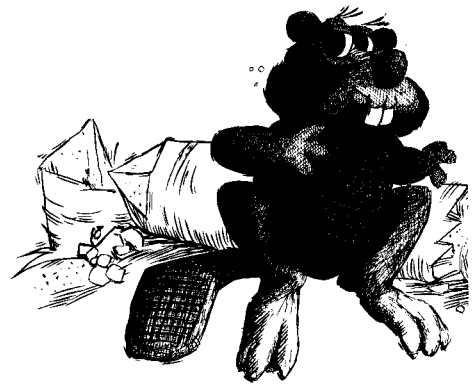
UNIVERSITY OF WATERLOO COMPUTER SCIENCE DEPARTMENT
UNIVERSITY OF WATERLOO COMPUTER SCIENCE DEPARTMENT
UNIVERSITY OF WATERLOO COMPUTER SCIENCE DEPARTMENT



On Rectangular Visibility

Mark H. Overmars
Derick Wood

Data Structuring Group
CS-86-58

October, 1986

On Rectangular Visibility *

Mark H. Overmars† Derick Wood‡

October 31, 1986

Abstract

Given a set of n points in the plane, two points are said to be *rectangular visible* if the orthogonal rectangle with the two points as opposite vertices has no other point of the set in its interior. In this paper it is shown that all pairs of rectangular visible points in a set of size n can be determined in $O(n \log n + k)$ time, where k is the number of reported pairs, using $O(n)$ space. Also, we consider the query problem: Given a set V and an arbitrary point p, determine those points in V that are rectangular visible from p. A dynamic data structure is described that uses $O(n \log n)$ space, has a query time of $O(k+\log^2 n)$ and an update time of $O(\log^3 n)$. Additionally, we extend the results to the 3-dimensional case.

Keywords and phrases: rectangular visibility; interval tree; minimal elements; stabbing problem.

1 Introduction

We introduce the following notion of visibility.

Definition 1.1 *Given a set of points V in the plane, a point p is said to be rectangular visible from a point q with respect to V if and only if there exists an orthogonal rectangle R that contains both p and q, but no other point of V.*

*This work was carried out while the first author visited the University of Waterloo. The first author was partially supported by The Netherlands Organization for the Advancement of Pure Research (ZWO). The second author was supported by a Natural Sciences and Research Council of Canada Grant No. A-5692.

†Department of Computer Science, University of Utrecht, P.O. Box 80.012, 3508 TA UTRECHT, The Netherlands.

‡Department of Computer Science, University of Waterloo, WATERLOO, Ontario, Canada N2L 3G1.

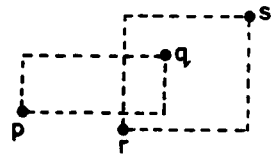


Figure 1: p sees q but r does not see s.

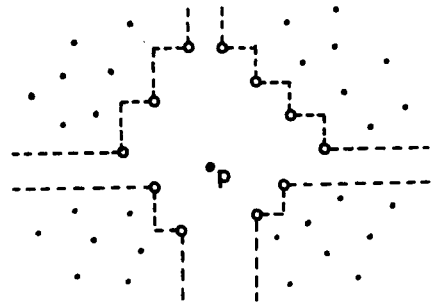


Figure 2: All points visible from p.

It is easy to see that p is rectangular visible from q if and only if the orthogonal rectangle with p and q as opposite vertices does not contain any other point from the set V. We will from now on use the term "see" when we mean "is rectangular visible from". See Figure 1 for an example in which p sees q but r does not see s. In Figure 2 all points in a set visible from p are shown. The notion of rectangular visibility has been used before, for example, [1] refer to rectangular visible as point separated. They prove that the maximal number of separated pairs of points is $n^2/4 + n - 2$, where n is the size of the point set. Also, in the art gallery problem, Keil [4] has used the notion of rectangular visibility.

In this paper we address the problem of computing pairs of points that can see each other. We consider two different problems:

1. The all pairs problem. Given a set of points V, determine all pairs of points in V that can see each other.

2. The query problem. Given a set of points V, how should it be represented such that, for a given point p, we can efficiently determine which points (in V) p can see.

Note that the query problem is equivalent to finding the largest empty orthogonal-convex orthogonal polygon containing p.

In Section 2 we consider the all pairs problem. We give an optimal solution to the problem, that is, a solution that runs in $O(k + \log n)$ time

using $O(n)$ space, where n is the number of points in the set and k is the number of reported pairs. For this purpose we use a plane sweep algorithm. In Section 3 we consider the query problem. A dynamic data structure is given, based on a structure for maintaining maximal elements by [7], that uses $O(n \log n)$ space, has a query time of $O(k + \log^2 n)$, and has an update time of $O(\log^3 n)$.

Next we consider the problem in three-dimensional space. Hence, we are given a set of points in 3-dimensional space and we say two points p and q are rectangular visible if there exists an orthogonal hyper-rectangle that contains both points, but no other points of the set. Again, we consider both the all pairs problem and the query problem.

In Section 4 we show that the all pairs problem in 3-dimensional space can be solved in time $O((n+k)\log^2 n)$ time using $O((n+k)\log n)$ space. In Section 5 we give a dynamic data structure for the query problem that uses $O(n \log^2 n)$ space, has a query time of $O((k+1)\log^2 n)$ and an insertion and deletion time of $O(\log^3 n)$.

Finally, in Section 6 we give some conclusions, extensions and open problems.

Throughout this paper we will assume that no two points lie on a horizontal or vertical line. At the appropriate places, it is indicated how the general case can be treated.

