Computational geometry on a grid
an overview

Mark H. Overmars

RUU-CS-87-4
February 1987

Rijksuniversiteit Utrecht

Vakgroep informatica

Budapestlaan 6 3584 CD Utrecht
Corr. adres: Postbus 80.012 3508 TA Utrecht
Telefoon 030-53 1454
The Netherlands

Computational geometry on a grid
an overview

Mark H. Overmars

February 1987

Abstract

In this paper an overview is given of a number of algorithms solving problems in computational geometry on a grid, i.e., in the case where objects have integer coordinates in some bounded universe. The emphasis is on simple, yet efficient solutions. Especially problems with some relevance to computer graphics are studied.

Simple and efficient algorithms are given for sorting of multi-dimensional pointsets, searching in such sets, finding a convex hull or maximal elements, finding intersections between rectangles (bounding boxes) and line segments (an important step in hidden line removal), etc.

The techniques show that in the restricted environment that often occurs in computer graphics (and other) applications, there is in general no need for the complicated or less efficient methods that solve the problems in general space.

1 Introduction.

Computational geometry normally concerns itself with geometric problems in arbitrary real (2- or multi-dimensional) space. On the other hand, in many practical applications geometric objects are normally taken from some bounded domain or can be rounded to some bounded domain. For example, in graphics coordinates of objects are often expressed in pixel coordinates or can, for example after a window to viewport mapping, be rounded to pixel coordinates. In VLSI-design objects and wires are normally alligned to a grid. And in for example robotics, where we need to find paths between obstacles, there is normally no problem in enlarging the obstacles slightly until they lie on a grid.

Hence, it seems useful to study problems from computational geometry in a bounded universe, i.e., on a grid. To state this more formally, let U be some integer. We assume that all points, endpoint of line segments, vertices of polygons, etc. have integer coordinates between 0 and $U-1$. In other words, the points are choosen from $[0..U-1]^d$, also denoted as U^d where d is the dimension. We

call U the *universe size*. For example, when the points are pixels on $1024 * 1024$ screen, $U = 1024$. Note that, with appropriate scaling and translation, U^d can approximate any bounded region of the real d-dimensional space. Note also that in this framework e.g. line segments are still real line segments that might intersect in some point that is not on the grid. We do not break objects down to pixels.

Computational geometry on a grid has recently been studied by a number of researchers (See e.g. [5,8,9,10,11,13,16].) In this overview we will describe a number of the results obtained so far, showing that on a grid many problems from computational geometry can by solved more efficiently and, often, in a much simpler way than in arbitrary real space. This is mainly due to the fact that we can use table look-up rather than searching in a number of applications and that there often is no need for rebalancing structures. This shows that it is practical to use these "restricted" solutions in situations in which objects do lie on a grid, as might be the case in many graphical application.

The emphasis in this paper will be on simple methods. In a number of cases we will not give the most efficient method when this method is very complicated, although we will give references to such results. We mainly treat problems that have some relevance to computer graphics although also some classical problems in computational geometry, like e.g. the convex hull problem, are treated.

This paper is organised as follows:

In Sections 2, 3 and 4 we treat two basic problems: sorting and searching. Most algorithms use sorting and searching as basic steps and, hence, it is important to investigate their complexity on a grid. We give efficient, simple methods for *lexicographical sorting* on a multi-dimensional grid and demonstrate some search structures, especially the *vanEmdeBoas tree* [23,24].

In Section 5 we treat the classical problem of finding the *convex hull* of a set of points on a planar grid. Moreover we look at related problems like finding *maximal elements*. We show that on a grid these problems can be solved in linear time.

In Sections 6,7 and 8 we treat *intersection problems*. First of all we look at the problem of finding all intersections in a set of axis-parallel rectangle. This is especially relevant in computer graphics applications where objects are surrounded by bounding boxes. Finding intersections between bounding boxes is the first step in finding intersections between the objects themselves. Secondly, we look at intersections in a set of line segments. Finding such intersections can be a first step in hidden line removal algorithms ([22]).

Finally, in Section 9 we give some concluding remarks.

The results in this paper assume that the size of the universe U is about the same size (or smaller) than the size of the set of objects. When U is much larger, the solutions are not very good. Note in this respect that U is the size of the universe, NOT the size of the grid which is e.g. U^2 in the 2-dimensional case.

In computational geometry, often, the assumption is made that no special case occur, like e.g. three points on a line, two points with the same x-coordinate, etc. On a grid such assumptions are highly unrealistic. Hence, in the presentation of

the algorithms we will try to make no such assumptions or we indicate how to solve the special cases.

2 Sorting.

In this section we will shortly recall two well-known results on sorting.

