# An Optimal Algorithm for Reporting Visible Rectangles

S. SIOUTAS, A. TSAKALIDIS, J. TSAKNAKIS, V. VASSILIADIS
Department of Computer Engineering and Informatics
University of Patras
26500, Patras
GREECE

*Abstract:-* We consider the following problem as defined by Grove et al. [5]: Given a set of n isothetic rectangles in 3D space determine the subset of rectangles that are not completely hidden. We present an optimal algorithm for this problem that runs in *O(nlogn)* time and *O(n)* storage. Our result is an improvement over the one of Grove et al. by a logarithmic factor in storage and is achieved by using a different approach. An analogous approach solves the problem for other kinds of objects too.

*Key-words:* Computer Graphics, 3D space, hidden surface removal, object space algorithms. CSCC'99 Proc.pp.2231-2236

## 1 Introduction

A ubiquitous task in computer graphics is that of rendering scenes of objects in 3D space. Part of this task is, given a viewing position, to determine the portions of the objects that are visible from the viewing position. In order only these to be displayed on the screen. This problem is commonly known as hidden surface removal or hidden line elimination depending on whether the displayed parts are surface patches or line segments. There are two main approaches to the problem: image space and object space. In the first approach, which do computer graphics algorithms implement, the projections of the objects on the screen are treated as raster images (i.e. images composed by pixels). As a consequence their output is a raster image too, and their running times depend on the raster size (screen resolution). On the other hand, algorithms in computational geometry treat the projections of the objects as vector images. Their output is a graph, called the visibility map, whose edges are visible segments of object's edges and whose vertices are visible vertices or cross points of visible edges. Their running times depend on the size of the visibility map.

Although there are certain advantages in using object space solutions like precise scaling of images without loss in detail, in practice image space algorithms have proved to be more successful because they can be efficiently implemented in hardware. An attempt to combine the two approaches is the object complexity model, introduced by Grove et al. in [5]. In this model the hidden surface removal problem is solved in two steps. In the first step the subset of objects that are not completely obscured by other objects is determined using object space methods. The size of this subset is called the object complexity of the scene, denoted q. In the second step an image space algorithm (like the z-buffer) is applied on this subset of objects only. Because of the first step, a significant speed up is expected in the execution of the image space algorithm. The challenge in this approach is to develop algorithms for the first step whose time can be bounded by a function of the object complexity q and the input size n. Standard object-space algorithms are inefficient for this purpose because their running times are lower bounded by the visibility map size k, which can be È($q^2$).

This approach can be efficiently realised for the window-rendering problem as shown in [5]. This is the problem of displaying on a screen the contents (text, images) of a set of n rectilinear windows which may partially overlap and for which a precedence order is known. It is easy to rephrase this as a hidden surface removal problem in which the objects are rectangles with sides parallel to the x-,y- axes. For the standard hidden surface problem the best solutions [1,4.12] run in *O((n + k)logn)* time and *O(nlogn)* storage. The storage can be reduced to *O(n)* by a simple technique as shown in [9]. By modifying appropriately the algorithm of Bern [1], Grove et al. showed that the object complexity version of the problem can be solved in optimal *O(nlogn)* time and *O(nlogn)* storage.

A space and time optimal algorithm for the problem has been given in [10]. This solution applies a similar idea as in [9] to the plane sweep algorithm of Grove et al. In this paper we propose an entirely different optimal solution which may be conceptually more appealing

and easy to implement. An advantage of our approach when compared to the one of [10] is that it requires a much simpler version of the linear Union-Find algorithm of [3], which can be easily implemented.

In the following section we describe a solution that matches the performance of [5] i.e. runs in *O(nlogn)* time and space. Our solution is based on the approach of Goodrich et al. [4], which we have modified appropriately. Then in section 3 it is shown how to reduce the storage to *O(n)* by using a similar idea as in [9]. Finally in the last section we show that the general approach can be applied effectively for other kinds of objects too.