2 Solutions to the All Pairs Problem

To solve the all pairs problem we use a plane sweep approach. We move a sweep line from left to right over the set of points. With the sweep line we keep an interval tree (see [2,5,9]) that stores for each point that lies to the left of the sweep line its so-called "visibility interval". For a point $p = (x, y)$ its *visibility interval* at position x' of the sweep line ($x' \geq x$) is defined as the interval $[y_1..y_2]$ such that p can see each point (x', y') with $y_1 \leq y' \leq y_2$. See Figure 3 for an example.

It is obvious that for $x' = x$ the visibility interval of p is $[-\infty.. + \infty]$. We will denote the visibility interval of p by VI_p.

We will now describe the algorithm in detail. The interval tree, associated with the sweep line will be denoted as T.

1. Sort all points by x-coordinate, in this way creating the list of x-positions, where the sweep line should stop.

2. Initialize an empty interval tree T.

3. For each point $p = (x, y)$ where the sweep line has to stop, do:

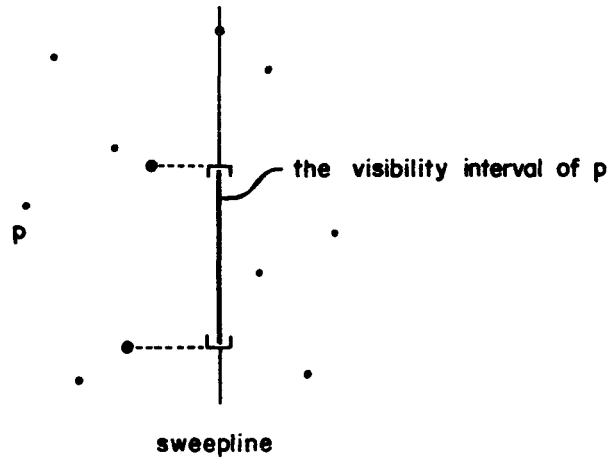


Figure 3: The visibility interval of p.

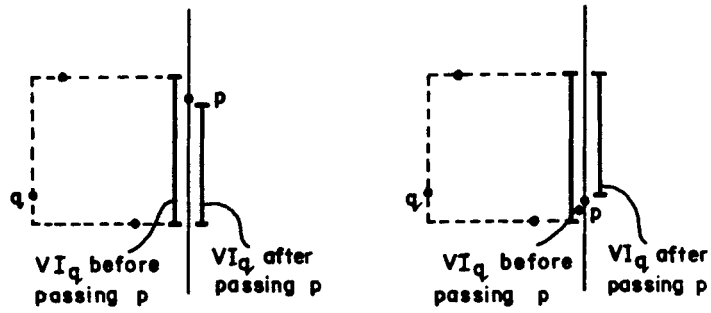


Figure 4: How the visibilty intervals can be blocked.

(a) Search in T for all intervals $VI_q = [y_{1q}..y_{2q}]$ that contain y. Report all the pairs (p, q).

(b) For each of these points q, change VI_q in the following way. If $y > y_q$ (the y-value of q) $VI_q := [y_{1q}..y]$, otherwise $VI_q := [y..y_{2q}]$.

(c) Insert the interval $VI_p = [-\infty.. + \infty]$.

Let us first give an intuitive understanding why this algorithm is correct. Clearly, all points q with $y \in VI_q$ can see p. This follows from the definition of VI_q. When the sweep line passes p, p will block the visible area for these points q. Figure 4 shows the two possible cases.

Theorem 2.1 *The above algorithm correctly reports all pairs (p, q) in which p can see q, once and only once.*

Proof: Note first that only pairs (p, q) with $x_q > x_p$ are reported. Hence, each pair is reported at most once.

i) If p can see q, (p, q) is reported (or (q, p)). Assume $x_p > x_q$. Because p can see q, y_p must be in VI_q when the sweep line stops at x_p. Hence, the query on the interval tree will report VI_q and (q, p) is reported.

ii) If (p, q) is reported, p can see q. Again, assume $x_p > x_q$. As (p, q) is reported, y_p must be in VI_q when the sweep line is at x_p. From the definition of VI_q it follows that p can see q.

Both proofs assume that the intervals VI_q are maintained correctly. This immediately follows from the method; see also Figure 4. □

Theorem 2.2 *Given a set of n points V, all pairs of points that can see each other can be computed in $O(n+k)\log n)$ time using $O(n)$ space, where k is the number of reported pairs.*

Proof: The sorting takes $O(n \log n)$ time. For each point we have to perform a query and an insertion in the interval tree and for each answer we have to change some interval, that is, insert an interval and delete one. The time bound follows.

The amount of space follows from the fact that the interval tree uses linear space. □

When more points are allowed to lie on one horizontal or vertical line we have to change the algorithm slightly. We again sort points by increasing x-coordinate. All points with the same x-coordinate we treat simultaneously. We first do Steps 3a and 3c for each of these points and next do Step 3b, for all the intervals found. This does not affect the time and space bounds.

When k is small this solution is good, unfortunately k can be $\Omega(n^2)$. Hence, it is interesting to look at solutions in which the time complexity is linear in the output size. To this end we design a new, in fact optimal, but much more complicated solution to the problem. This solution is based on the following lemma.

Lemma 2.3 *Two visibility intervals in the interval tree either do not intersect or one contains the other.*

Proof: Assume not. Hence one interval contains only one endpoint of the other; see Figure 5 for the situation.

There must have been some point q' to the right of q with y-value y_{q1} and there must have been some point p' to the right of p with y-value y_{p2}. Now assume q lies to the right of p. Then, also, q' lies to the right of p. Hence q' can be seen by p. As a result VI_p can no longer contain y_{q1} in its

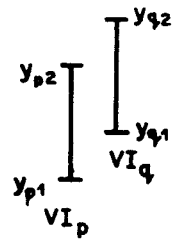


Figure 5: Visibility intervals cannot be incomparable.

