Computational Geometry and its application to Computer Graphics

Mark H. Overmars

RUU-CS-89-8
March 1989

Rijksuniversiteit Utrecht

Vakgroep informatica

Padualaan 14 3584 CH Utrecht
Corr. adres: Postbus 80.089, 3508 TB Utrecht
Telefoon 030-531454
The Netherlands

Computational Geometry and its application to Computer Graphics

Mark H. Overmars

RUU-CS-89-8
March 1989

Rijksuniversiteit Utrecht

Vakgroep informatica

Padualaan 14 3584 CH Utrecht
Corr. adres: Postbus 80.089, 3508 TB Utrecht
Telefoon 030-531454
The Netherlands

Computational Geometry and its application to Computer Graphics

Mark H. Overmars

Technical Report RUU-CS-89-8
March 1989

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

Computational Geometry and its application to Computer Graphics

Mark H. Overmars

1 Introduction

The area of *computational geometry* deals with the study of algorithms for problems concerning geometric objects like e.g. lines, polygons, circles, etc. in the plane and in higher-dimensional space. Since its introduction in 1976 by Shamos the field has developed rapidly and nowadays there are special conferences and journals devoted to the topic.

A recently produced bibliography contained over 2000 entries and this is only a small portion of the material actually published. Although so much work has been done and although computational geometry is rich with results only a very small number of textbooks exists. The most well-known are the works of Mehlhorn [57], Preparata and Shamos [67] and Edelsbrunner [22].

Clearly, a large number of problems in computer graphics deal with geometric objects as well. Examples are hidden surface removal, windowing problems, intersection problems, etc. Hence, computer graphics can benefit from the techniques developed in computational geometry.

Algorithms in computational geometry normally work in *object space*, i.e., objects are stored and treated in a structural way rather than being broken down to e.g. pixels as often happens in the *image space* algorithms that are mostly used in graphics. Treating objects in a structural way has a number of advantages. The most important advantage is that results and time complexity are independent of image sizes. Moreover, treating objects in object space normally gives more flexibility (e.g. rotating a scene is easy).

This paper gives a basic introduction into some of the techniques and data structures developed in computational geometry with emphasis on their use in computer graphics applications. Due to the limited space only global ideas are outlined and no details are presented. References are added, in particular in a subsection "Further Reading" at the end of each section, for those that are interested in learning more about the topics treated.

The paper is organized as follows.

In section 2 we consider problems concerning shapes of sets of objects. In particular we look at the problem of determining the convex hull of a set of points. This is a basic problem in computational geometry with many applications. We give a number of different algorithms for this problem, introducing different algorithmic techniques. As a second problem we look at triangulating pointsets and polygons.

In section 3 we concentrate on proximity problems where distance plays an important role. We introduce the Voronoi diagram, a powerful structure that stores distance information, and use it to solve the nearest neighbor searching problem. To obtain this solution some point location techniques are introduced.

In section 4 we consider the well-known windowing problem where we ask which part

of a picture is visible inside a given window. Data structures like k-d trees, range trees and segment trees are introduced for solving this problem.

In section 5 we treat intersection problems. First we concentrate on finding intersections between axis-parallel line segments and rectangles. Next we treat arbitrary line segments and other objects.

In section 6 we look at the hidden surface removal problem. We present two different object space algorithms. The first solution is based on projecting the objects and computing all intersections. The disadvantage of this method is that the number of intersections can be very large. The second solution avoids this problem. This surprisingly simple technique is purely dependent on the complexity of the visible scene.

Finally, in section 7, we give an overview of some important other directions of research in computational geometry and results obtained there.

2 Shape of objects

Given some object, e.g. a polygon, or a set of objects, e.g. points, it is often important to obtain information about the shape of such an object or a set. For example, one might want to obtain a bounded part of space that contains all the objects, or one might want to have a subdivision of the object in a set of smaller, simpler objects. In this section we deal with these two problems. First we concentrate on convex hulls of sets of points and of polygons. Next we look at triangulations, that subdivide pointsets or polygons into a number of non-overlapping triangles.

