International Journal of Computational Geometry & Applications

THE OBJECT COMPLEXITY MODEL FOR HIDDEN-SURFACE REMOVAL

EDWARD F. GROVE*
946, Tamarack Lane #13
Sunnyvale, CA 94086, U.S.A.
Email: `eddie@math.uri.edu`

T. M. MURALI†
Center for Geometric Computing
Department of Computer Science, Duke University
Box 90129, Durham, NC 27708–0129, U.S.A.
Email: `tmax@cs.duke.edu`

and

JEFFREY SCOTT VITTER‡
Center for Geometric Computing
Department of Computer Science, Duke University
Box 90129, Durham, NC 27708–0129, U.S.A.
Email: `jsv@cs.duke.edu`

Received 10 June 1997
Revised 16 February 1998
Communicated by H. Imai

ABSTRACT

We define a new model of complexity, called *object complexity*, for measuring the performance of hidden-surface removal algorithms. This model is more appropriate for predicting the performance of these algorithms on current graphics rendering systems than the standard measure of scene complexity used in computational geometry.
We also consider the problem of determining the set of visible windows in scenes consisting of n axis-parallel windows in $\mathbb{R}^3$. We present an algorithm that runs in optimal $\Theta(n \log n)$ time. The algorithm solves in the object complexity model the same problem that Bern[3] addressed in the scene complexity model.

Keywords: Object complexity, hidden-surface removal, window visibility, segment tree, computational geometry, computer graphics.

---

*This work was done when the author was at Duke University. Support was provided by Army Research Office grant DAAH04–93–G–0076.

†This author is affiliated with Brown University. Support was provided in part by National Science Foundation research grants CCR–9007851 and CCR–9522047, by Army Research Office grant DAAL03–91–G–0035, and by Army Research Office MURI grant DAAH04–96–1–0013.

‡Support was provided in part by National Science Foundation research grants CCR–9007851 and CCR–9522047, by Army Research Office grant DAAH04–93–G–0076, and by Army Research Office MURI grant DAAH04–96–1–0013.

1. The Object Complexity Model

How to render a set of opaque or partially transparent objects in $\mathbb{R}^3$ quickly and in a visually realistic way is a fundamental problem in computer graphics.[11,20] A central component of rendering is *hidden-surface removal:* given a set of objects, a viewpoint, and an image plane, compute the scene visible from the viewpoint as projected onto the image plane.

Various hidden surface algorithms have been proposed in the computational geometry literature. Such algorithms typically compute the *visibility map*, which is a partition of the image plane into regions with the property that in each region, at most one object is visible. The size of the visibility map is often called the *scene complexity.* Worst-case optimal algorithms are presented by Dévai and McKenna.[9,13] The running time of more recent algorithms depends on the input size and the scene complexity.[3,7,8,12,17,18,19] The fastest known algorithm for hidden-surface removal takes time $O(n^{2/3+\epsilon}k^{2/3} + n^{1+\epsilon})$, where n is the size of the input and k is the scene complexity.[1]

The computer graphics community, which is the source of the problem, has also studied hidden-surface removal extensively. Sutherland, Sproull and Schumacker[20] survey early hidden surface removal algorithms used in graphics. More recently, algorithms have been developed for walkthrough systems.[2,5,6,21] The aim here is to visually simulate the experience of walking inside an environment like a building using an architectural model of the building. The simulation achieves realism when 20–30 scenes are generated and displayed per second.

A conceptually simple solution to the hidden-surface removal problem is the z-buffer algorithm.[4,11] This algorithm sequentially processes the input objects; for each object, it updates the pixels of the image plane covered by the object, based on the distance information stored in the z-buffer. Assuming that the input objects are triangles, the time taken by the z-buffer algorithm is proportional to the number of triangles processed by it, except in the atypical case where the triangles are extremely large, when the processing cost is dominated by the number of pixels covered by the triangles.

