Generalized Hidden Surface Removal

Mark de Berg

RUU-CS-93-05
February 1993

Utrecht University

Department of Computer Science

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

Generalized Hidden Surface Removal

Mark de Berg

Technical Report RUU-CS-93-05
February 1993

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

ISSN: 0924-3275

Generalized Hidden Surface Removal

Mark de Berg*

Abstract

In this paper we study the following generalization of the classical hidden surface removal problem: given a set S of objects, a view point and a point light source, compute which parts of the objects in S are visible, subdivided into parts that are lit and parts that are not lit.

We prove tight bounds on the maximum combinatorial complexity of such views and give efficient output-sensitive algorithms to compute the views for three cases: (i) S consists of non-intersecting triangles, (ii) S consists of horizontal axis-parallel rectangles, (iii) S is the set of faces of a polyhedral terrain.

1 Introduction

One of the basic problems in computer graphics is the hidden surface removal problem: given a set of objects in 3-space and a view point p_{view}, compute which parts of the objects are visible from p_{view}. More formally, one wants to compute the visibility map of the objects, which is the subdivision of the viewing plane into maximal regions, such that in each region either no object is visible, or one object is visible. Because of its importance in computer graphics the hidden surface removal problem has been studied extensively, both from the practical side and from the theoretical side. However, hidden surface removal is only one step in drawing pictures of a scene; to obtain realistic images one should not only compute which parts of each object are visible, but also information about the intensity of the light that reflects from these visible parts. There are some 'practical' methods for obtaining this information (for example, ray tracing or radiosity), but these methods are not exact and computationally very expensive.

We start the investigation of this problem from the computational geometry point of view by considering the following problem: given a set S of objects in 3-space, a view point p_{view} and a point light source p_{light}, compute which parts of the objects are visible from p_{view}, subdivided into parts which are lit by p_{light} and parts which are not lit. (We are also interested in the parts which are visible but not lit, because we assume—as is common in computer graphics—the presence of ambient light.) In other words, we are interested in the subdivision of the viewing plane into maximal regions, such that in each region either no object is visible, or one object is visible and completely lit, or one object is visible and completely in shadow. In fact, we do not only want to know the regions, but we also want for each region the information which object is visible and whether it is lit. We call this subdivision the *generalized visibility map*. For clarity we call the subdivision obtained without considering lighting information the *standard visibility map*. We study generalized visibility maps for three different classes of objects, namely the general case where S consists of n non-intersecting triangles, the case where S consists of n axis-parallel rectangles parallel to the xy-plane, and the case where S is the set of faces of a polygonal terrain with a total of n edges. Next we discuss our results for each of these settings, and compare them to the known results on the standard hidden surface removal problem.

In Section 2 we consider the general case where we are given a set of non-intersecting triangles. It is easily seen that the maximum combinatorial complexity of the standard visibility map is $\Theta(n^2)$.

*Dept. of Computer Science, Utrecht University, P.O.Box 80.089, 3508 TB Utrecht, the Netherlands. Supported by the Dutch Organization for Scientific Research (N.W.O.), and by ESPRIT Basic Research Action No. 7141 (project ALCOM II: *Algorithms and Complexity*).

It is also known how to compute the standard map in $O(n^2)$ time [15]. In general, however, the complexity of the map will be smaller than quadratic. Hence, people have tried to develop *output-sensitive* algorithms, that is, algorithms whose running time depends on the total complexity k of the map. This turned out to be a difficult problem, and only recently an output-sensitive algorithm that can handle arbitrary (non-intersecting) triangles was presented by de Berg et al. [6]. The running time of their algorithm is $O(n^{1+\epsilon}\sqrt{k})$.[1] By implementing one of the data structures supporting the algorithm more efficiently, Agarwal and Matoušek [1] were able to improve the time bound to $O(n^{1+\epsilon} + n^{2/3+\epsilon}k^{2/3})$.

For the generalized hidden surface removal problem for non-intersecting triangles, we show the following results. First of all, we prove that the maximum combinatorial complexity of the generalized map is $\Theta(n^3)$. We also present a simple output-sensitive algorithm to compute the generalized map in $O(n^2 \log n + k)$ time. Finally, we argue that it will be very difficult to devise an output-sensitive algorithm whose running time is $o(n^2)$ for small k. We do this by showing that our problem is at least as hard as the famous problem of detecting whether a set of n triangles in the plane completely covers the unit square.

In Section 3 we consider a second class of objects: axis-parallel rectangles parallel to the xy-plane. As in the case of triangles, the maximum complexity of the standard map is $\Theta(n^2)$, whereas for the generalized map the bound is $\Theta(n^3)$. The best solutions known to date for computing the standard map of a set of axis-parallel rectangles run in time $O((n + k)\log n)$ [2, 7, 9], or in time $O(n^{1+\epsilon} + k)$ [9]; the algorithm that we present for the generalized case runs in time $O((n\sqrt{n} + k)\log^2 n)$.

Our solution is based on a data structure for so-called *union range queries*, which we believe to be of independent. The structure stores a semi-dynamic (insertions only) set of rectangles in the plane in such a way that the union of the rectangles inside a query rectangle can be found efficiently. (In fact, the structure can handle axis-parallel polygons as query ranges.) What makes devising this structure difficult is that we are not allowed to store the union explicitly, because the union can have quadratic complexity. Thus our structure stores the union in an implicit way; it uses $O(n\sqrt{n}\log n)$ storage, the amortized insertion time is $O(\sqrt{n}\log^2 n)$ and the query time is $O((l + 1)\log^2 n)$, where l is the complexity of the union inside the query rectangle. Our structure is closely related to a structure described by Overmars and Yap [17]. They have shown how to maintain the measure of a set of rectangles, using $O(\sqrt{n}\log n)$ time per update. We want to use the structure for answering union range queries, where the number of queries can be much larger than the number of updates. We therefore do not want to spend, say, $\Omega(\sqrt{n}\log n)$ time for a query (unless the reported union has large complexity, of course). This necessitates the use of additional machinery and a careful analysis.

Finally, in Section 4 we study polyhedral terrains. For this case we prove a $\Theta(n^2)$ bound on the maximum complexity of the generalized map, which is the same as the bound for the standard map. Katz et al. [13] have shown that the standard map can be computed in $O((n\alpha(n) + k)\log n)$ time. We give an algorithm for computing the generalized map whose running time is $O(n\log^2 n + n\sqrt{k\alpha(n)\log n} + k\log n)$.

The algorithm uses a data structure that maintains the upper envelope of a set of line segments in the plane under insertions, such that the l intersections of a query segment with the upper envelope can be computed efficiently. Again, the possible number number of changes in the envelope makes it necessary to store it implicitly. Our structure can achieve, for any $1 < m < n$, a query time of $O((m + l)\log n)$ with an update time of $O(\log n \log m + (n/m)\alpha(n/m))$. It uses $O(n\log m + n\alpha(n/m))$ storage.

Before we proceed let us introduce some notation that we use throughout the paper. S will be a set of non-intersecting polygons in 3-space, and p_{view} and p_{light} are points in 3-space. We call p_{view} the *view point* and p_{light} the *light source*. A projection with p_{view} as the center of projection is called a p_{view}*-projection*; the term p_{light}*-projection* is defined analogously. We define

[1] Such a bound means that, given any $\epsilon > 0$, one can implement the algorithm so that it runs in $O(n^{1+\epsilon}\sqrt{k})$ time. The constants hidden by the O-notation depend on ϵ.

a point p to be visible from p_{view} iff the interior of the line segment $\overline{p_{view}p}$ does not intersect any polygon in S; point p is lit by p_{light} iff the interior of the line segment $\overline{p_{light}p}$ does not intersect any polygon in S. Furthermore, let $\mathcal{V}_L(S, p_{view}, p_{light})$ be the set of points in 3-space that lie on a polygon in S, are visible from p_{view} and lit by p_{light}, and let $\mathcal{V}_N(S, p_{view}, p_{light})$ be the set of points that lie on a polygon in S are visible from p_{view} and not lit by p_{light}. Note that if we take the connected components of these sets, we get a collection of polygonal regions which are contained in the polygons of S. From now on we let $\mathcal{V}_L(S)$ and $\mathcal{V}_N(S)$ denote these regions, omitting the parameters p_{view} and p_{light} for the sake of brevity. If we take the p_{view}-projection of these regions onto the viewing plane then we obtain the generalized visibility map of S with respect to p_{view} and p_{light}, denoted by $\mathcal{M}(S, p_{view}, p_{light})$, or by $\mathcal{M}(S)$ for short. (Actually, the projections only give us the maximal regions where a single polygon is visible and completely lit, and the regions where a single polygon is visible and completely in shadow. The remaining regions, where no polygon is visible, are just the regions that form the complement of the projections.) Clearly, the problem of computing $\mathcal{M}(S)$ is equivalent to the problem of computing $\mathcal{V}_L(S)$ and $\mathcal{V}_N(S)$.

2 The General Case

In this section we prove tight bounds on the maximum complexity of the generalized visibility map $\mathcal{M}(S)$ of a set S of non-intersecting polygons in 3-space with n vertices in total, and we give an output-sensitive algorithm to compute the map. We start with the combinatorial bound. The proof is a somewhat simplified version of a proof given by Overmars [16].

Theorem 2.1 *Let S be a set of non-intersecting polygons in 3-space with n vertices in total. The maximum combinatorial complexity of the generalized visibility map $\mathcal{M}(S)$ is $\Theta(n^3)$. This bound also holds if S is a set of n axis-parallel rectangles that are parallel to the xy-plane.*

Proof: We first prove the upper bound. Consider a fixed polygon $\mathcal{P} \in S$. The edges of the regions of $\mathcal{V}_L(S)$ and $\mathcal{V}_N(S)$ on $\mathcal{P}$ are contained in a segment on $\mathcal{P}$ which is either an edge of $\mathcal{P}$, or the p_{view}-projection of an edge of another polygon in S onto $\mathcal{P}$, or the p_{light}-projection of an edge of another polygon onto $\mathcal{P}$. The vertices of the regions on $\mathcal{P}$ are intersections between two such segments. Clearly, there are $O(n)$ segments on $\mathcal{P}$, which proves that the maximum complexity of the regions of $\mathcal{V}_L(S)$ and $\mathcal{V}_N(S)$ on $\mathcal{P}$ is $O(n^2)$. The upper bound immediately follows.