2.1 Convex hulls

Let V be a set of points or other objects in the plane, the *convex hull* of V is the smallest convex object containing all the points or objects. (An object is convex iff for any two points in the object the line segment connecting the points lies completely inside the object.) The convex hull bounds the set of objects from the outside. See figure 1 for an example of the convex hull of a set of points.

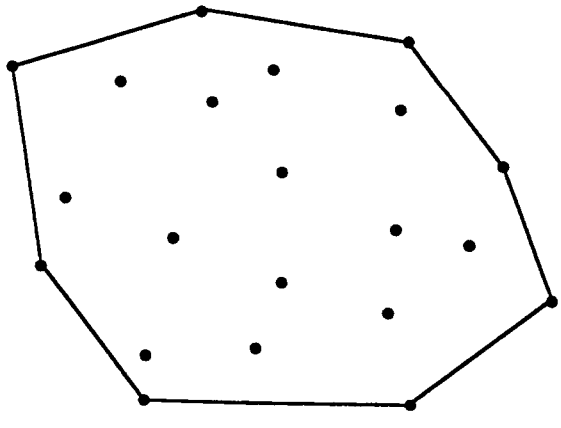

Figure 1: The convex hull.

Convex hulls play an important role in many applications. First of all they give some information about the shape of a set of objects. Secondly, convex objects are easy to

deal with. Hence, in many applications the convex hull of objects is often a good first approximation. For example, if a line segment does not intersect the convex hull, it surely does not intersect the object. So the convex hull is a good alternative for the *bounding box* that is often used in graphics. It is easy to compute, easy to use and it gives a better approximation for the object (e.g., the convex hull of a line segment is the segment itself).

For the moment let V be a set of points. Many algorithms are known for computing convex hulls of sets of points. We will mention a few different methods that use completely different techniques.

The first technique is based on one of the earliest methods by Graham [33]. First we sort all points by x-coordinate. This can be done in $O(n \log n)$ time. 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.)

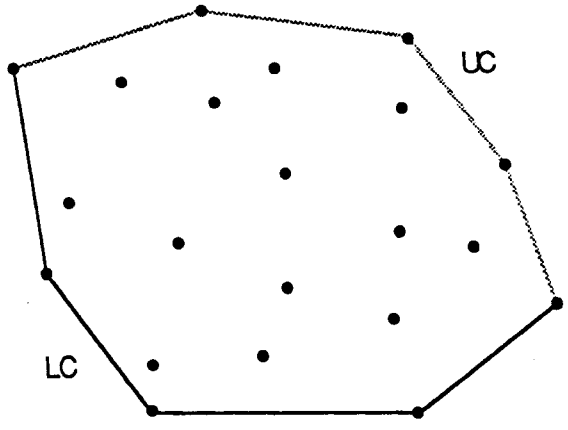

Figure 2: Convex chains.

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 [33]. 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 $i := i-1$, i.e., we consider the angle $p_{i-2}p_{i-1}p_{i+1}$. Again we check whether it is convex. If it is we can continue with p_{i+2}. Otherwise we throw away p_{i-1} and backup. It is easy to see that is this way we find the upper convex chain. This takes time $O(n)$ because at each step we either consider the next point or we throw away a point. This leads to the following result:

Theorem 2.1 *The convex hull of a set of n points in the plane can be computed in time $O(n \log n)$. When the points are sorted by x-coordinate the problem can be solved in linear time.*

This result is worst-case optimal, i.e., any convex hull algorithm takes time $\Omega(n \log n)$ in the worst case (see Yao [82]).

The following algorithm which is due to Jarvis [39] has a worst-case running time of $O(n^2)$ but works very good when the number of points on the convex hull is small (which is normally the case). Moreover, the method is very simple.

Let p_0 be the bottommost point in the set V. Clearly p_0 is part of the convex hull. Now let l be a horizontal line through p_0. We rotate l counterclockwise around p_0 until it hits a point p_1 of the set V. p_1 can be found by simply checking all points p_i and looking at the angle between p_0p_i and l. p_1 will also lie on the convex hull. Now we continue rotating l but this time around p_1. In this way we find a point p_2 that lies on the convex hull. Next we rotate around p_2, etc., until we finally get back to p_0. This algorithm is often called the gift-wrapping algorithm because we wrap a line around the set of points.