Theorem 2.1 *Given a set of n keys with a value between 0 and $U-1$, they can be sorted in time $O(n+U)$ using $O(n+U)$ storage.*

Proof. We make an array of U buckets and initialise them to be empty. Next we put each element in the set in its bucket using the key as array index. Finally we walk along the buckets in order, reporting the key when a bucket is not empty. The bounds follow trivially. □

On a two dimensional grid it is often important to sort the points *lexicographical*, i.e., $p = (p_1, p_2) < p' = (p'_1, p'_2)$ if $p_1 < p'_1$ or $p_1 = p'_1$ and $p_2 < p'_2$. To sort a set of points lexicographical we can use a simple version of bucket sort:

Theorem 2.2 *Given a set of n points in U^2, they can be sorted lexicographical in time $O(n+U)$ using $O(n+U)$ storage.*

Proof. We create two arrays of U buckets, but this time each bucket will contain a list of points. First we sort the points by second coordinate, using one of the arrays of buckets. If more points have the same second coordinate we put them in the list in the corresponding bucket. Next treating the points ordered by second coordinate we put them in the other array of buckets using their first coordinate as an index. When more points have the same first coordinate they will be put in the list in the bucket. Because we treat the points by increasing second coordinate, such a list corresponding to a particular first coordinate will contain the points in the order of their second coordinate. Hence, walking along the buckets and reporting the points in the order in which they appear in the lists will give the points in lexicographic order. The bounds follow trivially. □

On a d-dimensional grid we can use a similar technique. First we sort all points with respect to the dth coordinate. Next, treating the points in this order, we sort them with respect to the $d-1$th coordinate, next on the $d-2$th coordinate, etc. This immediately leads to the following result (noting that we need only two arrays of buckets):

Theorem 2.3 *Given a set of n points in U^d, they can be sorted lexicographical in time $O(d(n+U))$ using $O(n+U)$ storage.*

These results are, in fact, not optimal. Kirkpatrick and Reisch[13] have shown that a set of n keys in U can be sorted in time $O(n(1+\log\log_n U))$ using $O(n+U)$ storage. Their result can easily be extended to sort a set of points in U^d in time $O(dn(1+\log\log_n U))$ using $O(n+U)$ storage. On the other hand, when U is relatively small, this result is not any better and the method is more complicated.

3 Simple searching.

Let V be a set of n keys in U. We often need to store V such that questions like: Is $K \in V$, which keys in V lie in between A and B, what is the rank of a key K in V, etc., can be answered efficiently. In arbitrary real space search trees, like e.g. AVL-trees, are used for answering such questions. On a grid we can solve these problems much more efficiently.

Theorem 3.1 *Given a set V of n keys in U, we can store V using $O(n+U)$ storage such that given a key K we can determine in $O(1)$ time the largest key in V that is $\leq K$.*

Proof. The method is trivial. We make an array KEY of length U. In KEY[i] we store whether $i \in V$ and we store a pointer to the largest $j < i$ such that $j \in V$. Clearly, this takes $O(n+U)$ storage and queries can be answered in time $O(1)$. The structure can be built in time $O(n+U)$. □

A similar structure can be used to solve the ranking problem that asks how many keys are $\leq K$ in time $O(1)$ and the range searching problem that asks for all keys in $[A : B]$ in time $O(1+k)$ where k is the number of reported answers.

The structure is static, i.e., it is hard to insert or delete keys in the set V. For dynamic structures, see the next subsection.

The structure works fine when U is not much larger than n but when U is much larger the amount of storage required becomes too large. Fredman et al. [4] describe a technique, based on perfect hashing, that can answer queries of the form: is $K \in V$, in time $O(1)$ using only $O(n)$ storage. Unfortunately, their structure has a very high preprocessing time of $O(n^3 \log U)$. Moreover, as it is based on hashing, it cannot answer questions like: what is the largest key $\leq K$.

Willard[25] designed a structure, based on [4], that can answer such questions in time $O(\log\log U)$ using $O(n)$ storage. Unfortunately, also his structure has a very high preprocessing time.

4 The vanEmdeBoas tree.

The vanEmdeBoas tree [23,24] is an efficient dynamic data structure for storing a set of keys in a one-dimensional grid U, as long as U is not too large.Here we will