We now give an example that shows that the upper bound we just proved is tight. Our construction uses a set $R = \{r_1, r_2, \ldots, r_n\}$ of axis-parallel rectangles that are parallel to the xy-plane, a set $L = \{l_1, l_2, \ldots, l_n\}$ of n lines parallel to the x-axis and a set $M = \{m_1, m_2, \ldots, m_n\}$ of lines parallel to the y-axis, which are defined as follows. The set R consists of rectangles $r_i = [-1 : 1] \times [-1 : 1] \times -i$. The set L consists of the lines $l_j = (0, j\varepsilon, 0) + \lambda(1, 0, 0)$ and the set M consists of the lines $m_k = (k\varepsilon, 0, 1) + \lambda(0, 1, 0)$, where $\varepsilon = \frac{1}{2n}$ and $\lambda \in \mathbf{R}$. (In fact, we could also use very long and thin axis-parallel rectangles instead of lines.) The idea is to place $p_{light} = (x_l, 1/2, z_l)$ somewhere above L and $p_{view} = (1/2, y_p, z_p)$ somewhere above M such that every pair l_j, m_k induces a vertex on each of the r_i. Let us first make sure that the shadow cast by all the lines in L falls on each r_i. To this end we just have to choose z_l large enough, for example $z_l = n/2$. See Figure 1(a), where a view along the x-axis is given of the sets R (the fat line segments) and L (the dots) and the point p_{light} (the small circle). Notice that the shadows always fall on each r_i if we take $p_{light} = (x_l, 1/2, n/2)$, irrespective of the choice of x_l. In the same way we can ensure that every line in M obscures a part on each r_i from p_{view}, by taking $z_v = n/2$. We now have that the complexity of the regions which are visible and lit (and also of the regions which are visible and not lit) on the topmost rectangle r_1 is $\Omega(n^2)$. On other rectangles, however, this need not be the case: a rectangle r_i can be (partially) obscured by some $r_{i'}$, $i' < i$. But by choosing x_l and y_v large enough, we can make sure that the rectangles r_i do not influence each other. See Figure 1(b), where a view along the y-axis is given of the sets R and L and of the point p_{light}. (Note that all lines in L are parallel to the x-axis and at the same height, so only one of them is visible in this side view.) Thus the total complexity of $\mathcal{V}_L(R \cup L \cup M)$ and $\mathcal{V}_N(R \cup L \cup M)$ is $\Omega(n^3)$. □

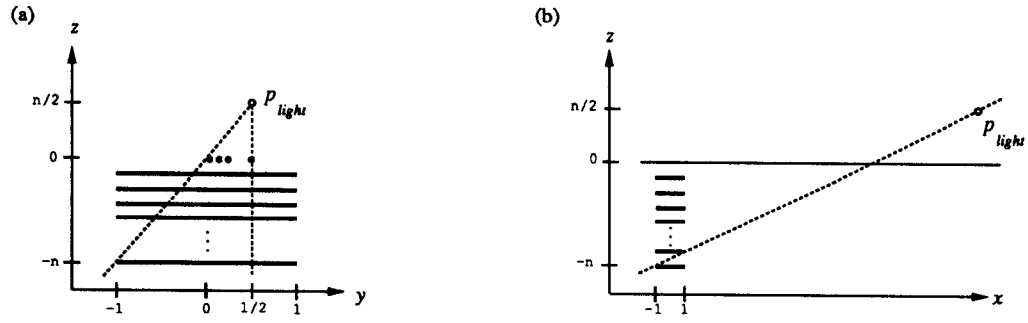

Figure 1: The lower bound construction.

Theorem 2.2 *Let S be a set of non-intersecting polygons in 3-space with n vertices in total. The generalized visibility map $\mathcal{M}(S)$ can be computed in $O(n^2 \log n + k)$ time, where k is the total complexity of the map.*

Proof: First compute the regions on each polygon in S which are visible from p_{view}, and the regions which are lit from p_{light}. Using the algorithm of McKenna [15] this can be done in $O(n^2)$ time in total. For a polygon $\mathcal{P} \in S$, let $V(\mathcal{P}) \subset \mathcal{P}$ be the visible portion of $\mathcal{P}$ and $L(\mathcal{P})$ the region that is lit. Notice that the regions $V(\mathcal{P}) \cap L(\mathcal{P})$ and $V(\mathcal{P}) - L(\mathcal{P})$ are exactly the regions of $\mathcal{V}_L(S)$ and $\mathcal{V}_N(S)$ that are located on $\mathcal{P}$. Using the red-blue intersection algorithm of Mairson and Stolfi [14] we can compute these regions in time $O(\ (|V(\mathcal{P})| + |L(\mathcal{P})|) \log(|V(\mathcal{P})| + |L(\mathcal{P})|) + k_{\mathcal{P}}\ )$, where $k_{\mathcal{P}}$ is the number of intersections between edges of $V(\mathcal{P})$ and edges of $L(\mathcal{P})$. Observe that every intersection between an edge of $V(\mathcal{P})$ and an edge of $L(\mathcal{P})$ is a vertex of the region that we want to compute. It follows that the total time to compute $\mathcal{V}_L(S)$ and $\mathcal{V}_N(S)$ is bounded by

$$\sum_{\mathcal{P}} O(\ (|V(\mathcal{P})| + |L(\mathcal{P})|) \log(|V(\mathcal{P})| + |L(\mathcal{P})|) + k_{\mathcal{P}}\ ) = O(n^2 \log n + k).$$

□

Although we now have an output-sensitive algorithm for the generalized hidden surface removal problem, its running time is not as good as what one might hope for. In particular, the algorithm needs $\Omega(n^2 \log n)$ time, even if the output complexity is only constant. However, we think that it will be very difficult to find an algorithm whose running time is much better, or, more precisely, to find an algorithm whose running time is $o(n^2)$ for small k. Our belief is based on the following observation.

Consider the well known *three collinear points problem*: given a set of n points in the plane, decide whether any three of them are collinear. This question can be answered in $O(n^2)$ time by constructing the dual arrangement (see [8]), and it is a famous open problem if this can be done in $o(n^2)$. In fact, the three collinear points problem is as least as hard as the "$A+B \cap C \neq \emptyset$?"-problem: given three sets A, B, C of n integers, are there $a \in A, b \in B, c \in C$ such that $a + b = c$? There are several other geometric problems that can be shown to be at least as hard as the "$A+B \cap C \neq \emptyset$?"-problem; some people call these problems *n^2-hard*. A well known example of an n^2-hard problem is the following *union-of-triangles problem*: given a set of triangles in the plane, does their union completely cover the unit square $[0:1] \times [0:1]$? We now show that the generalized hidden surface removal problem is at least as hard as this problem.

Let $T = \{t_1, t_2, \ldots, t_n\}$ be a set of triangles in the plane. Let ALG be an algorithm for the generalized hidden surface removal problem with running time $T(n, k)$, and let $f(n) = T(n, k)$ be the running time for constant k. We construct a set S of polygons in 3-space, such that the union-of-triangles problem for T can be solved by applying ALG to S. First, we place the light source

p_{light} at some point above the unit square $s_{unit} = [0:1] \times [0:1] \times 0$, say at $(1/2, 1/2, 10)$. Next we put for each triangle $t_i \in T$ a triangle $\tilde{t}_i$ which is parallel to and slightly above the xy-plane, such that the p_{light}-projection of $\tilde{t}_i$ onto the xy-plane is exactly t_i. We place all these triangles at different heights so that they do not intersect. Let $\widetilde{T}$ be the resulting set of triangles. Notice that T completely covers the unit square if and only if s_{unit} is completely covered by the shadows of the triangles in $\widetilde{T}$. We now place the view point p_{view} such that it can see the square s_{unit} completely from above. This can be achieved, for example, by giving p_{view} a positive z-coordinate which is smaller than the smallest z-coordinate of any of the triangles $\tilde{t}_i$. We can thus solve the union-of-triangles problem by computing the sets $\mathcal{V}_L(\widetilde{T} \cup \{s_{unit}\})$ and $\mathcal{V}_N(\widetilde{T} \cup \{s_{unit}\})$. However, we wanted to show that the generalized hidden surface removal problem is difficult even if the output complexity is small; in our example this can be large due to the regions on the triangles $\tilde{t}_i$. But this is easy to avoid: simply put a rectangle r_{shield} in front of the view point, such that r_{shield} obscures all the triangles $\tilde{t}_i$, but not s_{unit}. Now the output of the generalized hidden surface removal problem is the square s_{unit} subdivided into lit and non-lit parts plus the rectangle r_{shield}. To solve the union-of-triangles problem for T we run algorithm ALG on the set $\widetilde{T} \cup \{s_{unit}, r_{shield}\}$. If the algorithm halts within $f(n)$ time we know the answer to the union-of-triangles problem. But if the algorithm has not finished within $f(n)$ time, then we know that the output is not constant and we also know the answer. Hence, if we can solve the generalized hidden surface removal problem in $f(n) = o(n^2)$ time for constant k, then we can also solve the union-of-triangles problem in $o(n^2)$ time.

We have given strong evidence that it will be very difficult (perhaps even impossible?) to obtain an output-sensitive algorithm for the generalized hidden surface removal problem for an arbitrary set of triangles whose running time is $o(n^2)$ for small k. For special cases, however, such algorithms can be given, as is shown in the next two sections where we study axis-parallel rectangles and polyhedral terrains.

3 Axis-Parallel Horizontal Rectangles

Let S be a set of n axis-parallel rectangles that are parallel to the xy-plane. Recall from Theorem 2.1 that the maximum combinatorial complexity of the generalized map $\mathcal{M}(S)$ is $\Theta(n^3)$. In this section we give an efficient output-sensitive algorithm to compute $\mathcal{M}(S)$.

3.1 The Strategy

The strategy that we follow consists of two phases: in the first phase we compute which parts of the rectangles in S are visible from p_{view}, and in the second phase we determine which portions of these parts are lit by p_{light}.