Theorem 2.2 *The convex hull of a set of n points can be found in time $O(k.n)$ where k is the number of points on the convex hull.*

Proof. For each next point on the convex hull we have to spend $O(n)$ work on checking all points in the set. The bound follows. □

When the points are distributed in some uniform way the following algorithm of Overmars and van Leeuwen [63] based on an idea of Bykat [10] is useful. First determine the highest, bottommost, leftmost and rightmost point. Let these points be p_h, p_b, p_l and p_r. These points clearly belong to the convex hull. Now all points lie inside the axis-parallel rectangle R with p_h on the top boundary, p_b on the bottom boundary, etc. R is suddivided in five regions: V_{ins} being the central quadralateral and four triangle $V_1 ... V_4$. See figure 3. Clearly, all points in V_{ins} do not belong to the convex hull and can be removed. So we are left with the points in the four triangles $V_1 ... V_4$.

Now consider such a triangle V' with bottom line $\overline{p_ip_j}$. Find the point p_k in the triangle furthest away from $\overline{p_ip_j}$. This point must belong to the convex hull. This splits the triangle in four regions V'_{out} that lies further away than p_k, V'_{ins} being the triangle $\Delta p_ip_jp_k$ and the two other triangle V'_1 and V'_2. See figure 4. V'_{out} must be empty. The points in V'_{ins} cannot lie on the convex hull. So we are left with two new triangles V'_1 and V'_2 that are treated in the same way. Clearly, at each step of the algorithm a new point of the convex hull is found. Each such step takes time $O(n)$ in the worst case, so the worst-case total time is $O(k.n)$. (And indeed the method might take that many steps if every time all points lie in one of the remaining triangles.) But the expected time bound is a lot better. Note that the area of V_{ins}, the part that is removed in the first step, is of the same size as the remaining parts. Similar, when treating a triangle, the area of V'_{ins} is larger than the sum of the areas of V'_1 and V'_2. Hence, when the points are uniformly distributed in a convex region, the expected number of points remaining after each step of the algorithm is at least halved. This leads to an expected time of $O(n)$.

Theorem 2.3 *Given a set of n points uniformly distributed in a convex region of the plane, the convex hull can be found in expected time $O(n)$.*

Rather than taking convex hulls of sets of points, one can as well compute convex hulls of objects. An important case is the convex hull of a non-convex polygon. It is simple to see that the convex hull of a polygon is equal to the convex hull of the set of vertices of the polygon. Hence, we can compute the convex hull in the ways described above, yielding a worst-case time bound of $O(n \log n)$.

But one can do better. Using a clever way of walking around the polygon, starting at a vertex that lies on the convex hull, the convex hull can be wrapt around the polygon

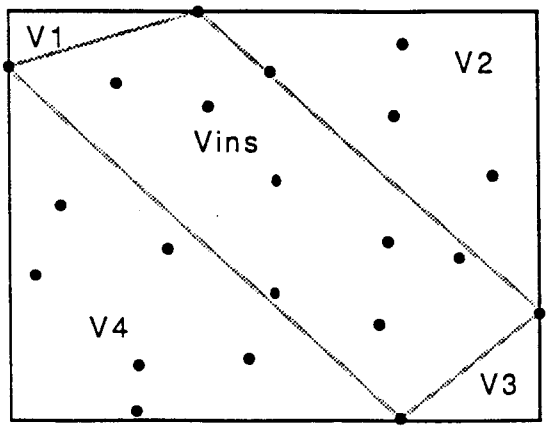

Figure 3: The initial situation.

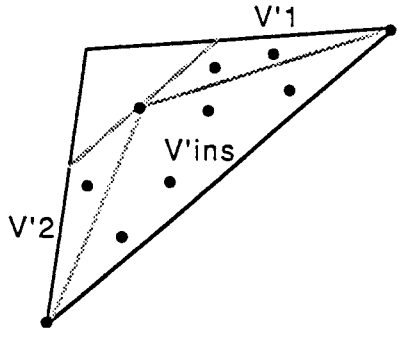

Figure 4: Treating a triangle.

in linear time. This technique was designed by Lee [50] (see also [67] section 4.1.4 for a description). Another technique has been developed by Graham and Yao [34]. We just state the result.

