JOURNAL OF COMPUTER AND SYSTEM SCIENCES **40**, 49–69 (1990)

Hidden Surface Removal for Rectangles

MARSHALL BERN

Xerox Palo Alto Research Center,
Palo Alto, California 94304

Received December 6, 1988; revised May 1, 1989

A simple but important special case of the hidden surface removal problem is one in which the scene consists of n rectangles with sides parallel to the x- and y-axes, with viewpoint at $z=\infty$ (that is, an orthographic projection). This special case has application to overlapping windows in computer displays. An algorithm with running time $O(n \log n + k \log n)$ is given for static scenes, where k is the number of line segments in the output. Algorithms are given for a dynamic setting (that is, rectangles may be inserted and deleted) that take time $O(\log^2 n \log \log n + k \log^2 n)$ per insert or delete, where k is now the number of visible line segments that change (appear or disappear). Algorithms for point location in the visible scene are also given.

1. INTRODUCTION

Imagine we are given a set of n opaque rectangles in 3-space, such that each rectangle has sides parallel to the x- and y-axes and a constant z-coordinate. No pair of rectangles intersects in a 2-dimensional region, though pairs may intersect along an edge. We would like to report the regions of the input rectangles visible from a point at $z=\infty$ (that is, lines of sight are parallel to the z-axis), or, equivalently, remove the hidden regions. Figure 1 shows a scene formed by eight rectangles; there are nine *visible regions*, the polygonal shapes bounded by solid lines. We consider two different settings: *static*, that is, all rectangles are known in advance, and *dynamic*, that is, rectangles may be inserted and deleted, and the visible scene must be updated after each change.

In this paper, we consider input rectangles to be featureless. Windows in computer displays, however, usually include contents, which may be either graphics or text. For this application, the output of our algorithms (that is, the coordinates of the visible regions) could be used to extract only the visible bits from back-up buffers in memory. It is sometimes desirable to subdivide visible regions into rectangles, for example, if the hardware supports a block transfer or *bitblt*; our algorithms can be easily adapted to do this. (See [M] for more on windows.) Though our work is theoretical, we believe that it offers practical improvements for hypothetical displays with a great many (hundreds of) windows.

Our algorithms are *output-sensitive*, that is, the running times depend on the size of the output and are much faster than the worst case of $O(n^2)$ for simple scenes

0022-0000/90 $3.00

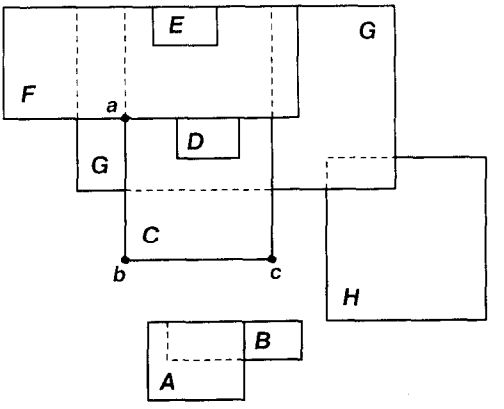

FIG. 1. A scene with hidden lines shown broken.

or changes. For general (static) scenes, already known are a worst-case optimal $O(n^2)$ algorithm [McK] and an "intersection-sensitive" algorithm with running time $O(n \log n + i)$, where i is the number of intersection in a projection plane [G]. For static rectangle scenes, there is also an output-sensitive algorithm, due to Güting and Ottmann [GO], with running time $O(n \log^2 n + k \log^2 n)$. Our static algorithm is faster than the algorithm of [GO], and in its simpler incarnation, also easier to program. The algorithm of [GO] is more general, handling "c-oriented" rectangles (rectangles aligned with a fixed number of orientations, not just horizontal and vertical). Recently, Preparata, Vitter, and Yvinec have independently improved on [GO], giving an algorithm with running time $O(n \log^2 n + k \log n)$ [PVY]. Atallah and Goodrich have given an algorithm with running time $O(n^{3/2} + k)$ [AG]. So far as we know, our paper includes the first theoretical treatment of the dynamic problem.

Sections 2 and 3 of this paper describe the static algorithm, which is a straightforward sweep algorithm using Bentley's segment tree as the primary data structure [PS]. Sections 4–7 develop the algorithms for the dynamic setting. These algorithms rely on higher dimensional data structures such as priority search trees [McC] and dynamic segment trees [EM, WL]. Finally, Section 8 discusses the problem of determining which rectangle is stabbed by a given line of sight (equivalently, which window contains a mouse-click), giving a method as fast as, and more space efficient than, general polygonal subdivision methods.

2. THE STATIC SETTING

In the static setting, each rectangle is given at the outset by five real numbers. In the pseudocode to follow, the five numbers specifying rectangle R, where $R = [x_1, x_2] \times [y_1, y_2] \times [z, z]$, are denoted by the Pascal notation of $R.x1$, $R.x2$,

$R.y1$, $R.y2$, and $R.z$. We shall refer to the line segment with endpoints at ($R.x1$, $R.y1$, $R.z$) and ($R.x1$, $R.y2$, $R.z$) as the *left* edge of R. *Right*, *bottom*, and *top* edges are defined similarly. Line segments parallel to the x-axis (y-axis) will be called *horizontal* (vertical). The z-coordinate will be called *height*, that is, if $R.z > R'.z$, we say that R is higher than R'. As usual in computational geometry, the comparison of two real numbers is assumed to take unit time.

Our algorithm conceptually sweeps a plane normal to the x-axis across the rectangles, maintaining the cross sections of rectangles currently cut by the sweep-plane in a segment tree whose skeleton has been precomputed. The skeleton of the segment tree is computed by sorting all $R.y1$ and $R.y2$ fields and then building a complete binary tree in which each leaf represents an *atomic* segment between two successive y-coordinates. Segment tree node v corresponds to a basic segment segment(v), which is the union of the atomic segments of the leaves in the subtree rooted at v. Atomic and basic segments are half-open, that is, they contain their left endpoints, but not their right endpoints.

To compute an "event schedule" for the sweep algorithm, we sort all $R.x1$ and $R.x2$ fields. As usual (see [PS] for similar sweep algorithms), when the sweep reaches the left edge of rectangle R, R is entered into the *rectangle list* of each node v such that [$R.y1$, $R.y2$] spans segment(v) but does not span segment(parent(v)). R is removed from these lists when the sweep reaches the right edge of R.