To simplify the exposition we assume that the z-coordinates of p_{view} and p_{light} are greater than the z-coordinates of the rectangles in S, and that all the rectangles lie above the xy-plane. These restrictions are straightforward to remove. Before we can give a more detailed description of the second phase of the algorithm, we need the following definitions. Let R be a set of axis-parallel polygons in the plane. A *union range query* on the set R with an axis-parallel query polygon $\mathcal{P}$ asks for $\mathcal{P} \cap (\bigcup R)$, that is, for the part of the union of R inside $\mathcal{P}$. We begin the second phase by initializing an empty structure $\mathcal{D}$ for union range queries. After that we process the rectangles $r \in S$ in order of decreasing z-coordinate—i.e. in order of increasing distance to p_{light}—as follows. First, we perform a union range query with the p_{light}-projection of every visible region on r (recall that these regions have been computed in the first phase) in the data structure $\mathcal{D}$. This gives us the parts of $\mathcal{V}_N(S)$—and, hence, of $\mathcal{V}_L(S)$—on r. Next, we insert the p_{light}-projection of r into $\mathcal{D}$. Thus the second phase is similar to the algorithm of Güting and Ottmann [10] for the standard hidden surface removal problem. (They treat the rectangles in order of increasing distance to the view point, and query with the rectangles to compute which parts are visible.) The main difference is that we are not allowed to maintain the union explicitly, as is done in [10].

Theorem 3.1 *Let S be a set of n axis-parallel rectangles that are parallel to the xy-plane. The generalized visibility map $\mathcal{M}(S)$ can be computed in $O((n\sqrt{n}+k)\log^2 n)$ time, where k is the total complexity of the map.*

Proof: Consider the algorithm given above. The correctness of the algorithm is evident. As for the time complexity, we note that the first phase can be done in $O((n + k_{vis})\log n)$, where k_{vis} is the complexity of the visible pieces, by a standard hidden surface removal algorithm for rectangles [2, 7, 9]. Note that $k_{vis} \leqslant k$. In the next subsection we describe a data structure $\mathcal{D}$ with $O(\sqrt{n}\log^2 n)$ amortized insertion time, and $O((m+l)\log^2 n)$ query time, where m is the number of vertices of the query polygon and l is the output complexity. (The structure also needs $O(n \log n)$ initializing time.) See Lemma's 3.10 and 3.15. Hence, the time needed for the second phase of the algorithm is bounded by

$$O(n\sqrt{n}\log^2 n) + \sum_{\mathcal{P}} O((|\mathcal{P}| + l_{\mathcal{P}})\log^2 n),$$

where we sum over all polygons $\mathcal{P}$ with which we query, and $l_{\mathcal{P}}$ is the complexity of the answer to the query. The time bound stated in the lemma now follows from the fact that $\sum_{\mathcal{P}} |\mathcal{P}| = k_{vis} \leqslant k$ and $\sum_{\mathcal{P}} l_{\mathcal{P}} \leqslant k$. □

3.2 Union Range Queries

Let R be a set of n axis-parallel rectangles in the plane. In this subsection we devise a data structure for so-called *union range queries* on R. Such queries ask for the part of the union of R inside an axis-parallel query polygon. More formally, the answer to a query with polygon $\mathcal{P}$ is $\mathcal{P} \cap (\bigcup R)$. We prove the following theorem.

Theorem 3.2 *Let R be a set of n axis-parallel rectangles in the plane. It is possible to store R into a data structure that uses $O(n\sqrt{n}\log n)$ storage, such that union range queries with an axis-parallel query polygon $\mathcal{P}$ can be answered in $O((|\mathcal{P}| + k)\log^2 n)$ time, where k is the size of the output. Rectangles can be inserted into the structure in $O(\sqrt{n}\log^2 n)$ amortized time, assuming we know all rectangles to be inserted in advance.*

3.2.1 The structure

Let us first describe the structure for a static set R. The basic ingredient that we use is a partitioning of the plane described by Overmars and Yap [17]. This partitioning is as follows. Sort the vertical[2] boundary edges of the rectangles in R by x-coordinate, and draw a vertical line through every $\sqrt{n}$-th vertical boundary edge. This partitions the plane into $O(\sqrt{n})$ vertical *slabs*. Each slab is divided further into $O(\sqrt{n})$ rectangular *cells* by drawing horizontal segments in the following way. First, we draw a segment through every horizontal boundary edge that has an endpoint inside the slab. The remaining horizontal boundary edges that cross the slab are sorted by y-coordinate, and a horizontal segment is drawn through every $\sqrt{n}$-th edge. We say that a rectangle r partially covers a cell of the partitioning if an edge of r intersects the interior of the cell. Overmars and Yap show that this partitioning has the following properties:

- There are $O(n)$ cells.
- Each rectangle in R partially covers $O(\sqrt{n})$ cells.
- No cell contains vertices in its interior.
- Each cell has $O(\sqrt{n})$ rectangles in R partially covering it.

[2] In this subsection, we call lines or segments that are parallel to the y-axis vertical, and lines or segments that are parallel to the x-axis horizontal.

One can associate a binary tree $\mathcal{T}$ with this partitioning. The tree consists of one *top tree* whose leaves—we call them *top leaves*—represent the slabs. The left-to-right order of the top leaves corresponds to the left-to-right order of the slabs. Each top leaf is the root of a *bottom tree* whose leaves—called *bottom leaves*—represent the cells in the corresponding slab, again in an ordered manner.

Before we describe the extra information that we need to be able to answer union range queries efficiently, it is convenient to introduce some notation. We denote the cell of the partitioning represented by a bottom leaf γ by $Cell(\gamma)$. An internal node ν of the tree $\mathcal{T}$ represents the region of the plane which is the union of the cells corresponding to the bottom leaves below ν; this region is also denoted by $Cell(\nu)$. Observe that $Cell(\nu)$ is always a rectangular region, so we can write $Cell(\nu) = [x(\nu) : x'(\nu)] \times [y(\nu) : y'(\nu)]$. Also note that for a top leaf ν $Cell(\nu)$ is just the vertical slab corresponding to ν. For a rectangle r we define $x(r)$, $x'(r)$, $y(r)$ and $y'(r)$ such that $r = [x(r) : x'(r)] \times [y(r) : y'(r)]$.

Let ν be a node (possibly a leaf) in $\mathcal{T}$. The set $R_{part}(\nu) \subset R$ is defined to be the set of rectangles that partially cover $Cell(\nu)$, and $R_{full}(\nu) \subset R$ is the set of rectangles that fully cover $Cell(\nu)$ but not $Cell(parent(\nu))$. In the sequel, we always restrict our attention to the part of these rectangles inside $Cell(\nu)$. Note that $R_{full}(\nu) = \emptyset$ for nodes ν in the top tree, assuming that the rectangles in R are bounded. Finally, let $R(\nu) = R_{part}(\nu) \cup R_{full}(\nu)$. For the nodes ν in the top tree (including the top leaves) we store the following information, see Figure 2.

- Let $Y_C(\nu)$ be the set of y-coordinates y^* such that $[x(\nu) : x'(\nu)] \times y^*$ is completely contained in $\bigcup R(\nu)$. $Y_C(\nu)$ consists of a set of intervals, which we store in a list $\mathcal{Y}_C(\nu)$. (In fact, we want to be able to do binary search on the endpoints of the intervals in $Y_C(\nu)$, so we implement $\mathcal{Y}_C(\nu)$ as a balanced binary tree. To avoid confusion with our main tree $\mathcal{T}$, however, we will call $\mathcal{Y}_C(\nu)$ a list anyway. The same remark holds for the other lists stored at nodes in $\mathcal{T}$.)
- Let $Y_P(\nu)$ be the set of y-coordinates y^* such that $[x(\nu) : x'(\nu)] \times y^*$ intersects $\bigcup R(\nu)$, that is, $Y_P(\nu) = \bigcup\{[y(r) : y'(r)] : r \in R(\nu)\}$. $Y_P(\nu)$ consists of a set of intervals, which we store in a list $\mathcal{Y}_P(\nu)$.

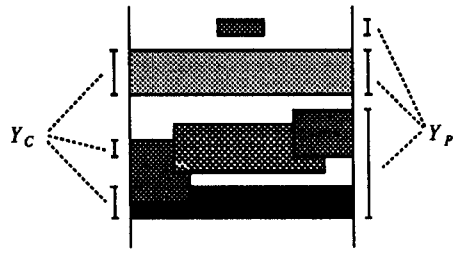

Figure 2: The lists $\mathcal{Y}_C(\nu)$ and $\mathcal{Y}_P(\nu)$ stored at node ν.

At the internal nodes ν of the bottom trees (excluding their roots, which are the top leaves), we store lists $\mathcal{X}_C(\nu)$ and $\mathcal{X}_P(\nu)$, which are defined analogously to $\mathcal{Y}_C(\nu)$ and $\mathcal{Y}_P(\nu)$, but with the roles of the x- and y-coordinate reversed.

Finally, at the bottom leaves γ we store the lists $\mathcal{X}_C(\gamma)$ and $\mathcal{Y}_C(\gamma)$.

Lemma 3.1 *The total size of $\mathcal{T}$ including all associated information is $O(n\sqrt{n}\log n)$.*

Proof: Every endpoint of an interval in a list $\mathcal{Y}_C(\nu)$ or $\mathcal{Y}_P(\nu)$ is the y-coordinate of some rectangle in $R(\nu)$. Similarly, every endpoint of an interval in a list $\mathcal{X}_C(\nu)$ or $\mathcal{X}_P(\nu)$ is the x-coordinate of some rectangle in $R(\nu)$. Hence, the total size of $\mathcal{T}$ is $\sum_\nu O(|R(\nu)|)$, which is $O(n\sqrt{n}\log n)$, see [17]. □

3.2.2 The query algorithm

We now describe the query algorithm for a rectangle q^* as query polygon; the generalization to axis-parallel polygons as query objects will be presented in Section 3.2.4. Before we describe the full query algorithm, let us see how to report $q \cap (\bigcup R(\gamma))$ for some rectangle q at a bottom leaf γ. Recall that $Cell(\gamma)$ has no vertices in its interior. This fact implies that it is quite easy to compute $q \cap (\bigcup R(\gamma))$ using the lists $\mathcal{X}_C(\gamma)$ and $\mathcal{Y}_C(\gamma)$. We leave the straightforward details of this to the reader and conclude:

Lemma 3.2 *Let q be a rectangle. The region $q \cap (\bigcup R(\gamma))$ at a bottom leaf γ can be reported in $O(\log n + k_\gamma)$ time, where k_γ is the complexity of $q \cap (\bigcup R(\gamma))$.*

The full query algorithm is a recursive procedure. Next we describe the actions performed by the query algorithm when a node ν in the top tree is visited by a query rectangle q. We distinguish three cases; we can decide which case occurs by searching with $y(q)$ and $y'(q)$ in the list $\mathcal{Y}_P(\nu)$.

case (i): $[y(q) : y'(q)] \cap Y_P(\nu) = \emptyset$.