Theorem 2.4 *Given a simple polygon in the plane with n vertices, the convex hull can be determined in time $O(n)$.*

2.2 Triangulation

As a second shape problem we consider the problem of subdividing a pointset or a polygon into small simple pieces, in our case triangles. In the case of a polygon we want to subdivide the interior into triangles using only the vertices of the polygon as vertices of triangles. When triangulating a set of points we want to subdivide the convex hull of the set into triangles such that all points are used as vertices of triangles. See figure 5 for examples of triangulations of a polygon and of a set of points. (Note that triangulations are not unique. The same polygon or set of points can be triangulated in many different ways.)

Triangulating a polygon or a pointset plays an important role in many applications. For example, once a polygon is triangulated many problems can be solved more efficiently.

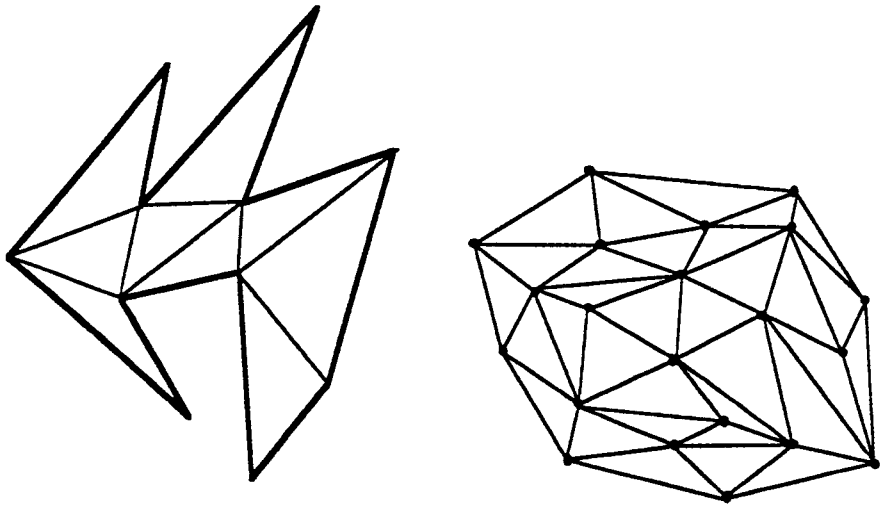

Figure 5: Triangulations.

In particular one can easily determine whether a point lies inside or outside the polygon. Triangulating sets of points is in particular important for interpolating information. When the points are places where some value is computed one can use a triangulation to decide between which values to interpolate when the value at another position is required. Simply determine the triangle the point lies in and interpolate between the vertices of the triangle.

To triangulate a set of points we use a scanline algorithm. We move a scanline from left to right over the set of points. At any moment we will take care that we have the subset to the left of the scanline triangulated. Moreover we maintain the convex hull of the points passed. Now assume the scanline stops at a point p_i. Let p_{i-1} be the last position where the scanline did stop. Connect p_i with p_{i-1}. p_{i-1} will lie on the convex hull of the points passed. Now walk along this convex hull starting at p_{i-1} in both directions connecting p_i with the points on the convex hull, until we reach the tangents from p_i to the convex hull. In this way we have updated the triangulation and the convex hull.

The amount of time spend is an initial $O(n \log n)$ to sort all the points by x-coordinate plus for each step an amount of work in the order of the number of edges added to the triangulation. As the triangulation is a planar graph the number of edges is linear and, hence, after sorting, we spend only $O(n)$ time in total.

Theorem 2.5 *Given a set of n points in the plane, they can be triangulated in time $O(n \log n)$. If the points are sorted by x-coordinate the method takes only linear time.*

Note that this also gives us an alternative method to computed the convex hull of a set of points.

Computing a triangulation of a polygon is more difficult. One way of solving the problem is to use a more general method that produces a so-called *constrained triangulation.* Here we are given a set of line segments (non-intersecting) and ask for a triangulation of the set of endpoints where the line segments must be edges of the triangulation. (This will also triangulate the outside of the polygon but this part can easily be removed.) We will only describe the basic ideas behind the method. For details see e.g. [67] section 6.2.2.