A very fast hidden-surface removal algorithm can be obtained by implementing the z-buffer in hardware. For example, the Silicon Graphics InfiniteReality system is capable of rendering more than seven million triangles per second.[15] Fast as the z-buffer is, datasets are becoming so huge that even the fastest z-buffer cannot render them in real time. Some aircraft models consist of tens of millions of triangles, and submarine models may have a billion triangles. This problem is compounded for interactive real-time applications like walkthrough systems.[2,21] In such applications, new scenes need to be generated about 20–30 times a second. Processing all the input through the z-buffer at these rates is currently not possible. If the visible scene is to be displayed in real time, it is imperative that the z-buffer should process only (a small superset of) the visible triangles. This strongly motivates the development of provably-fast algorithms for determining a small superset of the visible triangles (a.k.a. "occlusion culling") so that the requirements on z-buffers are eased.

Conceptually, algorithms that compute the visibility map are ideal for culling away invisible objects since they determine the exact set of visible objects. In practice, however, they are not appropriate for use with the z-buffer since their running time depends on scene complexity. The scene complexity for n objects can be $\Omega(n^2)$ in the worst-case; in such cases, it is likely that the z-buffer can process the original n objects much faster than it can process the $\Omega(n^2)$ faces of the visibility map.

Motivated by this disparity between the theoretical model of scene complexity and the performance characteristics of current graphics rendering hardware, we propose a more realistic model of complexity, called *object complexity*, in which the size of a scene is measured in terms of the number of objects visible in the scene. Object complexity is always at most n, the number of objects in the input and hence can be much less than the scene complexity (which can be $\Omega(n^2)$). This happens, for example, when the viewpoint is at $z = +\infty$ and the scene contains $n/2$ thin rectangles parallel to the x-axis lying directly above $n/2$ thin rectangles parallel to the y-axis. See Figure 1. Algorithms whose running time depends on scene complexity can be used trivially to determine visible objects by outputting all the objects that contain segments in the view. However, in the worst case, this technique might entail spending $\Omega(n^2)$ time to output only $O(n)$ distinct objects.

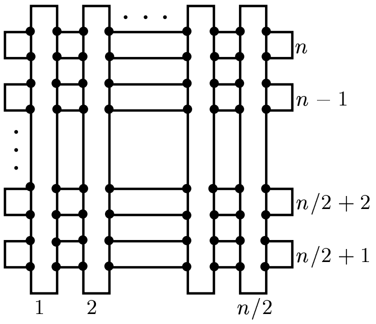

Fig. 1. A set of n objects with object complexity n and scene complexity $\Omega(n^2)$. Vertices of the visibility map are indicated by black dots.

Thus, in the object complexity model, our goal is to develop efficient algorithms for hidden-surface removal whose running time depends on the input size and the object complexity of the scene.

2. Hidden-Surface Elimination in Static Scenes

Our model of object complexity is relevant not only for dynamic scenes as mentioned above but also for static scenes like the one we address in this paper. Most machines do not have z-buffers and must resort to software z-buffers, or else they have hardware z-buffers that perform at a fraction of the speed of a state-of-the-art z-buffer like the InfiniteReality. In such cases, the speed of the rendering process is considerably heightened by a fast and efficient software algorithm which culls all but a small superset of the visible triangles and feeds only these to the z-buffer. Even in machines with state-of-the-art z-buffers, faster CPUs can put the bottleneck of rendering back on the z-buffer. Thus, an effective culling algorithm can produce a resulting speedup in rendering by decreasing the load on the z-buffer.

In this paper, we study the problem of finding the exact set of rectangles visible from the point $z = +\infty$ in a set of n rectangles with sides parallel to the x- and y-axes. We solve this problem in optimal $\Theta(n \log n)$ time. Bern[3] addresses this problem for the standard scene complexity model. Our algorithm is novel because we cannot afford to maintain information about all the visible segments explicitly (like he does). We maintain this information implicitly by using the segment tree in a clever manner. The following sections describe our technique. Section 3 defines the problem and proves a lower bound. The algorithm is described in Section 4. Section 5 contains the proof of correctness and the analysis of the running time. An improved, optimal algorithm is described in Section 6. Section 7 concludes.