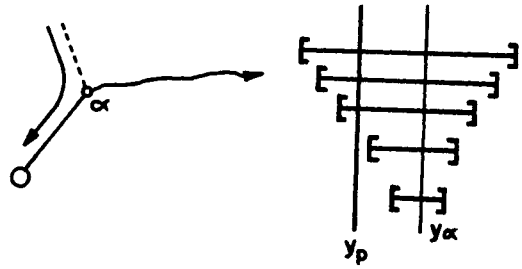


Figure 6: y_p goes left.

interior but it does. Therefore, we have a contradiction. The other cases are similar. □

As a result, the intervals in the interval tree have some extra properties. For example, all intervals stored at some internal node are all contained in one another.

To make the solution clearer we make two sweeps over the point set rather than one. In the first sweep we report only pairs (p, q) in which p can see q and q lies to the right and above p. In the second sweep, which is completely symmetrical, we report the pairs in which q lies below p.

We will only describe the first sweep. The method is exactly the same as in our first solution except that in Step 3b we know that $y > y_q$ and, hence, we always reduce the intervals to $[y_{1q}..y]$ and in Step 3c we insert the interval $VI_p = [y_p.. + \infty]$ rather than $[-\infty.. + \infty]$. In fact, the visibility intervals will always be bounded below by y_p.

To obtain a better time bound we will take a closer look at the interval tree and how we have to change the intervals. When we search in the tree with y_p (p is the point encountered by the sweep line) these are the possible cases:

1. y_p goes to the left son of an internal node; see Figure 6. As noted above the intervals stored at α are nested. We store them in a simple list, the largest element being the first one. Now, walking along the list we

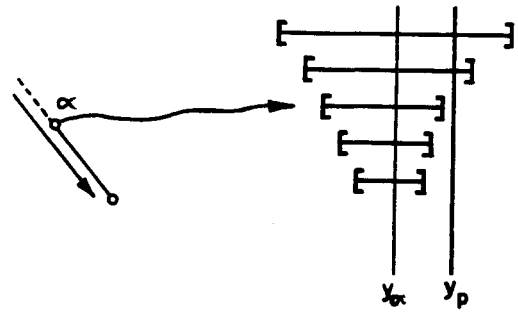


Figure 7: y_p goes right.

report all segments with left endpoint less than or equal to y_p. These segments will have their right endpoint changed to y_p. Hence, they can no longer be stored at α. So we remove them from α and put them in a list of intervals for which we have to find a new host. Clearly, the intervals in this list are nested and we keep them in nested order. The new intervals can be added at the bottom of the list. This can easily be seen as follows: Assume there is an interval VI_q in the list. Then VI_q was stored at a higher node in the tree. Let $VI_{q'}$ be the largest segment stored at α that has to be put into the list. As $VI_{q'}$ and VI_q have y_p in common either $VI_{q'}$ contains VI_q or VI_q contains $VI_{q'}$. We show that the first situation cannot happen. As VI_q was stored at a higher node β, in the tree it contained y_β and $VI_{q'}$ did not contain y_β. Hence, before the sweep line encountered p, VI_q contained $VI_{q'}$. This clearly cannot have been changed by reaching p. This shows that deleting the segments in Case (1) can be done in time proportional to the number of segments.

2. y_p goes to the right son of an internal node; see Figure 7. Again, we walk along the list and report all intervals that contain y_p. In all these intervals we change the right endpoint to y_p. Clearly, in this case, all segments can stay at α and remain in the same order. Hence, nothing else remains to be done.

After we have done this using $O(\#(\text{ answers}))$ time we have reported all answers and have changed all intervals. We are only left with a list of intervals, all with right endpoint y_p, that have to be inserted in the tree. To insert these intervals we again walk down the tree. It is easy to see that all the intervals will have to be stored at nodes on the search path of y_p at which y_p turns to the right son. Note also that the further the points are in the list, the deeper they will come in the tree. At each node α we walk simultaneous along the list and the list of intervals stored at α, inserting

intervals at the right place. All intervals we look at in the list will be stored at α (except for one). All intervals in the list of α we look at must have been reported at this moment. Hence, the total amount of time needed for all the insertions adds up to $O(\log n + \#(\text{reported points}))$.

There is no need to worry about rebalancing, because all points, and, hence, all endpoints of intervals, are known in advance. Therefore,we can construct an empty skeleton structure before doing any processing (see [2,5,9] for this technique).

Theorem 2.4 *Given a set of n points in the place, all pairs of points that can see each other can be reported in $O(n \log n + k)$ time using $O(n)$ space, where k is the number of reported pairs.*

Proof: Follows from the above discussion. □

It is easy to do both sweeps over the point set simultaneously. Also there is no need to search in the interval tree twice. As intervals only move down nodes in the tree we can do the insertions while reporting the answers.

The situation of more points lying on a horizontal or vertical line can be treated in the same way as stated above.

3 The Searching Problem

In this section we consider the problem of structuring a set of points in such a way that, given a query point p, we can efficiently determine all points that are rectangular visible from p. Let the query point p, be (x_p, y_p). We describe only how to determine all points $q = (x_q, y_q)$ with $x_q \geq x_p$ and $y_q \geq y_p$ that p can see. The points in the other quadrants can be treated in a similar way. Let V' be the subset of points $q \in V$ with $x_q \geq x_p$ and $y_q \geq y_p$. We say that a point r in V' is *minimal*, with respect to V', if there is no point s in V' with $x_s \leq x_r$ and $y_s \leq y_r$.

