that must be marked "touched" by the addition of R. We call **Touch**(h) at most twice, once for each leaf-level non-full strip, h, containing a vertical boundary of R.

Touch(h):
1. **For each** h' in $NFU(h)$ **do**
2. Remove h from $OD(h')$ and add h to $TD(h')$.
3. **If** h' is "open" **then**
4. Mark h' as "touched" and call **Touch**(h').

End-for

End Touch(h).

By a simple inductive argument one can show that, for each strip h that is an argument to the **Touch** procedure, h does not become full, since R cannot completely obscure h, by definition. There are a number of other strips in D that R can completely obscure, however. For this reason, we follow the above step by our second step, where we process all the "open" and "touched" strips in D that become "full" by the addition of R. In particular, we mark as "full" all the non-full strips in $P(R)$. These are all the strips in a $\text{Strip}(v)$ list for which R is the first rectangle added in the sweep

such that R covers v (in the segment tree sense) and R completely obscures h. Note that some of the strips in $P(R)$ may already be marked "full." For example, a strip h in $P(R)$ would become full if all the strips in Down(h) become full (by different rectangles).

As we mark each of the non-full strips h in $P(R)$ as "full" we update any other strips in D that become "full" because of h becoming full. There are two possible ways a strip h' could become full as a result of h becoming full. The first way is that h' belongs to a Down(h) list, where $h \in P(R)$ is the last non-full strip in Up(h'). For example, this situation would arise in the configuration of Fig. 3 should h_c be the last non-full strip in Up(h) and h_c is now being marked "full." The second way a strip h' could become full is that h' belongs to an Up(h) list, where and $h \in P(R)$ is the last non-full strip in Down(h'). For example, this situation would arise in the configuration of Fig. 3 should h_4 be the last non-full strip in Down(h) and h_4 is now being marked "full." Thus, we must update the shadow structure, D, for each previously non-full strip $h \in P(R)$ that we are now marking as "full," by alternately climbing D and descending D to cascade the effects of marking this h as "full." In particular, we do this by calling the following recursive procedures, **FullUp**(h) and **FullDown**(h), in turn, for each previously non-full $h \in P(R)$. Intuitively, **FullUp**(h) cascades the affect of marking h as "full" up D and **FullDown**(h) cascades the affect of marking h as "full" down D.

```
FullUp(h):
1.  For each h' in NFU(h) do
2.        If h was "open" then Remove h from OD(h').
3.        if h was "touched" then Remove h from TD(h').
4.        If OD(h') ∪ TD(h') = ∅ then
5.          Mark h' as "full" (for it is obscured by the strips in Down(h')).
6.          Call FullUp(h').
          End-if
    End-for
End FullUp(h).
```

Note that in Step 6 we do not also call **FullDown**(h'), for all of the strips in Down(h') are already full. Also note that we have omitted a test for the case when $OD(h') \neq \emptyset$ and the removal of h from $TD(h')$ leaves $TD(h') = \emptyset$. Such a case would require us to mark h' as "open". Fortunately, however, as we will show later, such a situation cannot occur, for once a strip is marked "touched" it remains touched until it becomes full.

Having given our **FullUp** procedure we next give the recursive procedure, **FullDown**, which we use to mark as "full" any strips below each non-full strip h_i that are now full.

FullDown(h):
1. **For each** h' in $OD(h) \cup TD(h)$ **do**
2. Remove h from $NFU(h')$.
3. **If** $NFU(h') = \varnothing$ **then**
4. Mark h' as "full" (for it is obscured by the strips in $Up(h')$).
5. Call **FullDown**(h').

End-if

End-for

End FullDown(h).

Note that in Step 5 we do not also call **FullUp**(h'), for all of the strips in Up(h') are already full. Performing these two procedures on all the h_i's marks as full all the strips in T that were previously non-full and become full by the introduction of the rectangle R.