# 2 An Incremental Approach

The problem we consider can be stated as follows: Given is a collection of n rectangles in 3-dimensional space which are parallel to the xy-plane and whose sides are parallel to the x-,y- axes. We are asked to determine the subset of rectangles that are not completely hidden by other rectangles when viewed from a point at $z = +\infty$. We will use the notation $R.x_1$, $R.x_2$, $R.y_1$, $R.y_2$, R.z for the x-, y- and z- coordinates of rectangle R, i.e. $R = [R.x_1, R.y_2] \times [R.y_1, R.y_2] \times R.z$.

We will use an incremental approach analogous to the one of Goodrich et al. This approach to hidden line elimination problems was introduced by Guting and Ottman in [6] and was also used in [7, 13, 14]. The rectangles will be processed in descanting z-order starting from the one nearest to the viewer, and a structure storing the union of the projections of all rectangles encountered so far, will be maintained. Following the terminology of [4] we call this union the shadow of the rectangles. Each time a new rectangle is encountered we will query the structure storing the shadow to determine if there are parts of the rectangle not covered by the shadow. If there are such parts then we add the rectangle to the set V of visible rectangles (initially this set is empty). Then the structure storing the shadow is updated to reflect the addition of the new rectangle, and the algorithm continues with the next rectangle.

By the above brief description it is clear that the algorithm correctly computes the subset of visible rectangles (which will be the set V after completion of the algorithm). We implement the structure for shadow maintenance as a simplified version of the Hive Tree of [4]. As shown in lemma 2.1 this structure can be built in *O(nlogn)* time using *O(nlogn)* storage. Furthermore in lemma 2.2 and corollary 2.5 it is shown that all queries and updates performed by the algorithm take *O(nlogn)* time after an *O(nlogn)* time preprocessing. Therefore, we have the main result of this section stated in the following:

Theorem 2.1 *The rectangles visible from $z = +\infty$ in. a set of n. iso-oriented rectangles in 3D space can be determined in O(nlogn) space and time.*

## 2.1 The Data Structure

As already mentioned the data structure is a simple version of the Hive Tree which is introduced in [4]. The Hive Tree is a segment tree augmented with auxiliary structures in its nodes to facilitate shadow maintenance. We will give an overview of this structure as needed for our purposes.

Let $x_1, x_2, x_{3,\ldots,} x_{2n}$ be the x-coordinates of the rectangles in ascending order (without loss of generality we assume that the x-coordinates are distinct). The skeleton of the structure is a segment tree T. T is a static binary balanced tree whose leaves are associated with the elementary x-ranges $[x_1, x_2]$, $[x_2, x_3]$, .. , $[x_{2n-1}, x_{2n}]$ in this order. Each node u of the tree has an associated *xrange(u)* equal to the union of the *xranges* of its two children. Known properties of the segment tree are: (i) at each level of the tree the xranges of the nodes constitute a partition of $[x_1, x_{2n}]$ (ii) the nodes whose xranges contain a value x lie in a root to leaf path, and (iii) any interval $[x_i, x_j]$ can be associated to *O(logn)* nodes u such that $xrange(u) \subseteq [x_i, x_j]$ and $xrange(parent(u)) \not\subset [x_I, x_j]$. Note that by property (iii) any interval can be partitioned to *O(logn.)* maximal subintervals (xranges).

In two dimensions each *xrange [xi, $x_j$]* defines a slab delimited by the two vertical lines $x = x_i$, $x = x_j$. Every rectangle *R* can be associated to *O(logn)* nodes according to $[R.x_1, R.x_2]$. For each node u we denote by S(u) the set of rectangles associated to u or to descendants of u. Note that rectangles in S(u) either intersect both boundaries of the slab of u, or have at least one vertical side inside the slab. Let $y_1, y_2, . ., y_p$ be the y-coordinates of rectangles of S(u). We draw horizontal lines $y = y_1$, $y = y_2$, . . . , $y = y_p$ partitioning the slab into p-1 horizontal strips. These strips are stored as a sorted list denoted as *Strip(u).* For each strip *h* in *Strip(u)* we store two lists *Up(h), Down(h). Up(h)* contains all strips in *Strip(parent(u))* that intersect h, while Down(h) contains all strips in *Strip(left_child(u)) ∪ Strip(right_child(u))* that intersect h. Note that strips in *Down(h)* form a partition of strip h. There is also an equivalence between Up and Down lists, specifically *h E Up(h')* if *h' ∈ Down(h).* Also note that *Down(h)* contains exactly two strips (one in each child). The *Up* and *Down* lists are implemented

