An Input-Size/Output-Size Trade-Off in the Time-Complexity of Rectilinear Hidden Surface Removal

M.T. Goodrich, M.J. Atallah, M.H. Overmars

RUU-CS-89-24
November 1989

University of Utrecht

Department of Computer Science

Padualaan 14, P.O. Box 80.089,
3508 TB Utrecht, The Netherlands,
Tel. : ... + 31 - 30 - 531454

An Input-Size/Output-Size Trade-Off in the Time-Complexity of Rectilinear Hidden Surface Removal

M.T. Goodrich, M.J. Atallah, M.H. Overmars

Technical Report RUU-CS-89-24
November 1989

Department of Computer Science
University of Utrecht
P.O.Box 80.089
3508 TB Utrecht
the Netherlands

An Input-Size/Output-Size Trade-Off in the Time-Complexity of Rectilinear Hidden Surface Removal

Michael T. Goodrich*

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

Mikhail J. Atallah†

Department of Computer Science
Purdue University
West Lafayette, Indiana 47907

Mark H. Overmars‡

Department of Computer Science
University of Utrecht
P.O. Box 80.089
3508 TB Utrecht, The Netherlands

Abstract

We present an algorithm for the well-known hidden-surface elimination problem for rectangles, which is also known as the window rendering problem. The time complexity of our algorithm is dependent on both the number of input rectangles, n, and on the size of the output, k. This is significant, for k is $\Theta(n^2)$ in the worst case, but for most problem instances k is much smaller than this (k can be $O(1)$ in some cases).

*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, and 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 No. 3075 (project ALCOM).

Our algorithm allows one to specify a trade-off between these two components of the running time with a parameter, t. When t is set to minimize the input-size component of the time complexity, then our algorithm runs in $O(n \log n + k \log n)$ time. This improves on the previous best known window-rendering algorithm. On the other hand, when t is set to minimize the output-size component of the time complexity, then our algorithm runs in $O(n^{1+\epsilon} + k)$ time, for any positive constant ϵ. In this case, our algorithm is the first output-sensitive window-rendering algorithm whose running time is $O(f(n) + k)$, where $f(n)$ is $o(n^2)$.

1 Introduction

The hidden-surface elimination problem is well known in computer graphics and computational geometry [6, 10, 14, 16, 17, 18, 21, 22, 23, 24]. 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 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 work-station. (See Figure 1.) Another situation where one often wishes to render such a collection of rectangles is in drafting software, where any time a rectangle R_1 is created, by the draftsman, before rectangle R_2 is created, then R_1 is "behind" R_2, unless the draftsman explicitly changes this ordering (e.g., by executing a "move to front" command on R_1 or, equivalently, a "send to back" command on R_2).

Using the terminology of [24], we are interested in the *object space* version of this problem. That is, we want a method that produces a device-independent, mathematically-based representation of the visible surfaces. One reason for our interest in an object space solution is that such a solution is not dependent on any specific method for rendering polygons nor on the number of pixels on a display screen (which seems to grow with each passing year). In addition, an object space

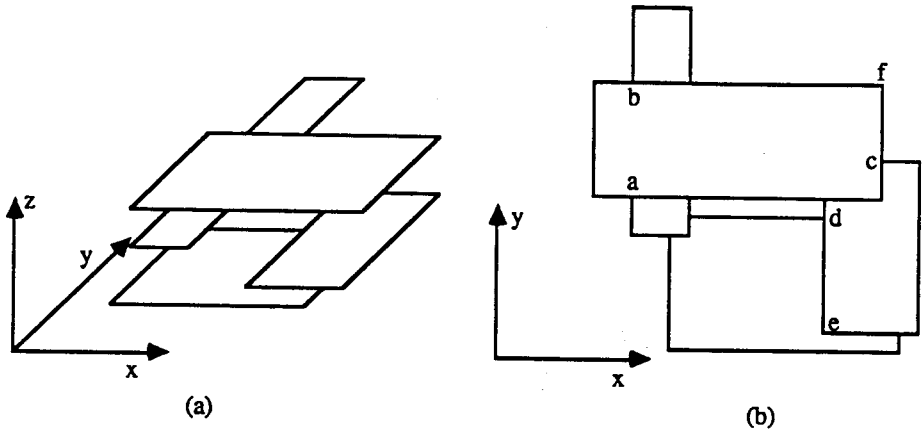

Figure 1: (a) isothetic rectangles; (b) their visible portion.

solution gives us a representation that is easily scaled and rotated.

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 [14] McKenna shows how to solve the general hidden-surface elimination problem in $O(n^2)$ time, generalizing an algorithm by Dévai [6] for the easier hidden-line elimination 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.) Both of these algorithms are worst-case optimal, because there are problem instances that have $\Theta(n^2)$ output size (e.g., a collection of rectangles that form a cross hatched pattern, as in Figure 2a.) Unfortunately, these algorithms always take $O(n^2)$ time [6, 14], even if the size of the output is very small. There are algorithms that run faster than $O(n^2)$ for certain problem instances, however. We review these next.

In [16] Nurmi gives an algorithm for general hidden-line elimination that runs in $O((n+I)\log n)$ time and $O((n+I)\log n)$ space, where I is the number of pairs of line segments whose projections on π intersect (I is $O(n^2)$). Schmitt [21] is able to achieve this same time bound for hidden-surface elimination using only $O(n+I)$ space. 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 [10] Güting and Ottmann address the window rendering problem (they are

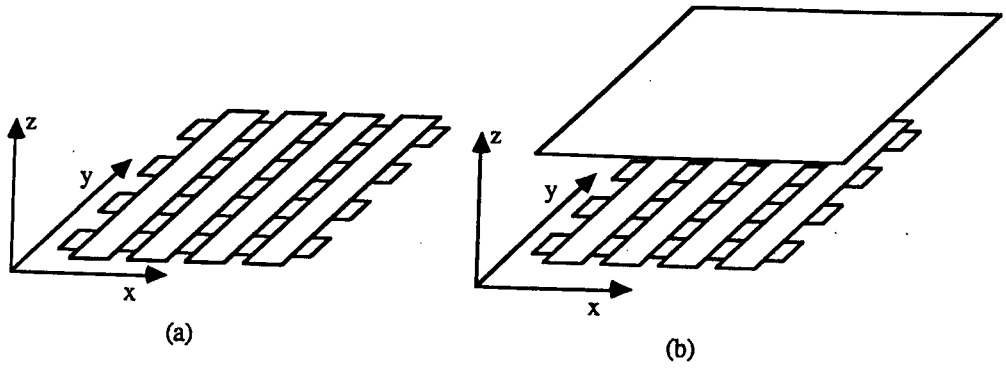

Figure 2: (a) Quadratic output size; (b) Small output size with quadratic I.

probably the first to study this important special case of hidden-surface elimination), giving an algorithm that runs in $O(n \log^2 n + I)$ time. In [8] Goodrich shows how to solve the window rendering problem in $O(n \log n \log \log n + I)$ time by reducing the window-rendering problem to an interval valuation problem [11]. Recently, Larmore [12] gave an improved algorithm for the interval valuation problem, which can be used to implement Goodrich's window-rendering algorithm in $O(n \log n + I)$ time. Recently, Doh [7] has also shown how to solve the window-rendering problem in $O(n \log n + I)$ time, using a different approach. All of these algorithms are optimal in the worst case and also take advantage of problem instances that are "simpler" than in the worst case, but they are not truly output-sensitive. Indeed, there are problem instances where these algorithms run in $O(n^2)$ time even though the output size is very small. For example, in the case where a large rectangle obscures a collection of cross hatched rectangles, as in Figure 2b, I is quadratic although the output size is $O(1)$.

There are methods whose running time depends on both the input size and output size, however. Indeed, in [10] 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 (recall that k is at worst $\Theta(n^2)$). Bern [4] and Preparata, Vitter, and Yvinec [20] 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. Also, in [18] Overmars and Sharir give an output-sensitive algorithm for the general hidden-surface elimination problem that runs in $O(n\sqrt{k} \log n)$ time

(which immediately implies a solution to the window-rendering problem). 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.* It is straightforward to show that in the time-complexity of any window rendering algorithm the input-size component must be $\Omega(n \log n)$ and the output-size component must be $\Omega(k)$. Thus, none of these algorithms have optimal input-size or output-size components.

In this paper we give an algorithm for the window rendering problem whose running time depends on both the input size and output size. Our algorithm allows one to specify the trade-off between these two components of the running time with a parameter, t. When t is set to minimize the input-size component of the time complexity, then our algorithm runs in $O(n \log n + k \log n)$ time, which optimizes the input-size component of the time-complexity while matching the output-size component in the time-complexities of the previous best known algorithms. On the other hand, when t is set to minimize the output-size component of the time complexity, then our algorithm runs in $O(n^{1+\epsilon} + k)$ time, for any positive constant ϵ, which optimizes the output-size component of the time-complexity while achieving an input-size component that is $o(n^2)$.