Lemma 3.1 *Let V' be the subset of points $q \in V$ with $x_q \geq x_p$ and $y_q \geq y_p$. The points in V visible from p are the points in V' that are minimal with respect to V'.*

Proof: See Figure 8 for an example. When $q \in V'$ is minimal, the rectangle between q and p cannot contain any other point in V'. Hence, q is visible. On the other hand, when q is not minimal there must be a point from V' and, hence, from V in this rectangle and, hence, q is not visible from p. □

Because of this lemma, it suffices to determine those points in this quadrant that are minimal with respect to the points in the quadrant.

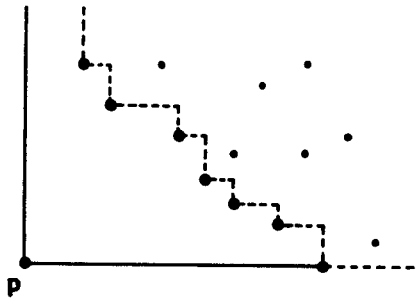


Figure 8: p sees minimal points.

We will first solve a subproblem. Let V be a set of points in the plane, we want to store them such that the following query problem can be solved efficiently: Given a vertical line L, letting V' be the set of points to the right of L, determine all points in V' that are minimal (with respect to V'). To solve this problem we store all points in a balanced binary search tree T, sorted with respect to their x-coordinate. To each internal node α of T we associate a balanced binary search tree B_α storing those points in T_α (the subtree of T rooted at α) that are minimal with respect to the set of points in T_α. The points in B_α are sorted in the order of their x-coordinate and, hence, also in the order of their y-coordinate. See Figure 9 for an example. It is clear that each point in the set is stored in at most $O(\log n)$ associated structures. Hence, the amount of space required is $O(n \log n)$.

To perform a query with a line L with x-coordinate x_L, search with x_L in T. Let $\alpha_1, \ldots, \alpha_i$ be the nodes that are right sons of nodes on the search path towards x_L, but are not on the search path themselves; see Figure 10 for an example. Clearly, the points in the subtrees $T_{\alpha_1}, \ldots, T_{\alpha_i}$ together contain each point to the right of x_L exactly once. Hence, we can restrict our attention to them. Moreover, it can easily be seen (see Figure 11) that each answer to the query must be a minimal element in its own subset. Hence, we only have to look at the elements in $B_{\alpha_1}, \ldots, B_{\alpha_i}$. Let us first look at B_{α_i} .

Now, all points in B_{α_i} are minimal, since their x-coordinates are less than the x-coordinates of all points in $B_{\alpha_{i-1}}, \ldots, B_{\alpha_1}$. Hence, they can be reported. Let q_i be the lowest such point. Now look at $B_{\alpha_{i-1}}$. All points in $B_{\alpha_{i-1}}$ that are lower than q_i are minimal and can be reported. This can be done by a reverse inorder traversal of $B_{\alpha_{i-1}}$ until a point above q_i is reached. Let q_{i-1} be the lowest point found in $B_{\alpha_{i-1}}$ ($q_{i-1} = q_i$ if no point is found). We continue this process in $B_{\alpha_{i-2}}$, etc. In this way all points are found in the time $O(\log n + k)$, where k is the number of answers.

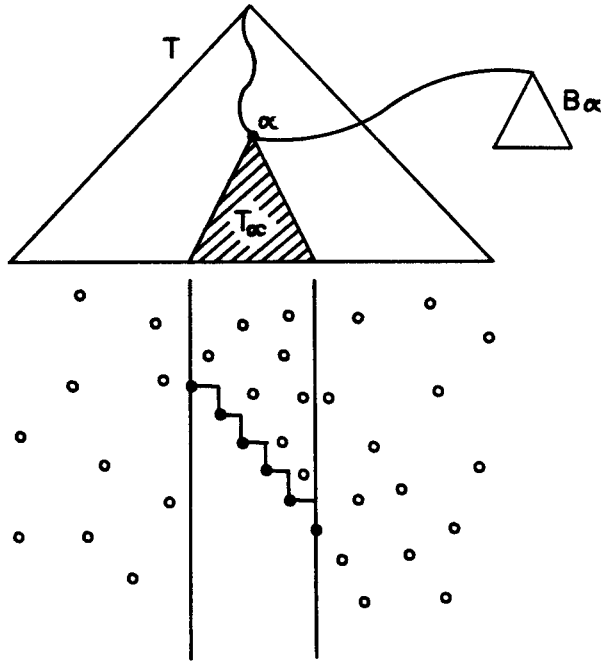


Figure 9: The trees T and B_α.

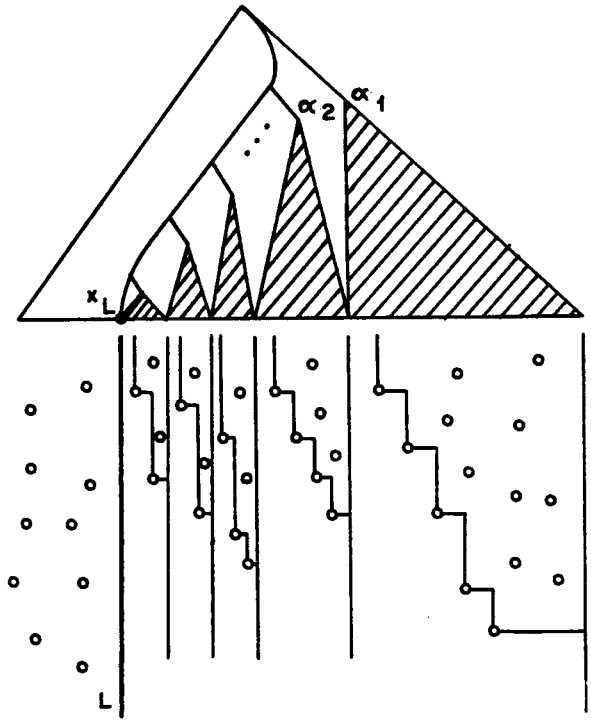


Figure 10: Searching with x_L in T.