In this case q is clearly disjoint from all the rectangles in $R(\nu)$, so we report $\emptyset$ and we are finished.

case (ii): $[y(q) : y'(q)] \cap Y_P(\nu) \neq \emptyset$ and $Cell(\nu)$ contains a vertex of q in its interior.

We first test whether $[y(q) : y'(q)] \subset Y_C(\nu)$, by searching with $y(q)$ and $y'(q)$ in the list $\mathcal{Y}_C(\nu)$. If this is true we report q and we are finished. Otherwise we continue the search in one or both children of ν: if $q \cap Cell(lc(\nu)) \neq \emptyset$ then we recurse into the left child $lc(\nu)$ of ν and if $q \cap Cell(rc(\nu)) \neq \emptyset$ then we recurse into the right child $rc(\nu)$ of ν.

case (iii): $[y(q) : y'(q)] \cap Y_P(\nu) \neq \emptyset$ and $Cell(\nu)$ does not contain a vertex of q in its interior.

We report $[x(\nu) : x'(\nu)] \times \{[y(q) : y'(q)] \cap Y_C(\nu)\}$; this computation can be done by searching with $y(q)$ and $y'(q)$ in $\mathcal{Y}_C(\nu)$. For each interval $[y : y']$ in $[y(q_\nu) : y'(q)] - Y_C(\nu)$ we recurse into the left child of ν with the query rectangle $q_l := [x(lc(\nu)) : x'(lc(\nu))] \times [y : y']$, and we recurse into the right child of ν with the query rectangle $q_r := [x(rc(\nu)) : x'(rc(\nu))] \times [y : y']$.

Nodes in the bottom tree are handled in a similar manner, with the roles of x- and y-coordinates reversed. The query algorithm starts by visiting the root of $\mathcal{T}$ with $q = q^*$. Let us first prove the correctness of the query algorithm.

Lemma 3.3 *$q^* \cap (\bigcup R)$ is the disjoint union of the rectangles reported by the query algorithm.*

Proof: First of all, it is easily seen that all rectangles reported by the algorithm are inside $q^* \cap (\bigcup R)$. Observe that if we report rectangles when visiting a node ν with some rectangle q, we only recurse with the parts of q that have not been reported yet. Thus the reported rectangles are disjoint from the ones reported in the subtree rooted at ν. Because they also lie inside $Cell(\nu)$, it follows that all reported rectangles are disjoint.

It remains to prove that all of $q^* \cap (\bigcup R)$ is reported. Suppose we visit node ν with query rectangle q. Let p be a point in $q \cap (\bigcup R(\nu))$ and let r be a rectangle in $R(\nu)$ containing point p. We shall prove by induction on the depth of the subtree rooted at ν that a rectangle containing point p is reported at ν or one of its descendants. The base case, where the depth equals zero and we are at a bottom leaf, is true by Lemma 3.2. So now consider an internal node ν. If $r \in R_{full}(\nu)$ then q itself, which is a rectangle containing p, is reported. So assume $r \in R_{part}(\nu)$ and no rectangle containing p is reported at ν. Then there will be a rectangle $q' \subseteq q$ that contains p and for which we recurse in a child μ of ν. From $r \in R_{part}(\nu)$ and $p \in r$, it follows that $r \in R(\mu)$. The induction hypothesis now tells us that a rectangle containing p will be reported when we recurse in μ. By setting $\nu = root(\mathcal{T})$ and $q = q^*$ we see that all of $q^* \cap (\bigcup R)$ is reported.

□

Next we analyze the time complexity of the query algorithm. The following lemma is easy to prove.

Lemma 3.4 *The time spent when we visit a node ν (not counting the time needed for the recursive calls) is $O(\log n + k_\nu)$, where k_ν is the number of rectangles reported at ν.*

Note that a node can be visited several times with different query rectangles $q \subset q^*$. The following lemma counts the total number of visits nodes in $\mathcal{T}$, where multiple visits to the same node are counted separately.

Lemma 3.5 *The total number of visits to nodes in $\mathcal{T}$ is $O(\log n + k_{rep} + k \log n)$, where k_{rep} is the total number of reported rectangles and k is the total complexity of $q^* \cap (\bigcup R)$.*

Proof: First note that the number of visits to nodes ν where the query rectangle has a vertex inside *Cell*(ν) is no more than $(\log n)$. Furthermore, the number of visits where we report a rectangle is trivially no more than k_{rep}.

So let us count the remaining visits. Let ν be a visited node and let q be a query rectangle with which we visit ν. Observe that we are either in case (i), or we are in case (iii) and we did not report any rectangle.

As for case (i), we charge this visit to the visit of *parent*(ν) by some rectangle $q' \supseteq q$. Note that q' always exists, is unique and gets charged at most $1 + l$ times, where l is the number of rectangles that were reported when q' visited *parent*(ν). Also observe that at *parent*(ν) case (i) did not occur for q'. Hence, the charging does not propagate any further. Thus we can account for all visits where case (i) occurs.

As for case (iii), assume that ν is a node in the top tree. Because we did not report anything when visiting ν with q, we know that there must be a vertical edge of $\bigcup R$ intersecting q. Hence, a vertex of $q^* \cap (\bigcup R)$ lies inside *Cell*(ν). If ν is a node in the bottom tree, there will be a horizontal edge of $\bigcup R$ intersecting q; in this case one can also argue that a vertex of $q^* \cap (\bigcup R)$ lies inside *Cell*(ν). For each vertex of $q^* \cap (\bigcup R)$ there are $O(\log n)$ nodes whose cells contain the vertex. Hence, the number of nodes where case (iii) applies and no rectangles are reported is $O(k \log n)$. □

Lemma 3.6 *For every rectangle $r = [x(r) : x'(r)] \times [y(r) : y'(r)]$ that is reported at a node ν in the top tree, it is true that either* Cell(ν) *or* Cell(sibling(ν)) *contains a vertex of $q^* \cap (\bigcup R)$ with y-coordinate $y(r)$ or $y'(r)$. Similarly, for every rectangle $r = [x(r) : x'(r)] \times [y(r) : y'(r)]$ that is reported at a node ν in the bottom tree, either* Cell(ν) *or* Cell(sibling(ν)) *contains a vertex of $q^* \cap (\bigcup R)$ with x-coordinate $x(r)$ or $x'(r)$.*

Proof: Let ν be a node in the top tree, and let q be a rectangle with which we visit ν. Suppose that we are in case (ii) and we report q. In this case the lemma must be true because the vertex of q contained in *Cell*(ν) is also a vertex of the original query rectangle q^*.

So now consider a rectangle r that is reported in case (iii) of the query algorithm, and let e be the bottom edge of r. Assume that e does not contain a vertex of $q^* \cap (\bigcup R)$. We claim that then the extension of e into *Cell*(*sibling*(ν)) must contain a vertex of $q^* \cap (\bigcup R)$. To see why this is true we distinguish two cases.

The first case is where another rectangle r' which contains e in its top edge has been reported at an ancestor of ν. Thus e is contained in the interior of $q^* \cap (\bigcup R)$. See Figure 3(a). Observe that the part of this bottom edge inside *Cell*(*parent*(ν)) cannot be fully contained in the interior of $q^* \cap (\bigcup R)$; otherwise a rectangle containing e would have been reported at an ancestor of ν. We conclude that there must be a vertex of $q^* \cap (\bigcup R)$ on the extension of e.

The second case is where e is not fully contained in the interior of $q^* \cap (\bigcup R)$. Note that if e is partially contained in the interior of $q^* \cap (\bigcup R)$ then e itself contains a vertex of $q^* \cap (\bigcup R)$. So assume that e is part of an edge of $q^* \cap (\bigcup R)$, as in Figure 3(b). Now the same argument applies as before: if the extension of e into *Cell*(*sibling*(ν)) does not contain an endpoint, then (a rectangle containing) it would have been reported at an ancestor, a contradiction.

For nodes ν in the bottom trees a similar argument holds, if we can show that each reported rectangle has a vertical edge in the interior of *Cell*(ν). But this must be true, because otherwise the reported rectangle completely spans the vertical slab that contains *Cell*(ν) and a rectangle

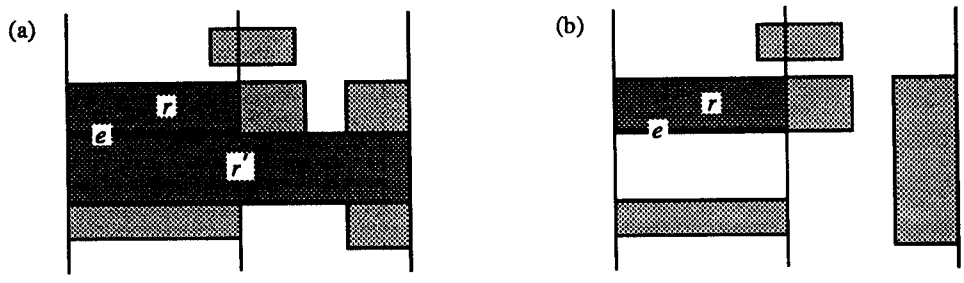

Figure 3: The two cases in the proof of Lemma 3.6.

containing this rectangle would have been reported at an ancestor of the top leaf corresponding to this slab. □

Lemma 3.7 *The total number of reported rectangles, k_{rep}, is $O(k \log n)$, where k is the total complexity of $q^* \cap (\bigcup R)$.*

Proof: By the previous lemma, we can charge every rectangle that is reported at node ν to a certain vertex of $q^* \cap (\bigcup R)$ that lies inside *Cell*(ν) or *Cell*(*sibling*(ν)). Moreover, this can be done in such a way that each vertex inside *Cell*(ν) gets charged at most four rectangles reported at ν or at *sibling*(ν) (in fact, at most one rectangle if we argue somewhat more carefully). Thus every vertex of $q^* \cap (\bigcup R)$ gets charged at most four times at every node on its search path, so $O(\log n)$ times in total. □

Finally, we are in the position to prove a bound on the query time.