3. Window Visibility Problem

Our input consists of n rectangles, each with sides parallel to the x- and y-axes. We want to report the set of rectangles visible from the point $z = +\infty$. This problem arises in windowing systems where windows are drawn on the screen according to a priority assigned to each window.

Each rectangle R is specified by five numbers, $R.x_1, R.x_2, R.y_1, R.y_2$, and $R.z$ such that $R = [x_1, x_2] \times [y_1, y_2] \times [z, z]$, where $x_1 < x_2$ and $y_1 < y_2$. If two edges belonging to different rectangles have different z-coordinates but project to the same segment in the xy-plane, the edge with larger z-coordinate is considered to obscure the edge with smaller z-coordinate.

Theorem 1 *In the algebraic decision tree model, any algorithm that determines which of n rectangles with sides parallel to the x- and y-axes are visible from $z = +\infty$ requires $\Omega(n \log n)$ tests.*

Proof. It is well known that the problem of determining whether all the members of a set of n real numbers are distinct has a lower bound of $\Omega(n \log n)$.[10]

Suppose we are given a set S of n real numbers. For every element x of S we create a square of side x whose top left corner is at (x, x). These squares are assigned distinct heights. See Figure 2. The point (a, a) on the square corresponding to an element a of S can be obscured only by another square with top left corner at (a, a). Hence, the members of S are distinct if and only if the algorithm to determine the visible rectangles reports n rectangles. $\square$

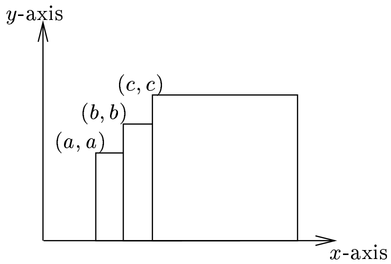

Fig. 2. The rectangle with top left corner (a, a) is not obscured by any other rectangle.

4. Algorithm

We sweep a plane perpendicular to the x-axis from $x = -\infty$ to $x = +\infty$. Event points of the sweep are the coordinates of the vertical edges (the edges parallel to the y-axis) of the rectangles. If more than one vertical edge share the same x-coordinate, they are processed in decreasing order of z-coordinate with right edges processed before left edges.[a] The intersections of the rectangles with the sweep plane are stored in a segment tree $\mathcal{T}$. which we define below. In what follows, the left and right children of node v in the segment tree are u and w, respectively.

Let Y be the set of y-coordinates of the endpoints of the horizontal edges (the edges parallel to the x-axis) of the n input rectangles. The elements of $Y \cup \{-\infty, \infty\}$ partition the y-axis into at most $2n+1$ intervals of the form $[y_i, y_{i+1}), 1 \le i \le 2n+1$, where $y_i, 1 \le i \le 2n+2$, is the ith smallest element in $Y \cup \{-\infty, \infty\}$. The segment tree $\mathcal{T}$ is a height balanced binary tree constructed on the elements of $Y \cup \{-\infty, \infty\}$. Each node v of $\mathcal{T}$ is associated with an interval called its *basic segment*, denoted by b_v. If v is the ith leaf of $\mathcal{T}$ (counting from left to right), then b_v is $[y_i, y_{i+1})$. If v is an internal node of $\mathcal{T}$, then $b_v = b_u \cup b_w$. See the books by Mehlhorn[14] and Preparata and Shamos[16] for more details on segment trees.

A *cross-section* is the one-dimensional intersection of a rectangle with the sweep plane. Each such cross-section is stored as $O(\log n)$ basic segments in $\mathcal{T}$.[16] The following fields are stored at each node v in $\mathcal{T}$.

(i) b_v: the basic segment associated with v.

(ii) v_{mid}: the midpoint of b_v.