How do we build z-coordinate information into this data structure? The first modification of the standard segment tree is to order the rectangle lists by decreasing height. For now we may assume these lists are heaps, implemented as implicit binary or 4-ary trees [T]. The second modification is to use the segment tree itself as a sort of heap. We attach additional fields, called H and L, to each node v in

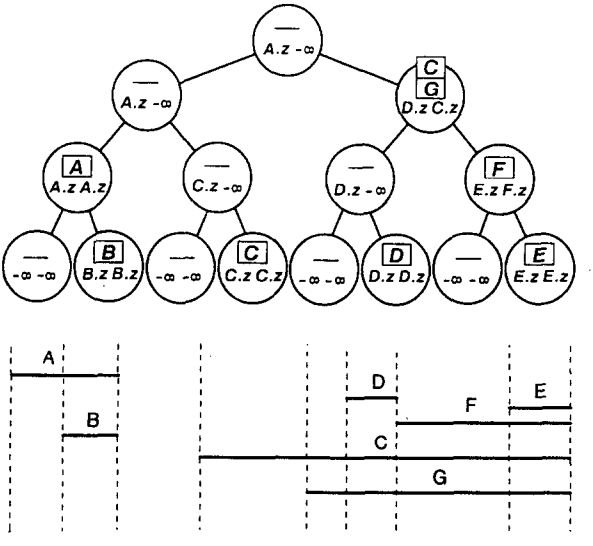

FIG. 2. End-on rectangles and modified segment tree.

the segment tree. The H field of node v stores the z-coordinate of the highest cross section listed at some node in the subtree rooted at v. The L field stores the z-coordinate of the lowest *visible* cross section, where visible means visible within the "subscene" at v; that is, some point is not obscured by the other cross sections listed within the subtree rooted at v. Figure 2 shows the segment tree for the scene of Figure 1 at a point in the sweep just after the left edge of D has been encountered. Below the segment tree the rectangles currently cut by the sweep-plane are shown end-on, that is, along lines of sight parallel to the x-axis. For each node, the rectangle list is represented by a stack of boxes, and the H field (L field) appears at the lower left (right). An H field of $-\infty$ indicates that the subscene is empty; an L field of $-\infty$ indicates some line of sight (parallel to the z-axis) continues forever.

The H and L fields for node v can be determined in $O(1)$ time, knowing the H and L fields for the children of v and the z-coordinate of the highest rectangle in v's list. As in the pseudocode below, $v.L$ is equal to either the z-coordinate of the highest rectangle (since a rectangle listed at v covers all lower rectangles listed at descendants of v) or the smaller of the two L fields of v's children.

In the pseudocode below, LChild(v) (resp. RChild(v)) returns the left (right) child of node v. Each node v has, along with H and L, the usual fields $y1$ and $y2$, containing the endpoints of segment(v), and *ymid*, which is such that segment(LChild)(v)) = $[v.y1, v.ymid)$. The rectangle list for v is called $v.heap$, and Top($v.heap$).z is assumed to be $-\infty$ if $v.heap$ is empty. If a rectangle R is contained in $v.heap$, we say v *lists* R. The procedure call LeftEdge(R, *true*, *root*) inserts rectangle R into the segment tree rooted at *root*, calls *LeftReport* to report output line segments, and then updates the H and L fields for all nodes visited in inserting R. Similarly, RightEdge(R, *true*, *background*, *root*) deletes rectangle R from the segment tree, calls *RightReport* to report output, and then updates the H and L fields for all nodes visited in deleting R. In this call, *background* is a fictitious rectangle with height $-\infty$.

```
procedure LeftEdge (R: rectangle, visible: boolean, v: segment tree node)
  if R.z < v.L then visible := false fi
  if R.y1 ≤ v.y1 and v.y2 ≤ R.y2 then
    insert R into v.heap
    if visible then LeftReport (R, v) fi
  else
    if R.y1 < v.ymid then LeftEdge (R, visible, LChild(v)) fi
    if v.ymid < R.y2 then LeftEdge (R, visible, RChild(v)) fi
  fi
  v.H := max{LChild(v).H, RChild(v).H, Top(v.heap).z}
  v.L := max{min{LChild(v).L, RChild(v).L}, Top(v.heap).z}

procedure RightEdge (R: rectangle, visible: boolean, R′: rectangle, v: segment tree node)
  if R.z < v.L then visible := false fi
  if R.y1 ≤ v.y1 and v.y2 ≤ R.y2 then
    delete R from v.heap
    v.H := max{LChild(v).H, RChild(v).H, Top(v.heap).z}
    if R′.z < Top(v.heap).z then R′ := Top(v.heap) fi
    if visible then RightReport (R, R′, true, v) fi
```

```
else
  if R'.z < Top(v.heap).z then R' := Top(v.heap) fi
  if R.y1 < v.ymid then RightEdge (R, visible, R', LChild(v)) fi
  if v.ymid < R.y2 then RightEdge (R, visible, R', RChild(v)) fi
fi
v.H := max{LChild(v).H, RChild(v).H, Top(v.heap).z}
v.L := max{min{LChild(v).L, RChild(v).L}, Top(v.heap).z}
```

Viewed segments—that is, the solid segments in Fig. 1—are reported in the following manner: a vertical segment is reported in $O(\log n)$ disjoint pieces when the sweep plane reaches its x-coordinate, and a horizontal segment is reported in one piece when the sweep plane reaches the x-coordinate of its right endpoint. For example, segment *ab* in Fig. 1 is reported in two pieces, corresponding to the two basic segments composing the y-extent of *ab*. Segment *bc* is reported in a single piece.

At a given point in the algorithm, a visible horizontal segment is *active* if it is currently cut by the sweep plane. A data structure, called the *active list*, stores active horizontal segments in y-coordinate order, so that segments with y-coordinates within a certain range can be reported together at the left edge of a higher rectangle. In LeftReport(R, v) below, active segments are reported at each v that corresponds to a basic segment of a visible piece of R's left edge. One could save up and combine the range queries induced by the basic segments into a single range query per visible piece. Then the k horizontal segments with right endpoints along a single visible piece can be reported in time $O(\log n + k)$ using a balanced binary tree such as a red–black tree [T] as the active list.

Alternatively, to avoid the programming difficulties of combining range queries and rebalancing a balanced tree, one can use the segment tree itself as the active list. Each segment tree node v is augmented with a field $v.A$. Field $v.A$ is an unordered set of all active segments with y-coordinate within segment(v). Set $v.A$ can be implemented as a linked list. Reporting k active segments with right endpoints along a single basic segment then takes time $O(k+1)$, and reporting all segments for an entire visible piece takes time $O(\log n + k)$. Storing or deleting an active segment takes time $O(\log n)$ using either method.

In the pseudocode below, each active segment s has two fields: the x-coordinate $s.x1$ of its left endpoints, and its y-coordinate $s.y$. In order to report visible regions rectangle by rectangle, one would add two more fields to store the names of the rectangles visible on either side of s.

```
procedure LeftReport (R: rectangle, v: segment tree node)
  if R.z < v.L then return fi
  if v.H < R.z then
    Output and remove from the active list each horizontal segment s with v.y1 ≤ s.y < v.y2
    Output vertical segment [(R.x1, v.y1), (R.x1, v.y2)]
    if v.y1 = R.y1 then Store a horizontal segment s with s.x1 := R.x1 and s.y := v.y1 fi
    if v.y2 = R.y2 then Store a horizontal segment s with s.x1 := R.x1 and s.y := v.y2 fi
```

```
  else
    LeftReport (R, LChild(v))
    LeftReport (R, RChild(v))
  fi

procedure RightReport (R: rectangle, R′: rectangle, atR: boolean, v: segment tree node)
  if R. < v.L then return fi
  if v.H < R.z and atR then
    Output vertical segment [(R.x2, v.y1), (R.x2, v.y2)]
    if v.y1 = R.y1 then Output a horizontal segment s with s.y := v.y1 fi
    if v.y2 = R.y2 then Output a horiozontal segment s with s.y := v.y2 fi
    atR := false fi
  if R′.z < Top(v.heap).z
    then R′ := Top(v.heap) fi
  if v.H < R′.z then
    if v.y1 = R′.y1 then Store a horizontal segment s with s.x1 := R.x2 and s.y := v.y1 fi
    if v.y2 = R′.y2 then Store a horizontal segment s with s.x1 := R.x2 and s.y := v.y2 fi
  else
    RightReport (R, R′, atR, LChild(v))
    RightReport (R, R′, atR, RChild(v))
  fi
```

3. Analysis

We first give a brief correctness argument, consisting of an explanation of the pseudocode above; we then give two theorems, "practical" and "theoretical."