as doubly linked lists. The following lemma bounds the size and construction time of the structure.

Lemma 2.1 *The Hive Tree for a collection of n rectangles in the plane can be constructed in O(nlogn) time and space.*

*Proof.* The proof follows from lemma 2.2 of [4] for the special case in which the skeleton of the structure is a binary tree. In brief the arguments go as follows: For the space notice that the size of all lists *Up(h)* is the same with the total size of lists *Down(h)* because of the equivalence property. The total size of *Down(h)* lists can be bounded by twice the size of all lists Strip(u) since each list has size 2. For the size of lists Strip(u) observe that each rectangle contributes two strip boundaries to each one of its associated nodes and their ancestors. Since there are *O(logn)* associated nodes whose ancestors lie in a forked path, each rectangle contributes *O(logn)* strip boundaries. Therefore the size of all lists and the size of the structure is *O(nlogn).* The lists can be constructed recursively, in time bounded by their size by constructing first the lists of the root and then using these lists to compute the lists of its children.

## 2.2 Determining Visible Rectangles

We process the rectangles in descending depth order. Let $R_1$, $R_2$, . . ., $R_{i-1}$ be an order of the rectangles in decreasing z-coordinate. At the time we process rectangle $R_i$ rectangles $R_1$, $R_2$,..., $R_{i-1}$ will have been inserted in the Hive Tree. The Hive Tree will be used to store a representation of the union of the processed rectangles. We denote this union by $U_i$. Processing of $R_i$ involves two steps: First we have to query the Hive Tree to determine if $R_i \cup U_i = U_i$. If so then the rectangle is completely obscured by the rectangles $R_1$, $R_2$, . . ., $R_{i-1}$ else there is a part of $R_i$ which is not hidden, in which case rectangle $R_i$ is reported as visible. Second, we have to update the Hive Tree in order to represent the union $U_{i+1} = U_i \cup R_i$, that is insert $R_i$ in the Hive Tree. We will show how to perform both steps in a single access of the Hive Tree.

A strip can be in one of three states full, open or touched. A strip is *full* if it is inside $U_i$ it is open, if it is outside $U_i$ and it is touched if it partially intersects $U_i$. It follows from the definitions that (i) if h is full (or open) then all strips $h' \in Down(h)$ are full (open) too, and (ii) if h is touched then all strips $h' \in Up(h)$ are touched too. Note that initially all strips are open while upon termination of the algorithm all strips have become full.

As in [4] we will use some additional lists for each strip h. These are *NFU(h)* which stores the strips in *Up(h)* that are not Full and *OD(h)* which stores the strips in *Down(h)* that are open. Initially *NFU(h) = Up(h)* and *OD(h) = Down(h).*

Now consider querying the structure with rectangle $R_i$. Rectangle $R_i$ can be associated to *O(logn)* nodes of the Hive Tree. The slabs of these nodes are disjoint and partition rectangle $R_i$ to *O(logn)* pieces. Each strip of these nodes, is either completely contained in $R_i$ or lies outside $R_i$. So the rectangle can be decomposed into a number of strips. The query algorithm must be able to determine if there is one such strip that is not full. If so, then there exists a part of $R_i$ not covered by $U_i$.

This can be done efficiently by the notion of principal rectangles as defined in [4]. For each strip h that belongs to node u the principal rectangle is the highest rectangle (the one with biggest z-coordinate) among the rectangles of S(u) that contain strip h. Note that each strip has at most one principal rectangle. Principal rectangles are computed in a preprocessing step. After having determined the principal rectangles it is easy to compute for each rectangle R a list P(R) of all strips for which R is a principal rectangle.