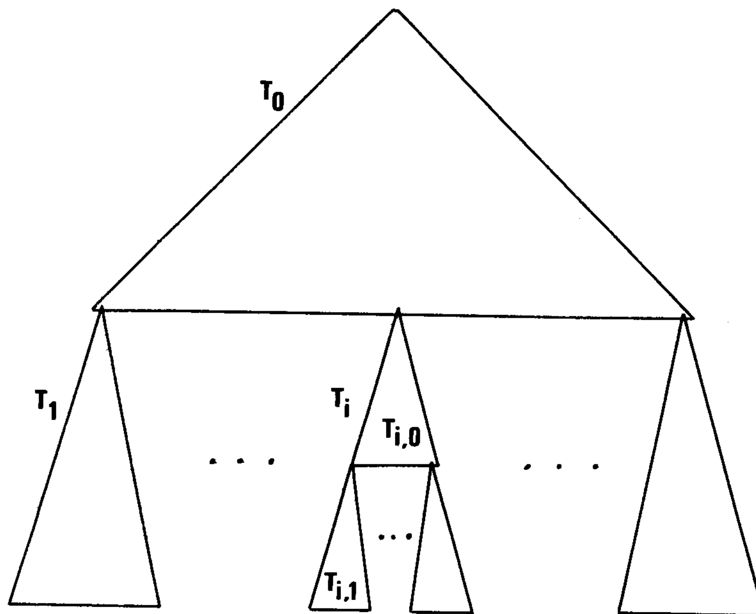

Figure 1: The partition of the tree.

describe a variation of the structure that obtains exactly the same time and space bounds but is easier to describe and implement. We will call it a vanEmdeBoas tree anyhow.

Let V be the set of points and let U be the universe. We construct a complete balanced binary tree T of depth $\log U$ on the universe. (For convenience we assume that $U = 2^i$ for some i.) Now you can think of this tree as having the following form. It consists of a top tree T_0 of depth $\lceil \frac{\log U}{2} \rceil$ having about $\sqrt{U}$ leaves and about $\sqrt{U}$ bottom trees T_1, T_2, ... also of size about $\sqrt{U}$. T_0 is a tree with a universe of size $\sqrt{U}$ being the indexes of the bottom tree. Also the bottom trees have a universe of size about $\sqrt{U}$. Each of the trees T_i is in the same way subdivided in a top tree $T_{i,0}$ and bottom trees $T_{i,1}$, $T_{i,2}$, ..., each of size $\sqrt[4]{U}$, etc. See figure 1 for the partition obtained.

Keys will be stored in the tree T in the following way. We split V in subsets V_1, V_2, ..., corresponding to the subtrees T_1, T_2, ... Each V_i we store in T_i in the following way. At the root of T_i we store the size of V_i. If V_i is empty, nothing else will be stored in T_i (although the subtree will be there). If V_i contains 1 or 2 keys, we store them at the root of T_i. If V_i contains more than 2 keys we store at the root the largest and smallest key in V_i. the other keys we store in T_i recursively in the same way. (I.e., we split V_i in subsets $V_{i,1}$, $V_{i,2}$, ... corresponding to $T_{i,1}$, $T_{i,2}$, ..., etc.) When V_i is not empty, we insert i in T_0. Finally, we link all keys in T in a double linked list such that, once we find a key, we can always find its two neighbors.

Clearly, the main T has $O(U)$ nodes and at each node at most 2 keys and the size are stored. Hence, the structure uses $O(U)$ storage. (Note that it will require $\Omega(U)$ storage, even when V is very small.)

The vanEmdeBoas tree is used for so-called neighbor queries that ask for some neighbor of a given query key K (i.e., the first key larger or smaller, or equal to K). Of course, because we linked the keys in a double linked list, once we have a neighbor we can always in $O(1)$ time find the other neighbor. This also allows us to find e.g. all keys between K_1 and K_2 in time $O(1)$ per answer once we found a neighbor of K_1.

To perform a neighbor query with key K we proceed in the following way. Determine the tree T_i K comes in. Because T is a complete binary tree this can be done in time $O(1)$. When T_i contains 1 or 2 keys (indicated with the root) one of the 1 or 2 keys stored at the root must be a neighbor and we are done. When T_i contains more than 2 keys, we first check whether one of the two keys stored at the root is a neighbor. (This can be done in $O(1)$ time by looking at them and their neighbors in the double linked list.) If so we are done. Otherwise the neighbor must be stored in T_i and we continue our search there recursively in the same way. (I.e., we look at the appropriate $T_{i,j}$, etc.) When T_i is empty, we search for a neighbor j of i in T_0. This means that T_j is the first non-empty subtree to the left or to the right of T_i. Assume $j < i$. In that case the largest key stored at the root of T_j is a neighbor of K. Similar, when $j > i$, the smallest key stored at T_j is a neighbor.

To estimate the time required, note that after $O(1)$ work we are left with a subtree of size $\sqrt{U}$. Hence, if $Q(U)$ denotes the query time over a tree of size U, we get $Q(U) = O(1) + Q(\sqrt{U})$, which leads to $Q(U)=O(\log\log U)$.

To perform an insertion of a key K we proceed in the following way. First we search for a neighbor K' of K in T. Next, determine the subtree T_i K must come in. Increase the size of T_i with 1. If T_i is empty store K at its root and insert i in T_0. If T_i contains 1 key just store K at its root. If T_i contains 2 or more keys, let A and B (with $A < B$) be the keys stored at its root. If $K < A$ store K at the root and insert A in T_i. If $K > B$ store K here and insert B in T_i. Otherwise insert K in T_i. Finally, using K' insert K in the double linked list.

