Two- and Three-Dimensional Point Location in Rectangular Subdivisions*

Mark de Berg
Department of Computer Science
Utrecht University

Marc van Kreveld
Department of Computer Science
Utrecht University

Jack Snoeyink
Department of Computer Science
University of British Columbia

Abstract

We apply van Emde Boas-type stratified trees to point location problems in rectangular subdivisions in 2 and 3 dimensions. In a subdivision with n rectangles having integer coordinates from $[0, U-1]$, we locate an integer query point in $O((\log\log U)^d)$ query time using $O(n)$ space when $d \leq 2$ or $O(n \log\log U)$ space when $d = 3$. Applications and extensions of this "fixed universe" approach include spatial point location using logarithmic time and linear space in rectilinear subdivisions having arbitrary coordinates, point location in c-oriented polygons or fat triangles in the plane, point location in subdivisions of space into "fat prisms," and vertical ray shooting among horizontal "fat objects." Like other results on stratified trees, our algorithms run on a RAM model and make use of perfect hashing.

1 Introduction

The point location problem—which seeks to preprocess a set of disjoint geometric objects to be able to determine quickly which object contains a query point—is an important and well-studied problem in computational geometry. The usual goal of such study is logarithmic-time algorithms and linear-space structures, since this is the lower bound for one-dimensional search in a comparison-based model. In two dimensions, researchers have developed several solutions that attain these bounds; see Preparata [16] for a survey. In three dimensions, these bounds have not yet been attained, even though recent work on dynamic planar point location has lead to advances in spatial point location. Goodrich and Tamassia's [9] method, which achieves $O(\log^2 n)$ query time using $O(n \log n)$ space, is the current best.

We consider the special case of rectangular subdivisions. For our purposes, a *rectangle* in d dimensions is the Cartesian product of d intervals that are closed on the left and open on the right. A *rectangular subdivision* is a partition of a rectangle R into disjoint rectangles $R_1, R_2, \ldots, R_n$ whose union covers R; the *size* of this subdivision is n. The problem of point location in a subdivision is

*This research was supported by the ESPRIT Basic Research Action No. 3075 (project ALCOM). The first author was also supported by the Dutch Organization for Scientific Research (N. W. O.) and the last by an NSERC Research Grant.

to report the rectangle R_i that contains a query point $q \in R$. Edelsbrunner, Haring, and Hilbert [5] extended a planar point location method of Edelsbrunner and Maurer [6] to solve point location in a d-dimensional rectangular subdivision in $O(\log^{d-1} n)$ query time. Their algorithm handles arbitrary coordinates and runs on a pointer machine.

We use a stronger model of computation, the random access machine (RAM), to support the perfect hashing of Fredman, Komlós and Szemerédi [7]. Furthermore, for the first half of this paper, we require that the rectangle corners and query points lie in a fixed size integer grid $[0, U-1]^d$. Stratified trees, a data structure introduced by van Emde Boas [19] and extended by him and others [10, 14, 21, 22], exploit the power of a RAM on a fixed universe. They have been used for log-logarithmic time queries in one-dimensional point location, more commonly known as searching a list for the successor of a query point. Müller [15] used a type of stratified tree as a two-dimensional point location structure, to answer queries in a rectangular subdivision of size n using $O((\log\log U)^2)$ time and $O(n \log U)$ space.

We give a new type of stratified tree that emphasizes a tradeoff between space and query time. In two dimensions, we can preserve Müller's $O((\log\log U)^2)$ query time using only $O(n \log\log U)$ space or reduce space to linear and increase the query time to $O(\log^\epsilon U)$, where ϵ is a postive constant. We can also achieve $O((\log\log U)^2)$ query time with linear space by extending van Emde Boas' pruning technique [20] to two dimensions. In three dimensions, we can extend the point location method but not the pruning: We achieve $O((\log\log U)^3)$ query time using $O(n \log\log U)$ space or $O(\log^\epsilon U)$ time using linear space, for any constant $\epsilon > 0$. Section 2 describes the data structure and subsections 2.1, 2.2 and 2.3 describe integer point location in rectangular subdivisions of one-, two-, and three-dimensional integer grids.

In Section 3 we apply the point location method to other problems that are not initially defined on fixed integer grids. In subsection 3.1, we normalize a three-dimensional rectangular subdivision having arbitrary coordinates to allow point location using linear space and logarithmic time. Point location in k rectangular subdivisions of $d \leq 3$ dimensions that have total size n takes $O(\log n + k(\log\log n)^d)$ time. In subsection 3.2, we perform point location among c-oriented polygons or fat triangles in the plane in $O((\log\log n)^2)$ time after a constant number of normalizations. This allows point location in a subdivision of 3-space into c-oriented or fat prisms in logarithmic time and linear space. In subsection 3.3, we perform vertical ray shooting queries among n horizontal objects in 3-space using $O(\log n (\log\log n)^2)$ time. If the objects are rectangles or c-oriented triangles, the space is $O(n \log n)$; and if the objects are fat triangles, the space is $O(n \log n \log\log n)$.

2 Stratified trees and point location

In this section we describe our variant of stratified trees and show how they can be used to solve point location problems efficiently on a RAM. Conceptually, a stratified tree is an interval tree T built on the universe $[0, U-1]$ with a search tree built on the levels of T. The actual implementation depends upon perfect hashing to reduce storage space. First we describe how we think of stratified trees and then how they are implemented. We assume that the universe size U is a power of 2 and take all logarithms as base 2.

An *interval tree T on* $[0, U-1]$ is a complete binary tree that stores intervals of $[0, U-1]$—our definition will be slightly different from that in Edelsbrunner [4]. Number the levels of the interval tree T from the root, level 0, to the leaves, level $\log U$. Number the leaves of T from left to right with integers from 0 to $U-1$. With the jth leaf we associate the range $\rho(j) = [j-1/2, j+1/2]$; an internal node τ of T is associated with the range $\rho(\tau)$ that is the union of the ranges of the leaves of the subtree rooted at τ. From Figure 1 you can see that the set of ranges on level ℓ partition the integers of $[0, U-1]$ into 2^ℓ equal-size subsets.

An interval $I = [i_{\min}, i_{\max})$ with integer bounds from $[0, U-1]$ *spans* a tree node $\tau \in T$ if I contains the range $\rho(\tau)$. Interval I is *contained in* τ if I is contained in $\rho(\tau)$. Interval I *cuts* node τ if $\rho(\tau)$ contains exactly one of the endpoints of I. We can further distinguish whether *I cuts τ on the left*, meaning that I contains the lower bound of $\rho(\tau)$, or whether *I cuts τ on the right*, meaning that I contains the upper bound of $\rho(\tau)$.