(iii) $\mathcal{H}_v$: a heap storing the cross-sections stored at v sorted in decreasing order of z. Each element of $\mathcal{H}_v$ has a flag *unrep* which is true if and only if that element has not been reported as visible so far. The query $\mathsf{top}(\mathcal{H}_v)$ returns the cross-section of maximum z coordinate stored in the heap.

(iv) l_v: the height of the lowest visible cross-section stored in the subtree rooted at v. If there is no such cross-section, then l_v is $-\infty$. Here, visibility is with respect to the cross-sections stored in the subtree rooted at v. Given the value of l_u and l_w, l_v can be calculated as follows:

$$l_v = \max\{\min\{l_u, l_w\}, \mathsf{top}(\mathcal{H}_v).z\}$$

(v) h_v: the height of the highest visible unreported cross-section (which must be the top of the heap of some node) in the subtree rooted at v. Again, visibility is with respect to the cross-sections stored in the subtree rooted at v. If there is no such cross-section, h_v is $-\infty$. The h_v field can be calculated as follows.

$h_v = \max\{h_u, h_w\}$;
if $(\mathsf{top}(\mathcal{H}_v).\mathit{unrep} = \mathit{false})$ **and** $(\mathsf{top}(\mathcal{H}_v).z > h_v)$ **then**
  $h_v = -\infty$;
else
  $h_v = \max\{h_v, \mathsf{top}(\mathcal{H}_v).z\}$;

[a] For a rectangle $[x_1, x_2] \times [y_1, y_2] \times [z, z]$, the *left edge* is the edge with x-coordinate x_1 and the *right edge* is the edge with x-coordinate x_2.

At each event point, the algorithm performs the following two actions:

(i) If an event point corresponds to the left edge of a rectangle R, the corresponding cross-section is inserted into $\mathcal{T}$ using procedure LEFT-INSERT (described below) and each basic segment it is divided into is checked for visibility.

(ii) If an event point corresponds to the right edge of a rectangle R, the corresponding cross-section is deleted from $\mathcal{T}$ using procedure RIGHT-DELETE (described below) and cross-sections which become visible as a result of this deletion are reported.

LEFT-INSERT(R, S, *root*), where S is the background rectangle (that is, $S.z = -\infty$) and *root* is the root of $\mathcal{T}$, inserts the cross-section of a rectangle R into $\mathcal{T}$ by dividing it into $O(\log n)$ cross-sections. At each node where the cross-section of R is stored, it is checked for visibility.

```
procedure LEFT-INSERT(R: rectangle, S: rectangle, v: segment tree node)
    if [R.y1, R.y2] ⊆ b_v then
        insert R into H_v and set the unrep field of R in H_v to true;
        if (R.z ≥ l_v) and (R.z ≥ S.z) then
            report R as visible and set the unrep field of R in H_v to false;
    else
        if (S.z < top(H_v).z) then S = top(H_v);
        if (R.y1 ≤ v_mid) then LEFT-INSERT(R, S, u);
        if (v_mid ≤ R.y2) then LEFT-INSERT(R, S, w);
    update l_v and h_v;
```

RIGHT-DELETE(R, S, *root*), where S and *root* are as defined in LEFT-INSERT, deletes the cross-section of rectangle R from $\mathcal{T}$. RIGHT-REPORT is called at each node where a visible cross-section of R is deleted.

```
procedure RIGHT-DELETE(R: rectangle, S: rectangle, v: segment tree node)
    if [R.y1, R.y2] ⊆ b_v then
        delete R from H_v;
        update h_v, l_v;
        if ((S.unrep = true) and (S.z ≥ l_v))
            report S as visible;
            set the unrep field of S in T to false;
        else
            if (R.z ≥ l_v) and (R.z ≥ S.z) then
                RIGHT-REPORT(S, v);
    else
        if (S.z < top(H_v).z) then S = top(H_v);
        if (R.y1 ≤ v_mid) then RIGHT-DELETE(R, S, u);
        if (v_mid ≤ R.y2) then RIGHT-DELETE(R, S, w);
    update l_v and h_v;
```