To delete a key K determine the subtree T_i K is in. Decrease the size of T_i. If the subtree contains 1 key, it must be K and K is stored at the root. Remove K here and remove i from T_0. If the subtree contains 2 keys, K is stored at the root and we can just remove it. Otherwise, if K is not stored at the root of T_i, delete K from T_i. If K is stored at the root, locate its neighbor in T_i (there must be such a neighbor). This can be done in time $O(1)$ using the double linked list. Replace K by its neighbor, and delete the neighbor from T_i. Finally, delete K from the double linked list.

The insertion and deletion time is bounded by $O(\log\log U)$ following the same arguments as in the query time. We conclude:

Theorem 4.1 *Let V be a set of keys in U. There exists a dynamic structure for storing V, requiring $O(U)$ storage such that insertions, deletions and neighbor queries take time $O(\log\log U)$.*

A number of improvements over the vanEmdeBoas tree have been suggested. Johnson[7] describes a modification to reduce the initialisation time and to perform some queue operations more efficiently. Willard[26] and Karlsson[8] describe dynamic structures on a grid that use only $O(n)$ storage. To obtain this lower space requirement, the update and query time of their methods is increased to $O(\sqrt{\log U})$. We won't go into details of their solutions because the methods are more complicated an unneccessary in the case U is of the same order as n.

5 Convex hulls and related problems.

As an example of how the fact that our points have integer coordinates can help us in deriving efficient algorithms we will consider the *convex hull* problem. Given a set of points (or a polygon), the convex hull is the smallest convex polygon containing all the points (or the polygon). A convex hull of a polygon is often a good first estimation for computing e.g. intersections or visibility. Computing the convex hull of a set of n points can be done in $O(n \log n)$ time (see e.g. [1,6,14,20]). For a simple polygon there exists a quite complicated method for computing the convex hull in linear time. We will show that on a grid there exists a very simple linear time solution to both problems. We will only treat the problem of finding the convex hull of a set of points. The convex hull of a polygon is the same as the convex hull of the set of vertices of the polygon.

Theorem 5.1 *Given a set V of n points in U^2, the convex hull of V can be computed in time $O(n + U)$.*

Proof. The method is based on [6]. First we sort all points by x-coordinate. This can be done in $O(n + U)$ time using Theorem 2.1. Clearly, the leftmost point p_l and the rightmost point p_r belong to the convex hull. The convex hull now consists of an upper chain UC that runs from p_l to p_r having all points below it, and a lower chain LC having all points above it. (See figure 2.)

We will only show how to compute UC. LC can be computed in exactly the same way. We will describe the method without much details. For detail see [6]. We walk from p_l to p_r. We will take care that when we are at some point p_i we found a convex arc from p_l to p_i with all points in between below it. Assume we have reached some point p_i. Now look at the angle $p_{i-1}p_ip_{i+1}$. If this angle is convex we can continue with $i := i + 1$. Otherwise, p_i will not belong to the upper convex hull and can be removed. We backup and continue with $i := i - 1$. It is easy to see that is this way, in $O(n)$ time, we find the upper convex chain. □

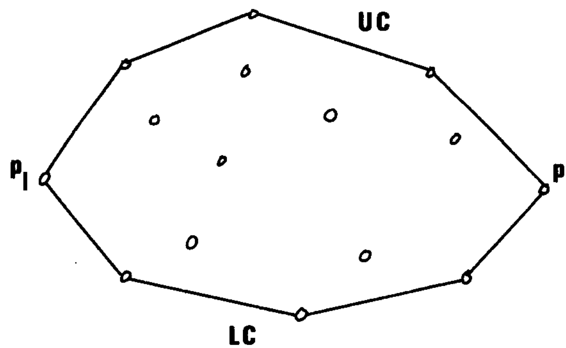

Figure 2: Convex chains.

In fact, the only place where we use the property that the point lie on a grid is when sorting the points. After the sorting the algorithms runs in $O(n)$ time independent of the fact whether the points lie on a grid.

Theorem 5.2 *A convex polygon (hull) on n points on a grid can be stored using O(U) storage such that for a given point p one can determine in O(1) time whether p lies inside or outside the polygon.*

Proof. Scanning from left to right over the polygon, we store for each vertical grid line the two intersection points of this line with the polygon. To search with a point p (not neccessarily on the grid) we determine the two vertical grid lines in between which p lies (or the one on which p lies. With the intersection points stored there we can easily determine whether p lies inside or outside the polygon. This clearly takes $O(1)$ time. □

The structure above can trivially be built in time $O(n + U)$.

As a second example let us consider the *maximal elements* problem. Given a set V of n points in the plane, a point $p = (p_1, p_2)$ in V is called maximal if there is no point in V with x-coordinate larger than p_1 and y-coordinate larger p_2. See figure 3.