The method consists of two parts. First we split the region into monotone polygons that consist of two vertically monotone chains of edges, i.e., any horizontal line intersects the polygon in one segment. Next each of these subpolygons is triangulated. This triangulation can be done in linear time during one simultaneous walk along the two monotone chains (see [31]).

To form the monotone polygons we first compute the convex hull of all the vertices. These will clearly be edges of the triangulation. Next we use a scanline technique. We move a scanline from the bottom to the top, stopping at each vertex. At any moment the scanline intersects a number of edges. For each part between two such edges we maintain the highest vertex in this part. Now assume we reach some vertex p. If p has both edges running downwards and upwards we simply adapt the information at the scanline and continue. If p has no edges above the scanline we also simply adapt the information at the scanline. If p has only edges above the scanline we connect p with the highest point in the interval p lies in. Next we adapt the information. It is easy to see that this takes time $O(n \log n)$ in total. After this step each vertex will have edges going downwards. In a second sweep, but this time from top to bottom we take care that each vertex also gets edges going upwards. As a result we will have only monotone polygons left.

Theorem 2.6 *Given a polygon with n edges, the inside of P can be triangulated in time $O(n \log n)$. If P is monotone the algorithm takes time $O(n)$.*

This is not the best possible. Tarjan and Van Wijk [75] have given a very complicated algorithm that runs in time $O(n\text{loglog}\, n)$. The question whether an $O(n)$ algorithm exists is still open.

2.3 Further reading

The most efficient convex hull algorithm for a set of points know today is by Kirkpatrick and Seidel [46] and runs in time $O(n \log k)$ where k is the number of points on the convex hull.

Algorithms for dynamically maintaining convex hulls under insertions and deletions of points can be found in [65, 59].

A generalization of convex hulls are the *alpha hulls* as defined by Edelsbrunner, Kirkpatrick and Seidel [24]. Alpha hulls resemble the shape of the set of points much better than convex hulls, allowing e.g. holes.

Many other triangulation methods for pointsets do exist, all obtaining triangulations with different properties. See section 3.2 for a different method.

There are also many different ways of subdiving polygons in smaller pieces. Sometimes additional vertices (called *Steiner points*) are allowed to obtain decompositions with particular properties. See e.g. Keil and Sack [43] or O'Rourke [58] for a number of different techniques.

3 Proximity

A number of problems in computational geometry deals with notions of *distance*, in other words, they are proximity problems. For example, one might ask for the two points in a set that lie nearest to one another (important in e.g. checking VLSI design constraints) or the points farthest away from one another. Another problem asks to store a set of points or other objects such that for a given query point the point in the set nearest to

this query point can be determined efficiently. This plays an important role in planning shortest routes for robots, picking graphical objects, finding the object in a database that satisfy some constraints as good as possible, etc.

In this section we will introduce the Voronoi diagram that can be used for solving many types of proximity problems. Next we will look at point location techniques that are neccessary to be able to use Voronoi diagrams but can be used in many other applications as well.

3.1 Voronoi diagram

Let V be a set of n points in the plane. Now subdivide the plane in n regions $R_1 ... R_n$ such that R_i consists of those points q for which $p_i \in V$ is the nearest neighbor. The subdivision we obtain is called the *Voronoi diagram* of the set of points, named after Voronoi [76]. See figure 6 for an example.

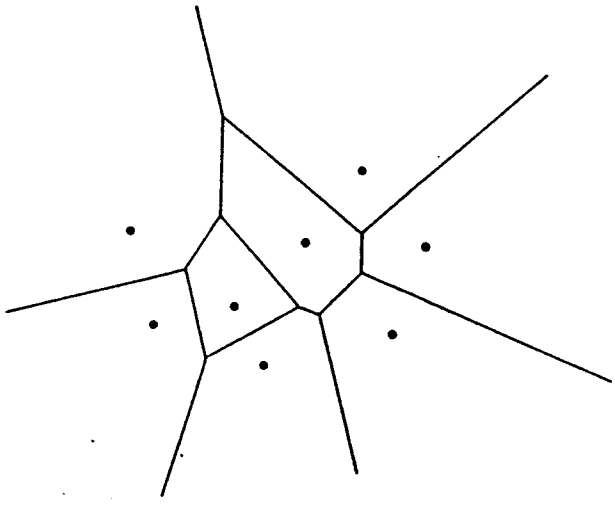

Figure 6: A Voronoi diagram.