Procedures *LeftEdge* and *RightEdge* are fairly straightforward adaptations of pseudocode from [PS]. In addition to inserting a new rectangle cross section in the usual manner, *LeftEdge* updates the H and L fields as discussed above and maintains a flag *visible*. The flag *visible* is meant to be true at node v if and only if some point on R within segment(v) is visible. That this is indeed the case follows from the correctness of the L fields along the path from the root to v. Similarly, it is not hard to confirm that procedure *RightEdge* correctly deletes a rectangle, updates H and L fields, and maintains *visible*. In addition, *RightEdge* also passes R', the highest rectangle found along the path from the root to a node listing R, to *RightReport*.

Now assume that the left edge of R is at least partially visible and let it be divided into alternating visible and invisible pieces. Imagine decomposing all visible pieces into basic segments, yielding the segments segment(v_1), segment(v_2), ..., segment(v_l); where each v_i is a node in the segment tree. Similarly, invisible pieces can be decomposed into basic segments segment(w_1), segment(w_2), ..., segment(w_m). Figure 3a shows an end-on view of rectangle R (not from the scene depicted in Figs. 1 and 2) with its division into basic segments. The crucial observation is that *LeftReport* stops its recursive search whenever it reaches one of the v_i or w_j nodes; thus, each piece of the left edge of R, even a piece hidden behind a complicated part of the scene, costs only $O(\log n)$ to discover. In the

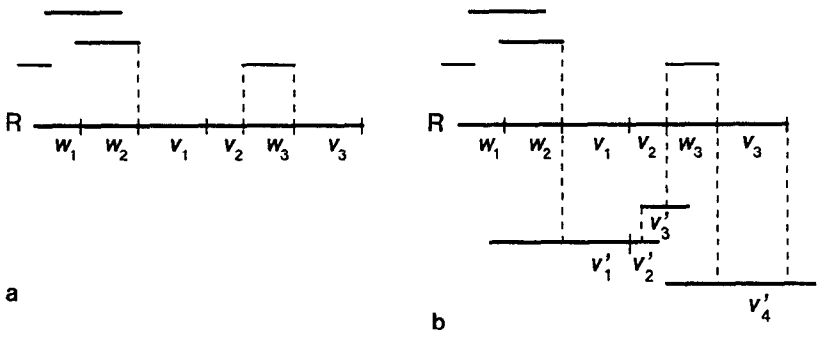

FIG. 3. (a) R's left edge divided into basic segments, (b) R's right edge and revealed rectangles beneath R.

proofs below, we call a node v *maximal* with respect to some property if v is the first vertex on the path from the root to v that satisfies that property.

LEMMA 1. LeftReport *explores a forest of subtrees of the segment tree. The roots of these subtrees are nodes that list R, the nodes* $v_1, v_2, ..., v_l$ *are leaves of these subtrees, and the remaining leaves are members of* $\{w_1, w_2, ..., w_m\}$. *Output is reported and the active region list is appropriately updated during this exploration.*

Proof. First notice that *LeftReport* is initially called at exactly those nodes that list R at which *visible* is true. Next notice that *LeftReport* visits each of the nodes v_i for $1 \leqslant i \leqslant l$. This statement follows from the observations: each v_i lies in a subtree rooted at a node v that lists R at which *visible* is true, and v_i is a maximal node with segment$(v_i) \subseteq [R.y1, R.y2)$ and $v_i.H < R.z$ both true (otherwise v_i would not be a basic segment of a visible piece). The recursive search stops at each node v_i. Then horizontal segments with endpoints along segment(v_i) are reported, a visible portion of the left edge R is reported along segment(v_i), and if necessary horizontal segments along the top or bottom of R are stored. Finally, if the recursive search does not find a node v_i, it will stop at a node w_j, because each w_j is a maximal node with L field greater than $R.z$. In this case, the active region list requires no modification, as R lies below the visible regions along segment(w_j). ∎

Figure 3b shows the situation handled by *RightReport*. Now the nodes v_i, $1 \leqslant i \leqslant p$, and w_j, $1 \leqslant j \leqslant q$, correspond to visible and invisible basic segments along the right edge of R. In addition, the visible pieces of rectangles along and below the right edge of R are divided into basic segments segment(v'_1), segment(v'_2), ..., segment(v'_r).

LEMMA 2. RightReport *explores a forest of subtrees of the segment tree. The roots of these subtrees are nodes that list R, each* v_i *is contained in some subtree, the nodes* $v'_1, v'_2, ..., v'_r$ *are leaves of these subtrees, and the remaining leaves are members*

of $\{w_1, w_2, ..., w_q\}$. *Output is reported and the active region list is appropriately updated during this exploration.*

Proof. *RightReport* is analogous to *LeftReport* except that it continues exploring below the "visible nodes" v_i. First observe that throughout the recursion, *RightReport* maintains R' to be the highest rectangle (besides R) listed on the path from the root to the current node v. Next notice that each v_j' is a maximal descendant of a visible node v_i such that $v_j'.H$ is smaller than the z-coordinate of the highest rectangle along the path from the root to v_j'. Thus, at each v_j', $v_j'.H < R'.z$, and horizontal segments for the revealed rectangle R' are appropriately stored. ∎

THEOREM 1. *Hidden surface removal for a static set of n rectangles can be computed in time* $O(n \log^2 n + k \log n)$, *where k is the length of the output, and space* $O(n \log n)$, *using "elementary" data structures (segment trees and heaps).*

Proof. The factor of $O(n \log^2 n)$ in the time bound is the time to insert and delete rectangles from heaps. The factor of $O(k \log n)$ follows from Lemmas 1 and 2 and the observation that the total processing time of *LeftReport* and *RightReport* is within a constant factor of the total number of basic segments in all visible and invisible pieces of left and right edges. The space bound follows from the fact that each rectangle is listed at $O(\log n)$ segment tree nodes. If the segment tree itself is used as the active list, then also observe that no more than $O(n)$ horizontal segments are active at any one time. ∎

The running time may be improved to $O(n \log n \log \log n + k \log n)$ by using van Emde Boas's $O(\log \log n)$ priority queues [VKZ] as the $v.heap$ data structures, but an even better theoretical result can be obtained by taking advantage of the off-line nature of the problem. For each node v in the segment tree, we consider the entire sequence of insertions and deletions of rectangles at v in order to determine Top($v.heap$), $v.H$, and $v.L$ at each x-coordinate.