To find the maximal elements we sort the points by decreasing x-coordinate. Now we walk along the points in this order, keeping track of the point with largest y-coordinate found so far. Let y this y-coordinate. In the beginning $y = -\infty$. When we come at a point $p = (p_1, p_2)$ we check whether $p_2 > y$. If so we report p as a maximal element and set $y = p_2$. And we continue with the next point. This can easily be done in $O(1)$ per point. Hence, the time for sorting dominates the run time of the algorithm.

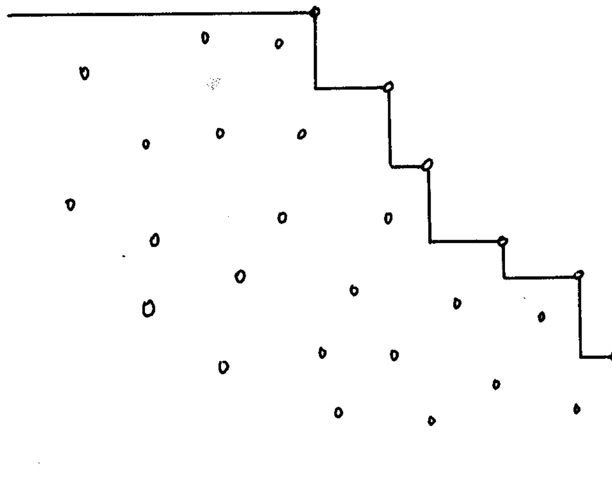

Figure 3: Maximal elements.

Theorem 5.3 *Given a set of n point in U^2, the maximal elements problem can be solved in time $O(n+U)$ using $O(n+U)$ storage.*

When U is large, the results in this section are not very good. Using the sorting method of [13] we can improve the bounds in such cases. See also [12].

6 The interval trie.

Let V be a set of n intervals in U, that might intersect or overlap. We want to store V such that we can perform *stabbing queries*, that ask for those intervals containing a given query point, efficiently. When the intervals have arbitrary real coordinates, this problem can be solved using a so-called *interval tree* using $O(n)$ storage in a query time of $O(k+\log n)$ where k is the number of reported intervals (see e.g. [21] for a description). In this subsection we will describe a solution on a grid, called the *interval trie*, that has slightly better assymptotic time complexity but might behave much better when the intervals are relatively small.

Let $F(U) = \log^{\epsilon} U$ for some $\epsilon > 0$. We split the set V in subsets $V_1, V_2, \ldots,$ where V_i contains all intervals of length between $F(U)^{i-1}$ and $F(U)^i$. Clearly, there are at most $O(\log U/\log F(U)) = O(\log U/\text{loglog}\, U)$ sets V_i. We will treat each V_i separately. For V_i we divide the universe in equal parts of size $F(U)^{i-1}$. Hence, each interval in V_i has its left endpoint in one part, overlaps at most $F(U)$ parts and has its right endpoint in another part. See figure 4. No interval can have its two endpoints in the same part. For each part we have three lists: B containing all the left endpoints, E containing all the right endpoints and O containing all the overlapping intervals. See figure 5. Both B and E we store as vanEmdeBoas

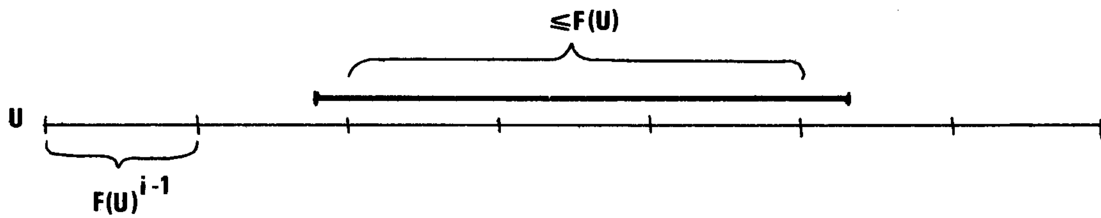

Figure 4: Parts overlapped by an interval.

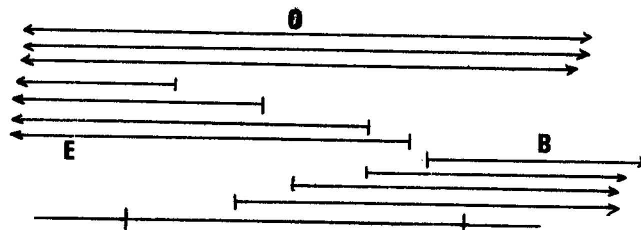

Figure 5: Intervals stored at a part.

trees, making it possible to insert and delete endpoints in time $O(\log\log U)$. O we store as a double linked list. Moreover, in B we store for each interval where it is stored in O lists.