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. 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.

This approach is similar to that used by Güting and Ottmann [10] and Preparata, Vitter, and Yvinec [20]. We achieve an improved running time over these previous implementations by using a data structure that we call the *hive tree.* Conceptually, the hive tree is a combination of the hive graph structure of Chazelle [5] and the segment tree structure of Bentley and Wood [3]. To solve the rectilinear hidden-surface elimination problem we augment the hive tree with a number of supporting auxiliary structures, where each structure is implemented with the most simple data structures: arrays and linked lists. Thus, our method should be fairly easy to program.

Our algorithm consists of two phases: (1) preparing for the sweep and (2) implementing the sweep. In order to concentrate on the primary structure of our algorithm, we begin by describing (in Section 3) a simple, but less-efficient, method for performing Phase 1, as well as describing a simple method for performing Phase 2, but for the simpler hidden-line elimination problem. These two phases combine to give a running time of $O(t(n^{1+1/t}\log n + n^{1+2/t} + k))$ for the hidden-line elimination problem for rectangles, where $2 \leq t \leq \log n$. Then, in Section 4 we show how to adapt our method to solve the harder hidden-surface elimination problem in this same bound by modifying the second phase of our algorithm. In Section 5 we show how to improve the running time of Phase 1 (the bottleneck phase), resulting in a running time that is $O(t(n^{1+2/t} + k))$ for rectilinear hidden-surface elimination. We begin our discussion, in the next section, by formally defining what constitutes a solution to the hidden-surface elimination problem for rectangles, and describing the hive tree structure.

2 Preliminaries

Suppose we are given a collection S of n non-intersecting isothetic rectangles in $\Re^3$, i.e., a collection of rectangles parallel to the xy-plane such that all edges are parallel to either the x- or 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.

2.1 Representing a Solution to the Window-Rendering Problem

There are many ways that one can specify what constitutes a solution to the hidden-surface elimination problem [10, 14, 17, 22, 23, 24]. 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. A solution to the hidden-surface elimination problem is given by $Hid(S)$, augmented so that each polygonal face of $Hid(S)$ stores the name of the rectangle of S that is visible in that face. We let $Vis(S)$ denote such an augmented graph for S, and use $Vis(S)$ to denote a solution to the hidden-surface elimination problem for S.

In order to better motivate our hidden-surface method, let us examine the structure of $Hid(S)$ and $Vis(S)$ more closely. For each vertex v of $Hid(S)$ (and $Vis(S)$) 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 $\top$, $\perp$, $\vdash$, 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 $\perp$), and *horizontal dead ends*, where the terminating segment is horizontal (i.e., $\vdash$ or $\dashv$). In Figure 1b, points e and f are corners, a is a $\top$, b is a $\perp$, c is a $\vdash$, and d is a $\dashv$. In that same figure, points a, b, c and d are dead ends: a and b are vertical dead ends, while c and d are horizontal dead ends. For convenience, we assume throughout the paper that the planar graph $Hid(S)$ lies in the xy-plane, so that any face of $Hid(S)$ is also in the xy-plane. Of course, for each such face in $Vis(S)$ we also know the name of the rectangle of S that is visible in it, and the z-coordinate of that rectangle.

There are a number of ways one can represent an embedded planar graph, such as $Hid(S)$ or $Vis(S)$. Three such representations are the "winged edge" structure of Baumgart [2], the "quad edge" structure of Guibas and Stolfi [9], and the "doubly-connected edge list" structure of Muller and Preparata [15, 19]. Our algorithm does not depend on which representation one chooses, so long as the representation allows one to determine each of the following in time proportional to its size:

1. all edges and faces adjacent to a given vertex v, as well as their orientation with respect to v,

2. all vertices and faces adjacent to a given edge e, as well as their orientation with respect to e, and

3. all vertices and edges that lie on the boundary of a given face f, in the order they occur around f.

Each of the mentioned representations provides this.

Given an isothetic rectangle R in $\Re^3$ we let $z(R)$ denote the z-coordinate of the plane to which R belongs. Similarly, for any point p in $\Re^3$, we use $x(p)$, $y(p)$, and $z(p)$ to denote the x-, y-, and z-coordinate of p, respectively. Our terminology implicitly assumes that the observer looking at the scene from $z = \infty$ has his body parallel to the y-axis, with both arms extended so they are parallel to the x-axis (the reader probably inferred this from the way we drew Figure 1b). Hence a *vertical* segment is parallel to the y-axis, whereas a *horizontal* segment is parallel to the x-axis. Similarly, we say that a plane is *vertical* (resp., *horizontal*) if it is parallel to the yz-plane (resp., xz-plane).

Before we give our hidden-line and hidden-surface methods, we describe the primary data structure we use in our algorithm, namely, the hive tree.

2.2 The Hive Tree Data Structure

In this section we define the hive tree data structure, and show how it can be efficiently constructed. It can be defined on a collection of horizontal and vertical segments in the plane, but, since all the horizontal and vertical segments in our case belong to rectangles, we define it on a collection, S, of n rectangles in the plane. We begin by projecting the vertical rectangle boundaries on the x-axis and placing 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". We use Π_v to denote the slab associated with the leaf v. 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 on these slabs in the natural way, so that each leaf is associated with a slab (t is a tunable parameter for the construction). Recall that an $n^{1/t}$-ary tree is a rooted tree such that each internal node has $n^{1/t}$ children (expect possibly nodes with children that are leaves). To simplify computations that we will have to 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.

For 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 $\mathcal{L}(\Pi_v)$ (resp., $\mathcal{R}(\Pi_v)$) denote the left (resp., right) vertical boundary of Π_v. Note that $\mathcal{L}(\Pi_v)$ (resp., $\mathcal{R}(\Pi_v)$) is a plane parallel to the yz-plane, and any rectangle R intersects this plane in a segment parallel to the y-axis.

For each v in T we define two sets, $Cover(v)$ and $End(v)$, which are motivated by similar sets defined for the segment tree data structure of Bentley and Wood [3]. Specifically, $Cover(v)$ stores all the rectangles that span Π_v (i.e., that intersect $\mathcal{L}(\Pi_v)$ and $\mathcal{R}(\Pi_v)$) but do not span Π_z, where z is the parent of v, and $End(v)$ stores all the rectangles that have a vertical boundary inside Π_v. Note that any rectangle in S can belong to at most $2\lceil t \rceil + 2$ of the $End(v)$ sets and no more than $(2\lceil t \rceil + 2)n^{1/t}$ of the $Cover(v)$ sets.

We partition each Π_v slab into horizontal *strips*, whose vertical boundaries are delimited by $\mathcal{L}(\Pi_v)$ and $\mathcal{R}(\Pi_v)$, 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 a $Y(h)$ and $Y(h')$, where $h \in Down(h')$. respectively.

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

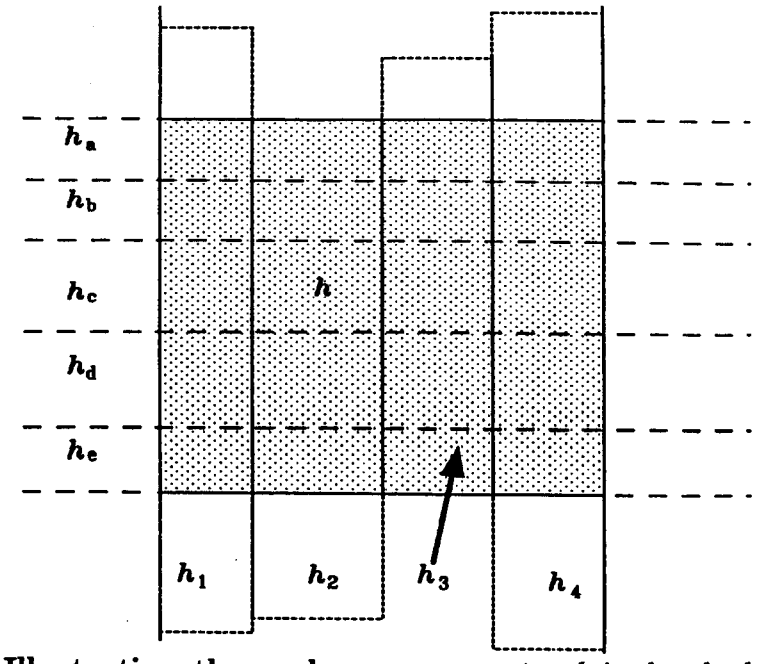

Figure 3: **Illustrating the enclosure property.** h is the shaded region, $Up(h) = \{h_a, h_b, h_c, h_d, h_e\}$, and $Down(h) = \{h_1, h_2, h_3, h_4\}$.