Lemma 3.8 *The query time for a union range query with query rectangle q^* in the structure described above is $O((k+1)\log^2 n)$, where k is the total complexity of $q^* \cap (\bigcup R)$.*

Proof: From Lemma 3.4 it follows that the query algorithm takes time $\sum_\nu O(\log n + k_\nu)$, where we sum over all visited nodes ν (counting multiply visited nodes separately) and k_ν is the number of rectangles reported at ν. Using Lemma's 3.5 and 3.7 we can bound this sum as follows.

$$\begin{array}{ll} \sum_\nu O(\log n + k_\nu) & = \\ (\text{number of visits}) \cdot O(\log n) + O(k_{rep}) & = \\ O((\log n + k_{rep} + k \log n) \cdot \log n) + O(k \log n) & = \\ O((k+1)\log^2 n) & \end{array}$$

□

3.2.3 Insertions

We now show how to update the data structure described above. We assume that we know all rectangles to be inserted in advance. This implies that we can construct the partitioning and build a balanced 'skeleton tree' in $O(n \log n)$ time before the insertions start. Hence, we need not worry about adapting the partitioning or about rebalancing operations during an insertion.

Before we proceed, let us introduce some notation. Let r be the rectangle to be inserted. We define R^{old} to be the set of rectangles present in the structure before the insertion of r, and

$R^{new} = R^{old} \cup \{r\}$ to be the new set of rectangles. Similar definitions are made for the sets that are stored at a node ν: $R^{old}(\nu)$ is the set of rectangles stored at node ν before the insertion of r, $Y_C^{old}(\nu)$ is the set of y-coordinates y^* such that $[x(\nu) : x'(\nu)] \times y^*$ is completely contained in $\bigcup R^{old}(\nu)$, and so on. Finally, we let $Y_C^{extra}(\nu) = Y_C^{new}(\nu) - Y_C^{old}(\nu)$.

The insertion of a rectangle r into the structure influences only the nodes ν such that $r \in R^{new}(\nu)$. There are $O(\sqrt{n}\log n)$ such nodes, and they can be identified in the same amount of time, see [17]. We show that the information at a node ν in the top tree can be updated in $O(\log n)$ amortized time. The information in the bottom trees can be updated in the same way.

Let ν be a node such that $r \in R^{new}(\nu)$. Updating the list $\mathcal{Y}_P(\nu)$ is easy: we can compute $Y_P^{new}(\nu) = Y_P^{old}(\nu) \cup [y(r) : y'(r)]$ by searching with $y(r)$ and $y'(r)$ in $\mathcal{Y}_P(\nu)$ and change $\mathcal{Y}_P(\nu)$ accordingly. The time that is needed is $O(\log n + i)$, where i the number of endpoints of intervals of $Y_P^{old}(\nu)$ that are contained in $[y(r) : y'(r)]$.

It remains to update the list $\mathcal{Y}_C(\nu)$. By definition we have $Y_C^{new}(\nu) = Y_C^{old}(\nu) \cup Y_C^{extra}(\nu)$, so let us see how to compute $Y_C^{extra}(\nu)$. Observe that the nodes ν in the top tree where $r \in R^{new}(\nu)$ form a subtree of $\mathcal{T}$, and that the leaves of this subtree are top leaves of $\mathcal{T}$. We update the lists $\mathcal{Y}_C(\nu)$ of these nodes in a bottom-up fashion, that is, we first update the nodes which are the leaves in this 'update tree' and then we work our way up.

Let ν be a leaf in the in the 'update tree'. If r does not have an endpoint inside $Cell(\nu)$ then life is easy: $Y_C^{extra}(\nu) = [y(r) : y'(r)] - Y_C^{old}(\nu)$. If r has an endpoint inside $Cell(\nu)$ things are much more difficult, but fortunately this can happen at only two of the leaves in the update tree. (Note that if we were updating a bottom tree, this case does not occur at all, since no cell in the partitioning contains a vertex in its interior.) We handle the case where a leaf of the update tree contains a vertex of r as follows. Let $R_V(\nu)$ be the set of rectangles that have a vertex inside $Cell(\nu)$, including rectangle r. From the properties of the partitioning it follows that $|R_V(\nu)| = O(\sqrt{n})$. Now compute the set $Y_V(\nu)$ of y-coordinates y^* such that $[x(\nu) : x'(\nu)] \times y^*$ is completely contained in $\bigcup R_V(\nu)$. $Y_V(\nu)$ consists of $O(\sqrt{n})$ intervals which can be computed in $O(\sqrt{n}\log n)$ time using a simple plane sweep algorithm. Finally, compute $Y_C^{new}(\nu) = Y_C^{old}(\nu) \cup Y_V(\nu)$ and update $\mathcal{Y}_C(\nu)$ accordingly. This takes $O(\sqrt{n}\log n + i)$ time, where i the number of endpoints of intervals of $Y_C^{old}(\nu)$ that are contained in $Y_V(\nu)$.

Finally, consider an internal node ν in the update tree. Again, the case where r does not have an endpoint inside $Cell(\nu)$ is easy. Now consider the case where r has an endpoint inside $Cell(\nu)$. Because we work bottom up, we may assume that we have $Y_C^{extra}(lc(\nu))$, $Y_C^{extra}(rc(\nu))$, $Y_C^{new}(lc(\nu))$ and $Y_C^{new}(rc(\nu))$ available. The list $\mathcal{Y}_C(\nu)$ is updated using the following lemma.

Lemma 3.9 *Let ν be an internal node in the update tree such that r has an endpoint inside* Cell(ν). *Then*

$$Y_C^{extra}(\nu) = \{Y_C^{extra}(\mathrm{lc}(\nu)) \cap Y_C^{new}(\mathrm{rc}(\nu))\} \cup \{Y_C^{extra}(\mathrm{rc}(\nu)) \cap Y_C^{new}(\mathrm{lc}(\nu))\}.$$

Proof: Note that every rectangle stored at ν is also stored at $lc(\nu)$ or it is disjoint from $Cell(lc(\nu))$. (This is true because we are at a node in the top tree; for nodes in the bottom tree this also holds unless $Cell(\nu)$ is completely covered, in which case there nothing to do.) Moreover, $R(lc(\nu)) \subset R(\nu)$. Analogous observations hold with respect to $rc(\nu)$. Now we can prove the lemma using elementary logic, as follows: From the above observations we have $Y_C(\nu) = Y_C(lc(\nu)) \cap Y_C(rc(\nu))$. It now follows that

$$Y_C^{extra}(\nu) \subseteq Y_C^{extra}(lc(\nu)) \cup Y_C^{extra}(rc(\nu))$$

and

$$Y_C^{extra}(\nu) \subseteq Y_C^{new}(\nu) = Y_C^{new}(lc(\nu)) \cap Y_C^{new}(rc(\nu)).$$

Hence,

$$Y_C^{extra}(\nu) \subseteq \{Y_C^{extra}(lc(\nu)) \cap Y_C^{new}(rc(\nu))\} \cup \{Y_C^{extra}(rc(\nu)) \cap Y_C^{new}(lc(\nu))\}.$$

The proof of "$\supseteq$" is equally simple so we leave it to the reader. □

We can now prove the following lemma.

Lemma 3.10 *Insertions into the structure described above can be performed in $O(\sqrt{n}\log^2 n)$ amortized time, assuming we know all rectangles to be inserted in advance.*

Proof: Consider the updating of the information in the top tree. We have already argued that the updating at the two leaves in the update tree whose cells contain a vertex of r takes $O(\sqrt{n}\log n)$ time. Let ν be one of the remaining nodes of the update tree. We prove the lemma for the updating of $\mathcal{Y}_C(\nu)$; similar arguments hold for $\mathcal{Y}_P(\nu)$. Let i_ν be the number of intervals in $Y_C^{extra}(\nu)$.

If r does not have an endpoint inside $Cell(\nu)$ then we can compute $Y_C^{extra}(\nu)$ in $O(\log n + i_\nu)$ time using binary search in $\mathcal{Y}_C(\nu)$.

Now assume that r has an endpoint inside $Cell(\nu)$. First, note that the number of intervals in $[Y_C^{extra}(lc(\nu)) \cap Y_C^{new}(rc(\nu))]$ and $[Y_C^{extra}(rc(\nu)) \cap Y_C^{new}(lc(\nu))]$ is $O(i_\nu)$. This implies that we can evaluate the expression of Lemma 3.9 in $O((1 + i_{lc(\nu)} + i_{rc(\nu)})\log n + i_\nu)$ time.

Observe that every endpoint of an interval in $Y_C(\nu)$ is the y-coordinate of a rectangle edge in $R(\nu)$. Moreover, there are no deletions, so once the endpoint induced by a rectangle disappears (that is, it is no longer an endpoint of an interval in $Y_C(\nu)$) it never reappears. To account for the extra time that we need on top of the $O(\log n)$ term, we can thus charge $O(\log n)$ time to $(i_{lc(\nu)} + i_{rc(\nu)} + i_\nu)$ rectangles in such a way that every rectangle gets charged no more than a constant number of times. Hence, the amortized time spent at node ν is $O(\log n)$.

For nodes in the bottom trees we can show in a similar way that we spend $O(\log n)$ amortized time for an update. (As remarked before, we do not have to consider the case of a bottom leaf whose cell contains a vertex of r.)

Recall that the number of nodes influenced by the insertion of rectangle r is $O(\sqrt{n}\log n)$ and that they can be found in the same amount of time. At every node, except for the (at most) two top leaves containing a vertex of r, we spend $O(\log n)$ amortized time. At the two top leaves we spend $O(\sqrt{n}\log n + i_\nu)$ time, which is $O(\sqrt{n}\log n)$ amortized. The lemma follows. □

3.2.4 Query polygons instead of query rectangles

In our application we need to be able to query with an axis-parallel polygon and not just with a rectangle. At first glance it may seem that one can simply decompose the query polygon into rectangles, but our query polygon is not guaranteed to have constant complexity so this approach fails: the sum of the complexities of the unions inside the rectangles can be much larger than the complexity of the union inside the polygon. See Figure 4, where each of the shaded rectangles of $\bigcup R$ is reported in many pieces. So we need a different approach. Let us first study how to avoid the

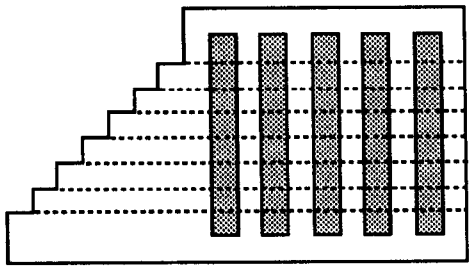

Figure 4: A decomposition giving a bad query time.

problem mentioned above at a bottom leaf γ. Let $\mathcal{P} \subset Cell(\gamma)$ be an axis-parallel query polygon, possibly with holes. We describe how to report the vertices of $\mathcal{P} \cap (\bigcup R(\gamma))$; it is straightforward to adapt the algorithm so that it reports the actual area as a collection of axis-parallel polygons.