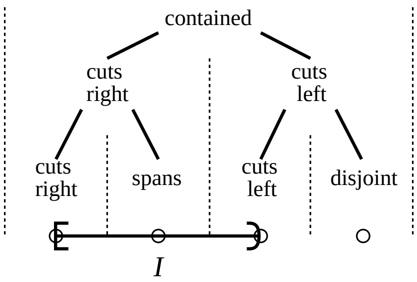

Figure 1: Interval tree T on $[0,3]$ and interval $I = [0,2)$

Let ℓ_I be the highest level at which interval I cuts a node of T; one can calculate ℓ_I as the most significant bit position at which $i_{\min}$ and $i_{\max}$ differ. As Figure 1 illustrates, the interval I is contained in the root and in one node per level down to level $\ell_I - 1$. From ℓ_I down to the leaves, I cuts two nodes per level, one on the right and one on the left, and spans any nodes in between.

We next form a *level-search tree* L, a balanced k-ary search tree on the levels of T. Figure 2 shows an interval tree with a ternary level-search tree. L is formed by assigning $k-1$ evenly spaced levels of T to the root L-node and recursively constructing k subtrees for the levels in between. (We refer to L-nodes of the level-search tree and T-nodes of the interval tree.) The height of L is $h = \Theta(\log\log U/\log k)$. For each L-node ν, define the T-depth(ν) as the number of the deepest level of T in the subtree rooted at ν.

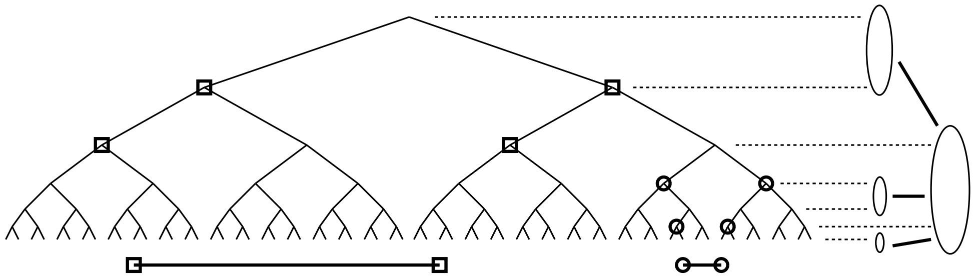

Figure 2: A stratified tree storing two intervals

To store an interval I in a van Emde Boas tree [19, 21], one would record the bounds of I in each T-node that I cuts. We use the level-search tree as a guide and store I only cut nodes certain levels of T as follows. Store interval I in level ℓ_I of T, and, for each L-node ν on the path in L from ℓ_I to the L-root, store I at level $1 + T$-depth(ν). Figure 2 illustrates the storage for two intervals. Notice that each interval is given to at most $2h$ T-nodes.

Recall that this was the conceptual view of stratified trees. If $n \ll U$, then most of the T-nodes do not receive intervals; to actually store these empty T-nodes would be wasteful. To implement stratified trees, we create only the level-search tree L and store only the *full* T-nodes of each level, that is, the T-nodes that receive at least one interval.

We label the T-nodes at level ℓ from 0 to $2^\ell - 1$; the label of the T-node that contains the integer $q \in [0, U-1]$ is the binary number represented by the first ℓ bits of q. We store the labels of full T-nodes and pointers to their intervals in a table using the perfect hashing scheme of Fredman, Komlós and Szemerédi [7]. (See also Mehlhorn and Näher [14].) This scheme stores m full nodes in $O(m)$ space and locates a stored node in $O(1)$ time. The deterministic preprocessing time is the minimum of $O(mU)$ and $O(m^3 \log U)$; the expected randomized preprocessing time is $O(m)$. Thus we have:

Theorem 2.1 *To store n intervals that partition $[0, U-1]$ in a stratified tree with a level-search tree of height h requires $O(nh + \log U)$ space and expected preprocessing time.*

Proof: The level-search tree structure takes $O(\log U)$ space, neglecting storage for associated levels. These levels store T-nodes containing $2hn$ intervals, thus, the maximum number of T-nodes and amount of storage is $O(nh)$ for all levels. The preprocessing is dominated by computing perfect hash tables; it is easy to assign intervals to levels and T-nodes in $O(nh)$ total time. ∎

Remark: The dynamic perfect hashing technique of Dietzfelbinger et al. [3] can be used to make these stratified trees dynamic. The amortized expected time to delete j intervals and replace them by k intervals that have the same union is $O((j+k)h)$. The space to store a level remains linear and the time to lookup whether a node is stored remains constant.

In the next subsections, we show how stratified trees answer point location queries in fixed universes.

2.1 One-dimensional point location

As a warm-up exercise for higher-dimensional point location, we show how to answer point location queries in one dimension using our variant of stratified trees. We prove Theorem 2.2.

Theorem 2.2 *Using a stratified tree on $[0, U-1]$ with a level-search tree of height $1 \le h \le \log\log U$, one can perform one-dimensional point location in an interval subdivision $I_1, I_2, \ldots, I_n$ using $O(nh)$ space and expected preprocessing time and $O(h(\log U)^{1/h})$ query time. By pruning, one can achieve $O(n)$ space and preprocessing time and $O(\log\log U)$ query time.*

When we have very few intervals, say $n = O(\log U)$, we punt the stratified trees and simply use a balanced binary search tree on the interval endpoints. This gives $O(\log\log U)$ query time using $O(n)$ space. Otherwise, we build a stratified tree using $O(nh)$ space according to Theorem 2.1.

Consider a stratified tree T-node τ and the (at most two) intervals it receives. If τ receives an interval I_j that cuts τ on the left (right), store I_j's upper (lower) bound. If τ receives no intervals, that is, if τ is empty, then some interval I_j spans $\rho(\tau)$.

Since an interval I_j is always given to two adjacent T-nodes at level ℓ_{I_j}, every point in I_j is in $I_j \cap \rho(\tau')$ for some full node τ'.

Suppose, just for one paragraph, that we had given each interval I_j to every T-node that it cuts—this is precisely what is done in forming a van Emde Boas tree [19, 21]. We could then determine if the interval containing a query integer q was stored above or below a level ℓ of the interval tree T by the following procedure: Take the label of q, which is the number determined by the first ℓ bits of q, and, by hashing in constant time, determine if the node $\tau \in T$ with that label is empty or is stored at level ℓ. If τ is empty, then q is inside an interval I_j that spans τ and, therefore, also spans τ's descendents—we need not search deeper in the interval tree. Otherwise, test q against the intervals cutting τ on the left and right. If either interval contains q, stop and report it; otherwise q is inside an interval I_j contained in τ and, therefore, also contained in τ's ancestors—we need not search higher in the tree T.