Proof: Let v and z be the nodes in T such that $h \in Strip(v)$ and $h' \in Strip(z)$. 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 $Cover(z) \cup End(z)$ (and similarly for v), it suffices to show that $Cover(v) \cup End(v) \subseteq Cover(z) \cup End(z)$. By the definition of Π_z, $End(v) \subseteq End(z)$, 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, $Cover(v) \subseteq End(z)$. ■

Corollary 2.2: *Let h be a strip such that in $h \in Up(h')$ 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 Figure 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 [5] 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 each (h, h') pair with $h \in Up(h')$ we have symmetric pointers between the copy of h in $Up(h')$ and the copy of h' in $Down(h)$.

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 $Cover(v)$ and $End(v)$ 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 $Strip(v)$ are a subset of the y-coordinates of the boundaries of the strips in $Strip(z)$, where z is v's parent. Our method, then, is to construct the $Strip(r)$ 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 $Strip(v_1), Strip(v_2), ..., Strip(v_{n^{1/t}})$, in turn, where $v_1, v_2, ..., v_{n^{1/t}}$ are the children of r. Given the lists $Cover(v_i)$ and $End(v_i)$ already constructed for each v_i, this is easy to do in $O(|Strip(r)|)$ time for each v_i. Repeating this recursively, for $v_1, ..., v_{n^{1/t}}$, constructs all the *Strip* lists in D. In addition, while we are copying out the strips from the *Strip* list 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 $Strip(v_i)$ and adding the $Strip(v_i)$ strips to the *Down* lists for the strips in $Strip(v)$. Since each recursive call takes $O(|Strip(v)|n^{1/t})$ time plus the 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 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 $Hid(S)$. 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

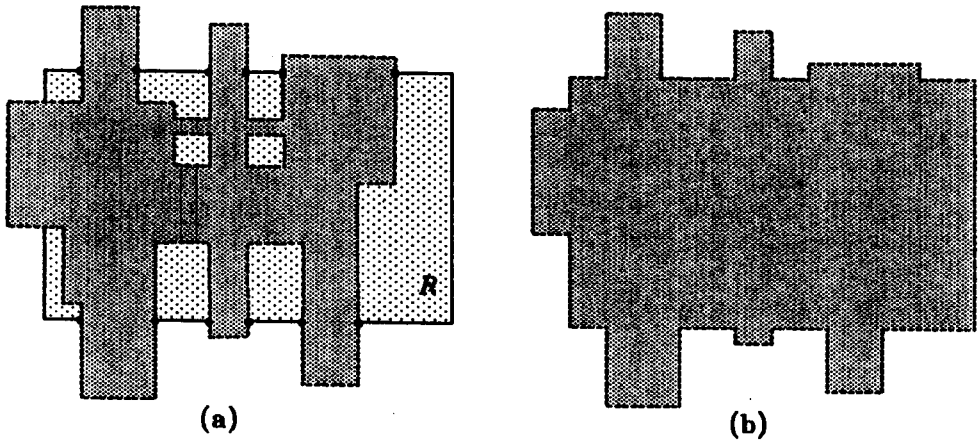

Figure 4: **The two shadow operations.** (a) **v-query**(R); (b) **add**(R)..

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 on 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 (if any) are not obscured by the shadow. (See Figure 4a.)

- **add**(R), update D so as to represent the shadow of $S' \cup \{R\}$ and assign $S' := S' \cup \{R\}$. (See Figure 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 $Hid(S)$ 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 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 $\bot$). 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 these points, 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 4 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 the "radix" sorting method 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 preparation phase 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 will show 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 preparation phase.

3.1 Phase 1: Preparing for the Sweep

As mentioned earlier, the first phase in our algorithm is to construct the augmented hive tree data structure for maintaining the shadow. So let T be a hive tree constructed on the projections of the rectangles in S on the xy-plane. We begin by describing how we augment 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 $h \in \mathit{Strip}(v)$ 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 $h \in \mathit{Strip}(v)$ 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 $\mathit{Strip}(v)$ we maintain a doubly-linked list, $NFU(h)$, which stores all the strips in $\mathit{Up}(h)$ that are not full.
- $TD(h)$: for each non-full h in $\mathit{Strip}(v)$ we maintain a doubly-linked list, $TD(h)$, which stores all the strips in $\mathit{Down}(h)$ that are touched.
- $OD(h)$: for each non-full h in $\mathit{Strip}(v)$ we maintain a doubly-linked list, $OD(h)$, which stores all the strips in $\mathit{Down}(h)$ that are open.

Initially, $NFU(h) = \mathit{Up}(h)$, $TD(h) = \emptyset$, and $OD(h) = \mathit{Down}(h)$ 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 $\mathit{Up}(h)$ and $\mathit{Down}(h)$ lists.

There is one more auxiliary structure that we add to T to help implement the space sweeping of Phase 2. 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 $\mathit{Strip}(v)$, the rectangle with largest z-coordinate (i.e., the first one to be added), over all rectangles that are in $\mathit{Cover}(v)$ and completely obscure h, is called the* principal rectangle *for h.*

Note that a strip h can have at most 1 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 $\mathcal{L}(\Pi_v)$ plane, of a collection of line segments parallel to the y-axis (the so-called "skyline problem" [13]). This step can easily be implemented in $O(n_v \log n_v)$ time for each v in T by a mergesort-like divide-and-conquer scheme, where $n_v = |Cover(v)|$. 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 $h \in Strip(v)$ 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 $O(n_v + |Strip(v)|)$ 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 will denote by D, for maintaining the shadow of S', as well as our method for its construction. We conclude this subsection, then, with the following lemma:

Lemma 3.1: *One can construct and initialize the data structure D, for maintaining the shadow of the rectangles S', in $O(tn^{1+1/t} \log n + tn^{1+2/t})$ time.* ∎

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**.

3.2 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 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 $Strip(v)$, and we could have easily precomputed all strips h in all $Strip(v)$ lists such that R is in $Cover(v)$ and such that R shares a horizontal boundary with h. Thus, searching through all such h's can be done in $O(tn^{1/t})$ time, given this precomputation. 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 $h \in Strip(v)$ 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 will 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 also determine 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 $h \in Strip(v)$ 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 $h \in Strip(w)$ 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 $O(t(n^{1/t} + k_R))$.

3.3 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 essentially 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 2 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 3 strips in the slab for such a leaf (i.e., $|Strip(v)| = 3$). 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 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 $Up(h')$ (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) 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).

Note that in this procedure we do not mark any strips as "full". This is because each bottom-level strip h properly contains the rectangle vertical boundary that defined it. Note that 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 $Strip(v)$ 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 Figure 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 Figure 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** $OD(h') \cup TD(h') = \emptyset$ **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. The proof of this claim is based on showing that 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** $NFU(h') = \emptyset$ **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.

3.4 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 over-estimate, however, for, as we show in this subsection, 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 σ = **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 $|NFU(h')| \leq |Up(h')|$, the most we can charge, then, is $O(|Up(h')|)$. By the previous observation, in the entire space sweep procedure we will 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 $O(\sum_{h \in D} |Up(h)|)$. 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 σ = **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 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 $O(|Up(h)|)$ (resp., $O(|Down(h)|)$). Thus, the total time we spend performing **FullUp** and **FullDown** operations is at most $O(\sum_{h \in D}(|Up(h)| + |Down(h)|))$. 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 theorem:

Theorem 3.2: *Given a collection S of n isothetic rectangle in $\Re^3$, one can construct $Hid(S)$ in $O(t(n^{1+1/t} \log n + n^{1+2/t} + k))$ time, where k is the size of the output and $2 \leq t \leq \log n$ is a tunable parameter.* ∎

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

4 Extending 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 will extend the method of the previous section to not only give us the graph of visible edges, but also the rectangle that is visible in each face of this graph. Recall that in the previous algorithm we swept through the collection of rectangles from front to back, maintaining the shadow as we swept. 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 would immediately know two of the three visible rectangles that are adjacent to v. In addition, for any visible rectangle corner vertex v, we would 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 determining 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)). Thus, there may be no proximity in the z-coordinate between v and its background rectangle. 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 $h' \in Down(h)$, and also a pointer from this copy of v to the 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 $Hid(S)$ which 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 theorem:

Theorem 4.2: *Given a collection S of n isothetic rectangle in $\Re^3$, one can solve the window-rendering problem for S in $O(t(n^{1+1/t}\log n + tn^{1_2/t} + k))$ time, where k is the size of the output and $2 \leq t \leq \log n$ is a tunable parameter.* ∎

By appropriate assignments to the parameter t, then, we immediately get the following corollaries. In the first corollary we use t to optimize the output-size component of the running time.

Corollary 4.3: *One can construct $Vis(S)$ in $O(n^{1+\epsilon} + k)$ for any positive constant $\epsilon \leq 1$.*

Proof: Take $2/t < \epsilon$. ∎

This corollary demonstrates the first output-sensitive window-rendering algorithm whose running time is $O(f(n) + k)$, where $f(n)$ is $o(n^2)$. The next corollary shows how one can alternately trade this off with the input-size component of the running time.