Given the list $P(R_i)$ it is trivial to perform the query: we visit all strips in $P(R_i)$ and check if there exists one which is not full. The correctness follows from the fact that all-remaining strips which are covered by $R_i$ and are not in $P(R_i)$ need not be examined because they are full (otherwise $R_i$ would be the principal rectangle for them).

It remains to show how to update the Hive Tree. Consider again the strips in $P(R_i)$. If all of them are full then nothing needs to be done (indeed adding $R_i$ does not change the union $U_i$). Otherwise, let h is a strip in P(R), which is not full. First we mark this strip as full. Then for each strip h' in *OD(h)* we remove h from *NFU(h').* If this results to *NFU(h') = 0* then we also mark h' as full, we remove h' from *OD(h)* and continue in the same way with strips in *OD(h').* We act analogously for strips in *NFU(h).* That is, for each strip h' in *NFU(h)* we remove h from *OD(h').* If this results to an empty list then we mark strip h' as full, we remove h' from *NFU(h)* and continue in the same way for strips in *NFU(h').*

This completes the description of the algorithm. A bound for the total time of queries and updates is provided by the following lemma.

Lemma 2.2 *The time spent for al1 queries and updates in the Hive Tree is bounded by the size of the Hive Tree.*

*Proof.* Each query processes the strips in $P(R_i)$ for each rectangle $R_i$. Because each strip has a unique principal rectangle it follows that each strip is processed once. So the query time is bounded by the

total size of lists Strip(u). Updates also process strips in *NFU(h)* and *OD(h)* for each visited strip h. This kind of processing may cascade to several ancestors and descendants of each associated node u. However, notice that for each visited strip h' a deletion is made from either *NFU(h')* or *OD(h')*. Since the total number of deletions during all updates can not exceed the total size of the lists *NFU* and *OD*, the claimed bound follows.

In the preprocessing step we have to compute for each strip h its principal rectangle. First we will need the following result.

Lemma 2.3 *We can compute in O(nlogn) time for each node u, the set S(u) of its associated rectangles, the depth order of rectangles in S(u) and the y-order of their horizontal boundaries.*

*Proof.* In a preliminary step we sort the y- and z-coordinates of the rectangles. For each rectangle we compute in *O(logn)* time a list of associated nodes in the Hive Tree. For each node in the tree we create two empty lists the *z-list* and the *y-list*. Then we process all rectangles in descending depth order and add each rectangle to the *z-lists* of its associated nodes. This produces in *O(nlogn)* time for each node a list of its associated rectangles in descending depth order. In the same way we can compute for each node its *y-list* containing the *y-order* of the boundaries of its associated rectangles.

Given the information computed in the previous lemma we can solve the principal rectangle problem for the strips of each node of the Hive Tree. The solution is described in the following:

Lemma 2.4 *Given the z- and the y-order of the boundaries of rectangles in S(u) the principal rectangle for each strip in Strip(u) can be computed in time O(|Strip(u)|).*

*Proof.* First we restrict our attention to the strips defined by the boundaries of rectangles in *S(u)* The strips in *Strip(u)* are contained in these strips, so their principal rectangles can be derived easily from the principal rectangles of the latter. Then the problem can be rephrased as a 2-dimensional interval visibility problem, specifically: given a set of horizontal segments in the y-z plane determine the visible parts of the segments from a point at $z = +\infty$. This problem can be solved using the linear time Union-Find algorithm of Gabow and Tarjan [3] in O(|S(u)|) time (see the off line min problem in [3]). Then in O(|Strip(u)|) time we can determine in a single pass of the list Strip(u) the principal rectangles for its strips.

As a consequence of the previous lemmas the preprocessing time can be bounded by the total size of the lists Strip(u) i.e.