In this version of the algorithm, each v keeps a list of values for each of Top($v.heap$), $v.H$, and $v.L$. Each entry in the list is a pair: the value of the field (either a rectangle or a single number) along with a range of x for which that value is valid. Lists are sorted so that ranges—which in each case form a partition of the entire range of x-coordinates—are increasing. These lists are computed in a second preprocessing phase—after computing the skeleton of the segment tree and the event schedule—by the method described below. Then the algorithm proceeds as before, only now no action need be taken to insert or delete a rectangle R from a data structure $v.heap$. A pointer into each list is maintained, and as the sweep proceeds these pointers are advanced in order to obtain current values of each of the three fields. The active region list remains unchanged.

The first step in the second preprocessing phase is to determine for each v a sorted list of $R.z$ values for all R ever stored at v. This can be accomplished in time $O(n \log n)$ by sorting all rectangles by $R.z$, then computing a list of relevant v's for each rectangle, and finally filling in the sorted lists for each v with a single pass

through the rectangle list. Similarly, we compute a sorted list of $R.x1$ and $R.x2$ values, corresponding to insertions and deletions (labeled so we can tell them apart), for each v.

Let us now focus on a single v. Since a sorted list of relevant $R.z$ values is known, we can represent these values by their *ranks*, that is integers between 1 and m (ties are broken arbitrarily), where m is the number of rectangles ever stored at v. Computing the list of Top($v.heap$) values now amounts to an off-line "extract maximum" problem: given a sequence of *insert* operations—the $R.x1$ values, *delete* operations—the $R.x2$ values, and *findmax* operations—performed say after each *insert* or *delete*, report the output of the *findmax* operations. This problem is slightly more general than a problem solved by Hopcroft and Ullman [AHU] and improved by Gabow and Tarjan [GT]. The same solution method using *Union-Find* works; we sketch it below.

LEMMA 3. *The output of a sequence σ of $O(m)$ insert, delete, and findmax operations on integers between* 1 *and m can be computed off-line in linear time.*

Proof. Assume for simplicity that each integer is inserted and deleted exactly once and that there are $2m$ *findmax* operations. We can write σ as $\sigma_1 F_1 \sigma_2 F_2 \cdots \sigma_m F_{2m} \sigma_{2m+1}$, where each σ_i consists only of *insert* and *delete* operations and each F_i stands for *findmax*. In an initial linear-time pass through σ, we compute for each integer j the index i_j such that j is inserted in σ_{i_j} and the index d_j such that j is deleted in σ_{d_j}. We also create $2m$ disjoint sets, one for each F_i. Each set is named by a "canonical index"; initially the set containing F_i is named i. We then execute the following code, where Find(i) returns the canonical index of the set containing F_i, and *Union* combines two sets (specified by canonical indices) and chooses the larger canonical index to be the canonical index of the combined set. (See [T] for more on *Union-Find.*)

```
A := array for storing output
for j = m, m − 1, ..., 1 do
   i := Find(i_j)
   while i < d_j do
    A[i] := j
    Union(i, Find(i + 1))
    i := Find(i + 1)
  od
od
```

Because the *Union* pattern is constrained, a linear-time version of *Union-Find* suffices [GT]. ∎

The final step of the preprocessing is to compute the lists of $v.H$ and $v.L$ values for each node v. These lists are computed in a postorder traversal of the segment tree. If v is a leaf then both lists are identical to the Top($v.heap$) list. If v is not a leaf, then computing the $v.H$ list amounts to a kind of merging of the H lists for

the children of v and the list of Top($v.heap$) values. A similar linear-time merge serves to compute the $v.L$ list.

The total length, over all v, of Top($v.heap$), $v.H$, and $v.L$ lists is $O(n \log n)$ as each rectangle is stored at $O(\log n)$ nodes. The computation of each list requires only linear time, thus we may conclude that the running time of the entire algorithm is now $O(n \log n + k \log n)$.

THEOREM 2. *Hidden surface removal for a static set of n rectangles can be computed in time $O(n \log n + k \log n)$, where k is the length of the output.*

In the algebraic decision tree model of computation, hidden surface removal for static rectangles must take at least $\Omega(n \log n + k)$ time as one can easily reduce element uniqueness to this problem. (See [PS] for background on this type of lower bound.)

4. THE DYNAMIC SETTING

The static algorithm above uses a data structure designed to hold 1-dimensional objects to solve a 3-dimensional (or, perhaps, 2.5-dimensional) problem. In order to give output-sensitive dynamic algorithms, we resort to a mixture of higher dimensional data structures. There are two primary data structures, each comprising a number of substructures, used by the algorithms given in Sections 5 and 6. In this section, we describe the structures from a functional point of view, deferring their design until Section 7.

The first data structure, denoted V, stores the visible scene. What it actually stores are the line segments bounding visible regions (called *visible line segments* below). Regions, even "background" regions of height $-\infty$, are considered separately, and each segment along a region boundary is stored once, along with the name of the rectangle visible within that region. (Thus a line segment in V is really a segment–region pair.) Endpoints are ordered so that the bounded region is on the right as the segment is traversed from the first endpoint to the second. Thus, viewed pieces of edges are split into two oppositely oriented sides at different heights (the z-coordinate of a segment is inherited from the rectangle visible within the bounded region), as in Fig. 4.

Data structure V supports the operations *VInsert* and *VDelete* that insert and delete segments, and three types of queries, called *Below*, *LeftmostOn*, and *Locate*. *Below* takes a rectangle R as its argument and returns all line segments in V that intersect the semi-infinite box $B(R) = [R.x1, R.x2] \times [R.y1, R.y2] \times [-\infty, R.z]$. LeftmostOn($R, s$), where s is a vertical line segment, returns the leftmost vertical line segment in V that has x-coordinate at least $s.x$ (where the fields of s are defined in the obvious way), has y-extent intersecting $[s.y1, s.y2]$, and bounds a visible region of R. (For us, visible segments are oriented, but intervals are not, thus $[s.y1, s.y2] = [s.y2, s.y1]$.) *Locate* takes an (x, y) coordinate pair as argument

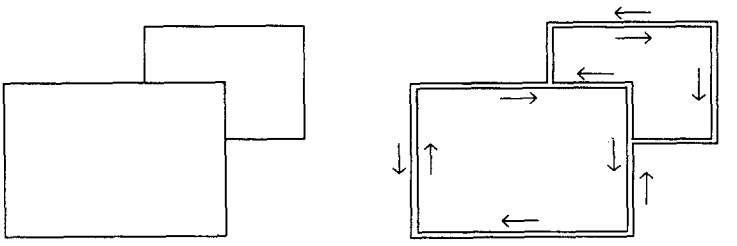

FIG. 4. *A* scene stored in *V* as 18 directed segments.

and returns the name of the highest rectangle along the line of sight through that coordinate pair.