We have not given each interval to every cut T-node, however, so we must remember intervals that we have seen as we traverse the level-search tree L. To answer a point location query for an integer q, we begin at the root of L and set the *search interval* $\mathcal{I} = [i_{\min} = 0, i_{\max} = U - 1]$.

With an L-node ν are associated $k-1$ levels of T, numbered $\ell_1, \ell_2, \ldots, \ell_{k-1}$. When the search reaches ν, we use the hash table for each level ℓ_j to determine if the T-node τ_j, whose label is the first ℓ_j bits of q, is empty or is present and stores one or two intervals. If some stored interval contains q, then we stop and report that interval. Otherwise, we must use $\mathcal{I}$ to decide in which child of the L-node ν to continue the search.

Let $i_{\max}$ be the minimum of $i_{\max}$ and all stored interval boundaries greater than q; let $i_{\min}$ be the maximum of $i_{\min}$ and all stored interval boundaries less than q. Thus, the search interval $\mathcal{I} = [\rangle_{\min}, \rangle_{\max}]$ does not intersect any of the stored intervals, but does contain the interval that contains q. We continue the search in the child of ν whose subtree contains level $\ell_{\mathcal{I}}$. We find this child by calculating the node numbers for $i_{\min}$ and $i_{\max}$ in levels $\ell_1, \ell_2, \ldots, \ell_{k-1}$. If $i_{\min}$ and $i_{\max}$ first fall in the same node at level ℓ_j, then we search recursively in the child below ℓ_j—that is, the first child ν' with T-depth$(\nu') < \ell_j$. If $i_{\min}$ and $i_{\max}$ fall into different nodes at each level, then we search the highest child of ν. Lemma 2.3 proves the correctness of this procedure.

Lemma 2.3 *Let ν be a node of the level-search tree L. Let $\mathcal{I}$ be the largest interval containing the query q that is disjoint from all intervals found in levels associated with L-nodes on the path to and including ν. The interval I_j that contains q can be found in the subtree of the highest child of ν below all associated levels having a T-node that contains $\mathcal{I}$.*

Proof: Think of adding the root and leaf levels to those associated with ν; then we can find two levels ℓ and ℓ', with one child between them, such that $\mathcal{I}$ is contained in a T-node τ at level ℓ and not contained in a T-node at level ℓ'.

We know that any interval stored in the stratified tree that intersects $\mathcal{I}$ is contained in $\mathcal{I}$—including the interval I_j that contains q. Since $\mathcal{I}$ is contained in τ, we need not search higher than level ℓ.

Now, consider the T-node $\tau' \in T$ that contains q at level ℓ'. The interval I_j either cuts or spans τ'—we shall prove that it spans τ'. Suppose, instead, that I_j cuts τ. Then $\mathcal{I}$ contains some interval I' that is stored in the stratified tree and cuts τ. But the highest level node that I' cuts must then be between ℓ and ℓ', so I' would be stored in node τ and would be found to contain q or to shorten the interval $\mathcal{I}$. Thus, I_j spans τ and also spans all descendents of τ—we need not search lower than ℓ'. This establishes the lemma. ∎

The proof of Theorem 2.2 is almost complete. For each of the h levels of the level-search tree, a query examines $k-1$ levels of the interval tree in constant time apiece. Query time is $O(hk)$ and space is $O(nh)$, where $h = \log\log U/\log k$. Varying the height parameter h gives a space/query time tradeoff: Choosing k a constant gives $O(\log\log U)$ query time and $O(n \log\log U)$ space. Choosing h a constant gives $O((\log U)^{1/h})$ query time and $O(n)$ space.

The tradeoff afforded by h is unnecessary for one-dimensional point location. Instead, one can use van Emde Boas' technique of *pruning* [20] to reduce the space to linear and increase the query time by only a constant factor. Choose every $\log\log U$th interval boundary to form $n/\log\log U$ super-intervals and store these super-intervals in a stratified tree using $O(n)$ space. Given a query, find the containing super-interval in the stratified tree, then use linear search to find the actual interval. The query time remains $O(\log\log U)$. This completes the proof of Theorem 2.2.

Because the union of rectangles is not a rectangle, pruning in higher dimensions is more difficult. At the end of the next section, we use the planar separator theorem to show that pruning is still possible in two dimensions. For three dimensions, however, we need the space/time tradeoff to attain linear space.

2.2 Two-dimensional point location

In this section, we show how to perform point location in a rectangular subdivision of the plane by using two layers of stratified trees. Theorem 2.4 improves a theorem of Müller [15].

Theorem 2.4 *Using stratified trees on $[0, U-1]^2$, one can perform 2-dimensional point location in a rectangular subdivision $R_1, R_2, \ldots, R_n$ in $O(h(\log U)^{1/h} \log\log U)$ query time using $O(nh)$ space and expected preprocessing time, for any integer $1 \le h \le \log\log U$. By pruning, one can achieve $O(n)$ space after $O(n \log n)$ deterministic and $O(n)$ expected preprocessing time.*

To perform point location in a rectangular subdivision of two dimensions, we form a stratified tree on the intervals of the x-axis in much the same way as in the previous section. We give each rectangle to the highest level nodes that its x-interval cuts and to the nodes cut on all other levels directly below the the path to the root of the level-search tree.

Consider a node τ in the interval tree T: it has range $\rho(\tau) = [x_{\min}, x_{\max}]$ and receives a set of rectangles $\mathcal{R}$ that cut it. (See Figure 3.) The intersection of the line $x = x_{\min}$ or $x = x_{\max}$ with $\mathcal{R}$ is a set of intervals—it is not a subdivision because there are gaps left by rectangles that span τ or that are stored elsewhere in the stratified tree. If we fill in these gaps, however, we can use one-dimensional point location to find the projection of a query point q onto the the lines $x = x_{\min}$ and $x = x_{\max}$. Since filling in the gaps at most doubles the number of intervals, we can locate both projections in $O(\log\log U)$ time using space proportional to the number of rectangles received by τ. This proves that the total space required is $O(nh)$.