Corollary 2.1 *The preprocessing time is bounded by the size of the Hive Tree.*

## 3 Space Reduction

To reduce the storage of the algorithm we will use the same idea as in [9], that is we will partition the scene into *O(logn)* vertical slabs and we will solve the problem independently in each slab in *O(n)* time and space. It is interesting to note that in our approach this simple partitioning scheme directly reduces the space to *O(n)* without further modifications of the previous approach.

Initially we sort the x- y- and z- coordinates of all rectangles in all slabs and we call the resulting orderings $O_x$, $O_y$, $O_z$ respectively. We define the slabs by drawing vertical lines in the x-y plane, so that in each slab there are no more than [n/logn] vertices of rectangles. In this way we have *O(logn)* slabs each containing *O(n/logn)* vertices. In each slab we distinguish two sets of rectangles denoted $S_p$ and $S_f$. The rectangles of $S_f$ span the x-range of the slab while the ones of $S_p$ have one or both of their vertical edges inside the slab. For both sets of rectangles we consider only the portions of rectangles inside the slab. Note that $|S_p| = O(n/logn)$ and $|S_f| = O(n)$. For each slab we apply the algorithm of the previous section with input the rectangles of $S_p \cup S_j$. The following lemma bounds the time and storage of this solution.

Lemma 3.1 *The visible rectangles in a single slab can be reported in O(n) space and time.*

*Proof.* First we will show that the hive tree has size *O(n)*. The storage for the rectangles of $S_p$ clearly is *O(n)* since there are *O(n/logn)* such rectangles. The rectangles of $S_f$ are all associated to a single node in the Hive Tree, its root, since they span the x-range of the slab. So the storage needed for them is also *O(n)*. By using the same algorithm as in lemma 2.1 the Hive Tree can be constructed in time bounded by its size if the y-order of rectangles of $S_f$ is known. This order can be derived in *O(n)* time from the ordering $O_y$ of all rectangles. So the Hive Tree can be built in *O(n)* time and storage. By lemma 2.2 the time of all queries and updates will be bounded by *O(n)*. However we still need a way to bound the time of the preprocessing. The preprocessing time will be *O(n)* if the time of lemma 2.3 is reduced to *O(n)*. The sets S(u) clearly can be computed in *O(n)* time. For all nodes except for the root the computations take *O(n)* time because there are

O(n/logn) rectangles. For the root the y- and z- order of rectangles of $S_f$ can be derived in *O(n)* time by the $O_y$ and $O_z$ orders of all rectangles. Now it's easy to bound the time of the whole algorithm as stated in the following.

Theorem 3.1 *The visible rectangles in a set of n iso-oriented rectangles can be determined in O(nlogn) time and O(n) storage.*

*Proof.* The total time for all slabs is *O(nlogn)*. The time to compute the orders $O_x$, $O_y$ and $O_z$ is *O(nlogn)* for sorting. The storage used during the process is always *O(n)* by the previous lemma.

The algorithm described above reports multiple times the rectangles that are visible in more than one slab. This can be avoided by using a table of size n to mark the reported rectangles. The rectangles are numbered from 1 to n and rectangle number indexes the table.

## 4 Extensions and Future Work

In this paper we have presented an incremental approach for determining visible rectangles in optimal time and storage. The main algorithm results from some direct modifications of a well-known algorithm for standard hidden surface removal for rectangles. Naturally the question is raised of whether other known algorithms for hidden surface removal can be appropriately modified to solve the object-complexity version of the problem for more general objects. There seems to be an inherent difficulty for most algorithms because their time complexities strongly depend on the size k of the visibility map. However the incremental approach we have presented, apart from being more appealing than the plane sweep approach of [5] it may also be more promising for other kinds of objects too.

Consider for example the case in which the objects of the scene have small union size. We can use the approach of [14], which is as follows: The objects are processed in descending depth order and the shadow (union) of processed objects is explicitly computed. However instead of processing one object in each step in that approach the visibility map of the next f objects is computed in *O(c)* time, where c is the current size of the shadow. Then, in a merging step which takes *O(clogc+k')* time, the visibility map is compared against the shadow to find the k' arts not obscured by the shadow. As shown in [14] this approach takes $O(n\sqrt{C}\log n+k)$ time where C is the maximum shadow size.