3.5. *Analyzing the Time Complexity of Shadow Updating*

A crude analysis of the time complexity of performing all the **Touch**, **FullUp**, and **FullDown** calls associated with a single **add**(R) is that each takes at most $O(tn^{1+1/t})$ time. Thus, an upper bound on the time we spend updating the shadow is $O(tn^{2+1/t})$, since we call **add**(R) once for each of the n rectangles in S. This is a significant overestimate, however, for, as we now show, the total time spent performing **add**(R) operations is $O(tn^{1+2/t})$, implying that a single **add**(R) has an amortized running time of $O(tn^{2/t})$.

One of the important factors in our analysis is the observation that once a strip becomes full it remains full for the rest of the computation. We also have a similar property for touched strips: namely, once a strip becomes touched it remains touched until it becomes full. Both of these observations follow from the fact that we never remove rectangles from the collection S' (whose shadow D represents); no operation we perform on D can reduce the portions of any strip that are obscured.

We use these observations to help us account for the work that is done by an operation $\sigma = \textbf{add}(R)$. Let us consider each sub-operation we perform for σ. The first sub-operation we perform is to visit the leaf-level strips for R's two vertical boundaries, marking these regions as "touched" (if they are not already full) and calling the recursive procedure **Touch**(h). For each recursive call of **Touch**(h') let us charge all the work done by this call to the strip h'. The total time required for any call of **Touch**(h'), not counting any recursive calls it generates, is $O(|NFU(h')|)$, for we perform $O(1)$ work for each strip in $NFU(h')$. Since $|NFU(h')| \leqslant |\text{Up}(h')|$, the most we can charge for any single cell, then, is $O(|\text{Up}(h')|)$. Since h' can become "touched" at most once, in the entire space sweep procedure we call **Touch**(h') on a strip h' in D at most once. Thus, the total time we spend on performing **Touch** operations during the sweep is $O(\sum_{h \in D} |\text{Up}(h)|)$. By

Lemma 2.1, any strip h can belong to at most $n^{1/t}$ of the Up(h') lists. Thus, since there are at most $O(tn^{1+1/t})$ strips in D, the total time we spend performing **Touch** operations is $O(tn^{1+2/t})$. Therefore, the amortized time complexity, per **add** operation, for any call to **Touch** is $O(tn^{2/t})$.

The other major sub-procedures we perform for $\sigma = \textbf{add}(R)$ are the **FullUp** and **FullDown** procedures, for marking as "full" all the open and touched strips that R obscures. Recall that we call these procedures for each strip h in a Strip(v) list, provided R covers v, R obscures h, and h is not full (i.e., $h \in P(R)$). Now we may also have considered some strips in $P(R)$ that were previously marked "full." But this is the only $P(R)$ list to which any such h is marked "full." But this is the only $P(R)$ list to which any such h could belong, so we can charge the cost of this $O(1)$-time test to h itself. Also recall that each such h is marked "full" before we call **FullUp**(h) and **FullDown**(h). Moreover, we call **FullUp**(h') or **FullDown**(h') recursively only if h' has just been marked "full" (hence, h' was previously not full). For each call (recursive or otherwise) of **FullUp**(h) or **FullDown**(h), let us charge the work of this call to the strip h. The total time required for the **FullUp** (resp., **FullDown**) call, not counting recursive calls, is at most $O(|\text{Up}(h)|)$ (resp., $O(|\text{Down}(h)|)$). Thus, the total time we spend performing **FullUp** and **FullDown** operations is at most $O(\sum_{h \in D} (|\text{Up}(h)| + |\text{Down}(h)|))$. By an argument similar to that above, this implies that the total time we spend performing these operations is $O(tn^{1+2/t})$. Therefore, the amortized time complexity, per **add** operation, for such a call is $O(tn^{2/t})$. Combining these observations with those made above, we have the following lemma:

LEMMA 3.2. *Given a collection S of n isothetic rectangles in $\Re^3$, and an augmented hive tree for the rectangles in S, one can construct* Hid(S) *in $O(t(n^{1+2/t} + k))$ time, where k is the size of the output and $2 \leqslant t \leqslant \log n$ is a tunable parameter.*

In the next section we show how to extend our method to the hidden-surface elimination problem for a set of rectangles.