RIGHT-REPORT(S, v) is called by RIGHT-DELETE at a segment tree node v where a visible cross-section of a just-deleted rectangle R was stored. RIGHT-REPORT reports all previously unreported cross-sections that become visible as a result of the deletion of R.

```
procedure RIGHT-REPORT(S: rectangle, v: segment tree node)
    if (h_v < S.z) return;
    if (S.z ≤ top(H_v).z) then
        if (top(H_v).unrep = true) and (top(H_v).z ≥ l_v)
            report top(H_v);
            top(H_v).unrep = false;
        S = top(H_v);
    RIGHT-REPORT(S, u);
    RIGHT-REPORT(S, w);
    update h_v;
```

5. Correctness and Analysis

Lemma 1 *The algorithm reports all and only visible cross-sections.*

Proof. The algorithm maintains the invariant that a cross-section s stored at a node v is obscured by the cross-sections stored in the subtree rooted at v iff $s.z$ is less than l_v. The segment tree has the property that for any node v of $\mathcal{T}$, $b_v \subset b_u$ when u is an ancestor of v in $\mathcal{T}$ and $b_v \supset b_u$ when u is a descendent of v in $\mathcal{T}$. This property implies that the highest cross-section s stored at a node v is invisible if and only if

(i) there exists a cross-section r stored at an ancestor of v such that $s.z < r.z$ or
(ii) $s.z < l_v$.

In the procedures LEFT-INSERT, RIGHT-DELETE and RIGHT-REPORT, when we are visiting node v, the rectangle S is the highest rectangle stored at an ancestor of v. Since the checks implied by the above statements are made using S and the l_v field before a cross-section is reported as visible, the algorithm reports only visible cross-sections.

A visible rectangle R has either its left edge visible or a portion of its interior visible. It is easy to see if R's left edge is visible, R is reported as visible when it is inserted into $\mathcal{T}$. If only an interior portion of R is visible, this portion must first become visible during the sweep when some visible rectangle S above R is deleted. Then a call to RIGHT-REPORT at some node in $\mathcal{T}$ where a cross-section of S is stored will report R as visible. This implies that all visible rectangles are reported. It is also clear that the *unrep* field ensures that each visible cross-section is reported only once. □

The analysis of the running time depends on the following key lemma. We say that a node is *marked* if a rectangle is reported as visible when the node is visited by RIGHT-REPORT.

Lemma 2 *Let $\mathcal{U}$ be the subtree of $\mathcal{T}$ explored by a single call to* RIGHT-REPORT. *If two leaves of $\mathcal{U}$ are siblings and unmarked, then their parent is marked.*

Proof. Let the two unmarked leaves be u and w, and let v be their parent. Let S_v, S_u and S_w be the values of S when RIGHT-REPORT visits v, u and w respectively. To show that v is marked we need to show that $\mathsf{top}(\mathcal{H}_v)$ is reported as visible when v is visited by RIGHT-REPORT. We can do this if we show that the following three facts are true:

(i) $\mathsf{top}(\mathcal{H}_v).z \geq S_v.z$,
(ii) $\mathsf{top}(\mathcal{H}_v).\mathit{unrep}$ is *true*, and
(iii) $\mathsf{top}(\mathcal{H}_v).z \geq l_v$.