To solve the object complexity version of the problem we can implement the merging step appropriately, so that it runs in *O(clogc)* time (the details are easy and are left to the reader). Then by the analysis of [14] the time is bounded by $O(n\sqrt{C}\log n)$. Now consider the classes of objects studied in [7]. For the case in which the objects are discs or convex homothetic objects *C* can be bounded by q [8], for the case of fat triangles C is bounded by *O(qloglogq)* [11], while for the case of polyhedral terrain C is bounded by *qa(q)*, where q is the object complexity and a is the inverse of Ackerman's function. So this approach yields an output sensitive algorithm with time $O(n\sqrt{q}\log n)$ for discs and convex homothetic objects, for fat triangles $O(n\sqrt{q\log\log q}\log n)$ and $O(n\sqrt{qa(q)}\log n)$ for terrains. This is optimal when y is constant and in the worst case (q = n) it is much more efficient than computing a visibility map of size $\Theta(n^2)$.

It is interesting to investigate whether this approach can solve the object complexity version of hidden surface removal for other kinds of objects as well. It is also interesting to look for more efficient algorithms for the case of fat triangles and the case of objects of small union size.

*References:*


[1] M. Bern *Hidden surface removal for rectangles,* J. Comp. Syst. Sci. 40 (1990) 49-69.

[2] R. Cole and M. *Sharir, Visibility problems for polyhedral terrains,* J. Symbolic Comput. 7 (1989), 11-30.

[3] H.N. Gabow and R.E. Tarjan, A linear time algorithm for a special case of disjoint set union. J. Comp. Syst. Sci., 1985, pp. 209-221.

[4] M. T. Goodrich, M.J. Attalah and M.H. Overmars, *An input size output size trade-off in the time complexity of rectilinear hidden surface removal,* Proc. ICALP'90, Lecture Notes in Comp. Sci. 443, Springer Berlin 1990, pp. 689-702.

[5] E.F. Grove, T.M. Murali and J.S. Vitter, *The object complexity model for hidden line elimination,* Proc. of the seventh Canadian Conf. on Comput. Geometry, 1995, pp. 273-278.

[6] R.H. Guting and T. Ottman, *New algorithms for special cases of the hidden line elimination, problem,* Comp. Vision, Graphics Image Process. 40 ( 1987), pp. 188-204.

[7] M. J. Katz, M. H. Overmars and M. Sharir *Efficient hidden, surface removal for objects of small union size,* Computational Geometry theory and Applications 2 (1992), pp. 223-234.

[8] K. Kedem, R. Livne, J. Pach and M. Sharir, *On the union of Jordan, regions and collision-free*

*transnational motion amidst polygonal obstacles*, Discrete Comput. Geom. 1, 1986, pp. 59-71.

[9] N. Kitsios and A. Tsakalidis, *Space optimal hidden line elimination for rectangles*, to appear in Information Processing Letters.

[10] C. Makris and A. Tsakalidis, *A space optimal algorithm for the object complexity of the hidden surface removal problem for rectangles*, CTI TR 96.12.41.

[11] J. Matousek, J. Pach, M. Sharir, S. Sifrony· and E. Welzl, *Fat triangles determine linearly many holes*, SIAM J. Comput., 1994, pp. 154-199.

[12] K. Mehlhorn, St. Naeher and C. Uhrig, *Hidden line elimination for iso-oriented rectangles*, Information Processing Letters 35, 1990, pp. 137-143.

[13] H. Schipper and M.H. Overmars, *Dynamic partition trees BIT 31*, 1991, pp. 421-436.

[14] M. Sharir, and M.H. Overmars, A simple output-sensitive algorithm for hidden surface removal, ACM Transactions on graphics, Vol.11,1992, pp. 1-11.