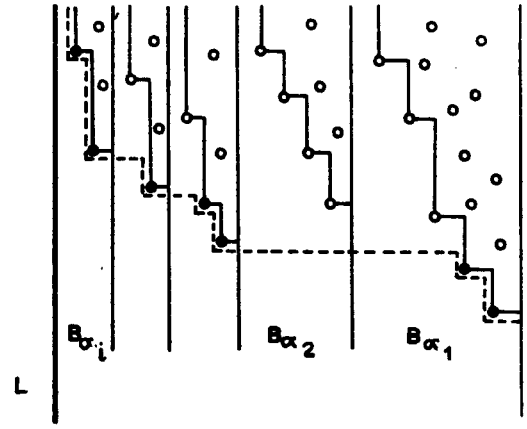


Figure 11: Answers are minimal elements.

Lemma 3.2 *Let V be a set of n points in the plane. We can represent them in a structure that requires $O(n \log n)$ space, such that those points in V that lie to the right of a vertical line L and are also minimal can be reported in $O(\log n + k)$ time.*

Proof: Follows from the above discussion. □

By a simple variation we can restrict our attention to points with x-coordinate less than or equal to x. To this end we first determine the subtree T_{α_j} that x lies in. In $\alpha_i, \ldots, \alpha_{j+1}$ we report the answers as we did above. However, in B_{α_j} we first search with x. We then perform reverse postordewr traversals of the subtrees hanging to the left of the search path from the frontier upwards. Again, we report points as long as they lie below q_{j+1}. The points on the search path are dealt with on a case by case basis. This solves the problem in the same time bounds.

To solve the original problem, we store all points in a balanced binary tree S, ordered by y-coordinate. With each internal node α we associate a structure T as described above, containing all points in the subtree rooted at α. Clearly, this structure takes $O(n \log^2 n)$ space.

To perform a query with a point $p = (x_p, y_p)$ we search with y_p in S. Let $\alpha_1, \ldots, \alpha_i$ be those nodes that are right sons of nodes on the search path, but are not on the search path themselves. Clearly, the subtrees $S_{\alpha_1}, \ldots, S_{\alpha_i}$ together contain all points above p. For each α_j we determine the leftmost point q_j, in S_{α_j} that lies to the right of p. If $x_{q_{j-1}} > x_{q_j}$, then $q_{j-1} := q_j$. These points can be determined in $O(\log n)$ time for each α_j by searching with x_p in the associated structure. It is easy to see that a point q in S_{α_j} is visible if and only if $x_q < x_{q_{j+1}}$, $x_q \geq x_p$, and q is minimal in the set of points in S_{α_j} that lie to the left of p. To find these points we use the structure

described above that performs such a query in time $O(\log n + \#(\text{answers}))$. We have to perform $O(\log n)$ such queries. Hence, the total amount of time required is $O(\log^2 n + k)$.

Theorem 3.3 *Given a set of n points in the plane, we can represent them in a structure requiring $O(n \log^2 n)$ space, such that those points that are rectangular visible from a query point p can be determined in $O(k + \log^2 n)$ time, where k is the number of reported points.*

Proof: Follows from the above discussion. □

This structure is static. We will change it slightly to make it dynamic and at the same time reduce the amount of space required.

We will, again, first look at the restricted problem of determining all points that are minimal at the right of a vertical line L. [7] describe a structure for dynamically maintaining the maximal elements of a set of points in $O(\log^2 n)$ update time using $O(n)$ space. The structure can, of course, also be used to store the minimal elements. It is almost the same as the structure T described above. The only difference is that the root of T, B_α contains all the minimal elements. At each other internal node α only those minimal elements are stored that are not minimal elements at the father of α. In this way, each point is stored only once and, hence, the space is linear. In [7] it is shown that it is possible to search in this tree in $O(\log n)$ time, reconstructing the structures B_α at each node α bordering the search path. Moreover, one can insert or delete a point and restructure the tree in $O(\log^2 n)$ time.

When we want to perform a query on such a tree with a line L at x_L we search with x, restoring the structures B_α bordering the search path. In this way, the structures B_{α_j} that we have to query are all available. We report the correct points in these structures and, next, walk upwards to restore the tree to its original shape. As we do not perform an update, this takes only $O(\log n)$ time. Hence, the query time is $O(\log n + \#(\text{answers}))$. Performing an insertion or deletion is described in [7] and takes $O(\log^2 n)$ time.

To solve the rectangular visibility problem we build a $BB[\alpha]$-tree holding the points sorted by y-coordinate. With each internal node we associate a structure as described above. Queries are performed in the same way as in the static case. To perform an update we search with the given point p in the tree. For each node on the search path we insert or delete p in the associated structure. Next we insert or delete p in the main tree. Keeping the tree balanced can be done using the techniques of [10], for example.

Theorem 3.4 *Given a set of n points, the rectangular visibility query problem can be solved dynamically using $O(n \log^2 n)$ space, with a query time of*

$O(k + \log^2 n)$, and an update time of $O(\log^3 n)$, where k is the number of reported points.

It is possible to reduce the update time at the cost of an increase in the query time. To this end we use a structure for range queries by [3]. This structure requiring $O(n \log n)$ space has an insertion and deletion time of $O(\log^2 n)$ and range queries can be performed in $O(k + \log n)$ time. The structure can be used to solve the following query problem in $O(\log n)$ time: Given a vertical line segment, determine the first point to be hit when moving the line segment to the right.