The second data structure, denoted *W*, stores all rectangles, whatever their visibility. *W* supports the operations *WInsert* and *WDelete*, and the queries *MaxInStrip* and *LeftmostAbove*. MaxInStrip(s), where s is a vertical segment, returns the highest rectangle intersecting the infinite strip $[s.x, s.x] \times [s.y1, s.y2] \times [-\infty, \infty]$. LeftmostAbove($s$), where s is a vertical segment, returns the rectangle R with minimum $R.x1$ such that the left edge of R intersects the semi-infinite box $[s.x, \infty] \times [s.y1, s.y2] \times [s.z, \infty]$.

5. INSERTING A RECTANGLE

Inserting a rectangle into the visible scene relies on data structure *V*. Roughly speaking, Insert(R) first performs the query Below(R) to find all affected line segments, deletes these, and then reinserts the pieces of segments that bound regions of R in the updated scene. Pseudocode for the dynamic algorithms is written at a higher level then pseudocode above:

```
procedure Insert (R: rectangle)
  S, C, C_in, C_out, B_int, B_ext: Set of segments; R': rectangle
  S := Below(R)
  if S = ∅ then
    R' := Locate(R.x1, R.y1)
    if R.z > R'.z then VInsert boundary segments of R (appropriately split) fi
  else
    VDelete members of S
    C := members of S that stab B(R)
    Cut members of C at the boundaries of B(R), producing C_in and C_out
    Discard doubled segments from C_in and S\C
    Merge segments of C_in and S\C if necessary
    Compute B_int and B_ext, boundary segments of R induced by C_in and C_out
    Update z and RName fields of segments in S\C, C_in, C_out, B_int, B_ext
    VInsert members of S\C, C_in, C_out, B_int, B_ext
  fi
  Output changes to visible scene
  WInsert(R)
```

Figure 5 shows a top view of the insertion of a rectangle R, shown by broken lines. The scene above R is shown by gray shading and by thin black lines; these lines are unaffected by the insertion. R will prove to be visible in two regions, lower left and upper right. The first line of *Insert* retrieves the set S of visible segments intersecting $B(R)$. If S is empty, then either R is completely visible, floating in the middle of some visible region, or R is completely hidden; *Locate* discriminates between these two cases. If S is nonempty, then all members of S are deleted from V and processed. In Fig. 5a, the bold black lines show members of S; those that cross broken segments are also members of C, the set of visible segments that lie only partially in $B(R)$. Bold line segments that appear doubled are those with both sides below R.

Figure 5b illustrates the step in *Insert* in which members of C are cut; C_{in} (C_{out}) consists of the portions of segments in C that lie inside (outside) $B(R)$. Figure 5c illustrates the next step, discarding doubled segments. In Figure 5d the two bold vertical segments in the lower left have been merged as these are pieces of the same border of a region of R. Figure 5d also shows the boundary of R divided into visible segments: B_{int}, the set of segments that bound visible regions of R, and B_{ext}, the set of segments that bound regions lying outside of $B(R)$. B_{int} and B_{ext} are computed by sorting C_{out} in order around the perimeter of R. The *RName* fields (the name of the rectangle seen to the right) of B_{ext} segments are determined using this sorted order and the *RName* fields of the C_{out} segments. Surviving segments of $S \backslash C$, C_{in}, and B_{int} all receive an *RName* field of R. In all cases, the z field of a segment gives the z-coordinate of the rectangle named in the *RName* field. Finally, output is reported, and R is inserted into W.

We say that a visible line segment s *changes* with an update if it is added or removed from V, or if any field of s changes. Thus, s is considered changed if it borders a newly visible region, even if a segment with identical x- and y-coordinate fields existed before the update. Referring to the pseudocode above, the set of changed line segments is precisely $S \cup T$, where T is the set of segments inserted into V in the last line of the *else* clause in *Insert*.

Let $k = |S|$; so long as $k > 0$, k is within a constant factor of the number of visible

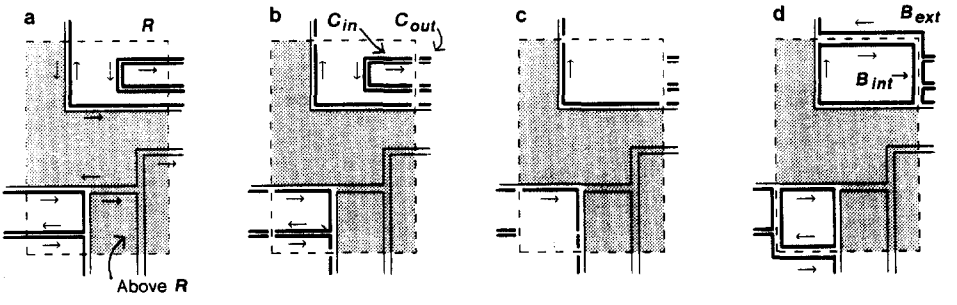

FIG. 5. (a) S is shown in bold. (b) Cutting C. (c) Doubled C_{in} lines removed. (d) B_{int} and B_{ext} added.

line segments that change. It is not hard to confirm that no step of *Insert* is slower than $O(k \log k)$, the time needed to sort S (to find doubled segments or compute B_{int} and B_{ext}). In Section 7, we give the times for *Below*, *VInsert*, *VDelete*, and *WInsert*; these determine the overall running time of *Insert*.

6. Deleting a Rectangle

Deleting a rectangle seems to be necessarily more complicated than inserting a rectangle. The call Delete(R) sweeps a *frontier* from left to right across R, maintaining the invariant that the scene to the left of the frontier is already computed. It advances the frontier with repeated *LeftmostOn*, *LeftmostAbove*, and *MaxInStrip* queries.

The frontier is a sequence of vertical line segments, $s_1, s_2, ..., s_l$, oriented so that $s_i.y1 < s_i.y2$ and such that the y-extents of the segments exactly cover the y-extent of R, that is, $s_1.y1 = R.y1$, $s_l.y2 = R.y2$, and $s_i.y2 = s_{i+1}.y1$ for each i from 1 to $l-1$. A frontier segment s_i also has a field $s_i.RName$; if this field contains the name of a rectangle R', then lines of sight passing just to the right of s_i intersect a visible region of R' and no other visible regions. We say that R' is visible *along* s_i. The *RName* field can also contain a marker *AboveR* which indicates that whatever is visible along s_i, maybe more than one region, has z-coordinate greater than $R.z$.

```
procedure Delete (R: rectangle)
  F, S: List of segments; s, s', s'': segment; R': rectangle
  WDelete(R)
  F := Initial Frontier (R)
  while ∃s in F such that s.x < R.x2 do
    s := Bottom leftmost segment of F
    if s.RName = AboveR then
      s' := LeftmostOn (R, s)
      s'' := part of s' within y-extent of s
      S := DivideUp(s''), possibly along with portions of s translated to s'.x
      Advance F to S
      Store active horizontal segments
      VDelete old visible segments, VInsert new segments, and report output
    else
      R' := LeftmostAbove(s)
      if R'.x1 < s.RName.x2 then
        S := s translated to left edge of R' and broken into pieces bordering R and R'
        Advance F to S
        Store active horizontal segments
        VDelete old visible segments, VInsert new segments, and report output
      else
        S := DivideUp(s'), where s' is translation of s to right edge of s.RName
        Advance F to S
        Store active horizontal segments
        VDelete old visible segments, VInsert new segments, and report output
    fi
  od
```