We call an edge e of $\mathcal{P}$ a *bottom edge* if e is horizontal and $int(\mathcal{P})$ lies above e. Furthermore, we define $Hist(e)$ to be the maximal histogram inside $\mathcal{P}$ with e as its base. In other words, $Hist(e)$

is the set of points that can be connected to e with a vertical segment that does not cross the boundary of $\mathcal{P}$. The idea of the query algorithm is to find the vertices of $\mathcal{P} \cap (\bigcup R(\gamma))$ inside each $Hist(e)$ separately, in the following manner.

Let $e = [x(e) : x'(e)] \times y(e)$ be a bottom edge of $\mathcal{P}$. First, the lowest vertices of $\mathcal{P} \cap (\bigcup R(\gamma))$ inside $Hist(e)$ (the ones with minimum y-coordinate) are found. This is done as follows. Define y^* to be the upper endpoint of the interval of $Y_C(\gamma)$ containing $y(e)$ if $y(e) \in Y_C(\gamma)$, and define $y^* := y(e)$ otherwise. Next, compute $e^* = [x(e) : x'(e)] \times y^* \cap Hist(e)$. Note that e^* may consist of several segments. For each such segment $s = [x(s) : x'(s)] \times y^*$ we compute the endpoints x of intervals in $X_C(\gamma)$ such that $x \in [x(s) : x'(s)]$. The points (x, y^*) that have have found are the lowest vertices of $\bigcup R$ inside $Hist(e)$.

To find the other vertices inside $Hist(e)$ we 'walk' upward from each of these lowest vertices, reporting other vertices of $\bigcup R(\gamma)$, until we leave $Hist(e)$. More precisely, starting at each lowest vertex (x, y^*) we report the points (x, y) where y is an endpoint of an interval in $Y_C(\gamma)$ and $(x, y) \in Hist(e)$. See Figure 5 for an illustration. It is easy to check that this way all the vertices

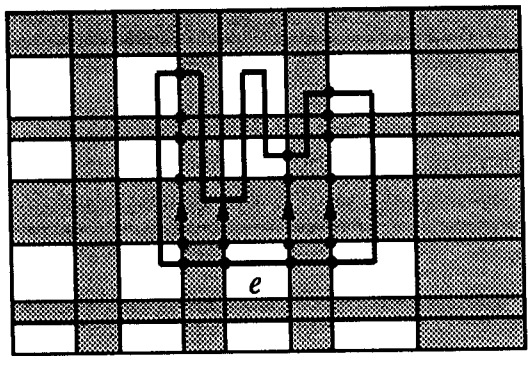

Figure 5: Reporting vertices inside a histogram.

of $\mathcal{P} \cap \bigcup R(\gamma)$ are correctly reported, except for some vertices that are on the boundary of $\mathcal{P}$. The latter type of vertices can be reported separately. Next we bound the time taken by the query algorithm.

Lemma 3.11 *The algorithm described above for reporting $\mathcal{P} \cap (\bigcup R(\gamma))$ at a bottom leaf γ can be implemented such that it runs in $O(|\mathcal{P}| \log n + k_\gamma)$ time, where $|\mathcal{P}|$ is the number of vertices of $\mathcal{P}$ and k_γ is the complexity of $\mathcal{P} \cap (\bigcup R(\gamma))$.*

Proof: The histograms $Hist(e)$ for the bottom edges e of $\mathcal{P}$ can be computed in $O(|\mathcal{P}|)$ time in total, by first computing the vertical adjacency map in $O(|\mathcal{P}|)$ time [3].

Let e be a bottom edge of $\mathcal{P}$. We will show that we spend $O(|Hist(e)| \log n + k_e)$ time to handle e, where k_e is the number of reported vertices inside $Hist(e)$, from which the lemma readily follows.

The computation of y^* can be performed in $O(\log n)$ time by a binary search in $\mathcal{Y}_C(\gamma)$, and after that we can compute e^* in $O(|Hist(e)|)$ time. Now let us see how much time we spent on a subsegment s of e^*. For each such segment s we can compute the endpoints x of intervals in $X_C(\gamma)$ such that $x \in [x(s) : x'(s)]$ in $O(\log n + l)$ time, where l is the number of reported endpoints, by searching in $\mathcal{X}_C(\gamma)$. Reporting the remaining points by walking upwards from these vertices can then done in constant time per reported point. (Recall that we already know the position of y^* in $\mathcal{Y}_C(\gamma)$.) Testing whether we leave $Hist(e)$ can be done in constant time, once we know the edge of $Hist(e)$ above (x, y^*). These edges can be computed for all (x, y^*) in advance, in $O(|Hist(e)| + k_\gamma)$ time in total. □

Now that we know how to treat the bottom leaves, let us consider a query in the full tree $\mathcal{T}$. A

query with query polygon $\mathcal{P}$ proceeds as follows. First, we decompose $\mathcal{P}$ into horizontal rectangles by computing the horizontal adjacency map. This can be done in linear time [3]. Next we start querying with the resulting rectangles. However, we do not treat them one at a time, but all simultaneously. If we arrive at a node ν in $\mathcal{T}$, and two rectangles q_i and q_j that share an edge both span $Cell(\nu)$, then we merge the two rectangles. In general, if there is a sequence of adjacent rectangles that all span $Cell(\nu)$, then we merge them into one rectangle. We then proceed as in the normal query algorithm for each of the resulting rectangles. When we arrive at a root node ν of a bottom tree, we collect all rectangles that are left at ν, form their union and compute a vertical decomposition of the union. We then continue in the same manner as in the top tree, that is, we merge rectangles whenever possible. When we finally arrive at some bottom leaf, we again collect all rectangles and we apply Lemma 3.2.

The correctness of the algorithm follows in the same way as in the case of query rectangles. To prove a bound on the query time we need the following lemma's, analogous to the ones used in the case of query rectangles.

Lemma 3.12 *The time spent at a node ν (not counting the time needed for the recursive calls) is $O(|\mathcal{P}_\nu| \log n + k_\nu)$, where $\mathcal{P}_\nu$ is the set of query rectangles reaching node ν and k_ν is the number of rectangles reported at ν.*

Proof: The merging that we have to perform at a node ν can easily be done in $O(|\mathcal{P}_\nu| \log n)$ time. (With some extra care it can even be implemented in $O(|\mathcal{P}_\nu|)$ time.) For a node which is not a bottom leaf it remains to apply the normal algorithm for each of the at most $|\mathcal{P}_\nu|$ resulting rectangles. By Lemma 3.4, this takes $O(\log n + l)$ time per rectangle, where l is the number of reported rectangles, summing up to $O(|\mathcal{P}_\nu| \log n + k_\nu)$ time in total. For a bottom leaf we spend the same amount of time, as has been shown in Lemma 3.11. □

Lemma 3.13 *Let $|\mathcal{P}_\nu|$ be the set of query rectangles considered at a node ν. Then we have $\sum_\nu |\mathcal{P}_\nu| = O(|\mathcal{P}| \log n + k_{rep} + k \log n)$, where k_{rep} is the total number of reported rectangles and k is the total complexity of $\mathcal{P} \cap (\bigcup R)$.*

Proof: The number of times we have a rectangle visiting a node ν where the rectangle has a vertex inside $Cell(\nu)$ is no more than $(|\mathcal{P}_\nu| \log n)$ in total. Furthermore, the number of visits where we report a rectangle is trivially no more than k_{rep}.

So let us count the remaining visits. Let ν be the visited node and let q be a query rectangle with which we visit ν.

First consider the case where q is involved in a merge with another rectangle q' at ν. This implies that q and q' cannot both span $Cell(parent(\nu))$ (otherwise they would have been merged earlier). Hence, we can charge this to $Cell(sibling(\nu))$ containing a vertex of $\mathcal{P}$.

If, on the other hand, q is not involved in a merge merge at ν, then we can use the same arguments as in the proof of Lemma 3.5: if we are in case (i), then we can charge this visit to the visit of $parent(\nu)$ by some rectangle $q' \supseteq q$, otherwise there must be a vertex of $\mathcal{P} \cap (\bigcup R)$ inside $Cell(\nu)$. □

Lemma's 3.6 and 3.7 (and their proofs) hold almost verbatim for a query polygon instead of a query rectangle. For the reader's convenience we state the equivalent of Lemma 3.7:

Lemma 3.14 *The total number of reported rectangles, k_{rep}, is $O(k \log n)$, where k is the total complexity of $\mathcal{P} \cap (\bigcup R)$.*

From these lemma's the total query time readily follows.

Lemma 3.15 *The query time for a union range query with an axis-parallel query polygon $\mathcal{P}$ in the structure described above is $O((|\mathcal{P}| + k) \log^2 n)$, where $|\mathcal{P}|$ is the number of vertices of $\mathcal{P}$ and k is the total complexity of $\mathcal{P} \cap (\bigcup R)$.*

4 Terrains

In this section we study the generalized hidden surface removal problem for the set S of faces of a polyhedral terrain. A polyhedral terrain is defined as the graph of a continuous, piecewise linear function $z = F(x, y)$ defined over the entire xy-plane. We assume that the faces of the terrain are convex.

4.1 The combinatorial bound

Recall that for a set of arbitrary triangles—and even for a set of axis-parallel rectangles—the maximum combinatorial complexity of $\mathcal{M}(S)$ is $\Theta(n^3)$. For terrains, however, the maximum complexity is considerably less, as we show next. We will find it sometimes convenient to drop the distinction between view point and light source, and use the terms 'visible from' or 'shadow cast by' for both.

Let e be an edge of the terrain. We hang a curtain from e, which we denote by $curt(e)$. A curtain is an unbounded polygon with three edges; one of these edges, the top edge, is the edge of the terrain and the other two edges, the vertical edges, are parallel to the z-axis and extend downward to minus infinity. Curtains have been used for ray shooting problems in terrains [4, 6]. For our complexity proof we imagine that the faces of the terrain themselves do not cast shadows, only the curtains do. Observe that this does not change the union of the shadows.