Since both u and w are leaves of $\mathcal{U}$, we know from the pseudo-code for RIGHT-REPORT that

$$h_u < S_u.z \text{ and } h_w < S_w.z. \tag{1}$$

We also know from the pseudo-code for RIGHT-REPORT that

$$S_u.z = S_w.z = \max\{S_v.z, \mathsf{top}(\mathcal{H}_v)\}. \tag{2}$$

Since v is not a leaf of $\mathcal{U}$, $h_v > S_v.z \geq -\infty$. Hence, the definition of h_v implies that

$$h_v = \max\{h_u, h_w, \mathsf{top}(\mathcal{H}_v).z\}. \tag{3}$$

Now we are ready to prove that $\mathsf{top}(\mathcal{H}_v).z \geq S_v.z$. Since v is not a leaf of $\mathcal{U}$, we know that

$$h_v \geq S_v.z. \tag{4}$$

By (3), we have that $h_v = h_u$ (the case $h_v = h_w$ is symmetric) or that $h_v = \mathsf{top}(\mathcal{H}_v).z$. We first prove that $h_v = \mathsf{top}(\mathcal{H}_v).z$ by showing that assuming $h_v = h_u$ leads to a contradiction. If $h_v = h_u$, then (3) implies that

$$h_u \geq \mathsf{top}(\mathcal{H}_v).z. \tag{5}$$

Now, (2) implies that $S_u.z = S_v.z$ (i.e., $S_u.z \geq \mathsf{top}(\mathcal{H}_v).z$) or $S_u.z = \mathsf{top}(\mathcal{H}_v).z$ (i.e., $S_u.z \geq S_v.z$). We consider each case in turn.

(i) $S_u.z = S_v.z \geq \mathsf{top}(\mathcal{H}_v).z$: Using (1) and (4), we get a contradiction since $h_v = h_u < S_u.z = S_v.z \leq h_v$.
(ii) $S_u.z = \mathsf{top}(\mathcal{H}_v).z \geq S_v.z$: Using (5) and (1), we get a contradiction since $\mathsf{top}(\mathcal{H}_v).z \leq h_u < S_u.z = \mathsf{top}(\mathcal{H}_v).z$.

Hence $h_v = \mathsf{top}(\mathcal{H}_v).z$. By (4), $h_v \geq S_v.z$. Hence, we have

$$\mathsf{top}(\mathcal{H}_v) \geq S_v.z. \tag{6}$$

(6) and (2) now imply that $S_u.z = S_w.z = \mathsf{top}(\mathcal{H}_v)$. Combining with (1), we obtain

$$\mathsf{top}(\mathcal{H}_v) > \max\{h_u, h_w\}.$$

Since $h_v > -\infty$ (otherwise v would have been a leaf of $\mathcal{U}$), the definition of h_v implies that either $\mathsf{top}(\mathcal{H}_v).\mathit{unrep}$ is *true* or $\mathsf{top}(\mathcal{H}_v).z \leq \max\{h_u, h_w\}$. We just showed that $\mathsf{top}(\mathcal{H}_v) > \max\{h_u, h_w\}$. Hence $\mathsf{top}(\mathcal{H}_v).\mathit{unrep}$ is *true*.
To complete the proof, we show that

$$\mathsf{top}(\mathcal{H}_v).z \geq l_v.$$

By the definition of h_v and l_v, it is clear that $h_v \geq l_v$. Similarly, $h_u \geq l_u$ and $h_w \geq l_u$. Since $\mathsf{top}(\mathcal{H}_v) > \max\{h_u, h_w\}$, it follows that $\mathsf{top}(\mathcal{H}_v) > \max\{l_u, l_w\}$. From the definition of l_v, we know that

$$l_v = \max\{\min\{l_u, l_w\}, \mathsf{top}(\mathcal{H}_v).z\},$$

which implies that $l_v = \mathsf{top}(\mathcal{H}_v).z$. $\square$

It is now an easy exercise to show that if a subtree traversed by a call to RIGHT-REPORT has k marked nodes, then the subtree has $O(k \log n)$ nodes.

Theorem 2 *The rectangles visible from $z = +\infty$ in a set of n rectangles with sides parallel to the x- and y-axes can be reported in $O(n \log^2 n)$ time. The space used is $O(n \log n)$.*

Proof. The space taken by $\mathcal{T}$ is clearly $O(n \log n)$ because each cross-section is stored at $O(\log n)$ nodes in $\mathcal{T}$.