4. EXTENDING OUR METHOD TO HIDDEN-SURFACE ELIMINATION

The method of the previous section gave us Hid(S). In this section we show how to adapt our method to give us Vis(S). That is, we extend the method of the previous section to give us not only the graph of visible edges, but also the rectangle that is visible in each face of this graph. We can easily modify our method so as to store with each vertical edge of the shadow the name (and z-coordinate) of the rectangle that determined that

edge (this essentially "comes for free"). Thus, whenever we use the Search procedure to locate vertices of Hid(S) we can actually get some information about Vis(S). In particular, with each horizontal dead end v (i.e., a vertex of the form $\vdash$ or $\dashv$) in Hid(S) we immediately know two of the three visible rectangles that are adjacent to v. In addition, for any visible rectangle corner vertex v, we immediately know one of the two visible rectangles that are adjacent to v (i.e., the rectangle with v as its corner point). The difficulty, then, is to determine the identity of the unknown adjacent visible rectangle. Viewed another way, the problem that remains is to determine the "background" rectangle for v.

The main obstacle to determine the background rectangle R' for a vertex v in Hid(S) is that, in our space-sweep procedure, R' may not be added to the shadow until long after the rectangle that discovered v (i.e., the rectangle R such that v was one of the vertices returned by **v-query**(R)). We can modify our procedure to overcome this obstacle, however.

Our solution is to augment D so as to also store all the vertices of Hid(S) for which we have yet to determine their background rectangle. We call these the *incomplete vertices* in D. Intuitively, our method for maintaining the incomplete vertices is to have the search procedure "leave a trail" in D of the vertices it discovers. We then augment the **FullUp** and **FullDown** procedures to tag each incomplete vertex v they encounter as "complete" and identify v's background as the current rectangle (for which we are performing the **add** operation). We give the details below.

Recall that the Search(s, h) procedure is called on each strip h that the segment s covers (in the segment-tree sense). Also recall that for each strip h' in $TD(h)$ (the touched strips below h) we recursively call Search(s, h'). We now augment the procedure so that when all the recursive calls return we copy all the discovered answers into a list $I(h)$, which will always contain all the incomplete vertices in h. We represent $I(h)$ as a doubly linked list. In addition, for each v in $I(h)$ we store a pointer to the copy of v in $I(h')$, where $h' \in \text{Down}(h)$, and also a pointer from this copy of v to its copy in $I(h)$. This does not alter the time complexity of the Search procedure, for we will store at most t copies of any incomplete vertex and the adding of m new items to an $I(h)$ list can easily be done in $O(m)$ time.

As mentioned above, we also modify the **FullUp** and **FullDown** procedures to tag incomplete vertices that they discover. More precisely, any time we mark a strip h as "full" because of the addition of a rectangle R we immediately search through the list $I(h)$ and tag each vertex v as having R as its background rectangle. In addition, for each v in $I(h)$ we remove all copies of v in D by following the up and down pointers associated with each v in $I(h)$. This takes $O(t)$ time for each v in $I(h)$. At the end of the space-sweep procedure, when all the rectangles in S have been incorporated into the shadow, we tag all the remaining incomplete vertices in D

as having $-\infty$ (i.e., the true background) as their background rectangle. In the lemma below we show that these modifications are sufficient for solving the hidden-surface elimination problem.

LEMMA 4.1. *Given the above modifications, the space sweep algorithm correctly determines the adjacent visible rectangles for each vertex of* Hid(S).

Proof. Suppose there is a vertex v of Hid(S) that is labeled with an incorrect background rectangle R. Let R' be the true background rectangle for v. There are two cases:

Case 1. $z(R') > z(R)$. Then R' is added to the shadow before R. Moreover, since R' is the background rectangle for v, v must be stored as an incomplete vertex in D at the time we add R' to D. By definition, R' contains v (in its projection on the xy-plane). Thus, when we add R' to D we must mark as "full" some strip that contains v. But this strip must contain v in its $I(h)$ list. Therefore, we remove all copies of v in D before R is added ($\rightarrow\leftarrow$).