Corollary 4.4: *One can construct $Vis(S)$ in $O(n \log^2 n + k \log n / \log \log n)$ time.*

Proof: Take $t = \log n / \log \log n$, so $n^{1/t} = \log n$. ∎

This latter result matches the input-size component of Güting and Ottmann [10] and Preparata, Vitter, and Yvinec [20], and improves the output-size component of the running times of methods by Preparata, Vitter, and Yvinec [20] and Bern [4]. It does not improve input-size component of the running time of Bern's algorithm, however. We can further improve our procedure, however, to optimize the input-size component of the running time, without increasing the output-size component. We describe the modifications necessary for achieving this in the next section.

5 Optimizing the Input-Size Component

In this section we show how to modify the first phase of our algorithm to achieve a running time for the entire algorithm of $O(t(n^{1+2/t} + k))$. Taking $t = \log n$ implies a running time of $O((n + k) \log n)$ for the window-rendering problem, which improves the best previous method by Bern [4].

There is one primary bottleneck to achieving an $O(n \log n)$ input-size component in our running time, and that is in the construction of 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. We would like to bring this from $O(tn^{1/t} \log n)$ to $O(tn^{1+2/t})$. Doing this requires the use of more complicated data

structures than those we have used so far. Thus, the discussion of this section may be of more theoretical than practical interest.

In the description that follows let us concentrate on a particular node v in D. Recall that since each rectangle in $Cover(v)$ spans Π_v, the computation of Vis_v is equivalent to a 2-dimensional visibility problem. Namely, it is equivalent to determining the upper envelope of a set of segments parallel to the y-axis in the yz-plane. Also note that we can normalize 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 [25, 26], where $n_v = |Cover(R)|$. 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 **min** 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. But this is not efficient enough, however, for $\sum_{v \in D} n_v$ is $O(tn^{1+1/t})$; hence, this approach would result in a running time of $O(tn^{1+1/t} \log\log n)$ for Phase 1. Thus, we must be more clever in how we construct the Vis_v's. Our method, then, is as follows.

1. We mark each node that is on a level of T which is a multiple of $\log\log n$ 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 $i + \log\log n$ 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 $\mathcal{L}(\Pi_z)$ plane of the segments formed by intersecting $\mathcal{L}(\Pi_z)$ with the rectangles in $End(v)$, ignoring the rectangles in $End(v)$ that do not intersect $\mathcal{L}(\Pi_z)$. 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 $t/\log\log n$ 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 merging the solutions already at the leaves of T_z in a pair-wise fashion up T_z. 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 $O(n_z \log n^{1/t} \log\log n) = O((n_z/t)\log n \log\log n)$, 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 $t/\log\log n$ of these (leaf) super node lists, $\sum_z n_z = nt/\log\log n$; 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 $\mathcal{L}(\Pi_v)$ plane) of the segments formed by intersecting $\mathcal{L}(\Pi_v)$ 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 by a simple "plane-sweeping" procedure, using the priority queue data structure of van Emde Boas [25, 26], where m_v is the number of rectangles involved for v. We can use this data structure for all the operations in this sweep, because each sweep operation can be implemented using $O(1)$ $insert(i)$, $delete(i)$, and/or max operations, where $i \in [1,n]$ (assuming the rectangle z-coordinates are normalized to integers in the range $[1,n]$). 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 $\mathcal{L}(\Pi_v)$ plane) of the segments formed by intersecting $\mathcal{L}(\Pi_v)$ 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 outside 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: *One can construct all the $P(R)$ lists in $O(tn^{1+2/t})$ time.*

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

This immediately gives us the following theorem:

Theorem 5.2: *Given a collection S of n isothetic rectangle in $\Re^3$, one can construct $Vis(S)$ in $O(t(n^{1+2/t} + k))$ time, where k is the size of the output and $2 \leq t \leq \log n$ is a tunable parameter.* ∎

Corollary 5.3: *One can construct $Vis(S)$ in $O((n + k) \log n)$ time.*

Proof: Take $t = \log n$. ∎

Note that this bound matches output-size component of the algorithm [4, 20], but improves the input-size component to $O(n \log n)$, which is optimal.

6 Conclusion

In this paper we have given an algorithm for solving the hidden-surface elimination problem for rectangles. Our method is parameterized by a number, t, which specifies the trade-off between the input-size, n, and output-size, k, in the running time of the algorithm. We can choose t to achieve a time complexity of $O(n^{1+\epsilon} + k)$, for any positive constant ϵ, thus achieving an optimal output-size component, or a time-complexity of $O((n + k) \log n)$, which achieves an optimal input-size component. One of the main ideas in our method is the use of a geometric data structure we called the *hive tree*, which may have applications to other rectilinear problems.

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.

Acknowledgements

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

References

[1] A.V. Aho, J.E. Hopcroft, and J.D. Ullman, *The Design and Analysis of Computer Algorithms*, Addison-Wesley, Reading, Mass., 1974.

[2] B.G. Baumgart, "A Polyhedron Representation for Computer Vision," *Proc. 1975 AFIPS National Computer Conf., 44*, AFIPS Press, 1975, 589–596.

[3] J.L. Bentley and D. Wood, "An Optimal Worst Case Algorithm for Reporting Intersections of Rectangles," *IEEE Trans. on Computers*, Vol. C-29, 1980, 571–577.

[4] M. Bern, "Hidden Surface Removal for Rectangles," *Proc. 4th ACM Symp. on Computational Geometry*, 1988, 183–192.

[5] B. Chazelle, "Filtering Search: A New Approach to Query-Answering," *SIAM J. Comput.*, Vol. 15, 1986, 703–724.

[6] F. Dévai, "Quadratic Bounds for Hidden-Line Elimination," *Proc. 2nd ACM Symp. on Computational Geometry*, 1986, 269–275.

[7] J.I. Doh, "Visibility Problems for Orthogonal Objects in Two- or Three-Dimensions," to appear in *The Visual Computer.*

[8] M.T. Goodrich, "A Polygonal Approach to Hidden-Line Elimination," *Proc. of 25th Annual Allerton Conference on Comm., Control, and Computing*, 1987, 849–858.

[9] L.J. Guibas and J. Stolfi, "Primitives for the Manipulation of General Subdivisions and the Computation of Voronoi Diagrams," *ACM Trans. on Graphics*, Vol. 4, 1985, 75–123.

[10] R.H. Güting and T. Ottmann, "New Algorithms For Special Cases of the Hidden Line Elimination Problem," *Computer Vision, Graphics, and Image Processing*, Vol. 40, 1987, 188–204.

[11] D.S. Hirschberg and D.J. Volper, "Improved Update/Query Algorithms for the Interval Valuation Problem," *Information Processing Letters*, Vol. 24, 1987, 307–310.

[12] L. Larmore, "An Optimal Query-Update Structure for the Interval Valuation Problem," manuscript, 1989.

[13] U. Manber, *Introduction to Algorithms: A Creative Approach*, Addison-Wesley, Reading, Mass., 1989.

[14] M. McKenna, "Worst-case Optimal Hidden-Surface Removal," *ACM Transactions on Graphics*, Vol. 6, 1987, 19–28.

[15] D.E. Muller and F.P. Preparata, "Finding the Intersection of Two Convex Polyhedra," *Theoretical Computer Science*, Vol. 7, 1978, 217–236.

[16] O. Nurmi, "A Fast Line-Sweep Algorithm For Hidden Line Elimination," *BIT*, Vol. 25, 1985, 466–472.

[17] T. Ottmann and P. Widmayer, "Solving Visibility Problems by Using Skeleton Structures," *Proc. 11th Symp. on Math. Foundations of Computer Science*, 1984, 459–470.

[18] M.H. Overmars and M. Sharir, "Output-Sensitive Hidden Surface Removal," *Proc. 30th IEEE Symp. on Foundations of Computer Science, 1989*, in press.

[19] F.P. Preparata and M.I. Shamos, *Computational Geometry: An Introduction*, Springer-Verlag, New York, 1985.

[20] F.P. Preparata, J.S. Vitter, and M. Yvinec, "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, 1988.

[21] A. Schmitt, "On the Time and Space Complexity of Certain Exact Hidden Line Algorithms," Universität Karlsruhe, Fakultät für Informatik, Report 24/81, 1981.

[22] A. Schmitt, "Time and Space Bounds for Hidden Line and Hidden Surface Algorithms," *EUROGRAPHICS '81*, 43–56.

[23] S. Sechrest and D.P. Greenberg, "A Visibility Polygon Reconstruction Algorithm," *ACM Transactions on Graphics*, Vol. 1, 1982, 25–42.

[24] I.E. Sutherland, R.F. Sproull, and R.A. Schumacker, "A Characterization of Ten Hidden-Surface Algorithms," *Computing Surveys*, Vol. 6, 1974, 1–25.