We can now solve the visibility problem in the following way: store all the points in such a structure. To perform a query, start with the vertical line through the query point and move it to the right. In $O(\log n)$ time the first point to be hit can be determined. This shrinks the line on one side. The obtained line segment is again moved to the right until it hits a point, etc. In the same way the line is moved to the left. In this way all k points that can be seen are found in $O((1 + k) \log n)$ time.

Theorem 3.5 *Give a set of n points, the rectangular visibility query problem can be solved dynamically using $O(n \log n)$ space and having a query time of $O((1 + k) \log n)$ and an update time of $O(\log^2 n)$, where k is the number of reported answers.*

When k is small, this method is better than the one described above.

4 Rectangular Visibility in 3 Dimensions

We will now concentrate on the 3-dimensional case. Hence, we are given a set V of n points in 3-dimensional space and we ask for all pairs $p, q \in V$ such that the hyper-rectangle with p and q as opposite vertices contains no other points in V. The technique we use is similar to the one described in Section 2. We perform a space sweep with a plane in the positive z-direction over the set of points. For each point p that has been passed, we keep track of its visibility area, that is, the area on the sweep plane such that any point in this area is rectangular visible from p. Initially, when the sweep plane is at p, its visibility area is the whole sweep plane. After passing more points, the visibility area looks like Figure 12. To simplify the description of the method we only consider points that lie to the left and above p. The other points can be found in a similar way. The visibility area of p now looks as shown in Figure 13. To store the visibility area we split it into a number of overlapping rectangles: R_u is the rectangle with p as bottom left corner that goes up to infinity, R_r is the rectangle that goes to the right to infinity

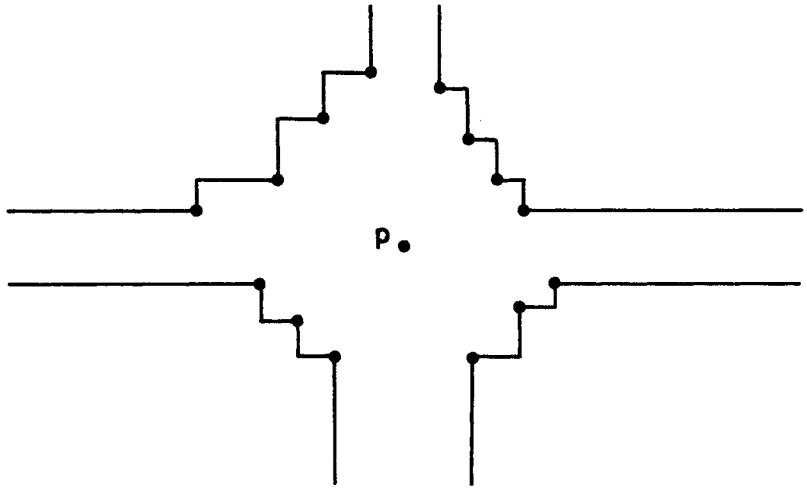


Figure 12: The visibility area of p.

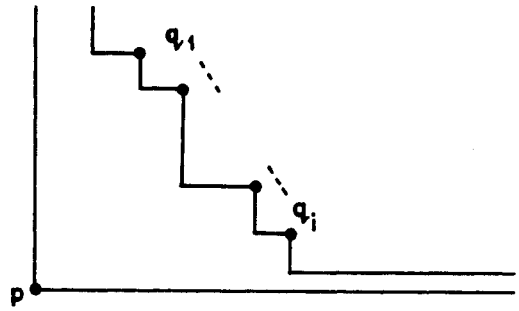


Figure 13: The visibility area of p in the positive quadrant.

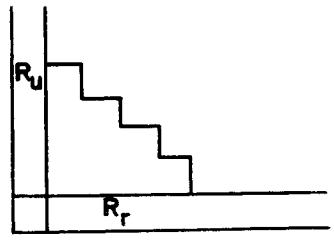


Figure 14: Storing the visibility area.

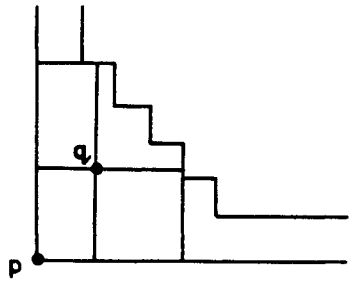


Figure 15: Modifying the visibility area.

(see Figure 14) and $R_1, \ldots, R_i$, are the rectangles that have p and $q_1, \ldots, q_i$ as vertices.

We store all these rectangles, for all points passed by the sweep plane, in one 2-dimensional stabbing tree, that is, a structure that holds a set of rectangles such that for a given point p we can efficiently determine which rectangles contain p. Such a tree exists and it uses $O(n \log n)$ space, has a query time of $O(\log^2 n + k)$, and update time $O(\log^2 n)$ when the set contains n rectangles. (In our application the set contains up to $n + k$ rectangles, where k is the total number of answers.)

When the sweep plane reaches a point q we search with q in the stabbing tree to determine the rectangles it contains. With each rectangle we keep track of which visibility area it is a part of, so that we can report the correct pair of visible points. As the rectangles overlap we might report a pair more than once, but this can be avoided. Next we have to update the visibility areas. To this end we remove, from the stabbing tree, all rectangles that have been found. Because they contain q they are no longer completely part of a visibility area. Instead we insert for each answer two new rectangles; see Figure 15. These rectangles are, first, the rectangle with p as its leftmost bottommost point, q on the right boundary, and as top boundary the top boundary of the highest rectangle found and, second, the rectangle with p as its leftmost bottommost point, q on the top boundary, and as right boundary the right boundary of the rightmost rectangle. It is easy to see that in this way the set of rectangles is maintained correctly.