Case 2. $z(R') < z(R)$. Then R' is added to the shadow after R, and R removed all copies of v before R' was added. But the fact that R' is v's true background vertex implies that v's projection on R' is not obstructed by v's projection on R. Thus, R cannot contain v (in its projection on the xy-plane). But this implies that R cannot obscure any strip that contains v, contradicting the assumption that R removed all copies of v before R' was added ($\rightarrow\leftarrow$).

This completes the proof. ∎

Having established the correctness of our modifications, we have the following lemma:

LEMMA 4.2. *Given a collection S of n isothetic rectangles in $\Re^3$, and an augmented hive tree constructed for the rectangles in S, one can solve the window-rendering problem for S in $O(t(n^{1+2/t}+k))$ time, where k is the size of the output, and $2 \leqslant t \leqslant \log n$ is a tunable parameter.*

Proof. The proof follows immediately from the above description and Lemma 3.2. ∎

Combining this lemma with Lemma 3.1, we have the following theorem:

THEOREM 4.3. *Given a collection S of n isothetic rectangles in $\Re^3$, one can solve the window rendering problem for S in $O(t(n^{1+1/t}\log n + n^{1+2/t}+k))$ time.*

Note that the input-size component of the above running time is a $\log n$ factor from that claimed in the introduction (with $r = t/2$). In the next section we show how to eliminate this $\log n$ factor.

5. Improving the Running Time

In this section we show how to modify the pre-processing for our algorithm to achieve a running time for the entire algorithm of $O(t(n^{1+2/t} + k))$. Recalling the analysis given previously, the only obstacle to achieving this time bound is that of constructing all the $P(R)$ lists, where we compute for each v in D a solution, Vis_v, to the hidden-surface elimination problem for the rectangles in Cover(v), restricted to Π_v. As we show, achieving an improved running time for this step requires the use of more sophisticated techniques than those we have used so far.

5.1. *A Modest Improvement*

We can achieve a modest improvement by noting that we can simplify the problem by normalizing the rectangles so that their z-coordinates fall in the range $[1, n]$ (in a preprocessing step that requires $O(n \log n)$ time). This immediately implies that we can construct all the Vis_v's in $O(n_v \log\log n)$ time by a simple plane-sweeping procedure using the priority queue data structure of van Emde Boas [29, 30], where $n_v = |\text{Cover}(R)|$. In particular, we can sweep the yz-plane from $y = -\infty$ to $y = +\infty$ with a line parallel to the z-axis, maintaining the collection of rectangles "stabbed" by this line. At each rectangle endpoint we perform a **max** operation to determine the visible rectangle at this point, and then perform the appropriate **insert** or **delete** operation to maintain the collection of rectangles stabbed by this line. This is not sufficient for our goals, however, for $\sum_{v \in D} n_v$ is $O(tn^{1+1/t})$; hence, this approach would result in a running time of $O(tn^{1+1/t} \log\log n)$. Thus, we must be more clever in how we construct the Vis_v's.

5.2. *A Coordinated Attack*

Our approach to achieving $O(t(n^{1+2/t}))$ time for the entire pre-processing step is to coordinate the construction of all the Vis_v's, instead of viewing the pre-processing for each v in T as an isolated problem. We also use a $\lceil \log\log n \rceil$-stratification paradigm [7]. Our method is as follows:

0. We begin by normalizing the z-coordinates of the rectangles in S to be integers in the range $[1, n]$. This takes $O(n \log n)$ time [1].

1. We mark each node that is on a level of T that is a multiple of

$\lceil \log \log n \rceil$ as a *super node*, where, to avoid confusion, we use T to denote the underlying ($n^{1/t}$-ary) tree for D. For each super node v, on level i, we let T_v denote the subtree of T rooted at v and having the super nodes at level $i + \lceil \log \log n \rceil$ as its leaves (the root is on level 0).