[25] P. van Emde Boas, "Presevering Order in a Forest in Less than Logarithmic Time and Linear Space," *Information Processing Letters*, Vol. 6, 1977, 80–82.

[26] P. van Emde Boas, R. Kaas, and E. Zijlstra, "Design and Implementation of an Efficient Priority Queue," *Math. Systems Theory*, Vol. 10, 1977, 99–127.

An Input-Size/Output-Size Trade-Off in the Time-Complexity of Rectilinear Hidden Surface Removal

M.T. Goodrich, M.J. Atallah, M.H. Overmars

RUU-CS-89-24
November 1989

University of Utrecht

Department of Computer Science

Padualaan 14, P.O. Box 80.089,
3508 TB Utrecht, The Netherlands,
Tel. : ... + 31 - 30 - 531454

An Input-Size/Output-Size Trade-Off in the Time-Complexity of Rectilinear Hidden Surface Removal

M.T. Goodrich, M.J. Atallah, M.H. Overmars

Technical Report RUU-CS-89-24
November 1989

Department of Computer Science
University of Utrecht
P.O.Box 80.089
3508 TB Utrecht
the Netherlands

An Input-Size/Output-Size Trade-Off in the Time-Complexity of Rectilinear Hidden Surface Removal

Michael T. Goodrich*

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

Mikhail J. Atallah†

Department of Computer Science
Purdue University
West Lafayette, Indiana 47907

Mark H. Overmars‡

Department of Computer Science
University of Utrecht
P.O. Box 80.089
3508 TB Utrecht, The Netherlands

Abstract

We present an algorithm for the well-known hidden-surface elimination problem for rectangles, which is also known as the window rendering problem. The time complexity of our algorithm is dependent on both the number of input rectangles, n, and on the size of the output, k. This is significant, for k is $\Theta(n^2)$ in the worst case, but for most problem instances k is much smaller than this (k can be $O(1)$ in some cases).

*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, and 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 No. 3075 (project ALCOM).

Our algorithm allows one to specify a trade-off between these two components of the running time with a parameter, t. When t is set to minimize the input-size component of the time complexity, then our algorithm runs in $O(n \log n + k \log n)$ time. This improves on the previous best known window-rendering algorithm. On the other hand, when t is set to minimize the output-size component of the time complexity, then our algorithm runs in $O(n^{1+\epsilon} + k)$ time, for any positive constant ϵ. In this case, our algorithm is the first output-sensitive window-rendering algorithm whose running time is $O(f(n) + k)$, where $f(n)$ is $o(n^2)$.

1 Introduction

The hidden-surface elimination problem is well known in computer graphics and computational geometry [6, 10, 14, 16, 17, 18, 21, 22, 23, 24]. 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 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 work-station. (See Figure 1.) Another situation where one often wishes to render such a collection of rectangles is in drafting software, where any time a rectangle R_1 is created, by the draftsman, before rectangle R_2 is created, then R_1 is "behind" R_2, unless the draftsman explicitly changes this ordering (e.g., by executing a "move to front" command on R_1 or, equivalently, a "send to back" command on R_2).

Using the terminology of [24], we are interested in the *object space* version of this problem. That is, we want a method that produces a device-independent, mathematically-based representation of the visible surfaces. One reason for our interest in an object space solution is that such a solution is not dependent on any specific method for rendering polygons nor on the number of pixels on a display screen (which seems to grow with each passing year). In addition, an object space

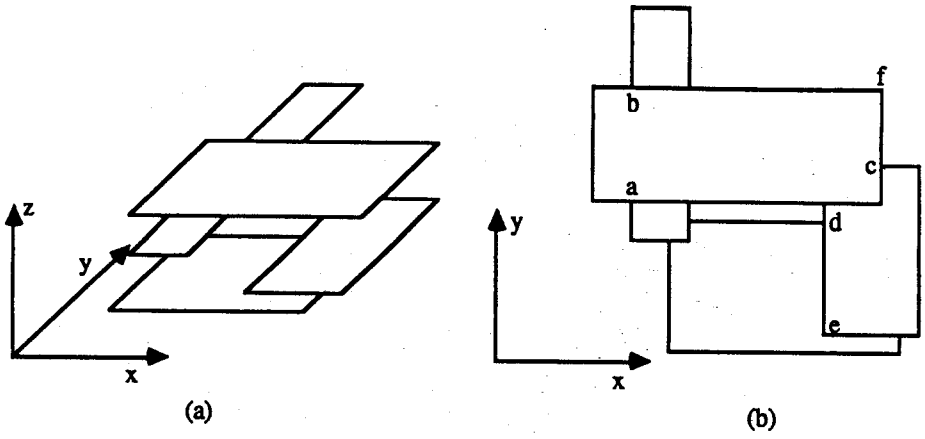

Figure 1: (a) isothetic rectangles; (b) their visible portion.

solution gives us a representation that is easily scaled and rotated.

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 [14] McKenna shows how to solve the general hidden-surface elimination problem in $O(n^2)$ time, generalizing an algorithm by Dévai [6] for the easier hidden-line elimination 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.) Both of these algorithms are worst-case optimal, because there are problem instances that have $\Theta(n^2)$ output size (e.g., a collection of rectangles that form a cross hatched pattern, as in Figure 2a.) Unfortunately, these algorithms always take $O(n^2)$ time [6, 14], even if the size of the output is very small. There are algorithms that run faster than $O(n^2)$ for certain problem instances, however. We review these next.

In [16] Nurmi gives an algorithm for general hidden-line elimination that runs in $O((n+I)\log n)$ time and $O((n+I)\log n)$ space, where I is the number of pairs of line segments whose projections on π intersect (I is $O(n^2)$). Schmitt [21] is able to achieve this same time bound for hidden-surface elimination using only $O(n+I)$ space. 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 [10] Güting and Ottmann address the window rendering problem (they are

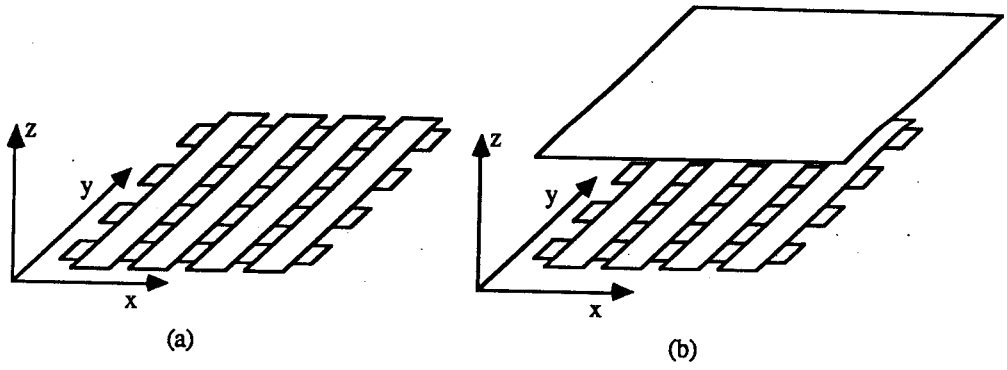

Figure 2: (a) Quadratic output size; (b) Small output size with quadratic I.

probably the first to study this important special case of hidden-surface elimination), giving an algorithm that runs in $O(n \log^2 n + I)$ time. In [8] Goodrich shows how to solve the window rendering problem in $O(n \log n \log \log n + I)$ time by reducing the window-rendering problem to an interval valuation problem [11]. Recently, Larmore [12] gave an improved algorithm for the interval valuation problem, which can be used to implement Goodrich's window-rendering algorithm in $O(n \log n + I)$ time. Recently, Doh [7] has also shown how to solve the window-rendering problem in $O(n \log n + I)$ time, using a different approach. All of these algorithms are optimal in the worst case and also take advantage of problem instances that are "simpler" than in the worst case, but they are not truly output-sensitive. Indeed, there are problem instances where these algorithms run in $O(n^2)$ time even though the output size is very small. For example, in the case where a large rectangle obscures a collection of cross hatched rectangles, as in Figure 2b, I is quadratic although the output size is $O(1)$.

There are methods whose running time depends on both the input size and output size, however. Indeed, in [10] 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 (recall that k is at worst $\Theta(n^2)$). Bern [4] and Preparata, Vitter, and Yvinec [20] 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. Also, in [18] Overmars and Sharir give an output-sensitive algorithm for the general hidden-surface elimination problem that runs in $O(n\sqrt{k} \log n)$ time

(which immediately implies a solution to the window-rendering problem). 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.* It is straightforward to show that in the time-complexity of any window rendering algorithm the input-size component must be $\Omega(n \log n)$ and the output-size component must be $\Omega(k)$. Thus, none of these algorithms have optimal input-size or output-size components.