Observation 4.1 *Consider the p_{view}-projection of a curtain $curt(e)$ onto a plane h. Let s be a segment on h that intersects the p_{view}-projection of e and whose endpoints are not contained in the p_{view}-projection of $curt(e)$. Then s intersects the p_{view}-projection of a vertical edge of $curt(e)$. The same statement holds for p_{light}-projections.*

Now we are ready to prove a bound on the complexity of $\mathcal{M}(S)$.

Theorem 4.1 *Let S be the set of faces of a polyhedral terrain, and let n be the total number of edges of the terrain. The maximum combinatorial complexity of the generalized visibility map $\mathcal{M}(S)$ is $\Theta(n^2)$.*

Proof: The lower bound trivially follows from the fact that even the region visible from one point can have quadratic complexity.

We now prove the upper bound. Consider the set of vertices of $\mathcal{M}(S)$. Initially, we color all vertices red. Our upper bound proof proceeds by coloring $O(n^2)$ vertices (plus some additional points which may not be vertices of $\mathcal{M}(S)$) green in a way to be described next. Then we argue that the number of remaining red vertices is no more than $O(n^2)$.

Let f be a face of the terrain. Define $E_f(p_{view})$ to be the set of edges bounding the region on f visible from p_{view}, and define $E_f(p_{light})$ to be the set of edges bounding the region on f lit by p_{light}. Furthermore, define V_f to be the set of points which are either the endpoint of an edge in $E_f(p_{view})$ or $E_f(p_{light})$, or the intersection between an edge in $E_f(p_{view})$ and an edge in $E_f(p_{light})$. Clearly, the set of vertices of $\mathcal{M}(S)$ that are located on f is a subset of V_f. Let us color some of the vertices in V_f.

1. Color all the vertices of edges in $E_f(p_{view})$ and $E_f(p_{light})$ green.

2. For each vertex v of an edge a in $E_f(p_{view})$ such that v is not lit by p_{light} do the following. Move on a starting from v until an edge of $E_f(p_{light})$ is hit. If no edge is hit before we reach the other endpoint of a, then are ready. Otherwise we color the vertex of V_f where we hit the edge green.

3. Let $\pi_f(curt(e))$ be the p_{light}-projection of a curtain $curt(e)$ onto f. $\pi_f(curt(e))$ is a convex polygon whose vertices are either p_{light}-projections of vertices of $curt(e)$ onto f, or intersections of p_{light}-projections of one of the edges of $curt(e)$ with an edge of f, or vertices of f. For each vertex w of $\pi_f(curt(e))$ which is not a vertex of f and which is not visible from

p_{view}, we color at most four vertices in V_f green, as follows. Let a_1 and a_2 be the two edges of $\pi_f(curt(e))$ incident to w. Move on a_1, starting at w, until an edge of $E_f(p_{view})$ is hit. If no edge is hit before we reach the other endpoint of a_1, then we are done with a_1. So assume that we hit edge b of $E_f(p_{view})$. Start moving on b from point p to the right and to the left, and color the first points of V_f that we encounter on b green. Repeat this procedure for the other edge a_2 that is incident to w.

The number of green vertices can be estimated as follows. In the first two steps we color per face $O(|E_f(p_{view})| + |E_f(p_{light})|)$ vertices green. For all faces f this adds up to

$$\sum_f O(|E_f(p_{view})| + |E_f(p_{light})|) = O(n^2)$$

green vertices in total. In the last step we color $O(n)$ vertices per face f, which also adds up to $O(n^2)$ green vertices in total.

It remains to bound the number of vertices that are still red. Let v be such a vertex, let f be the face containing v. Because of the vertices colored in the first step, v must be the intersection between an edge a_1 in $E_f(p_{view})$ and an edge a_2 in $E_f(p_{light})$. Notice that a_1 is contained in the p_{view}-projection of some edge e_1 of the terrain, and that a_2 is contained in the p_{light}-projection of an edge e_2. Now imagine moving from v along a_1 into the shadow of p_{light}. Because of the vertices colored green in the second step, and because v is still red, we know that we must leave this shadow before we encounter an endpoint of a_1. Hence, the endpoints of a_1 do not lie in the p_{light}-projection of $curt(e_2)$. See Figure 6, where the the lower endpoint of a_1 does not lie in the (lightly shaded) p_{light}-shadow. Observation 4.1 now tells us that a_1 intersects the p_{light}-projection

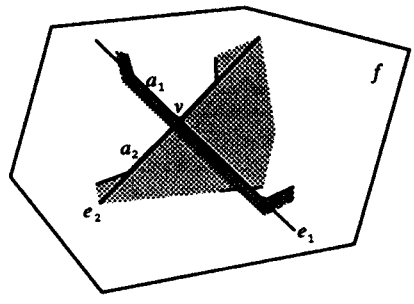

Figure 6: Situation for a red vertex.

s_2 of a vertical edge bounding $curt(e_2)$. Now imagine moving along s_2 from its intersection point with a_1 into the shadow of p_{view}. Because of the vertices colored in step three, we know that we must leave this shadow before we encounter an endpoint of s_2. Hence, the endpoints of s_2 do not lie inside the p_{view}-projection of $curt(e_2)$. Using Observation 4.1 once more, we conclude that s_2 intersects the p_{view}-projection s_1 of a vertical edge bounding $curt(e_1)$.

The upper bound now follows if we can show the following: the total number of intersections between the p_{view}-projections and the p_{light}-projections of the vertical segments that bound the curtains $curt(e)$, summed over all faces f of the terrain, is $O(n^2)$. This can be seen as follows. The p_{view}-projection (p_{light}-projection) of a vertical edge is contained in the vertical plane through p_{view} (p_{light}) and the edge. Hence, the intersection of the p_{view}-projection of a vertical edge and the p_{light}-projection of another vertical edge is contained in a vertical line. But any vertical line intersects only one face of the terrain, and so the projections can intersect on only one face. □

4.2 The algorithm

The algorithm that we use to compute $\mathcal{M}(S)$ is the same as we used in the axis-parallel case. In the first phase we compute which parts of the terrain are visible from p_{view}. In the second phase we treat the faces in order of increasing distance to p_{light}, maintaining the shadow of the faces, and querying with the visible parts on the faces to determine which pieces are lit by p_{light}. More precisely, the order that we use is an order on the vertical projections of the faces of the terrain with respect to the vertical projection of p_{light}. Note that it may be necessary to split the terrain with a vertical plane through p_{light} to ensure the existence of a depth order. The second phase is the same as the algorithm of [18], who solve the standard hidden surface removal problem, except that they can query with the faces themselves and are allowed to maintain the shadow explicitly.

This algorithm requires a data structure such that the part of the lower envelope of a set of line segments in the plane—the projections of edges of the terrain—below a query segment can be reported efficiently. (Note that these queries are actually union-range queries on the projections of the curtains hanging from the edges of the terrain, where the query polygon is also a curtain.) In the next subsubsection we will present a data structure that uses $O(n \log m + n\alpha(n/m))$ storage—where m is a parameter that can be chosen in the range $1 \cdots n$—such that the part of the upper envelope of E below a query segment can be computed in $O((m+l)\log n)$ time, where l is the size of the output of the query. Segments can be inserted into the structure in $O((n/m)\alpha(n/m) + \log n \log m)$ time. This leads to the following theorem.

Theorem 4.2 *Let S be the set of faces of a polyhedral terrain, and let n be the total number of edges of the terrain. The generalized visibility map $\mathcal{M}(S)$ can be computed in $O(n \log^2 n + n\sqrt{k\alpha(n)}\log n + k \log n)$ time, where k is the total complexity of the map.*

Proof: Let $k_{vis} \leqslant k$ be the complexity of the visibility map of the terrain, as seen from p_{view}. This map is computed in the first phase of the algorithm, taking $O((n\alpha(n) + k_{vis})\log n)$ time [13]. The time we spend in the second phase is dominated by the time we need to perform $O(n)$ insertions into the structure for maintaining the shadow, and $O(k_{vis})$ queries. This adds up to

$$O((n^2/m)\alpha(n/m) + n \log n \log m) + O((k_{vis}m + k)\log n)$$

for the second phase. By setting

$$m = \lceil n\sqrt{\alpha(n)/(k_{vis}\log n)}\rceil,$$

we obtain a running time for the algorithm as claimed. □

4.2.1 Implicitly maintaining the upper envelope

Let E be a set n segments in the plane, and let $\mathcal{E}(E)$ denote the upper envelope of E. We want to store E in a semi-dynamic data structure such that the part of $\mathcal{E}(E)$ below a query segment q can be computed efficiently. Let us first study the somewhat simpler problem, where we only want the intersection points between $\mathcal{E}(E)$ and q. Because the structural change in $\mathcal{E}(E)$ may be linear when we add a segment to E we have to maintain E implicitly. We do this by partitioning the problem into subproblems. The subproblems are solved using the following two simple lemma's.

Lemma 4.1 *The upper envelope of a set of n line segments can be stored in a structure that uses $O(n\alpha(n))$ storage, such that the l intersections of a query line segment with the envelope can be found in $O((l+1)\log(n/l))$ time. A segment can be inserted into the structure in $O(n\alpha(n))$ time.*

Proof: It is well known that the upper envelope of n line segments has complexity $O(n\alpha(n))$ [12]. Chazelle and Guibas [5] gave a structure for segment intersection queries in a simple polygon that uses linear storage and has $O((l+1)\log(n/l))$ query time. To insert a segment we first determine the new upper envelope in $O(n\alpha(n))$ time, by querying with the segment. Then we completely rebuild the structure, which can be done in time linear in the size of the envelope [11]. □

Lemma 4.2 *The upper envelope of a set of n lines can be stored in a structure that uses $O(n)$ storage, such that the at most two intersections of a query line segment with the envelope can be found in $O(\log n)$ time. A line can be inserted into the structure in $O(\log n)$ time.*

Proof: The upper envelope of a set of lines is the intersection of the positive half-planes bounded by the lines. Using dualization the problem becomes that of maintaining the convex hull of a set of points. Because we only have insertions, this problem is easily solved with a structure of linear size with $O(\log n)$ update and query time. □

Let m be a parameter, with $1 \leqslant m \leqslant n$. (Different choices of m will give us different trade-offs between storage and query time.) Let us first consider a fixed set E. Partition the plane into m vertical slabs $Slab_1, \ldots, Slab_m$, such that the interior of each slab contains at most n/m endpoints of the segments in E. Let $E_i \subset E$ be the set of segments that have an endpoint inside $Slab_i$. For each set E_i we explicitly maintain the upper envelope—more precisely, the portion of the envelope inside $Slab_i$—using the structure of Lemma 4.1.