2. For each super node v, let z be the nearest super node ancestor of v (so v is a leaf in T_z). We construct Vis_Left_Long(v) and Vis_Right_Long(v), where Vis_Left_Long(v) is a representation of the upper envelope in the $\mathscr{L}(\Pi_z)$ plane of the segments formed by intersecting $\mathscr{L}(\Pi_z)$ with the rectangles in End(v), ignoring the rectangles in End(v) that do not intersect $\mathscr{L}(\Pi_z)$. Intuitively Vis_Left_Long(v) is the upper envelope of the "long" rectangles in End(v). Vis_Right_Long(v) is defined similarly. Since the horizontal boundaries of the rectangles in End(v) are given in sorted order in Strip(v), we can extract a y-sorted listing of the boundaries of rectangles in each End(v) in $O(n^{1+1/t})$ time (for all v's). Given these lists we can then construct Vis_Left_Long(v) and Vis_Right_Long(v) in $O(n_v \log \log n)$ time for each v, where n_v is the number of rectangles involved for v, by the plane-sweeping method described above. Since a rectangle R can be involved in at most $t/\lceil \log \log n \rceil$ of these computations, this also takes $O(n^{1+1/t})$ time.

3. For each node v that is not a super node we let z be the nearest super node ancestor of v (so v is an internal node in T_z). We construct Vis_Left_Long(v) and Vis_Right_Long(v), as defined in the previous step. We perform this computation for each z by applying the mergesort-like procedure of Section 3.2 to the solutions already at the leaves of T_z (combining solutions up the tree using a $n^{1/t}$-way merge). Since the height of each T_z is $O(\log \log n)$, and each node in T_z has $n^{1/t}$ children this step takes $O(n_z \log n^{1/t} \log \log n) = O((n_z/t) \log n \log \log n)$ time, where n_z is the number of rectangles which are stored in the leaves of T_z (in Vis_Left_Long(v) and Vis_Right_Long(v) lists) at the beginning of this step. Since a rectangle R can be contained in at most $t/\lceil \log \log n \rceil$ of these (leaf) super node lists, $\sum_z n_z = nt/\lceil \log \log n \rceil$; hence, the total time for this step is $O(n \log n)$.

4. For each node v that is not a super node (hence, has a nearest super node ancestor z), we construct Vis_Cover_Short(v), where Vis_Cover_Short(v) is a representation of the upper envelope (in the $\mathscr{L}(\Pi_v)$ plane) of the segments formed by intersecting $\mathscr{L}(\Pi_v)$ with the rectangles in Cover(v) that have both of their vertical boundaries properly contained in Π_z. This can be done in $O(m_v \log \log n)$ time using the method given above (in Section 5.1), where m_v is the number of rectangles involved for v. Since any rectangle can cover at most $O(n^{1/t} \log \log n)$ nodes in this way, this step can be implemented in $O(n^{1+1/t}(\log \log n)^2)$ time.

5. For each node v we compute Vis_v, the upper envelope (in the $\mathscr{P}(\Pi_v)$ plane) of the segments formed by intersecting $\mathscr{P}(\Pi_v)$ with the rectangles in Cover(v). We do this by initializing Vis_v to be Vis_Cover_Short(v) and iteratively merging the current Vis_v with each upper envelope Vis_Left_Long(w) (resp., Vis_Right_Long(w)) such that w is a sibling of v and w is to the right (resp., left) of v. Since any rectangle that covers v either has both its vertical boundaries in Π_z or has one in a Π_w (where w is a sibling of v) and the other ouside of Π_z, this gives us Vis_v for each v in T. Note that each segment in such a Vis_Left_Long(w) or Vis_Right_Long(w) list will be examined at most $O(n^{1/t})$ times by v. Thus, each segment in a Vis_Left_Long(w) or Vis_Right_Long(w) list will be examined at most $O(n^{2/t})$ times ($O(n^{1/t})$ times for each sibling of w). In addition, each segment in Vis_Cover_Short(v) will be examined at most $O(n^{1/t})$ times (only by v). Any rectangle R can contribute a segment to at most $O(t)$ Vis_Left_Long(w) or Vis_Right_Long(w) lists and at most $O(n^{1/t})$ Vis_Cover_Short(v) lists. Thus, this step takes at most $O(tn^{1+2/t})$ time.