Each of the $O(n)$ calls to LEFT-INSERT and RIGHT-DELETE takes $O(\log^2 n)$ time since $O(\log n)$ nodes are visited in each call and $O(\log n)$ time is spent at each node in updating the heap stored at that node. Hence the total time spent in calls to LEFT-INSERT and RIGHT-DELETE is $O(n \log^2 n)$. Lemma 2 implies that RIGHT-REPORT will traverse a tree of size $O(l \log n)$ to report l previously unreported cross-sections. Since each visible rectangle is reported $O(\log n)$ times, calls to RIGHT-REPORT take $O(k \log^2 n)$ time, where k is the number of visible rectangles. Since k is at most n, the total time is $O(n \log^2 n)$. $\square$

6. An Improved Algorithm

In this section, we improve the running time of the algorithm to $O(n \log n)$. When a rectangle is reported for the first time by the above algorithm, in $O(\log n)$ time all cross-sections corresponding to it can be marked as reported (using the *unrep* field). Since these cross-sections are the leaves of a subtree of $\mathcal{T}$ of size $O(\log n)$, the h_v values in the tree can be updated to reflect the changes to $\mathcal{T}$ in $O(\log n)$ time. This reduces the $O(k \log^2 n)$ component of the running time (which is hidden by $O(n \log^2 n)$ in Theorem 2) to $O(k \log n)$.

To reduce the time taken by the rest of the algorithm to $O(n \log n)$, we use Bern's trick.[3] He notes that anytime a node v of $\mathcal{T}$ is visited, it is enough to know just the value of $\mathsf{top}(\mathcal{H}_v)$ rather than what is stored in the entire heap. At each node v of the segment tree, the modified algorithm stores a list of values of $\mathsf{top}(\mathcal{H}_v)$. Each entry in the list has a range of x values for which it is valid. The modified algorithm simulates the old algorithm exactly except that no insertions and deletions are made into the heaps and whenever the value at the top of a heap is needed, the correct value is taken from the corresponding list.

Once the skeleton of $\mathcal{T}$ and the event schedule have been determined, for all nodes v in $\mathcal{T}$, we calculate a sorted list of $R.z$ values for all rectangles R ever stored at v. We can do this in $O(n \log n)$ time. We also keep a sorted list of $R.x_1$ and $R.x_2$ values for each node corresponding to the insertions and deletions made at that node.

For a single node v, we can represent the sorted list of $R.z$ values by ranks between 1 and m, where m is the total number of rectangles stored at v. Computing the list of $\mathsf{top}(\mathcal{H}_v)$ values now is an off-line "extract-maximum" problem. Bern shows how a sequence of $O(m)$ insert, delete and find-max operations on integers between 1 and m can be processed in $O(m)$ time.

The total length of all $\mathsf{top}(\mathcal{H}_v)$ lists is $O(n \log n)$ since each rectangle is stored at $O(\log n)$ nodes. The computation of each list requires time linear in the length of the list. Combining with Theorem 1, we have the following theorem.

Theorem 3 *The rectangles visible from $z = +\infty$ in a set of n rectangles with sides parallel to the x- and y-axes can be reported in $\Theta(n \log n)$ time. The space used is $O(n \log n)$.*

7. Conclusions

We have developed a new model of complexity for measuring the performance of hidden-surface removal algorithms. This model, called the object complexity model, is motivated by the characteristics of graphics rendering hardware like the z-buffer. Our model is appropriate for both dynamic and static hidden-surface elimination. We believe that this model measures the performance of a hidden-surface elimination algorithm much more realistically than the standard computational geometry model of scene complexity.

We have also presented a simple, easy-to-implement algorithm under this new model to report the set of rectangles visible from the point $z = +\infty$. All these rectangles are parallel to the xy-plane and have sides parallel to the x- and y-axes. This algorithm runs in $\Theta(n \log n)$ time and uses $O(n \log n)$ space.