To perform a stabbing query with point p we query each V_i separately. In V_i we determine the part p lies in in time $O(1)$. We report all intervals in the corresponding O list. In B we start at the leftmost endpoint. If it lies left of p we report it and continue with the next endpoint. If not there are no answers in B. Similar in E. It is easy to see that in this way we find all answers in V_i in time $O(1)$ plus the number of answers found. As we have to query $O(\log U/\log\log U)$ sets the query time is bounded by $O(k + \log U/\log\log U)$ where k is the number of answers found.

To insert or delete an interval we determine the set V_i the interval belongs to (looking at its length). We insert or delete the interval in the appropriate B and E structure. Moreover we have to add it to at most $F(U)$ O-lists. It is easy to see that this takes time $O(\log\log U + F(U)) = O(\log^\epsilon U)$.

Theorem 6.1 *Given a set of n intervals on a grid U, for each $\epsilon > 0$ there exists a dynamic structure for solving the stabbing problem, that can be updated in time $O(\log^\epsilon n)$, with a query time of $O(k + \log U/\log\log U)$, where k is the number of reported answers.*

Although this is only a minor improvementin query time, in practical situations the structure might perform much better. When all intervals are short, only for small i, V_i will contain intervals. As the query time is, in fact, bounded by the number of non-empty sets V_i in such situations it will be much lower.

7 Intersections between rectangles.

Intersection problems are very important in computer graphics. For example, many hidden-surface and hidden-line removal algorithm are based on finding intersections of geometric objects. In this section we will consider intersections between axis-parallel rectangles. Axis-parallel rectangles occur especially when working with bounding boxes. Only those objects whose bounding boxes intersect need to be considered further. In the next section we treat the more general problem of finding all intersections in a set of line segments.

To solve the intersection problem for axis-parallel rectangles efficiently on a grid we will use the vanEmdeBoas tree and the interval trie described above.

We will first consider the following subproblem: Let V be a set of n horizontal and vertical line segments, report all intersections between them. To solve this problem we use a plane sweep approach. We sweep a scanline from left to right, stopping at each endpoint of a line segment. With the scanline we keep a vanEmdeBoas tree T, storing all horizontal line segments that are intersected by the scanline.

So we start by sorting all endpoint of the line segments in left to right order. When more endpoint have the same x-coordinate we first store the left endpoints of horizontal line segments, then the vertical line segments with this x-coordinate and finally the right endpoints of horizontal line segments. We initialise T to be empty.

For each endpoint p belonging to a segment s we do the following: If p is the left endpoint of s we search with p in the tree to see whether it lies on any other horizontal line segment. If so we report it. Next we insert s in the vanEmdeBoas tree. If p is a right endpoint we delete s from T. Finally, if s is a vertical line segment $\overline{(x, y_1)(x, y_2)}$ we search with y_1 in T and walk along the list of stored segments, reporting all segments that lie between y_1 and y_2.

It is easy to see that the operations take time $O(\text{loglog}\, U)$ each (plus the number of answers reported).

Theorem 7.1 *Given a set V of n horizontal and vertical line segments in U^2, all k intersections between them can found in time $O(k + U + n\text{loglog}\, U)$.*

Two axis-parallel rectangles intersect if either they intersect in the boundary, or one lies completely inside the other. Given a set V of rectangles we can find those pairs that intersect in the boundary by considering each rectangle as four line segments and finding the intersections between the line segments in the way described above. So we are left with the problem of finding all pairs where one is completely contained in the other. Clearly, if a rectangle is completely contained in the other, its leftbottom corner lies in the other rectangle. So we can as well formulate the problem as follows: Given a set of points and a set of axis-parallel rectangles, report all pairs (p, R) where p lies in R.

To solve this subproblem, we again use a plane sweep approach, sweeping a scanline from left to right over the grid, but this time we store with the scanline an interval trie T (as described in the previous section) storing the projections of the rectangles intersecting the scanline. We sort all points and left and right boundaries of the rectangles by x-coordinate. When more have the same x-coordinate we store first the left boundaries, then the points, and finally the right boundaries. We initialise T to be empty. If we reach a left boundary, we insert the interval in the interval trie. If we reach a right boundary, we delete it. Finally, if we meet a point, we perform a stabbing query with the point. Using the results from the previous section, noting that $\log^{\epsilon} U = O(\log U/\text{loglog}\, U)$, we conclude

Theorem 7.2 *Given a set of n points and axis-parallel rectangles in U^2, all k pairs (p, R) with p contained in R can be found in time $O(k + U + n \log U/\text{loglog}\, U)$.*

Combining Theorems 7.1 and 7.2 we obtain:

Theorem 7.3 *Given a set of n axis-parallel rectangles in U^2, all k intersecting pairs can be found in time $O(k + U + n \log U/\text{loglog}\, U)$.*

As noted in Section 6, the time will, in practise, often be much better.