In this paper we give an algorithm for the window rendering problem whose running time depends on both the input size and output size. Our algorithm allows one to specify the trade-off between these two components of the running time with a parameter, t. When t is set to minimize the input-size component of the time complexity, then our algorithm runs in $O(n \log n + k \log n)$ time, which optimizes the input-size component of the time-complexity while matching the output-size component in the time-complexities of the previous best known algorithms. On the other hand, when t is set to minimize the output-size component of the time complexity, then our algorithm runs in $O(n^{1+\epsilon} + k)$ time, for any positive constant ϵ, which optimizes the output-size component of the time-complexity while achieving an input-size component that is $o(n^2)$.

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. 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.

This approach is similar to that used by Güting and Ottmann [10] and Preparata, Vitter, and Yvinec [20]. We achieve an improved running time over these previous implementations by using a data structure that we call the *hive tree.* Conceptually, the hive tree is a combination of the hive graph structure of Chazelle [5] and the segment tree structure of Bentley and Wood [3]. To solve the rectilinear hidden-surface elimination problem we augment the hive tree with a number of supporting auxiliary structures, where each structure is implemented with the most simple data structures: arrays and linked lists. Thus, our method should be fairly easy to program.

Our algorithm consists of two phases: (1) preparing for the sweep and (2) implementing the sweep. In order to concentrate on the primary structure of our algorithm, we begin by describing (in Section 3) a simple, but less-efficient, method for performing Phase 1, as well as describing a simple method for performing Phase 2, but for the simpler hidden-line elimination problem. These two phases combine to give a running time of $O(t(n^{1+1/t}\log n + n^{1+2/t} + k))$ for the hidden-line elimination problem for rectangles, where $2 \leq t \leq \log n$. Then, in Section 4 we show how to adapt our method to solve the harder hidden-surface elimination problem in this same bound by modifying the second phase of our algorithm. In Section 5 we show how to improve the running time of Phase 1 (the bottleneck phase), resulting in a running time that is $O(t(n^{1+2/t} + k))$ for rectilinear hidden-surface elimination. We begin our discussion, in the next section, by formally defining what constitutes a solution to the hidden-surface elimination problem for rectangles, and describing the hive tree structure.

2 Preliminaries

Suppose we are given a collection S of n non-intersecting isothetic rectangles in $\Re^3$, i.e., a collection of rectangles parallel to the xy-plane such that all edges are parallel to either the x- or 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.

2.1 Representing a Solution to the Window-Rendering Problem

There are many ways that one can specify what constitutes a solution to the hidden-surface elimination problem [10, 14, 17, 22, 23, 24]. 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. A solution to the hidden-surface elimination problem is given by $Hid(S)$, augmented so that each polygonal face of $Hid(S)$ stores the name of the rectangle of S that is visible in that face. We let $Vis(S)$ denote such an augmented graph for S, and use $Vis(S)$ to denote a solution to the hidden-surface elimination problem for S.

In order to better motivate our hidden-surface method, let us examine the structure of $Hid(S)$ and $Vis(S)$ more closely. For each vertex v of $Hid(S)$ (and $Vis(S)$) 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 $\top$, $\perp$, $\vdash$, 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 $\perp$), and *horizontal dead ends*, where the terminating segment is horizontal (i.e., $\vdash$ or $\dashv$). In Figure 1b, points e and f are corners, a is a $\top$, b is a $\perp$, c is a $\vdash$, and d is a $\dashv$. In that same figure, points a, b, c and d are dead ends: a and b are vertical dead ends, while c and d are horizontal dead ends. For convenience, we assume throughout the paper that the planar graph $Hid(S)$ lies in the xy-plane, so that any face of $Hid(S)$ is also in the xy-plane. Of course, for each such face in $Vis(S)$ we also know the name of the rectangle of S that is visible in it, and the z-coordinate of that rectangle.

There are a number of ways one can represent an embedded planar graph, such as $Hid(S)$ or $Vis(S)$. Three such representations are the "winged edge" structure of Baumgart [2], the "quad edge" structure of Guibas and Stolfi [9], and the "doubly-connected edge list" structure of Muller and Preparata [15, 19]. Our algorithm does not depend on which representation one chooses, so long as the representation allows one to determine each of the following in time proportional to its size:

1. all edges and faces adjacent to a given vertex v, as well as their orientation with respect to v,

2. all vertices and faces adjacent to a given edge e, as well as their orientation with respect to e, and

3. all vertices and edges that lie on the boundary of a given face f, in the order they occur around f.

Each of the mentioned representations provides this.

Given an isothetic rectangle R in $\Re^3$ we let $z(R)$ denote the z-coordinate of the plane to which R belongs. Similarly, for any point p in $\Re^3$, we use $x(p)$, $y(p)$, and $z(p)$ to denote the x-, y-, and z-coordinate of p, respectively. Our terminology implicitly assumes that the observer looking at the scene from $z = \infty$ has his body parallel to the y-axis, with both arms extended so they are parallel to the x-axis (the reader probably inferred this from the way we drew Figure 1b). Hence a *vertical* segment is parallel to the y-axis, whereas a *horizontal* segment is parallel to the x-axis. Similarly, we say that a plane is *vertical* (resp., *horizontal*) if it is parallel to the yz-plane (resp., xz-plane).

Before we give our hidden-line and hidden-surface methods, we describe the primary data structure we use in our algorithm, namely, the hive tree.

2.2 The Hive Tree Data Structure

In this section we define the hive tree data structure, and show how it can be efficiently constructed. It can be defined on a collection of horizontal and vertical segments in the plane, but, since all the horizontal and vertical segments in our case belong to rectangles, we define it on a collection, S, of n rectangles in the plane. We begin by projecting the vertical rectangle boundaries on the x-axis and placing 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". We use Π_v to denote the slab associated with the leaf v. 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 on these slabs in the natural way, so that each leaf is associated with a slab (t is a tunable parameter for the construction). Recall that an $n^{1/t}$-ary tree is a rooted tree such that each internal node has $n^{1/t}$ children (expect possibly nodes with children that are leaves). To simplify computations that we will have to 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.

For 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 $\mathcal{L}(\Pi_v)$ (resp., $\mathcal{R}(\Pi_v)$) denote the left (resp., right) vertical boundary of Π_v. Note that $\mathcal{L}(\Pi_v)$ (resp., $\mathcal{R}(\Pi_v)$) is a plane parallel to the yz-plane, and any rectangle R intersects this plane in a segment parallel to the y-axis.

For each v in T we define two sets, $Cover(v)$ and $End(v)$, which are motivated by similar sets defined for the segment tree data structure of Bentley and Wood [3]. Specifically, $Cover(v)$ stores all the rectangles that span Π_v (i.e., that intersect $\mathcal{L}(\Pi_v)$ and $\mathcal{R}(\Pi_v)$) but do not span Π_z, where z is the parent of v, and $End(v)$ stores all the rectangles that have a vertical boundary inside Π_v. Note that any rectangle in S can belong to at most $2\lceil t \rceil + 2$ of the $End(v)$ sets and no more than $(2\lceil t \rceil + 2)n^{1/t}$ of the $Cover(v)$ sets.

We partition each Π_v slab into horizontal *strips*, whose vertical boundaries are delimited by $\mathcal{L}(\Pi_v)$ and $\mathcal{R}(\Pi_v)$, 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 a $Y(h)$ and $Y(h')$, where $h \in Down(h')$. respectively.

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

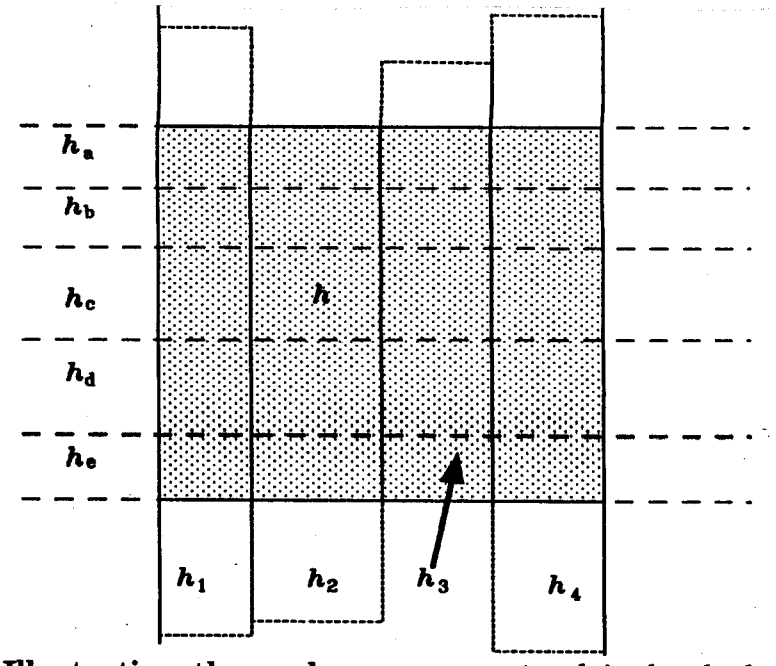

Figure 3: **Illustrating the enclosure property.** h is the shaded region, $Up(h) = \{h_a, h_b, h_c, h_d, h_e\}$, and $Down(h) = \{h_1, h_2, h_3, h_4\}$.