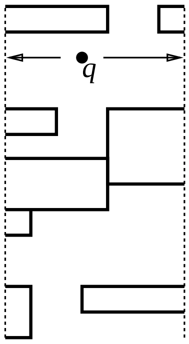

Figure 3: Rectangles given to τ

If the projection of q lies in a y-interval of a rectangle $R \in \mathcal{R}$, then we can check in constant time if q also lies in the x-interval of R. Thus, to locate the rectangle containing a query point q, we begin at the root of the level-search tree and set the line segment $\mathcal{I}$ to the portion of the horizontal line through q with x coordinates in $[0, U-1]$. At a level-search tree node ν, we use hashing to obtain the node containing q at each of the $k-1$ levels associated with ν and use one-dimensional point location to check for rectangles containing q. If none is found, we shrink the segment $\mathcal{I}$ to lie between the closest rectangles intersecting $\mathcal{I}$ to the right and left of q. We continue the search in the child of ν directly below the lowest associated level that has a node containing $\mathcal{I}$. Again, Lemma 2.3 proves the correctness of this procedure.

For each of the h levels of the level-search tree, a query examines $k-1$ levels of the tree in $O(\log\log U)$ time apiece. Thus, query time is $O(hk\log\log U)$, where $h = \log\log U/\log k$. Except for the pruning, this establishes Theorem 2.4. Choosing k a constant gives a $O((\log\log U)^2)$ query time algorithm using $O(n\log\log U)$ space. Choosing h a constant gives a $O((\log U)^{1/h}\log\log U)$ algorithm with linear space.

Remark: If we use dynamic perfect hashing [3] in the primary structure, then an operation that replaces j rectangles by k rectangles that have the same union induces $O((j+k)h)$ changes in 1-dimensional structures. Each of these takes $O(1)$ expected amortized time, if dynamic, pruned stratified trees are used for the 1-dimensional subproblems. Thus, the entire space/query time tradeoff can be achieved. Unfortunately, the 2-dimensional pruning described below cannot be used in a dynamic setting.

For the remainder of this section, we develop a two-dimensional analogue of van Emde Boas' pruning technique to reduce the space to linear while increasing the query time by only a constant factor. Specifically, we prove that we can collect the rectangles $R_1, R_2, \ldots, R_n$ into groups of size $O((\log\log U)^2)$ and cover these groups by a new subdivision into $m = O(n/\log\log U)$ rectangles ${R'}_1, {R'}_2, \ldots, {R'}_m$ such that every new rectangle R'_i intersects only one group. We can store the new rectangles ${R'}_1, {R'}_2, \ldots, {R'}_m$ in a point location structure that uses $O(n)$ space and find the rectangle R'_i containing a query point in $O((\log\log U)^2)$ time. Rectangle R'_i tells us a unique group of $O((\log\log U)^2)$ rectangles that we can search exhaustively. Thus, the pruning part of Theorem 2.4 will be established when we prove Lemma 2.5.

Lemma 2.5 *Let rectangles $R_1, R_2, \ldots, R_n$ form a rectangular subdivision. In linear time, one can partition these n rectangles into sets $\mathcal{R}_j$ of size $O((\log\log U)^2)$ and find a new subdivision $R'_1, R'_2, \ldots, R'_m$ of at most $m = O(n/\log\log U)$ rectangles such that each new rectangle R'_i intersects the rectangles of only one set $\mathcal{R}_j$.*

Proof: Lipton and Tarjan's planar separator Theorem [12] states that any planar graph on v vertices has a subset of $2\sqrt{2v}$ vertices whose removal separates the graph into components with at most $2v/3$ vertices each.

The dual graph of the rectangular subdivision $R_1, R_2, \ldots, R_n$—the graph whose vertices are rectangles and whose edges join rectangles that share a portion of a boundary—is planar. While a connected component of this graph has $v > (\log\log U)^2$ vertices (rectangles), we apply the planar separator theorem to remove a small set of vertices so that no component remaining has more than $2v/3$ vertices. Using Goodrich's algorithm [8] we can compute all these planar separators in linear time.

When the algorithm terminates, we collect all rectangles whose corresponding dual vertices are removed into a set C. Removing the rectangles of C from the subdivision leaves connected groups of size at most $(\log\log U)^2$. For each group, We take the union and decompose it into rectangles by computing its *vertical adjacency map*—making vertical cuts through the reflex vertices of the boundary of the union. This can be done by a simple sweep if the boundary vertices are sorted. Let D be the set of all rectangles formed in this manner. Figure 4 illustrates a small rectangular subdivision and a decomposition into rectangles of C and D.

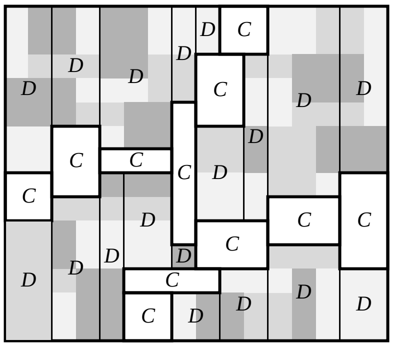

Figure 4: The rectangles of C and D

We let $C \cup D$ be our new subdivision. Because each rectangle of D lies inside a union of at most $(\log\log U)^2$ rectangles, it is clear that a new rectangle intersects at most $(\log\log U)^2$ old rectangles. We must show that the number of rectangles $|C \cup D|$ is $O(n/\log\log U)$. It is sufficient to bound $|C|$ since the number of rectangles in D is proportional to the boundary complexity of the union of connected components, which is proportional to $|C|$.

Following Lipton and Tarjan [12], we form a tree on the set of all components constructed by the algorithm: a component that splits is the parent of the resulting components. We number each component in the tree by the greatest distance to a leaf. Leaves, components that are not split, are numbered 0.

The components numbered 1 are split once, thus they have size at least $(\log\log U)^2$. By induction, the components numbered i each have size at least $(3/2)^{i-1}(\log\log U)^2$. Since a vertex is in at most one component with a given number, there are $m_i \le n(2/3)^{i-1}/(\log\log U)^2$ components numbered i.

Let us look at the contribution to C from the m_i components that are numbered i. If n_j is the size of the jth component numbered i, then the contribution to C is $\sum_{1\le j\le m_i} 2\sqrt{2n_j}$. Because the sum $\sum_{1\le j\le m_i} n_j \le n$, the Cauchy-Schwarz inequality states that the maximum contribution occurs when the components have equal sizes; the contribution is at most $m_i 2\sqrt{2n/m_i} = 2\sqrt{2nm_i}$. Summing over all positive component numbers gives the maximum total contribution:

$$\sum_{i\geq 1} 2\sqrt{2nm_i} \leq \sum_{i\geq 1} 2\sqrt{2}\left(\frac{2}{3}\right)^{(i-1)/2} \frac{n}{\log\log U} = O(n/\log\log U).$$

This completes the proofs of Lemma 2.5 and of Theorem 2.4. ∎

Remark: On integer grids, this data structure can improve many algorithms that use point location as a subroutine. For a simple example, reporting the k horizontal segments that intersect a vertical query segment can be performed in $O((\log\log n)^2 + k)$ time by preprocessing Chazelle's hive graph [2].