8 Intersections between line segments.

In this subsection we determine intersections among n line segments which are arbitrary oriented, but still are defined by endpoints at integer coordinates between 0 and $U-1$. Until recently, the best algorithm to find all intersections required $O((k+n)\log n)$ time where k is the number of reported intersections (see Bentley and Ottmann[3] or Preparata and Shamos[21]). Chazelle[2] was the first to provide a solution with a running time of the form $O(k+f(n))$, where $f(n)$ is a subquadratic function of n. In [2] $f(n)=n\log^2 n/\text{loglog}\, n$.

The line segment intersection problem is important in a number of Computer Graphics applications. For example, Schmitt[22] uses the line segment intersection problem as an important step in a hidden line removal algorithm. The complexity of the line segment intersection problem is dominant in his solution. Hence, any improvement on the intersection problem improves his hidden surface removal algorithm. Also other hidden surface removal algorithms use line segment intersection as an important step.

We will show that, in the case the line segments have endpoints on an integer grid, all intersections can be found in time $O(k+U+n\log n)$ using $O(n+U)$ space. To this end we will adapt the technique in [3] to work on a grid, replacing search trees by table lookup.

For each x-coordinate x we store three unordered bags of points. BEGIN(x) contains all line segments with a left endpoint with x-coordinate x. END(x) contains all line segments with a right endpoint with x-coordinate x. INTER(x) contains all intersections found so far with x-coordinate in the interval $[x,x+1)$ (intersections can have coordinates in between grid points). When a line segment is vertical with x-coordinate x, we store it in both the BEGIN(x) and the END(x) bag.

To initialise the structures, each line segment is stored in the appropriate BEGIN and END bag (by using table lookup) and all INTER bags are made empty.

Next we move a scanline from left to right stopping at each x-coordinate. With the scanline we keep a balanced binary search tree T (e.g. an AVL-tree) storing all line segments that are intersected by the scanline in the order of intersection in the internal nodes. We will take care that the INTER bags always contain exactly the intersections between neighboring segments in T. At each stop position x we perform the following three actions:

1. Insert all line segments that are stored in BEGIN(x) in the tree T. For each such line segment s we locate its two neighbors s_1 and s_2 in T. We compute the intersection between s and s_1 and (if any) insert it in the appropriate INTER bag. Similar, we compute the intersection between s and s_2 and store it. Finally, we compute the intersection between s_1 and s_2. Because these segments are no longer neighbors, we have to remove the intersection from the INTER bag (if there was an intersection). Keeping with each

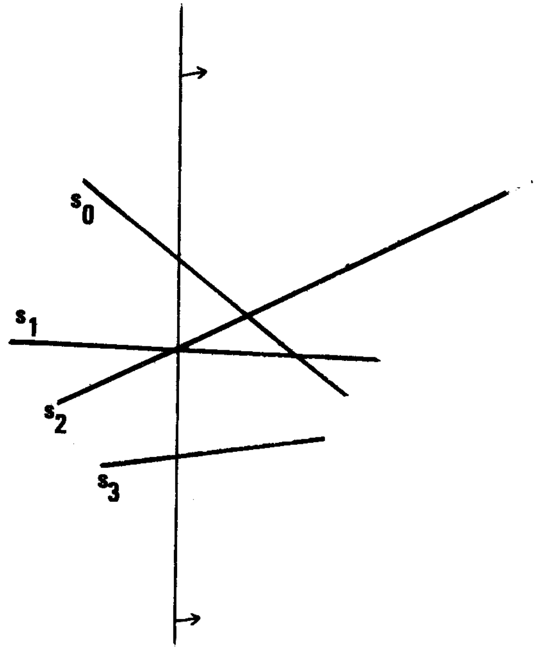

Figure 6: Intersection of line segments.

two neighbors in T a pointer to their intersection in the INTER bag this intersection can be removed in time O(1). It is easy to see that the total operation takes time $O(\log n)$ per segment inserted.

2. Delete all line segments that are stored in END(x) from T. For each such segment s we locate its two neighbors s_1 and s_2 in T. As these segments now become neighbors we compute their intersection and, if it exists and lies to the right of x we store it in the appropriate INTER bag. This takes time $O(\log n)$ per segment deleted.

3. For each intersection in INTER(x) we do the following. Let the intersection be between s_1 and s_2. s_1 and s_2 must be neighbors in T. Let s_0 be the other neighbor of s_1 and s_3 be the other neighbor of s_2. See figure 6 for the situation. First of all we report the intersection. Secondly, we switch s_1 and s_2 in T. As s_0 and s_1 are no longer neighbors, we remove their intersection (if any) from the appropriate INTER bag. Similar for s_2 and s_3. We also remove the intersection between s_1 and s_2. As s_0 and s_2 have now become neighbors we compute their intersection and, if it exists and HAS NOT BEEN TREATED BEFORE, put it in the appropriate INTER bag (which might be INTER(x)). Similar for s_1 and s_3. When we keep a pointer from each intersection to the appropriate place in T and the other way round, this total operation can be carried out in time O(1) per intersection.