Therefore, we have the following lemma:

LEMMA 5.1. *Given a collection S of n isothetic rectangles in $\Re^3$, one can construct an augmented hive tree, D, for the rectangles in S in $O(tn^{1+2/t})$ time, for $2 \leqslant t \leqslant \log n$.*

Proof. The proof follows from Lemma 2.3 and the fact that $n \log n + n^{1+1/t}(\log \log n)^2$ is $O(tn^{1+2/t})$ for $2 \leqslant t \leqslant \log n$. ∎

Incidentally, a method of Bern [5] and Mehlhorn [17], which was discovered independent of the above method, can also be used with Lemma 2.3 to derive Lemma 5.1. Their method depends on the union-find data structuring techniques of Gabow and Tarjan [10], for they both reduce the skyline problem to an off-line min problem. In any case, having established Lemma 5.1, one immediately has the following theorem:

THEOREM 5.2. *Given a collection S of n isothetic in $\Re^3$, one can solve the window-rendering problem for S in $O(r(n^{1+1/r} + k))$ time, where k is the size of the output and $1 \leqslant r \leqslant \log n$ is a tunable parameter.*

Proof. Apply Lemmas 5.1 and 4.2, taking $r = t/2$. ∎

5.3. *Tuning the Algorithm "On the Fly"*

Having established a method that can be tuned by a parameter, r, one can use this to derive an improved window-rendering algorithm for all values of k. We give this result as a corollary to Theorem 5.2:

COROLLARY 5.3 (Paterson [22]). *One can solve the window-rendering problem for S in* $O(n \log n + k(\log n/\log(1 + k/n)))$ *time.*

Proof. The method is to iteratively update the value for r on the fly. We run the algorithm with different values of r: the ith time, we use $r = \log n/2^i$ and let the algorithm run for $\tau(n, i)$ time where $\tau(n, i) = c2^{(2^i - i)} n \log n$ and c is a constant (any c will do). As soon as the ith run of the algorithm takes longer than $\tau(n, i)$ time steps, we stop it and launch the $(i+1)$th one (using $r = \log n/2^{i+1}$ and $\tau(n, i+1) = c2^{2^{i+1} - i - 1} n \log n$). Should r ever become equal to 2 (i.e., $i = \log \log n$), then we simply let the algorithm complete (we no longer interrupt it). A straightforward analysis shows that this strategy results in the time bound claimed. ∎

Thus, we can solve the window-rendering problem in time that is both $O((n+k)\log n)$ and $O(n^{1+\varepsilon} + k)$ for any positive constant ε. We leave open the following question: Can one solve the hidden-surface elimination problem for rectangles in $O(n \log n + k)$ time? Such an algorithm would be the best possible for all values of k, for it would optimize both components of the running time.

ACKNOWLEDGMENTS

We thank Michael McKenna for several helpful discussions and S. Rao Kosaraju for his never-ending encouragement.

RECEIVED September 20, 1988; FINAL MANUSCRIPT RECEIVED June 25, 1991

REFERENCES

1. AHO, A. V., HOPCROFT, J. E., AND ULLMAN, J. D. (1974), "The Design and Analysis of Computer Algorithms," Addison–Wesley, Reading, MA.
2. BAUMGART, B. G. (1975), A polyhedron representation for computer vision, *in* "Proceedings, 1975 AFIPS National Computer Conference, 44," AFIPS Press, pp. 589–596.
3. BENTLEY, J. L., AND WOOD, D. (1980), An optimal worst case algorithm for reporting intersections of rectangles, *IEEE Trans. Comput.* **C-29**, 571–577.
4. BERN, M. (1988), Hidden surface removal for rectangles, *in* "Proceedings, 4th ACM Symposium on Computational Geometry," pp. 183–192.
5. BERN, M. Hidden surface removal for rectangles, manuscript (an improved version of [4]).
6. CHAZELLE, B. (1986), Filtering search: A new approach to query-answering, *SIAM J. Comput.* **15**, 703–724.
7. CHAZELLE, B. (1984), Intersecting is easier than sorting, *in* "16th ACM Symposium on Theory of Computing," pp. 125–134.
8. DÉVAI, F. (1986), Quadratic bounds for hidden-line elimination, *in* "Proceedings, 2nd ACM Symposium on Computational Geometry," pp. 269–275.