2.3 Three-dimensional point location

By now, the method for three-dimensional point location should come as no surprise: use a stratified tree with two-dimensional point location as secondary structures at each node. That the method breaks down in four dimensions may come as more of a surprise. We merely outline the proof of Theorem 2.6.

Theorem 2.6 *Using stratified trees on* $[0, U-1]$*, one can perform 3-dimensional point location in a rectangular subdivision* $R_1, R_2, \ldots, R_n$ *in* $O(h(\log U)^{1/h}(\log\log U)^2)$ *query time using* $O(nh)$ *space, for any integer* $1 \leq h \leq \log\log U$*.*

Proof: Again, we form a stratified tree on the intervals of the x-axis and give each rectangular box to the highest level nodes that its x-interval cuts and to the nodes cut on all other levels directly below the path to the root of the level-search tree. Let $\mathcal{R}$ be the set of boxes that cut a node $\tau \in T$ on the right (the set cutting τ on the left is handled similarly). All boxes of $\mathcal{R}$ intersect the plane through the right boundary of the interval $\rho(\tau)$, forming a set of 2-dimensional rectangles. We extend this to a rectangular subdivision of the plane by forming the vertical adjacency map—making vertical cuts from the corners of rectangles of $\mathcal{R}$. This process of "filling the gaps" in the plane increases the number of rectangles by a constant factor. We can then use two-dimensional point location to find the projection of a query point q onto the planes bounding $\rho(\tau)$ in $O((\log\log U)^2)$ time and linear space. This proves that the total space required is $O(nh)$.

If the projection of q lies in a 2-dimensional rectangle, then we can check in constant time if q lies in the 3-dimensional rectangular box that created it. Thus, to locate the rectangular box containing a query q, we begin at the root of the level-search tree and set the line segment $\mathcal{I}$ to the portion of the line through q and parallel to the x-axis that has x-coordinates in $[0, U-1]$. At a level-search tree node ν, we use hashing to obtain the node containing q at each of the $k-1$ levels associated with ν and use two-dimensional point location to check for boxes containing q. If none does, we shrink the segment $\mathcal{I}$ to lie between the closest boxes intersecting $\mathcal{I}$ to the right and left of q. We continue the search in the child of ν directly below the lowest associated level that has a node containing $\mathcal{I}$. Lemma 2.3 proves the correctness of this procedure. The analysis of running time is the same as in the previous section except for substituting two-dimensional point location with pruning as the secondary structure. This establishes Theorem 2.6. ∎

Varying the height parameter h gives a space/query time tradeoff: Choosing h a constant gives linear space and $O((\log U)^{1/h}(\log\log U)^2)$ query time. Choosing $h = \lfloor \log\log U \rfloor$ gives $O((\log\log U)^3)$ query time and $O(n \log\log U)$ space.

This method cannot be extended to higher dimensions because we can no longer "fill the gaps" using linear space. There are sets of m boxes in 3-dimensions that are contained only in rectangular subdivisions of size $\Omega(m^{3/2})$.

3 Application to other domains

The previous section developed data structures for point location problems in fixed universes; the problem domains were fixed size integer grids. In this section, we look at some easy applications of these data structures to problem domains that are not (initially) fixed grids. These include a logarithmic-time linear-space point location structure for rectangular subdivisions of three dimensions using arbitrary coordinates, locating a single point in several subdivisions, point location among c-oriented polygons or fat triangles in the plane and among prisms with c-oriented or fat bases, and vertical ray shooting among horizontal rectangles or c-oriented or fat triangles. Our approach is two-pronged: First, to extract one or more grids from a problem and preprocess them for point location. Second, to normalize a query point to these grids by binary search and then perform point locations. We gain by having only one search for the normalization of a query point.

3.1 Point location in rectangular subdivisions of real 3-space

We begin with two simple examples of normalization. The first is the problem that motivated this research.

Theorem 3.1 *Given a three-dimensional rectangular subdivision of size n having arbitrary real coordinates, we can answer point location queries in $O(\log n)$ time on a RAM using linear space and $O(n \log n)$ expected preprocessing time.*

Proof: Every rectangular box is a product of intervals from the three coordinate directions. For each of the three coordinate directions, form a sorted and ranked list of the bounds of these intervals and replace each real interval $[a, b)$ with the integer interval $[rank(a), rank(b))$. This takes $O(n \log n)$ preprocessing.

The rectangular subdivision can now be considered as a subdivision of the integer grid whose maximum coordinate is n. We can therefore apply Theorem 2.6 and compute a linear space point location structure that reports the rectangle containing a query grid point in $O(\log n)$ time. (If the height of level-search trees in all dimensions are taken to be $h = 3$, for example, then these bounds are attained without pruning and the expected time to build the data structures is $O(n)$.)

To answer a query for a real point q, we normalize q to a grid point: we replace each coordinate of q by its rank, which we determine by a binary search in the list of bounds for that coordinate. Then we report the box that contains this grid point. Both normalization and grid-point location take $O(\log n)$ time. ∎

A related (and trivial) example deals with locating a point in several subdivisions of total size n. All bounds of rectangle intervals from a given axis can be collected into one sorted and ranked list and all normalizations can performed on this list.

Theorem 3.2 *A query point can be located in k rectangular subdivisions of $d \leq 3$ dimensions that have total size n in $O(\log n + k(\log\log n)^d)$ time.*

Remark: By collecting the planar subproblems that arise in the skewer trees of Edelsbrunner, Haring, and Hilbert [5] and applying Theorem 3.2, we obtain an $O((\log n)^{d-2}(\log\log n)^2)$ point location method for rectangular subdivisions of $d > 2$ dimensions.

3.2 Point location among c-oriented polygons and fat triangles

In this section we explore a method to extend the rectangular point location scheme to a set $\mathcal{P}$ of polygons in the plane that have disjoint interiors. Suppose we can find a rectangular subdivision of the plane such that each rectangle in the subdivision intersects only a constant number of polygons of $\mathcal{P}$; we call such a subdivision a *sparse rectangularization* $SR(\mathcal{P})$ of the set $\mathcal{P}$ of polygons. Then we could answer a point location in $\mathcal{P}$ by locating the query point q in $SR(\mathcal{P})$ and comparing q to the polygons of $\mathcal{P}$ that intersect the rectangle containing q.