Proof: Let v and z be the nodes in T such that $h \in Strip(v)$ and $h' \in Strip(z)$. 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 $Cover(z) \cup End(z)$ (and similarly for v), it suffices to show that $Cover(v) \cup End(v) \subseteq Cover(z) \cup End(z)$. By the definition of Π_z, $End(v) \subseteq End(z)$, 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, $Cover(v) \subseteq End(z)$. ■

Corollary 2.2: *Let h be a strip such that in $h \in Up(h')$ 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 Figure 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 [5] 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 each (h, h') pair with $h \in Up(h')$ we have symmetric pointers between the copy of h in $Up(h')$ and the copy of h' in $Down(h)$.

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 $Cover(v)$ and $End(v)$ 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 $Strip(v)$ are a subset of the y-coordinates of the boundaries of the strips in $Strip(z)$, where z is v's parent. Our method, then, is to construct the $Strip(r)$ 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 $Strip(v_1), Strip(v_2), ..., Strip(v_{n^{1/t}})$, in turn, where $v_1, v_2, ..., v_{n^{1/t}}$ are the children of r. Given the lists $Cover(v_i)$ and $End(v_i)$ already constructed for each v_i, this is easy to do in $O(|Strip(r)|)$ time for each v_i. Repeating this recursively, for $v_1, ..., v_{n^{1/t}}$, constructs all the *Strip* lists in D. In addition, while we are copying out the strips from the *Strip* list 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 $Strip(v_i)$ and adding the $Strip(v_i)$ strips to the *Down* lists for the strips in $Strip(v)$. Since each recursive call takes $O(|Strip(v)|n^{1/t})$ time plus the 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 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 $Hid(S)$. 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

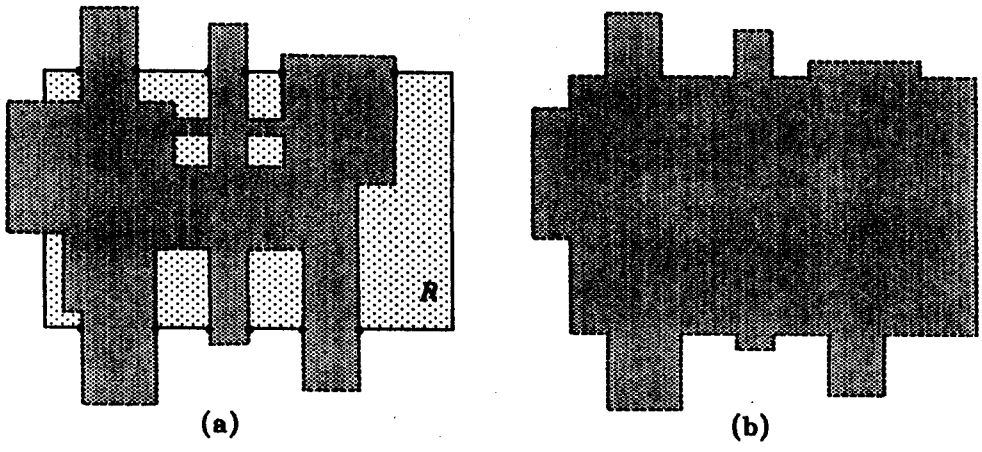

Figure 4: **The two shadow operations.** (a) **v-query**(R); (b) **add**(R)..

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 on 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 (if any) are not obscured by the shadow. (See Figure 4a.)

- **add**(R), update D so as to represent the shadow of $S' \cup \{R\}$ and assign $S' := S' \cup \{R\}$. (See Figure 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 $Hid(S)$ 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 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 $\bot$). 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 these points, 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 4 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 the "radix" sorting method 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 preparation phase 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 will show 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 preparation phase.

3.1 Phase 1: Preparing for the Sweep

As mentioned earlier, the first phase in our algorithm is to construct the augmented hive tree data structure for maintaining the shadow. So let T be a hive tree constructed on the projections of the rectangles in S on the xy-plane. We begin by describing how we augment 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, $NFU(h) = Up(h)$, $TD(h) = \emptyset$, and $OD(h) = Down(h)$ 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.

There is one more auxiliary structure that we add to T to help implement the space sweeping of Phase 2. 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 1 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 $\mathcal{L}(\Pi_v)$ plane, of a collection of line segments parallel to the y-axis (the so-called "skyline problem" [13]). This step can easily be implemented in $O(n_v \log n_v)$ time for each v in T by a mergesort-like divide-and-conquer scheme, where $n_v = |Cover(v)|$. 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 $h \in Strip(v)$ 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 $O(n_v + |Strip(v)|)$ 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 will denote by D, for maintaining the shadow of S', as well as our method for its construction. We conclude this subsection, then, with the following lemma:

Lemma 3.1: *One can construct and initialize the data structure D, for maintaining the shadow of the rectangles S', in $O(tn^{1+1/t} \log n + tn^{1+2/t})$ time.* ∎

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**.

3.2 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 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 $Strip(v)$, and we could have easily precomputed all strips h in all $Strip(v)$ lists such that R is in $Cover(v)$ and such that R shares a horizontal boundary with h. Thus, searching through all such h's can be done in $O(tn^{1/t})$ time, given this precomputation. 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 $h \in Strip(v)$ 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 will 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 also determine 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 $h \in \mathit{Strip}(v)$ 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 $h \in \mathit{Strip}(w)$ 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 $O(t(n^{1/t} + k_R))$.

3.3 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 essentially 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 2 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 3 strips in the slab for such a leaf (i.e., $|\mathit{Strip}(v)| = 3$). 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 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 $Up(h')$ (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) 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).

Note that in this procedure we do not mark any strips as "full". This is because each bottom-level strip h properly contains the rectangle vertical boundary that defined it. Note that 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 $Strip(v)$ 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 Figure 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 Figure 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** $OD(h') \cup TD(h') = \emptyset$ **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. The proof of this claim is based on showing that 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** $NFU(h') = \emptyset$ **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.

3.4 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 over-estimate, however, for, as we show in this subsection, 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 σ = **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 $|NFU(h')| \leq |Up(h')|$, the most we can charge, then, is $O(|Up(h')|)$. By the previous observation, in the entire space sweep procedure we will 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 $O(\sum_{h \in D} |Up(h)|)$. 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 σ = **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 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 $O(|Up(h)|)$ (resp., $O(|Down(h)|)$). Thus, the total time we spend performing **FullUp** and **FullDown** operations is at most $O(\sum_{h \in D}(|Up(h)| + |Down(h)|))$. 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 theorem:

Theorem 3.2: *Given a collection S of n isothetic rectangle in $\Re^3$, one can construct $Hid(S)$ in $O(t(n^{1+1/t} \log n + n^{1+2/t} + k))$ time, where k is the size of the output and $2 \leq t \leq \log n$ is a tunable parameter.* ■

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

4 Extending 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 will extend the method of the previous section to not only give us the graph of visible edges, but also the rectangle that is visible in each face of this graph. Recall that in the previous algorithm we swept through the collection of rectangles from front to back, maintaining the shadow as we swept. 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 would immediately know two of the three visible rectangles that are adjacent to v. In addition, for any visible rectangle corner vertex v, we would 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 determining 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)). Thus, there may be no proximity in the z-coordinate between v and its background rectangle. 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 $h' \in Down(h)$, and also a pointer from this copy of v to the 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 $Hid(S)$ which 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 theorem:

Theorem 4.2: *Given a collection S of n isothetic rectangle in $\Re^3$, one can solve the window-rendering problem for S in $O(t(n^{1+1/t}\log n + tn^{1_2/t} + k))$ time, where k is the size of the output and $2 \leq t \leq \log n$ is a tunable parameter.* ∎

By appropriate assignments to the parameter t, then, we immediately get the following corollaries. In the first corollary we use t to optimize the output-size component of the running time.

Corollary 4.3: *One can construct $Vis(S)$ in $O(n^{1+\epsilon} + k)$ for any positive constant $\epsilon \leq 1$.*

Proof: Take $2/t < \epsilon$. ■

This corollary demonstrates the first output-sensitive window-rendering algorithm whose running time is $O(f(n) + k)$, where $f(n)$ is $o(n^2)$. The next corollary shows how one can alternately trade this off with the input-size component of the running time.