9. DOH, J. I. to appear, Visibility problems for orthogonal objects in two- or three-dimensions, *Visual Computer*.
10. GABOW, H. N., AND TARJAN, R. E. (1983), A linear-time algorithm for a special case of disjoint set union, *in* "15th ACM Symposium on Theory of Computing," pp. 246–251.
11. GOODRICH, M. T. (1987), A polygonal approach to hidden-line elimination, *in* "Proceedings of 25th Annual Allerton Conference on Communication, Control, and Computing," pp. 849–858.
12. GUIBAS, L. J., AND STOLFI, J. (1985), Primitives for the manipulation of general subdivisions and the computation of Voronoi diagrams, *ACM Trans. Graphics* **4**, 75–123.
13. GÜTING, R. H., AND OTTMANN, T. (1987), New algorithms for special cases of the hidden line elimination problem, *Comput. Vision Graphics Image Process.* **40**, 188–204.
14. LARMORE, L. (1989), An optimal query-update structure for the interval valuation problem, manuscript.
15. MANBER, U. (1989), "Introduction to Algorithms: A Creative Approach," Addison-Wesley, Reading, MA.
16. MCKENNA, M. (1987), Worst-case optimal hidden-surface removal, *ACM Trans. Graphics* **6**, 19–28.
17. MEHLHORN, K. (1989), private communication.
18. MULLER, D. E., AND PREPARATA, F. P. (1978), Finding the intersection of two convex polyhedra, *Theoret. Comput. Sci.* **7**, 217–236.
19. NURMI, O. (1985), A fast line-sweep algorithm for hidden line elimination, *BIT* **25**, 466–472.
20. OTTMANN, T., AND WIDMAYER, P. (1984), Solving visibility problems by using skeleton structures, *in* "Proceedings, 11th Symposium on Mathematical Foundations of Computer Science," pp. 459–470.
21. OVERMARS, M. H., AND SHARIR, M. (1989), Output-sensitive hidden surface removal, *in* "Proceedings, 30th IEEE Symposium on Foundations of Computer Science," in press.
22. PATERSON, M. (1989), private communication.
23. PREPARATA, F. P., AND SHAMOS, M. I. (1985), "Computational Geometry: An Introduction," Springer-Verlag, New York.
24. PREPARATA, F. P., VITTER, J. S., AND YVINEC, M. (1988), "Computation of the Axial View of a Set of Isothetic Parallelepipeds," Laboratoire d'Informatique de L'Ecole Normal Supérieure, Départment de Mathématiques et d'Informatique, Report LIENS-88-1.
25. SCHMITT, A. (1981), "On the Time and Space Complexity of Certain Exact Hidden Line Algorithms," Universität Karlsruhe, Fakultät für Informatik, Report 24/81.
26. SCHMITT, A. (1981), Time and space bounds for hidden line and hidden surface algorithms, *in* "EUROGRAPHICS '81," pp. 43–56.
27. SECHREST, S., AND GREENBERG, D. P. (1982), A visibility polygon reconstruction algorithm, *ACM Trans. Graphics* **1**, 25–42.
28. SUTHERLAND, I. E., SPROULL, R. F., AND SCHUMACKER, R. A. (1974), A characterization of ten hidden-surface algorithms, *Comput. Surv.* **6**, 1–25.
29. VAN EMDE BOAS, P. (1977), Preserving order in a forest in less than logarithmic time and linear space, *Inform. Process. Lett.* **6**, 80–82.
30. VAN EMDE BOAS, P., KAAS, R., AND ZIJLSTRA, E. (1977), Design and implementation of an efficient priority queue, *Math. Systems Theory* **10**, 99–127.