Not every set $\mathcal{P}$ admits a sparse rectangularization: if, for example, the set contains a vertex with more than a constant number of incident polygons then no sparse rectangularization of $\mathcal{P}$ exists. Therefore we study the restricted class of c-oriented polygons; polygons whose edges are parallel to a fixed set of c orientations, for some constant c. Furthermore, rather than looking for a single sparse rectangularization, we partition a set of c-oriented polygons into a constant number of sets of quadrilaterals, each of which admits a sparse rectangularization. The size of all rectangularizations (the total number of rectangles) will be linear in the number of polygon edges. Different rectangularizations will use different orientations for their axes; a query point must be normalized in several new orientations. As we have seen in Section 3.1, however, applications that perform several point locations gain by performing the normalizations only once.

Let $\mathcal{P}$ be a set of c-oriented polygons with disjoint interiors and assume, without loss of generality, that one of the c possible orientations is parallel to the x-axis. To obtain a family of sparse rectangularizations, we decompose each polygon P of $\mathcal{P}$ into trapezoids (some of which can degenerate into triangles) by slicing through each vertex of P with the longest horizontal segment that is contained in P. Each trapezoid has (one or) two edges that are horizontal, which we call its top and bottom edges, and a left and a right edge. Since $\mathcal{P}$ is c-oriented, we can partition the resulting set of trapezoids into $c-1$ subsets $\mathcal{P}_1, \ldots, \mathcal{P}_{c-1}$ according to the orientation of their left edge.

Consider one subset $\mathcal{P}_i$. Compute the horizontal trapezoidation of the left edges of trapezoids in $\mathcal{P}_i$, as shown in Figure 5. The planar subdivision thus obtained is called the horizontal adjacency map of the left edges of $\mathcal{P}_i$, denoted $ADJ(\mathcal{P}_i)$, because the endpoints of each left edge are connected to the horizontally adjacent left edges. If we apply a skew transformation to make the left edges vertical, then the horizontal adjacency map is a rectangularization of $\mathcal{P}_i$. The next lemma proves that it is a sparse rectangularization.

Lemma 3.3 $\mathrm{ADJ}(\mathcal{P}_i)$ *is a sparse rectangularization of $\mathcal{P}_i$ with linear size.*

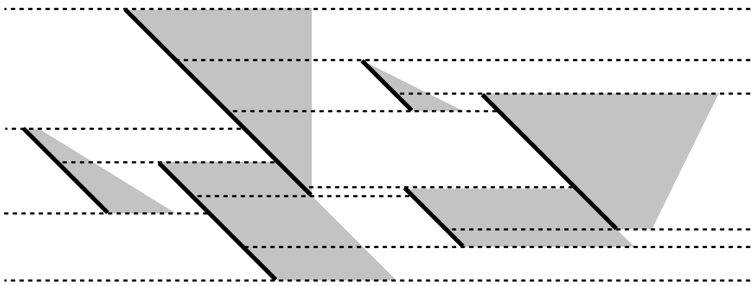

Figure 5: The horizontal adjacency map for the left edges of the shaded trapezoids

Proof: Since the numbers of trapezoids, edges, and vertices in $ADJ(\mathcal{P}_i)$ are proportional to the number of left edges in $\mathcal{P}_i$, the horizontal adjacency map $ADJ(\mathcal{P}_i)$ is a rectangularization of linear size.

To prove the sparseness of $ADJ(\mathcal{P}_i)$, we show that each rectangle $R \in ADJ(\mathcal{P}_i)$ intersects at most one trapezoid of $\mathcal{P}_i$. By construction, the only vertices of trapezoids in $\mathcal{P}_i$ that touch R lie on the horizontal edges of R. Thus, there is at most one trapezoid whose left edge is the left boundary of R. Since rectangle R is not considered to include its right boundary, any trapezoid that extends to the right from this boundary does not intersect R. ∎

Lemma 3.3 enables us to perform fast point location in $\mathcal{P}_i$: first we perform fast point location in $ADJ(\mathcal{P}_i)$ and then we test in constant time whether the query point is inside the trapezoid that intersects the rectangle of $ADJ(\mathcal{P}_i)$ that contains the query point. Observe that the query point needs to be normalized in both the y direction and in the direction perpendicular to the left edges of $\mathcal{P}_i$ in order to perform fast point location in $ADJ(\mathcal{P}_i)$. Applying this scheme to each $\mathcal{P}_i$ leads to the following theorem.

Theorem 3.4 *After c normalizations, one can perform 2-dimensional point location among disjoint c-oriented polygons with n vertices in $O((\log\log n)^2)$ query time using $O(n)$ space and $O(n\log n)$ expected preprocessing time.*

We can obtain the same result for fat triangles, where a triangle is *fat* if every internal angle contains at least one of a set of c fixed orientations. The more common definition [13], that each internal angle to be greater than some fixed angle θ, is equivalent up to a multiplicative constant.

Corollary 3.5 *After c normalizations, one can perform 2-dimensional point location among n disjoint fat triangles in $O((\log\log n)^2)$ query time using $O(n)$ space and $O(n\log n)$ expected preprocessing time.*

Proof: Given a fat triangle, first cut it into two triangles by a c-oriented cut through a vertex, then make c-oriented cuts through the remaining two original vertices. This results in four triangles t_1, t_2, t_3 and t_4 such that each t_i has two c-oriented edges. Once all triangles have been cut, group them into $c(c-1)$ classes based on the orientations of the c-oriented edges. For each class, apply a skew transformation so that the two c-oriented edges are horizontal and vertical, then compute a vertical adjacency map. A rectangle in this adjacency map intersects at most two triangles, so the sparse rectangularization method can be used for point location. ∎

One way to extend these algorithms to three dimensions is to consider subdivisions into prisms whose bases are parallel c-oriented polygons or fat triangles. The next theorem is a generalization of Theorem 3.1.

Theorem 3.6 *In a subdivision of 3-space into prisms whose bases are parallel c-oriented polygons, with constant c, or parallel fat triangles we can perform point location in $O(\log n)$ time using linear space and $O(n \log n)$ expected preprocessing time.*

Proof: Transform the problem so the prism axes are parallel to the z-axis and the bases are parallel to the xy-plane. Then, sort and rank the coordinates of prism vertices along the z-axis and perpendicular to the c orientations in the xy-plane. We can use the ranks as integer coordinates for the vertices. We can also normalize a query point in $O(\log n)$ time by a constant number of binary searches.

Form a stratified tree, with height parameter $h = 2$, on the z-direction as in Section 2.3. The secondary structures of this stratified tree are point location structures for c-oriented polygons or fat triangles that were developed in this section. Point location in this structure takes $o(\log n)$ time after normalization.