References

1. P. K. Agarwal and J. Matoušek, "Ray shooting and parametric search," *SIAM J. Comput.*, **22** (1993), 794–806.
2. J. M. Airey, "Increasing Update Rates in the Building Walkthrough System with Automatic Model-space Subdivision and Potentially Visible Set Calculations," Ph.D. Thesis, Dept. of Computer Science, University of North Carolina, Chapel Hill, 1990.
3. M. Bern, "Hidden surface removal for rectangles," *J. Comput. Syst. Sci.*, **40** (1990), 49–69.
4. E. Catmull, "A Subdivision Algorithm for Computer Display of Curved Surfaces," Ph.D. Thesis, Computer Science Dept., University of Utah, Salt Lake City, 1974.
5. S. Coorg and S. Teller, "Temporally coherent conservative visibility," *Proc. 12th Annu. ACM Sympos. Comput. Geom.*, 1996, pp. 78–87.
6. S. Coorg and S. Teller, "Real-time occlusion culling for models with large occluders," *Proc. 1997 Symp. on Interactive 3D Graphics*, 1997, pp. 83–90.
7. M. de Berg, "Generalized hidden surface removal," *Comput. Geom. Theory Appl.*, **5** (1996), 249–276.
8. M. de Berg, D. Halperin, M. Overmars, J. Snoeyink, and M. van Kreveld, "Efficient ray shooting and hidden surface removal," *Algorithmica*, **12** (1994), 30–53.
9. F. Dévai, "Quadratic bounds for hidden line elimination," *Proc. 2nd Annu. ACM*

Sympos. Comput. Geom., 1986, pp. 269–275.

10. D. P. Dobkin and R. J. Lipton, "On the complexity of computations under varying sets of primitives," *J. Comput. Syst. Sci.*, **18** (1979), 86–91.
11. J. D. Foley, A. van Dam, S. K. Feiner, and J. F. Hughes, *Computer Graphics: Principles and Practice*, (Addison-Wesley, Reading, 1990).
12. M. J. Katz, M. H. Overmars, and M. Sharir, "Efficient hidden surface removal for objects with small union size," *Comput. Geom. Theory Appl.*, **2** (1992), 223–234.
13. M. McKenna, "Worst-case optimal hidden-surface removal," *ACM Trans. Graph.*, **6** (1987), 19–28.
14. K. Mehlhorn, *Data Structures and Algorithms*, (Springer-Verlag, Berlin, 1984), Volumes 1–3.
15. J. S. Montrym, D. R. Baum, D. L. Dignam, and C. J. Migdal, "InfiniteReality: A Real-Time Graphics System," *Proc. SIGGRAPH 97*, in *Comput. Graph. Proc.*, Annual Conference Series, (ACM SIGGRAPH, New York, 1997), pp. 293–302.
16. F. P. Preparata and M. I. Shamos, *Computational Geometry: An Introduction*, (Springer-Verlag, New York, 1985).
17. F. P. Preparata and J. S. Vitter, "A simplified technique for hidden-line elimination in terrains," *Proc. 9th Sympos. Theoret. Aspects Comput. Sci.*, in *Lecture Notes Comput. Sci.*, Vol. 577, (Springer-Verlag, Berlin, 1992), pp. 135–146.
18. F. P. Preparata, J. S. Vitter, and M. Yvinec, "Output-sensitive generation of the perspective view of isothetic parallelepipeds," *Algorithmica*, **8** (1992), 257–283.
19. J. H. Reif and S. Sen, "An efficient output-sensitive hidden-surface removal algorithms and its parallelization," *Proc. 4th Annu. ACM Sympos. Comput. Geom.*, 1988, pp. 193–200.
20. I. E. Sutherland, R. F. Sproull, and R. A. Schumacker, "A characterization of ten hidden-surface algorithms," *ACM Comput. Surv.*, **6** (1974), 1–55.
21. S. J. Teller, "Visibility Computations in Densely Occluded Polyhedral Environments," Ph.D. Thesis, Dept. of Computer Science, University of California, Berkeley, 1992.