Corollary 4.4: *One can construct $Vis(S)$ in $O(n \log^2 n + k \log n / \log \log n)$ time.*

Proof: Take $t = \log n / \log \log n$, so $n^{1/t} = \log n$. ■

This latter result matches the input-size component of Güting and Ottmann [10] and Preparata, Vitter, and Yvinec [20], and improves the output-size component of the running times of methods by Preparata, Vitter, and Yvinec [20] and Bern [4]. It does not improve input-size component of the running time of Bern's algorithm, however. We can further improve our procedure, however, to optimize the input-size component of the running time, without increasing the output-size component. We describe the modifications necessary for achieving this in the next section.

5 Optimizing the Input-Size Component

In this section we show how to modify the first phase of our algorithm to achieve a running time for the entire algorithm of $O(t(n^{1+2/t} + k))$. Taking $t = \log n$ implies a running time of $O((n + k) \log n)$ for the window-rendering problem, which improves the best previous method by Bern [4].

There is one primary bottleneck to achieving an $O(n \log n)$ input-size component in our running time, and that is in the construction of 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. We would like to bring this from $O(tn^{1/t} \log n)$ to $O(tn^{1+2/t})$. Doing this requires the use of more complicated data

structures than those we have used so far. Thus, the discussion of this section may be of more theoretical than practical interest.

In the description that follows let us concentrate on a particular node v in D. Recall that since each rectangle in $Cover(v)$ spans Π_v, the computation of Vis_v is equivalent to a 2-dimensional visibility problem. Namely, it is equivalent to determining the upper envelope of a set of segments parallel to the y-axis in the yz-plane. Also note that we can normalize 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 [25, 26], where $n_v = |Cover(R)|$. 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 **min** 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. But this is not efficient enough, however, for $\sum_{v \in D} n_v$ is $O(tn^{1+1/t})$; hence, this approach would result in a running time of $O(tn^{1+1/t} \log\log n)$ for Phase 1. Thus, we must be more clever in how we construct the Vis_v's. Our method, then, is as follows.

1. We mark each node that is on a level of T which is a multiple of $\log\log n$ 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 $i + \log\log n$ 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 $\mathcal{L}(\Pi_z)$ plane of the segments formed by intersecting $\mathcal{L}(\Pi_z)$ with the rectangles in $End(v)$, ignoring the rectangles in $End(v)$ that do not intersect $\mathcal{L}(\Pi_z)$. 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 $t/\log\log n$ 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 merging the solutions already at the leaves of T_z in a pair-wise fashion up T_z. 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 $O(n_z \log n^{1/t} \log\log n) = O((n_z/t)\log n \log\log n)$, 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 $t/\log\log n$ of these (leaf) super node lists, $\sum_z n_z = nt/\log\log n$; 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 $\mathcal{L}(\Pi_v)$ plane) of the segments formed by intersecting $\mathcal{L}(\Pi_v)$ 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 by a simple "plane-sweeping" procedure, using the priority queue data structure of van Emde Boas [25, 26], where m_v is the number of rectangles involved for v. We can use this data structure for all the operations in this sweep, because each sweep operation can be implemented using $O(1)$ $insert(i)$, $delete(i)$, and/or max operations, where $i \in [1,n]$ (assuming the rectangle z-coordinates are normalized to integers in the range $[1,n]$). 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 $\mathcal{L}(\Pi_v)$ plane) of the segments formed by intersecting $\mathcal{L}(\Pi_v)$ 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 outside 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: *One can construct all the $P(R)$ lists in $O(tn^{1+2/t})$ time.*

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

This immediately gives us the following theorem:

Theorem 5.2: *Given a collection S of n isothetic rectangle in $\Re^3$, one can construct $Vis(S)$ in $O(t(n^{1+2/t} + k))$ time, where k is the size of the output and $2 \leq t \leq \log n$ is a tunable parameter.* ∎

Corollary 5.3: *One can construct $Vis(S)$ in $O((n + k) \log n)$ time.*

Proof: Take $t = \log n$. ∎

Note that this bound matches output-size component of the algorithm [4, 20], but improves the input-size component to $O(n \log n)$, which is optimal.

6 Conclusion

In this paper we have given an algorithm for solving the hidden-surface elimination problem for rectangles. Our method is parameterized by a number, t, which specifies the trade-off between the input-size, n, and output-size, k, in the running time of the algorithm. We can choose t to achieve a time complexity of $O(n^{1+\epsilon} + k)$, for any positive constant ϵ, thus achieving an optimal output-size component, or a time-complexity of $O((n + k) \log n)$, which achieves an optimal input-size component. One of the main ideas in our method is the use of a geometric data structure we called the *hive tree*, which may have applications to other rectilinear problems.

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.

Acknowledgements

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

References

[1] A.V. Aho, J.E. Hopcroft, and J.D. Ullman, *The Design and Analysis of Computer Algorithms*, Addison-Wesley, Reading, Mass., 1974.

[2] B.G. Baumgart, "A Polyhedron Representation for Computer Vision," *Proc. 1975 AFIPS National Computer Conf., 44*, AFIPS Press, 1975, 589–596.

[3] J.L. Bentley and D. Wood, "An Optimal Worst Case Algorithm for Reporting Intersections of Rectangles," *IEEE Trans. on Computers*, Vol. C-29, 1980, 571–577.

[4] M. Bern, "Hidden Surface Removal for Rectangles," *Proc. 4th ACM Symp. on Computational Geometry*, 1988, 183–192.

[5] B. Chazelle, "Filtering Search: A New Approach to Query-Answering," *SIAM J. Comput.*, Vol. 15, 1986, 703–724.

[6] F. Dévai, "Quadratic Bounds for Hidden-Line Elimination," *Proc. 2nd ACM Symp. on Computational Geometry*, 1986, 269–275.

[7] J.I. Doh, "Visibility Problems for Orthogonal Objects in Two- or Three-Dimensions," to appear in *The Visual Computer.*

[8] M.T. Goodrich, "A Polygonal Approach to Hidden-Line Elimination," *Proc. of 25th Annual Allerton Conference on Comm., Control, and Computing*, 1987, 849–858.

[9] L.J. Guibas and J. Stolfi, "Primitives for the Manipulation of General Subdivisions and the Computation of Voronoi Diagrams," *ACM Trans. on Graphics*, Vol. 4, 1985, 75–123.

[10] R.H. Güting and T. Ottmann, "New Algorithms For Special Cases of the Hidden Line Elimination Problem," *Computer Vision, Graphics, and Image Processing*, Vol. 40, 1987, 188–204.

[11] D.S. Hirschberg and D.J. Volper, "Improved Update/Query Algorithms for the Interval Valuation Problem," *Information Processing Letters*, Vol. 24, 1987, 307–310.

[12] L. Larmore, "An Optimal Query-Update Structure for the Interval Valuation Problem," manuscript, 1989.

[13] U. Manber, *Introduction to Algorithms: A Creative Approach*, Addison-Wesley, Reading, Mass., 1989.

[14] M. McKenna, "Worst-case Optimal Hidden-Surface Removal," *ACM Transactions on Graphics*, Vol. 6, 1987, 19–28.

[15] D.E. Muller and F.P. Preparata, "Finding the Intersection of Two Convex Polyhedra," *Theoretical Computer Science*, Vol. 7, 1978, 217–236.

[16] O. Nurmi, "A Fast Line-Sweep Algorithm For Hidden Line Elimination," *BIT*, Vol. 25, 1985, 466–472.

[17] T. Ottmann and P. Widmayer, "Solving Visibility Problems by Using Skeleton Structures," *Proc. 11th Symp. on Math. Foundations of Computer Science*, 1984, 459–470.

[18] M.H. Overmars and M. Sharir, "Output-Sensitive Hidden Surface Removal," *Proc. 30th IEEE Symp. on Foundations of Computer Science, 1989*, in press.

[19] F.P. Preparata and M.I. Shamos, *Computational Geometry: An Introduction*, Springer-Verlag, New York, 1985.

[20] F.P. Preparata, J.S. Vitter, and M. Yvinec, "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, 1988.

[21] A. Schmitt, "On the Time and Space Complexity of Certain Exact Hidden Line Algorithms," Universität Karlsruhe, Fakultät für Informatik, Report 24/81, 1981.

[22] A. Schmitt, "Time and Space Bounds for Hidden Line and Hidden Surface Algorithms," *EUROGRAPHICS '81*, 43–56.

[23] S. Sechrest and D.P. Greenberg, "A Visibility Polygon Reconstruction Algorithm," *ACM Transactions on Graphics*, Vol. 1, 1982, 25–42.

[24] I.E. Sutherland, R.F. Sproull, and R.A. Schumacker, "A Characterization of Ten Hidden-Surface Algorithms," *Computing Surveys*, Vol. 6, 1974, 1–25.

[25] P. van Emde Boas, "Presevering Order in a Forest in Less than Logarithmic Time and Linear Space," *Information Processing Letters*, Vol. 6, 1977, 80–82.

[26] P. van Emde Boas, R. Kaas, and E. Zijlstra, "Design and Implementation of an Efficient Priority Queue," *Math. Systems Theory*, Vol. 10, 1977, 99–127.