Since each prism base is stored in at most two secondary structures, the entire structure takes linear space and can be built in $O(n \log n)$ expected time. ∎

3.3 Vertical ray shooting queries

The problem of vertical ray shooting among horizontal objects in space can be seen as a generalization of 3-dimensional point location in a subdivision. In this section we apply the fast point location technique to speed up vertical ray shooting queries among horizontal rectangles or horizontal c-oriented or fat triangles. Because one normalization can serve for several point locations, we can improve query times from $O(\log^2 n)$ to $O(\log n (\log \log n)^2)$ for these problems. Similar improvements are possible for any structure that uses a (rectangular or c-oriented) point location structure as an associated structure. Preparata, Vitter, and Yvinec [18] have used this observation and stratified trees to speed up their hidden surface routines for parallel rectangles [17].

The problem we study is this: Let S be a set of horizontal objects (parallel to the xy-plane) in 3-space. We want to preprocess S such that the first object hit by a ray directed vertically downward (parallel to the z-axis) can be determined efficiently. First we consider horizontal axis-parallel rectangles and later horizontal c-oriented polygons and fat triangles. Because we use only standard data structures such as binary search trees and segment trees, we keep the description of our structures short and leave the details to the reader.

Let $S = \{R_1, \ldots, R_n\}$ be a set of n horizontal axis-parallel rectangles, and assume that we have sorted and ranked the y- and z-coordinates so that they are integers. For a rectangle $R_i = [x_i, x_i') \times [y_i, y_i') \times z_i$, we call $[x_i, x_i')$ the *x-interval* of R_i. The structure for answering vertical ray shooting queries in S will be a two-level tree. The main tree is a segment tree T on the x-intervals of the rectangles. For a node ν in T, let S_ν be the subset of rectangles in S whose x-intervals span the interval range of ν, but not the range of the parent of ν. The set of rectangles S_ν is stored as follows at ν: Project the rectangles onto the yz-plane and construct the vertical adjacency map of the resulting set of line segments. Preprocess this adjacency map, which is a rectangular subdivision, for fast point location according to Theorem 2.4.

A ray shooting query with a vertical ray α is performed as follows. First, the y- and z-coordinates of the starting point of α are normalized by binary search. Then we search with the x-coordinate of the starting point in the segment tree. At every node ν on the search path, we perform a ray shooting query with the projection of α onto the yz-plane in the projection of S_ν; this is done by locating the normalized projection of the starting point in the vertical adjacency map stored at ν. Finally, we compare the $O(\log n)$ rectangles that we have found and select the one that is hit first.

Theorem 3.7 *Vertical ray shooting queries in a set of n horizontal axis-parallel rectangles in 3-space can be answered in $O(\log n(\log\log n)^2)$ time using $O(n \log n)$ space. Preprocessing takes $O(n \log^2 n)$ expected time.*

Proof: The vertical adjacency map at node ν has size $|S_\nu|$ and it can be constructed by a simple sweep line algorithm in time $O(|S_\nu| \log |S_\nu|)$. By Theorem 2.4, and the fact that each rectangle is stored in at most $2 \log n$ nodes, the bounds on the space and the construction time follow.

The time for a ray shooting query is dominated by the point location queries that we perform at the $O(\log n)$ nodes on the search path in the segment tree, each taking $O((\log\log n)^2)$ time by Theorem 2.4. ∎

Next, we turn our attention to vertical ray shooting queries among horizontal c-oriented triangles, where c is a constant. Observe that we cannot use the same approach as among horizontal rectangles; a horizontal rectangle is intersected by a vertical ray if and only if it is intersected in both the projection onto the xz-plane and the projection onto the yz-plane, which is not true for triangles. Instead, we use a theorem of Alt et al. [1], which says that the boundary complexity of the union of a set of homothetic triangles is linear in their number.

We partition the triangles into $\binom{c}{3}$ sets, depending on the orientations of the edges; each set S consists of homothetic triangles. We can answer the ray shooting query on each set independently and choose the best result.

The structure for vertical ray shooting queries in S is a two-level structure. The main tree is a binary search tree T on the z-coordinates of the triangles. With each node ν of T we associate the set S_ν of triangles whose z-coordinates are stored in the leaves of the subtree rooted at ν. At ν we store the union of the projections onto the xy-plane of the triangles in S_ν, preprocessed for fast point location according to Theorem 3.4. Beside this two-level structure we have c binary search trees to perform the normalizations that are needed for the fast point location structure.

A query with a vertical ray α is performed as follows. First, we normalize the x- and y-coordinates of α. Then we search with the z-coordinate of the starting point of α in T. Let $\nu_1, \ldots, \nu_t$ be an enumeration in depth-increasing order of the nodes that are the left child of a node on the search path but that are not on the search path themselves. Notice that the set $\bigcup_{1 \le i \le t} S_{\nu_i}$ is exactly the set of triangles that have smaller z-coordinate than the starting point of α. Also notice that if α intersects triangles in both S_{ν_i} and S_{ν_j}, with $i < j$, then the triangles in S_{ν_i} are intersected first. Hence, a ray shooting query can be answered as follows. Test S_{ν_1}, S_{ν_2}, $\ldots$, until the first S_{ν_i} is found that contains at least one triangle intersected by α. Then start walking down the subtree rooted at ν_i, turning right whenever at least one triangle is intersected at the right child of the current node, and turning left otherwise. The leaf where the search ends will contain the first triangle that is hit. To test whether a set S_ν contains at least one triangle that is intersected, we can use the fast point location structure associated with ν: if the intersection of α with the xy-plane is contained in the union of the projected triangles then at least one triangle is stabbed, otherwise α misses all triangles.