Theorem 4.1 *Given a set V of n points in 3-dimensional space, all pairs of rectangular visible points can be determined in $O((n + k) \log^2 n)$ time using $O((n + k) \log n)$ space.*

Proof: With each rectangular visible pair found we insert two rectangles. Moreover, for each point encountered by the sweep plane we insert a rectangle, perform a query, and remove some rectangles. Hence, the total number of rectangles that are inserted is $O(n + k)$ and the total insertion time is

$O((n+k)\log^2 n)$. The same holds for the total deletion time. The total time required for queries is $O(n\log^2 n)$, because there are n queries. The space bound follows from the maximal number of rectangles. (Of course we have to execute the algorithm four times, once for each quadrant.) □

Unfortunately, the algorithm does not generalize to multi-dimensional space in an easy way.

5 The Query Problem in 3-Dimensional Space

We now consider the query problem in 3-dimensional space. Given a point $p = (x, y, z)$ we will only show how to determine those points $q = (x', y', z')$ that are visible from p with $x' > x$, $y' > y$, and $z' > z$. The points in the seven other octants can be found in a similar way. To solve the problem we need the following theorem.

Theorem 5.1 *Given a set of n points in 3-dimensional space, we can store them using $O(n\log^2 n)$ space, such that, given an orthogonal rectangle, parallel to the xy-plane, we can determine in $O(\log^2 n)$ time the first point we hit when moving the rectangle in the positive z-direction. The structure is dynamic and has an update time of $O(\log^3 n)$.*

Proof: This can be done using the 3-dimensional version of the structure for range searching in [3]. □

To solve the visibility problem we store all points in this structure. To perform a query with a point $p = (x, y, z)$ we start with the rectangle given by $[x..+\infty, y..+\infty]$ at position z and move it in the positive z-direction until it hits a point $q_1 = (x_1, y_1, z_1)$. We report this point and "split" the rectangle into two rectangles: $R_1 = [x..x_1, y..+\infty]$ and $R_2 = [x_1..+\infty, y..y_1]$. Now, we move these two rectangles until one of them hits a point $q_2 = (x_2, y_2, z_2)$. If $x_2 \geq x_1$, then we split R_2 into $[x_1..x_2, y..y_1]$ and $[x_2..+\infty, y..y_2]$. Otherwise, that is, $x_2 < x_1$, we need to modify both R_1 and R_2. R_1 is replaced by the rectangle $[x..x_2, y..+\infty]$ and R_2 by the rectangle $[x_2..+\infty, y..y_2]$. We move the remaining rectangles until they hit a point, repeating the process until no new points are hit. The union of the rectangles always forms the visibility area of p (in this octant) at the current z-position; see Figure 16. After some time a newly hit point may cause a drastic modification of the set of rectangles. For example, consider the before and after pictures of Figure 17 and Figure 18 on hitting point q_i with $x_i < x_2$. The only rectangles that are not modified are those to the left of q_i and those to the right of q_i, that have y-value less than y_i. The rectangle that q_i lies in is split and the others are thrown away.

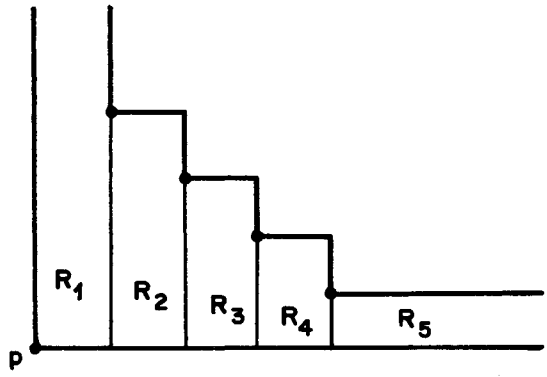


Figure 16: The union of rectangles is the visibility area.

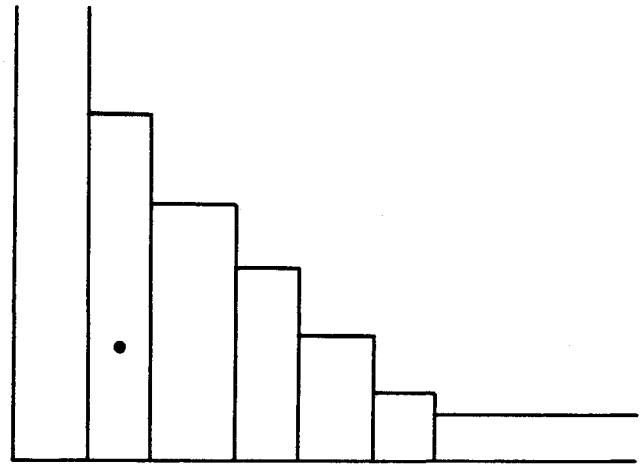


Figure 17: Before hitting q_i.

Hence, all points that are visible from p and lie in this octant are correctly reported.