Delete starts by removing R from data structure W. It then calls a function *InitialFrontier* (pseudocode omitted) to set up an initial frontier $s_1, s_2, ..., s_l$, where each $s_i.x = R.x1$. That is, the initial frontier lies entirely along the left edge of R, though it may be divided into a number of segments with different z and *RName* fields. *InitialFrontier* would use *LeftmostOn* queries to find the visible portions of R's left edge; these portions are then subdivided by *DivideUp* according to what is about to become visible beneath R. The remaining invisible portions of the left edge of R, if any, are added to the frontier with *RName* field set to *AboveR*.

```
function DivideUp (s: frontier segment): list of frontier segments
  S,S′: List of segments; s, s′: segment; R′: rectangle
    Create empty segment lists S and S′
    Insert s into S
    while S is nonempty do
      s := any segment in S
      Delete s from S
      R := MaxInStrip(s)
      s′ := portion of s above R
      s′.RName := R
      s′.z := R.z
      Insert s′ into S′ and remaining parts of s into S
    od
    return(S′)
```

Once the initial frontier is computed, the sweep begins. The bottom leftmost (that is, the (x, y)-lexicographically first) segment s of the frontier is selected, and we *advance* F to the right along s, that is, replace s by a sequence of segments lying to its right. There are three ways we can advance.

Figure 6a illustrates the first way, handled in the four lines after the first *then* in *Delete*, in which the scene visible along s is above R. *LeftmostOn* discovers the visible segment s' bounding a region of R that would be first touched by a translation of s to the right; s' is then trimmed, if necessary, to fit within the y-extent of s, yielding s''. *DivideUp* is called to discover the scene below s''. The portion of the frontier segment s with the same y-extent as s'' is then replaced by the sequence S of frontier segments computed by *DivideUp*. The remaining portions of s, if any, are

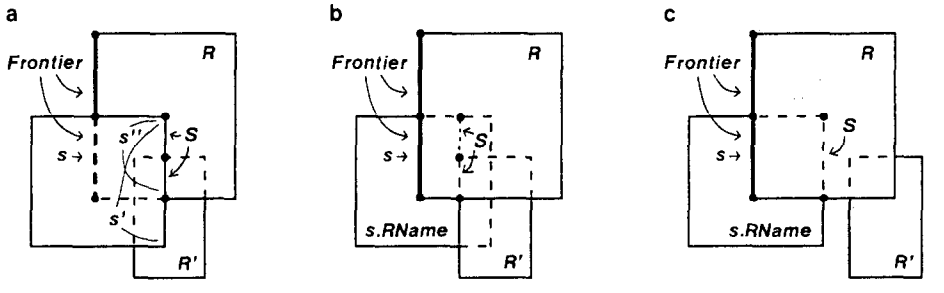

FIG. 6. (a) *s.RName* = *AboveR*. (b) Encountering R'. (c) Right edge of *s.RName*.

simply advanced in x-coordinate, with *RName* fields still set to *AboveR*. In Fig. 6a, S contains two segments, the *RName* fields of which are *background* and R', respectively. A horizontal visible segment along the top of the rectangle above R is now inserted into V with *RName* field set to *background*. The two vertical visible segments of S are also inserted into V; moreover, the one with the smaller y-coordinate is also merged with an existing segment in V.

In both the second and third ways to advance, the region visible along s lies below R. Figure 6b illustrates the second way, in which another rectangle R', either above or below R, with left edge to the right of s, partially covers the region visible along s. R' is discovered by *LeftmostAbove*. In this case, s is advanced and broken into pieces bounding $s.RName$ and R'.

In the third way, shown in Figure 6c, the right edge of $s.RName$ is encountered before a higher rectangle. *DivideUp* is used to discover the rectangles lying below this right edge. In the case illustrated, *DivideUp* would return a list S, containing only a single segment with an *RName* field of *background*.

We must be able to insert and delete frontier segments and also find bottom leftmost frontier segments efficiently. To avoid excessive fragmentation, we also need to merge adjacent segments s_i and s_{i+1} if these two segments have identical x, z, and *RName* fields. A balanced binary tree such as a red–black tree supports insertion, deletion, and merging in logarithmic time. We use such a tree for each list of segments in the pseudocode above. In addition if each node of the tree stores the bottom leftmost segment within its subtree, we can find the overall bottom leftmost segment in $O(1)$ time. The key observation needed to give an output sensitive bound on the running time of *Delete* is the following.

LEMMA 4. *The total number of calls to* LeftmostOn, LeftmostAbove, *and* MaxInStrip *in an execution of* Delete *is linear in the number of visible line segments that change.*

Proof. First we assert that the bottom leftmost frontier segment s is always at least *half-maximal*, that is, lengthening s in one direction or the other either makes two regions visible along s or extends s beyond the top or bottom edge of R. Every segment is half-maximal (in fact, *maximal*) in the initial frontier. When a segment s is replaced by more than one segment to the right of s, each new frontier segment is maximal, except for the top and bottom new segments which may be only half-maximal. When s is replaced by exactly one segment s' to its right, then s' is either half-maximal or it is a proper subsegment of a visible segment, in which case, s' will merge with another segment (either immediately or after a neighboring segment advances) before it becomes bottom leftmost. Second, we assert that every half-maximal frontier segment shares an endpoint with a visible line segment that changes. This assertion holds for each segment in the initial frontier. It also holds for each half-maximal segment that results from advancing and merging operations, as each advancing operation moves to a line segment that changes. For example, in Fig. 6b, the upper segment of S will sometime serve as the bottom, leftmost seg-

ment; its lower endpoint is coincident with the upper left corner of R'. Notice that the *AboveR* marker is crucial to this second assertion, as it allows the frontier to skip over unchanged parts of the scene. Together our two assertions imply that the number of segments that ever serve as s in the main *while* loop is linear in the number of visible line segments that change.

LeftmostOn and *LeftmostAbove* are each called at most once per execution of the main *while* loop, thus we need only bound the number of calls to *MaxInStrip*. *MaxInStrip* is repeatedly called by *DivideUp* to break a vertical segment s' into frontier segments. Each of these frontier segments will, at some time, become bottom leftmost, except for the top and bottom pieces of s' which may merge with other frontier segments. We charge each *MaxInStrip* call to the discovered frontier segment, except for the two calls that discover top and bottom pieces, which we charge to the current bottom leftmost frontier segment. Altogether, each bottom leftmost segment s is charged for at most three *MaxInStrip* calls. ∎

7. Data Structures

In this section we discuss the substructures of V and W that handle each of the five queries. Of course, *VInsert* and *VDelete* (*WInsert* and *WDelete*) must update each of the substructures of V (W). All the substructures use a multilevel tree scheme; that is, there is an outer tree in which each node includes an inner tree, and so on. A difficulty with these schemes is that a rebalancing operation, such as a rotation, on the outer tree may require that inner trees be completely rebuilt. Willard and Lueker show how to use weight-balanced binary trees (BB(α) trees) to ensure that rebalancing d-level trees is not too expensive, giving insertion and deletion times of $O(\log^d n)$ [WL]. We use two and three-level BB(α) tree structures (as in [EM]), and also two-level structures in which the outer level is a BB(α) tree and the inner trees are McCreight's priority search trees (as in [McC, E]). For the best asymptotics, the innermost tree in a multilevel BB(α) structure will be the priority queue used in dynamic fractional cascading [FMN, CG]. Below we call a BB(α) tree a *dynamic segment tree*, a slight misnomer when the ultimate data objects are points rather than segments. By *list*, we mean a balanced binary search tree, supporting insertion, deletion, and look-up by key in logarithmic time.