The main problem with the above method is that intersections between x and

$x+1$ are NOT treated in left to right order but in the order in which they appear in INTER(x). Therefore it is very important that we only insert intersections in the INTER bags if they have not been treated before. Otherwise the algorithm could loop forever, reporting the same intersections over and over again. To avoid this we have to check whether the intersection has indeed not been treated before. To this end we use an idea of Myers[17]. Assume we want to add an intersection between s_0 and s_2 to INTER(x) (when it has to be added to another INTER bag this is alway correct). Now look at the order in which s_0 and s_2 have to be at position $x+1$. If they already are in this order we must have treated their intersection before. Otherwise we can insert it in INTER(x). This test will take only O(1) time.

We still have to prove that the method correctly reports all intersection between x and $x+1$ and that T will have the right shape afterwards. This follows from the following two observations: (i) When an intersection is reported it does exist and has not been reported before. (ii) When INTER(x) is empty this means that each two neighbors in T either do not intersect between x and $x+1$ or that their intersection has been treated. For details see [10].

Theorem 8.1 *Given a set of n line segments on U^2, we can report all intersections between line segments in time $O(k+U+n\log n)$ using space $O(n+U)$, where k is the number of reported intersections.*

Proof. The tree T uses only O(n) storage. The bags need storage O($n+U$) because there are $3U$ bag that store $2n$ endpoints and at most n intersections.

The query time follows from the above observations. Each insertion and deletion takes time O($\log n$), each intersection takes time O(1) and we have to spend at least O(1) time for each bag. □

When U is large, this method is not very good. In [10] a second method is presented that takes time O($(k+n)\text{loglog}\, U + n\log n$).

9 Conclusions and extensions.

In this paper we have demonstrated that techniques from computational geometry, combined with the property that objects have integer coordinates, can lead to simple but efficient solutions for many geometric problems that are related to computer graphics. We believe that these techniques should be applicable in computer graphics applications and can lead to powerful, simple methods that are also very efficient.

Although the presentation of the algorithms does not go into implementation details we think that, because of the simplicity of the solutions, it should not be much work to come to efficient implementations. We realise that in practise,

problems normally are not as simple as stated in this paper. Still we believe that the techniques presented will be useful.

This paper only lists a number of example problems with their solutions. Similar methods can be used to solve other geometric problems on a grid as well. For example, Karlsson and Munro[9] consider the problem of finding nearest neighbors on a grid and Müller[16] discusses the point location problem on a grid. In Overmars[18,19] the range searching problem on a two-dimensional grid is solved in a (theoretically) very efficient way. Also multi-dimensional problems can be treated in an efficient way. See e.g. Karlsson and Overmars[11] and Gabow e.a.[5].

Theoretical, the techniques presented are not optimal. For example, as mentioned, sorting can be done more efficiently using a method of Kirkpatrick and Reisch[13]. Although this gives assymtotically faster results, in practise, due to the complexity of the methods, this will not improve the methods. Therefore we have choosen to present the methods using the more simple techniques.

References

[1] Andrew, A.M., Another efficient algorithm for convex hulls in two dimensions, *Info Proc. Lett.* 8 (1979), 108-109.

[2] Chazelle, B., Intersecting is easier than sorting, *Proc. 16th Annual ACM Symp. on Theory of Computing,* 1984, 125-134.

[3] Bentley, J.L. and T.A. Ottmann, Algorithms for reporting and counting geometric intersections, *IEEE Trans. on Computers* 28 (1979), 643-647.

[4] Fredman, M.L., J. Komlos and E. Szemeredi, Storing a sparse table with O(1) worst case access time, *J. ACM* 31 (1984), 538-544.

[5] Gabow,H.N., J.L. Bentley and R.E. Tarjan, Scaling and related techniques for geometric problems, *Proc. 16th Annual ACM Symposium on Theory of Computing,* 1984, 135-143.

[6] Graham, R.L., An efficient algorithm for determining the convex hull of a finite planar set, *Info. Proc. Lett.* 1 (1972), 132-133.

[7] Johnson, D.B., A priority queue in which initialization and queue operations take O(loglog D) time, *Math. Systems Theory* 15 (1982), 295-310.

[8] Karlsson, R.G., *Algorithms in a restricted universe,* Ph.D. thesis, Techn. Rep. CS-84-50, Dept. of Comp. Science, University of Waterloo, 1984.

[9] Karlsson, R.G., and J.I. Munro, Proximity on a grid, *Proc. 2nd Symp. on Theoretical Aspects of Comp. Science,* Springer-Verlag Lect. Notes in Comp. Science 182, 1985, 187-196.

[26] Willard, D.E., New trie data structures which support very fast search operations, *J. Comput. Syst. Sci.* 28 (1984), 379-394.