Theorem 5.2 *There exists a structure for storing a set of n points in 3-dimensional space, using $O(n \log^2 n)$ space, such that rectangular visibility queries can be solved in time $O((1+k)\log^2 n)$, where k is the number of reported answers. The insertion and deletion time is $O(\log^3 n)$.*

Proof: The amount of space required and the insertion and deletion time follow from the preceding theorem. So we only have to prove the bound on the query time. Note that each time we find an answer, the number of query rectangles increases by at most two. To estimate the total amount of time required we have to look at the total number of queries with rectangles. There is at least a first one. Next, when we find an answer, we split one

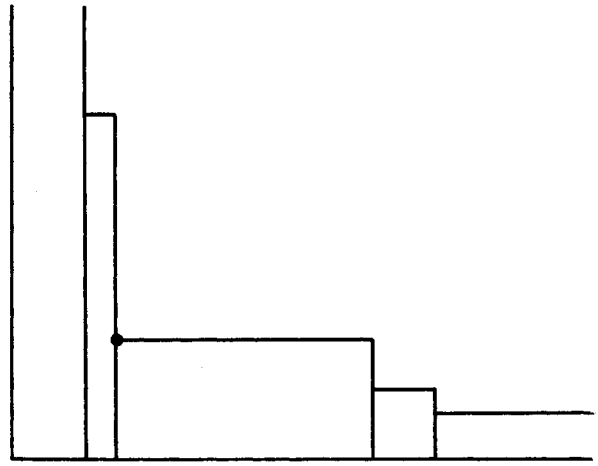


Figure 18: After hitting q_i.

rectangle into two and perform a query with each of these two new rectangles. Also, we have, perhaps, to modify other rectangles. When we find another point we again split a rectangle and perform two new queries, etc. Therefore, to obtain a rapid query time, we have to provide an efficient implementation of the intuitive rectangle-sweep-and-split process introduced above.

To find the first point that is hit by the set of rectangles, we perform a separate query for each rectangle and store the answers in a priority queue, say, with respect to their z-coordinates. Hence, performing a min query produces the sought point. Now, this hit point causes one rectangle to split and, perhaps, causes the removal of other rectangles. To determine which rectangle is split and which are removed, we store the "active" hit points and, hence, the rectangles, in a balanced binary search tree organized by x-values, and perform suitable queries on it. To remove a rectangle, we must delete it from the set of rectangles and remove its entry in the priority queue. Therefore, we link rectangles in the search tree to their hit points in the priority queue. The split rectangle is also removed and is replaced by two new rectangles. We must add these to the set of rectangles, find their hit points, and insert them into the priority queue. Finally, we need eight such structures rather than one, one for each octant defined by the query point.

Observe that each hit point introduces at most two new rectangles and each rectangle determines at most one hit point before it is either split or removed. We charge a rectangle's removal to the hit point that created it in the first place. Now, finding a hit point requires $O(\log^2 n)$ time by Theorem 5.1 and by noting that the other costs mentioned above are $O(\log n)$. Hence,

because the total number of queries is $1 + 2k$ (per octant) and each query takes $O(\log^2 n)$ time, the total query time follows immediately. □

Unfortunately, the query time is highly dependent on the number of answers. A structure with a query time that is only linear in the number of answers would be desirable but is unavailable at present.

6 Conclusions, Extensions, and Open Problems

In this paper we have introduced a new notion of visibility, *rectangular visibility*, and we have given a number of algorithms and data structures to deal with it. Although related to many other rectangle problems, rectangular visibility seems to be harder because the visibility of two points is a global property of the whole set, rather than a local property of the two points. Nevertheless, an optimal solution was devised for finding all pairs of visible points in a set, using a plane sweep approach and some special techniques for performing updates in an interval tree. Also, an efficient solution for the query problem was given, based on a structure for maintaining maximal elements in [7]. In the 3-dimensional case the problem really becomes harder. Again solutions have been given, but their efficiency depends on the number of visible pairs.

Rectangular visibility can be extended and restricted in many ways. In [6] the special case in which the rectangle is a square is handled. Rectangular visibility is a special case of connectivity as introduced in [8]. In [8] a number of extensions and generalizations are treated, for example, rectangular visibility in a set of line segments.

A large number of open problems remain. First of all, most of the bounds stated in this paper are not (proven to be) optimal. Hence, there is the question of improving the bounds, especially in the 3-dimensional case it would be interesting to have bounds that are only linearly dependent on the number of answers. Second, there is the question of solutions in higher dimensional space. The methods described for 2- and 3-dimensional space do not generalize to multi-dimensional space. The reason is that encountering an answer changes the visibility "area" for a point in quite a drastic way, therefore other techniques have to be developed.

An obvious extension to the notion of rectangular visibility is that three or more points are said to be rectangular visible if there is a rectangle that contains all of them and no other points. Many more extensions and open problems can be found in [8].

References

[1] N. Alon, Z. Füredi, and M. Katchalski. Separating pairs of points by standard boxes. *European Journal of Combinatorics*, 6:205–210, 1985.

[2] J.L. Bentley and D. Wood. An optimal worst case algorithm for reporting intersections of rectangles. *IEEE Transactions on Computers*, EC-29:571–576, 1980.

[3] H. Edelsbrunner. A note on dynamic range searching. *Bulletin of the European Association for Theoretical Computer Science*, 15:34–40, 1981.

[4] J.M. Keil. Minimally covering a horizontally convex orthogonal polygon. In *Proceedings of the Second ACM Symposium on Computational Geometry*, pages 43–51, 1986.

[5] K. Mehlhorn. *Data Structures and Algorithms 3: Multi-Dimensional Searching and Computational Geometry.* Springer-Verlag, New York, 1984.

[6] J.I. Munro, M.H. Overmars, and D. Wood. *Square Visibility.* Technical Report , Department of Computer Science, University of Waterloo, 1986.

[7] M.H. Overmars and J. van Leeuwen. Maintenance of configurations in the plane. *Journal of Computer and System Sciences*, 23:166–204, 1981.

[8] M.H. Overmars and D. Wood. *Connectivity and its Generalizations.* Technical Report , Department of Computer Science, University of Waterloo, 1986.

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

[10] D.E. Willard and G.S. Lueker. Adding range restriction capability to dynamic data structures. *Journal of the ACM*, 32:597–617, 1985.