Below query. The *Below* query must return each segment intersecting the semi-infinite box $B(R)$ below R. We show how to build a data structure for horizontal segments; vertical segments are handled analogously. If a horizontal segement s intersects $B(R)$, either (1) the endpoint $(s.x1, s.y, s.z)$ lies inside $B(R)$, or (2) s stabs a face of $B(R)$ normal to the x-axis. To handle case (1), we build a dynamic segment tree by x-coordinate in which each node v has an associated priority search tree by y and z-coordinates. That is, node v corresponds to segment(v) and has a field $v.tree$ that is a priority search tree holding all endpoints with $x1$ field within segment(v). A priority search tree answers queries of the form "return all points

with y-coordinate within the range $[R.y1, R.y2]$ that have z-coordinate less than $R.z$" in time $O(\log n + k)$, where k is the length of the output [McC]. A given *Below* query generates a priority search tree query at each of the $O(\log n)$ node corresponding to basic segments of $[R.x1, R.x2]$.

Case (2), that is, s stabs a face of $B(R)$, is handled in a dual manner. We build a dynamic segment tree by x-coordinate in which each node has an associated priority search tree. Each line segment s is stored at nodes v in the outer tree that correspond to basic segments of s; at v, s is stored in $v.tree$ according to its y- and z-coordinates. A query follows a path from the root to a leaf in the outer tree, exploring each node v such that segment(v) contains the x-coordinate of the query face; at each v, $v.tree$ answers a semi-infinite range query as in case (1). Altogether, in each case, we have insert and delete times of $O(\log^2 n)$ and query times of $O(\log^2 n + k)$, as priority search trees support insert and delete in $O(\log n)$ time [McC].

LeftmostOn query. LeftMostOn(R, s) returns the leftmost segment s' bounding a visible region of R such that $s.x \leqslant s'.x$ and the y-extents of s and s' intersect. Since *LeftmostOn* queries for different rectangles are entirely separate, we create a separate data structure for each rectangle R, and a dictionary (another balanced binary tree) for the "names" of rectangles. After $O(\log n)$ time for the dictionary look-up, we find the data structure for R.

The data structure for R will again be divided into two cases. In case (1), an endpoint of segment s' has y-coordinate within the y-extent of s. Thus, the query object is a line segment and the data are points. We use a dynamic segment tree by y-coordinate, in which each node v stores all endpoints with y-coordinate within segment(v) in a list sorted by x-coordinate. This gives time $O(\log^2 n)$ for insert and delete, and $O(\log^2 n)$ query time. All times are improvable to $O(\log n \log\log n)$ with dynamic fractional cascading, as both updates and queries fit the pattern of repeatedly searching sorted lists, which are connected in a tree, using the same key.

In case (2), s' spans the y-extent of s. We can now use either endpoint of s as the query object and let the data be vertical line segments. Again, a dynamic segment tree by y-coordinate, in which each segment list is sorted by x-coordinate, will handle this case with $O(\log^2 n)$ insert, delete, and query time. Once again, all times are improvable to $O(\log n \log\log n)$.

Locate query. *Locate* returns the visible rectangle stabbed by a given line of sight. To do this, we find the leftmost visible line segment with y-extent containing the y-coordinate of the line of sight, and with x-coordinate at least the x-coordinate of the line of sight. The *RName* field of the appropriately oriented segment answering this query gives the name of the desired rectangle. This query is handled by the same type of data structure as case (2) of *LeftmostOn*, only for the *Locate* query all visible segments, not just those bordering regions from a single rectangle must be included.

Altogether, the slowest steps of *VInsert* and *VDelete* are the insertions and deletions to the data structures for *Below* which take time $O(\log^2 n)$. The *Below* query

runs in time $O(\log^2 n + k)$, while *Locate* and *LeftmostOn* take $O(\log n \log\log n)$ in their fast and $O(\log^2 n)$ in their slow incarnations. Space requirements are $O(n + q \log n)$, where n (resp. q) is the number of rectangles (visible segments) present.

MaxInStrip query. *MaxInStrip* returns the highest rectangle intersecting a given strip normal to the x-axis, with edges parallel to the z-axis. Two cases: either (1) one of the edges of the strip stabs the highest rectangle, or (2) some edge of that rectangle stabs the strip. Case (1) is handled by a dynamic segment tree by y-coordinate, in which each node has a dynamic segment tree by x-coordinate, in which, in turn, each node has a list of rectangles sorted by decreasing z-coordinate. A query, defined by a line parallel to the z-axis (one of the strip's edges), visits a root to leaf path in the outer tree, a root to leaf path in each of $O(\log n)$ inner trees, and, finally, looks at the highest rectangle in each visited node's rectangle list. This solution gives query time of $O(\log^2 n)$ and update time of $O(\log^3 n)$, this second time improvable to $O(\log^2 n \log\log n)$ with dynamic fractional cascading. Case (2) can be handled in a dual manner by a similar three-level tree. Times are the same as in the first case.

LeftmostAbove query. *LeftmostAbove* returns the leftmost rectangle R above and to the right of a given vertical query segment s. Again, two cases: (1) either some endpoint of R's left edge lies within the semi-infinite box above and to the right of s, or (2) R's left edge stabs one of the faces of this box. The first case is handled with a dynamic segment tree by y, in which each node has a priority search tree by x and $-z$. An endpoint is stored at each node v in the outer tree along a root to leaf path; at each v it is stored in the priority search tree $v.tree$. A query segment s finds each node v such that segment(v) is a basic segment of the y-extent of s, and then asks for the endpoint within $v.tree$ whose x value is minimal within a query box above (that is, below the limit $-s.z$) and to the right of the query point $(s.x, s.z)$. A priority search tree can find the point of minimum x in such a semi-infinite box in time $O(\log n)$. Case (2) is also handled in a dual manner. In both cases, query, insert, and delete times are all $O(\log^2 n)$.

Altogether then, data structure W can be implemented in space $O(n \log^2 n)$ to support *MaxInStrip* and *LeftmostAbove* queries in time $O(\log^2 n)$. The time to insert or delete a rectangle is $O(\log^3 n)$, improvable to $O(\log^2 n \log\log n)$ using dynamic fractional cascading.