Theorem 3.8 *Vertical ray shooting queries among a set of n horizontal c-oriented triangles in 3-space can be answered in time $O(\log n(\log\log n)^2)$ with a structure that uses $O(n \log n)$ space. This structure can be built in $O(n \log^2 n)$ expected time.*

Proof: The union of the projections of the homothetic triangles in S_ν is a set of 3-oriented polygons, whose total complexity is $O(|S_\nu|)$ by a theorem of Alt et al. [1]. This bound, together with Theorem 3.4 and the fact that $\sum_\nu |S_\nu| = O(n \log n)$ establishes space bounds. All unions can be constructed in $O(n \log^2 n)$ time by a divide and conquer algorithm similar to that of Kedem et al. [11]. Computing the point location structures takes additional $O(n \log^2 n)$ deterministic and $O(n \log n)$ expected preprocessing time. Because all normalizations are performed beforehand, the point location queries take $O((\log\log n)^2)$ time, leading to a total query time of $O(\log n(\log\log n)^2)$. ∎

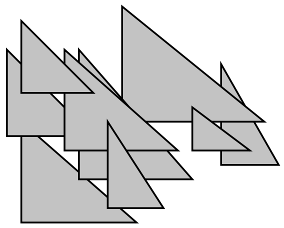

Figure 6: Triangles of a set S

For fat triangles we have a similar result with a slightly worse space bound because the union of fat triangles can have superlinear complexity. We attain the result in a similar fashion: First we break our triangles into triangles with two sides parallel to a given set of c orientations, as in Corollary 3.5. Then we partition the resulting triangles into $4\binom{c}{2}$ sets; the triangles of a given set S have two edges whose orientations come from a fixed set of c orientations and whose directions from their intersection point are the same. As Figure 6 illustrates, the triangles of S can be viewed as triangles whose lower and leftmost edges are parallel to the x and y axes after a suitable affine transformation. We will independently answer vertical ray shooting queries on each of the sets S and report the best answer.

To answer ray shooting queries among the triangles of S we use the tree of unions, as in Theorem 3.8.

Theorem 3.9 *Vertical ray shooting queries among a set of n horizontal fat triangles in 3-space can be answered in $O(\log n(\log\log n)^2)$ time and $O(n \log n \log\log n)$ space. Preprocessing takes $O(n \log^2 n \log\log n)$ expected time.*

Proof: By a theorem of Matoušek et al. [13], the union of the triangles in S_ν is bounded by $O(|S_\nu| \log\log |S_\nu|)$ line segments. Since all lower and rightmost edges of of triangles in S_ν are parallel to the x- or y-axis (under a suitable affine transformation), the horizontal adjacency map of the union consists of trapezoids with known top, bottom and left edges. Thus, we can extend the horizontal adjacency map to a sparse rectangularization and perform each point location in $O((\log\log n)^2)$ query time after normalization. Because all normalizations are performed beforehand, the total query time is $O(\log n(\log\log n)^2)$.

Storage space is dominated by the complexity of all unions, which is $O(n \log n \log\log n)$. All unions can be constructed in $O(n \log^2 n \log\log n)$ time by a divide and conquer algorithm [13] and computing the point location structures takes additional $O(n \log^2 n \log\log n)$ deterministic and $O(n \log n \log\log n)$ expected preprocessing time. ∎

Acknowledgements

We thank Kurt Mehlhorn for discussions on perfect hashing, Mark Overmars for discussions of fat objects, and the referee for references [17, 18] and other helpful comments.

References

[1] H. Alt, R. Fleischer, M. Kaufmann, K. Mehlhorn, S. Näher, S. Schirra, and C. Uhrig. Approximate motion planning and the complexity of the boundary of the union of simple geometric figures. In *Proc. 6th Ann. ACM Symp. Comp. Geom.*, pages 281–289, 1990.

[2] B. Chazelle. Filtering search: A new approach to query-answering. *SIAM J. Comp.*, 15(3):703–723, 1986.

[3] M. Dietzfelbinger, A. Karlin, K. Mehlhorn, F. Meyer auf der Heide, H. Rohnert, and R. E. Tarjan. Dynamic perfect hashing: Upper and lower bounds. In *Proc. 29th FOCS*, pages 524–531, 1988. Revised version: Bericht Nr. 77, Reihe Informatik, Paderborn, Januar 91.

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

[5] H. Edelsbrunner, G. Haring, and D. Hilbert. Rectangular point location in d dimensions with applications. *Computer Journal*, 29:76–82, 1986.

[6] H. Edelsbrunner and H. A. Maurer. A space-optimal solution of general region location. *Theoretical Comp. Sci.*, 16:329–336, 1981.

[7] M. L. Fredman, J. Komlós, and E. Szemerédi. Storing a sparse table with $O(1)$ worst case access time. *JACM*, 31(3):538–544, 1984.

[8] M. T. Goodrich. Planar separators and parallel polygon triangulation. In *Proc. 24th Ann. ACM STOC*, pages 507–516, 1992.

[9] M. T. Goodrich and R. Tamassia. Dynamic trees and dynamic point location. In *Proc. 23rd Ann. ACM STOC*, pages 523–533, 1991.

[10] D. Johnson. A priority queue in which initialization and queue operations take $O(\log\log D)$ time. *Math. Systems Theory*, 15:295–309, 1982.

[11] K. Kedem, R. Livne, J. Pach, and M. Sharir. On the union of Jordan regions and collision-free translational motion amidst polygonal obstacles. *Disc. & Comp. Geom.*, 1:59–71, 1986.

[12] R. J. Lipton and R. E. Tarjan. Applications of a planar separator theorem. In *Proc. 18th FOCS*, pages 162–170, 1977.

[13] J. Matoušek, N. Miller, J. Pach, M. Sharir, S. Sifrony, and E. Welzl. Fat triangles determine linearly many holes. In *Proc. 32nd FOCS*, pages 49–58, 1991.

[14] K. Mehlhorn and S. Näher. Bounded ordered dictionaries in $O(\log\log N)$ time and $O(n)$ space. *Info. Proc. Let.*, 35:183–189, 1990.

[15] H. Müller. Rasterized point location. In H. Noltemeier, editor, *Proc. WG 85*, pages 281–294. Trauner Verlag, 1985.

[16] F. P. Preparata. Planar point location revisited. *Int. J. Found. Comp. Sci.*, 1(1):71–86, 1990.

[17] F. P. Preparata, J. S. Vitter, and M. Yvinec. Computation of the axial view of a set of isothetic parallelepipeds. *ACM Trans. Graph.*, 9:278–300, 1990.

[18] F. P. Preparata, J. S. Vitter, and M. Yvinec. Output-sensitive generation of the perspective view of isothetic parallelepipeds. *Algorithmica*, 8:257–283, 1992.

[19] P. van Emde Boas. Preserving order in a forest in less than logarithmic time. In *Proc. 16th FOCS*, pages 75–84, 1976.

[20] P. van Emde Boas. Preserving order in a forest in less than logarithmic time and linear space. *Info. Proc. Let.*, 6:80–82, 1977.

[21] P. van Emde Boas, R. Kaas, and E. Zijlstra. Design and implementation of an efficient priority queue. *Math. Systems Theory*, 10:99–127, 1977.

[22] D. E. Willard. Log-logarithmic worst-case range queries are possible in space $\Theta(N)$. *Info. Proc. Let.*, 17:81–89, 1983.