Apart from the structures that we have for each set E_i, we also need to handle the line segments that completely cross a slab. This is done using a segment tree $\mathcal{T}$ built on the x-ranges of the slabs. Thus the leaves of $\mathcal{T}$ represent the slabs $Slab_i$, where the leftmost leaf represents the leftmost slab, and so on. An internal node ν of $\mathcal{T}$ represents the vertical slab $Slab(\nu)$ which is the union of the slabs represented by the leaves in the subtree rooted at ν. We associate with a node ν the set $E(\nu) \subset E$ of segments that span $Slab(\nu)$ but not $Slab(parent(\nu))$. The part of $\mathcal{E}(E(\nu))$ inside $Slab(\nu)$ is maintained using the structure of Lemma 4.2.

A query with segment q is performed by propagating q down in the tree $\mathcal{T}$ in the following manner. Suppose we are at a node ν in $\mathcal{T}$. We compute the part $q' \subset q$ that lies above $\mathcal{E}(E(\nu))$ and we continue the search in the two children of ν with the relevant part of q'. More precisely, if $q' \cap Slab(lc(\nu)) \neq \emptyset$ then we search in the left child $lc(\nu)$ with the segment $q' \cap Slab(lc(\nu))$, and if $q' \cap Slab(rc(\nu)) \neq \emptyset$ then we search in the right child $rc(\nu)$ with $q' \cap Slab(rc(\nu))$. This way we have computed for each slab $Slab_i$ the subsegment q_i of q that is above all the segments that completely cross $Slab_i$. It remains to compute and report the intersections of q_i with $\mathcal{E}(E_i)$.

The following lemma readily follows from Lemma's 4.1 and 4.2.

Lemma 4.3 *The structure described above uses $O(n \log m + n\alpha(n/m))$ storage. With this structure one can compute the l intersections of a query segment q with the upper envelope of E in $O(m \log n + l \log(n/m))$ time.*

It remains to study the dynamic behavior of the structure. We assume that we know all n segments that will be inserted into E in advance. Thus we can perform the partitioning into slabs and build a skeleton tree $\mathcal{T}$ in advance. This means that we do not have to worry about rebalancing when we insert a segment later on. Now suppose we want to insert a segment e. We have to update (at most) two structures storing the sets E_i that contain e, and $O(\log m)$ structures at nodes ν where $e \in E(\nu)$. By Lemma's 4.1 and 4.2 this takes $O((n/m)\alpha(n/m) + \log n \log m)$ time in total.

Lemma 4.4 *It is possible to insert a segment into the structure described above in $O((n/m)\alpha(n/m) + \log n \log m)$ time, assuming we know all segments to be inserted in advance.*

Combining these results we obtain the following theorem.

Theorem 4.3 *Let E be a set of n segments in the plane, and let m be a parameter between 1 and n. It is possible to store E into a data structure that uses $O(n \log m + n\alpha(n/m))$ storage, such that the l intersections of a query segment with the upper envelope of E can be computed in $O(m \log n + l \log(n/m))$ time. Segments can be inserted into the structure in $O((n/m)\alpha(n/m) + \log n \log m)$ time, assuming we know all segments to be inserted in advance.*

Recall that we actually want the part of $\mathcal{E}(E)$ below a query segment q, not just the intersections between $\mathcal{E}(E)$ and q. We now describe how to obtain this extra information.

Let $\overline{p_1p_2}$ be a maximal portion of q which is above $\mathcal{E}(E)$. Note that p_1 and p_2 are either endpoints of q or intersections of q with $\mathcal{E}(E)$. Let v_1 and v_2 be the points on $\mathcal{E}(E)$ whose x-coordinate is the same as the x-coordinate of p_1 and p_2, respectively. The idea of the algorithm is to 'walk' along $\mathcal{E}(E)$ from v_1 to v_2. We show how to compute the first vertex w of $\mathcal{E}(E)$ to the right of v_1. By repeating this procedure we can walk along $\mathcal{E}(E)$ until we reach v_2. Let $e \in E$ be the edge that contains v_1. Notice that w is either an endpoint of e or an intersection between e and another edge $e' \in E$. Define $\tilde{e}$ to be the part of e to the right of v. We perform a query with $\tilde{e}$ in our tree $\mathcal{T}$ in the way described above. However, we make sure that we find the rightmost intersection of $\tilde{e}$ with $\mathcal{E}(E)$ first[3], and then we stop. This way we can find the rightmost intersection in $O((s+1)\log n)$ time, where $s \leqslant m$ is the number of slabs crossed by $\tilde{e}$ in which there is no intersection. But we do not want to spend, say, $O(m \log n)$ time for every segment on the part of the upper envelope that we have to report. But fortunately, once we cross a slab we will not return to it again. Indeed, this not only holds if we consider reporting the part of the upper envelope below one 'visible' portion of q, but it holds if we consider all such portions. Hence, we have the following corollary.

Corollary 4.1 *Let E be a set of n segments in the plane, and let m be a parameter between 1 and n. It is possible to store E into a data structure that uses $O(n \log m + n\alpha(n/m))$ storage, such that the part of $\mathcal{E}(E)$ below a query segment q can be reported in $O((m+l)\log n)$ time, where l is the size of the output. Segments can be inserted into the structure in $O((n/m)\alpha(n/m) + \log n \log m)$ time, assuming we know all segments to be inserted in advance.*

5 Concluding remarks

We have generalized the classical hidden surface removal problem to take lighting considerations into account. More precisely, we have studied the generalized visibility map of sets of polyhedral objects with respect to a view point and one point light source, which is the subdivision of the viewing plane into maximal regions, such that in each region either no object is visible, or one object is visible and completely lit, or one object is visible and completely in shadow. We proved tight bounds on the maximum combinatorial complexity of such views and gave efficient output-sensitive algorithms to compute the views for several settings of the problem.

This has been an initial study of such problems any many questions are still open. For example, it might be possible to improve the time bounds that we have obtained—especially in the case of axis-parallel rectangles and in the case of terrains. One possible direction to improve these results would be to design more efficient data structures for union range queries on sets of rectangles or curtains in the plane. Furthermore, it would be interesting to generalize the problem even further, to allow more than one light source, or other light sources than point light sources.

Acknowledgment

I would like to thank the people who attended the Second Utrecht Workshop on Computational Geometry and its Application for discussions on this subject. Special thanks go the Mark Overmars and Micha Sharir for suggesting the approach taken in Section 4, and to Emo Welzl for discussing the data structure for maintaining the upper envelope of a set of line segments.

References

[1] P.K. Agarwal and J. Matoušek, Ray Shooting and Parametric Search, *Proc. 24th ACM Symp. on Theory of Computing*, 1992, pp. 517–526.

[3] Actually, we are looking for the first intersection with $\mathcal{E}(E - \{e\})$, but one easily sees that the fact that $e \in E$ does not impose any serious difficulties.

[2] M. Bern, Hidden Surface Removal for Rectangles, *J. Comp. Syst. Sciences* **40** (1990), pp. 49–69.

[3] B. Chazelle, Triangulating a Simple Polygon in Linear Time, *Discr. & Comp. Geometry* **6** (1991), pp. 485–524.

[4] B. Chazelle, H. Edelsbrunner, L.J. Guibas and M. Sharir, Lines in Space — Combinatorics, Algorithms and Applications, *Proc. 21st ACM Symp. on Theory of Computing*, 1989, pp. 382–393.

[5] B. Chazelle and L.J. Guibas, Visibility and intersection problems in plane geometry, *Discr. & Comp. Geometry* **4** (1989), pp. 551–581.

[6] M. de Berg, D. Halperin, M.H. Overmars, J. Snoeyink and M. van Kreveld, Efficient Ray Shooting and Hidden Surface Removal, *Proc. 7th ACM Symp. on Computational Geometry*, 1991, pp. 21–30. To appear in *Algorithmica*.

[7] M. de Berg and M.H. Overmars, Hidden Surface Removal for c–Oriented Polyhedra, *Computational Geometry: Theory and Applications* **1** (1992), pp. 247–268.

[8] H. Edelsbrunner, *Algorithms in Combinatorial Geometry*, Springer-Verlag, Berlin, 1987.

[9] M.T. Goodrich, M.J. Atallah and M.H. Overmars, An Input-Size/Output-Size Trade-Off in the Time-Complexity of Rectilinear Hidden Surface Removal, *Proc. 17th Int. Coll. on Automata, Languages and Programming*, Lecture Notes in Computer Science 443, 1990, pp. 689–702.

[10] R.H. Güting and T. Ottmann, New Algorithms for Special Cases of the Hidden Line Elimination Problem, *Comp. Vision, Graphics and Image Processing* **40** (1987), pp. 188–204.

[11] L.J. Guibas, J. Hershberger, D. Leven, M. Sharir and R. E. Tarjan, Linear-Time Algorithms for Visibility and Shortest Path Problems Inside Triangulated Simple Polygons, *Algorithmica* **2** (1987), pp. 209–233.

[12] S. Hart and M. Sharir, Non-Linearity of Davenport-Schinzel Sequences and of Generalized Path Compression Schemes, *Combinatorica* **6** (1986), pp. 151–177.

[13] M.J. Katz, M.H. Overmars and M. Sharir, Efficient Hidden Surface Removal for Objects with Small Union Size, *Computational Geometry: Theory and Applications* **2** (1992), pp. 223–234.

[14] H. Mairson and J. Stolfi, Reporting and counting intersections between two sets of line segments, *Theoretical Foundations of Computer Science and CAD*, R.A. Earnshaw (Ed.), NATO ASI Series, Vol. F-40, Springer Verlag, 1988, pp.307–326.

[15] M. McKenna, Worst-Case Optimal Hidden Surface Removal, *ACM Trans. Graphics* **6** (1987) pp. 19–28.

[16] M.H. Overmars, *personal communication.*

[17] M.H. Overmars and C.K. Yap, New Upper Bounds in Klee's Measure Problem, *SIAM J. Comput.* **20** (1991), pp. 1034–1045.

[18] J. Reif and S. Sen, An Efficient Output-Sensitive Hidden Surface Removal Algorithm and its Parallelization, *Proc. 4th ACM Symp. on Computational Geometry*, 1988, pp. 193–200.