THEOREM 3. *Hidden surface removal for a dynamic set of rectangles can be computed in time $O(\log^3 n + k \log^2 n)$ per insertion or deletion ($O(\log^2 n \log\log n + k \log^2 n)$ using dynamic fractional cascading) and space $O(n \log^2 n + q \log n)$, where k is the number of visible line segments that change, and n and q are, respectively, the number of rectangles and the number of visible line segments at the time of the update.*

Proof. *Insert* uses a single *Below* query, a single *WInsert* operation, and $O(k)$ *VInsert* and *VDelete* operations. These cost $O(\log^2 n + k)$, $O(\log^3 n)$ (or

$O(\log^2 n \log\log n)$ with fractional cascading), and $O(k \log^2 n)$ time, respectively. As we saw in Section 5, *Insert also involves* $O(k \log k)$ other work. *Delete* uses a single *WDelete* operation, $O(k)$ *LeftmostOn*, *LeftmostAbove*, and *MaxInStrip* queries, and $O(k)$ *VInsert*, *VDelete*, and frontier segment merging operations (implicit in the advancing). Altogether we obtain the claimed time and space bounds. ∎

In windowed systems, a rectangle is typically constrained to be completely visible when inserted. By using a data structure such as the finger search tree in [GMPR], which supports "insert at front" and "report the first item" in $O(1)$ time, and delete in $O(\log n)$ time, we obtain the following theorem.

THEOREM 4. *If each rectangle is completely visible when it is inserted, insertions can be handled in time* $O(\log^2 n + k \log^2 n)$ *without fractional cascading.*

8. MOUSE-CLICK LOCATION

Typically windowed systems are used on workstations equipped with a mouse. Mouse-clicks switch the keyboard input stream from one window to another, so a fundamental problem is to locate the window containing a given mouse-click, that is, which rectangle is the highest along a given line of sight.

For the static case, notice that our modified segment tree, as it exists between successive x-coordinates x_i and x_{i+1}, can handle query points with x-coordinate between x_i and x_{i+1} in logarithmic time. Such a query considers the top rectangle in each heap along a root-leaf path and picks the highest top rectangle. An efficient solution to the problem for arbitrary x is obtained by making the segment tree *persistent* [DSST, ST]. More precisely, the correct values of Top(v.*heap*) should be accessible in $O(1)$ time per node after an initial $O(\log n)$ binary search by x-coordinate. Sarnak and Tarjan show how to use "path-copying" along with a small number of extra pointers per node to achieve this goal without increasing the space complexity [ST]. Notice that only Top(v.*heap*), rather than all of v.*heap*, need be copied for each v. If we are using the "theoretical" version of the segment tree, in which lists of Top(v.*heap*) values are already compiled, then persistence amounts to providing "bridges" between the list at v and the lists at v's children, as in (static) fractional cascading [CG]. Notice that in either case, we obtain a better space bound, namely $O(n \log n)$, than can be obtained by treating a complicated rectangle scene as a general polygonal subdivision.

For the dynamic case, the query *Locate*, described above, gives an efficient, though not optimal, solution.

THEOREM 5. *Which window contains a given mouse-click can be determined in time* $O(\log n)$ *in the static and* $O(\log^2 n)$ ($O \log n \log\log n$) *using dynamic fractional cascading*) *in the dynamic case. Adding this capability does not change other space or time bounds.*

9. Conclusions

We have given algorithms that solve window management problems efficiently in a theoretical sense. The number of windows in computer displays, however, is currently not large enough to warrant the use of these algorithms. Our algorithms might someday find use in cartographic applications or in VLSI design tools for many-layer technologies.

We mention some avenues for further research. Naturally, it would be nice to give algorithms with better time or space bounds. Two questions concerning the static case seem particularly interesting. It is possible to reduce the $k \log n$ factor to k without increasing the dependence on n by more than a logarithmic factor? Does there exist an optimal, $O(n)$-space, $O(\log n)$-query-time data structure for answering mouse-click queries? Cleaner algorithms for the dynamic case, even with the same time and space bounds, would be an improvement. Finally, we would like to see output-sensitive algorithms for more general hidden surface removal problems.

Acknowledgments

I thank David Dobkin, Dan Greene, and Leo Guibas for many valuable discussions and a referee for a very careful reading.

References

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

[AG] M. J. Atallah and M. T. Goodrich, "Output-Sensitive Hidden Surface Elimination for Rectangles," Tech. Report 88-13, Computer Science Dept., Johns Hopkins University, 1988.

[CG] B. Chazelle and L. J. Guibas, Fractional cascading. I. A data structuring technique, *Algorithmica* **1** (1986), 133–162.

[DSST] J. R. Driscoll, N. Sarnak, D. Sleator, and R. E. Tarjan, Making data structures persistent, *in* "Proceedings, 18th ACM Symp. on Theory of Computing, 1986," pp. 109–121.

[E] H. Edelsbrunner, A note on dynamic range searching, *Bull. EATCS* **15** (1981), 34–40.

[EM] H. Edelsbrunner and H. A. Maurer, On the intersection of orthogonal objects, *Inform. Process. Lett.* **13** (1981), 177–181.

[FMN] O. Fries, K. Mehlhorn, and S. Näher, Dynamization of geometric data structures, *in* "Proceedings, 1st ACM Symp. on Comp. Geometry, 1985," pp. 168–176.

[GT] H. Gabow and R. E. Tarjan, A linear-time algorithm for a special case of disjoint set union, *in* "Proceedings, 15th Annual ACM Symp. on Theory of Computing, 1983," pp. 246–251.

[G] M. T. Goodrich, A polygonal approach to hidden-line elimination, *in* "Proceedings, 25th Allerton Conf. on Comm., Control, and Comput. 1987," pp. 849–858.

[GMPR] L. J. Guibas, E. M. McCreight, M. F. Plass, and J. R. Roberts, A new representation for linear lists, *in* "Proceedings, 9th ACM Symp. on Theory of Computing, 1977," pp. 49–60.

[GO] R. H. Güting and T. Ottmann, New algorithms for special cases of the hidden line elimination problem, *in* "Proceedings, Symp. on Theor. Aspects of Comput. Sci., 1985."

[McC] E. M. McCreight, Priority search trees, *SIAM J. Comput.* **14** (1985), 257–276

[McK] M. McKenna, Worst-case optimal hidden surface removal, *ACM Trans. Graphics* **6** (1987), 19–28.

[M] B. MYERS, A complete and efficient implementation of covered windows, *Computer* **19** (1986), 57–69.

[PS] F. P. PREPARATA AND M. I. SHAMOS, "Computational Geometry: An introduction," Springer-Verlag, Berlin/New York, 1985.

[PVY] F. P. PREPARATA, J. S. VITTER, AND M. YVINEC, Computation of the axial view of a set of isothetic parallelepipeds, manuscript, 1988.

[ST] N. SARNAK AND R. E. TARJAN, Planar point location using persistent search trees, *Comm. ACM* **29** (1986), 669–679.

[T] R. E. TARJAN, "Data Structures and Network Algorithms," SIAM, Philadelphia, 1983.

[VKZ] P. VAN EMDE BOAS, B. KAAS, AND E. ZIJLSTRA, Design and implementation of an efficient priority queue, *Math. Systems Theory* **10** (1977), 99–127.

[WL] D. E. WILLARD AND G. S. LUEKER, Adding range restriction capability to dynamic data structure, *J. Assoc. Comput. Mach.* **32** (1985), 597–617.