A Voronoi diagram has a number of interesting properties. All regions are convex and the region R_i contains p_i and no other point of V. Some of the regions are unbounded. These correspond to the points on the convex hull of V. An edge of the Voronoi diagram is a part of the bisector between the points in the two region on both sides of the edge. The number of edges and nodes in the Voronoi diagram is linear. When no three points lie on a circle the nodes have degree 3. A node corresponds to a point that lies on the same shortest distance from three points in V.

Shamos and Hoey [73] were the first to realize that the Voronoi diagram is a powerful tool in computational geometry. They gave the first algorithm to compute the Voronoi diagram of n points in time $O(n \log n)$. The basic idea of the method is a divide and conquer approach. We will only briefly sketch the algorithm. For details see e.g. [67] chapter 5. We split the set of points with a vertical line in two equal sized subsets V_1 and V_2 and recursively construct the Voronoi diagram VD_1 of V_1 and VD_2 of V_2. To merge the two Voronoi diagrams, note that all edges in VD are either edges of VD_1 or of VD_2 or are bisectors between points in V_1 and V_2. These new edges will form a chain that is monotone in the y-direction. We first determine the half-infinite edges that have to be added. To this end we compute the two tangents to the convex hulls of V_1 and V_2 and take the perpendicular bisectors of these tangents. Let l be the top bisector. l will lie in a

region R_i of VD_1 and a region R_j of VD_2. Hence, l is the bisector of p_i and p_j. Determine the first place where l intersects the boundary of R_i or R_j. When l first intersects R_i determine the new region R_k of VD_1 we come in. Adapt l and make it the bisector of p_k and p_j. Similar when we hit a region of VD_2 first. In this way we continue until we reach the bottom half-infinite ray. The new edges can be calculated in time $O(n)$ total.

Theorem 3.1 *The Voronoi diagram of a set of n points in the plane can be determined in time $O(n \log n)$.*

As a first application consider the nearest neighbor problem in which we are interested in finding the two points in the set that lie nearest together. It is easy to see that the Voronoi regions of the two nearest points must share an edge. Hence, we simply check for each edge the distance between the two points on both sides and take the minimum of all pairs. This clearly takes $O(n)$ once the Voronoi diagram has been constructed because the number of edges in the Voronoi diagram is linear.

In fact, for each point p in the set, the nearest neighbor of p must share an edge with p. Hence, in the same way, we can find the nearest neighbor for each point in V.

Theorem 3.2 *Let V be a set of n points in the plane. One can determine for each point in V its nearest neighbor in V in time $O(n \log n)$.*

Using Voronoi diagrams is very practical. The algorithms to construct them are not hard to implement (although they might look that way). Guibas and Stolfi [36] show how to do this implementation. Another approach is to use a quite different method (using the plane sweep technique) that was designed by Fortune [30].

3.2 Delaunay triangulation

Let V be a set of points. We call two points *neighbors* when their Voronoi regions share an edge. Now assume that no four points in V lie on a circle. As a result, if we connect all neighbors with straight line segments we get a triangulation of the pointset. This triangulation is called a *Delaunay triangulation* after Delaunay who studied such triangulations already a long time ago ([18]).

Theorem 3.3 *The Delaunay triangulation of a set of points can be constructed in time $O(n \log n)$.*

The Delaunay triangulation has a number of interesting properties. The most important one is probably that the circle with the three vertices of a triangle of the triangulation on its boundary contains no other point of the set. This follows from the fact that this circle has one of the vertices of the Voronoi diagram as its center and, hence, this center lies on the same minimal distance to the three points. This property makes the Delaunay triangulation useful in interpolation applications (see section 2.2).

3.3 Point location

One important proximity problem is the *nearest neighbor searching problem* (also called the post office problem): Given a set of points (post offices) V store them such that for a given point p we can efficiently determine the point (post office) nearest to p.

To solve this one could construct the Voronoi diagram of the set V. Now the problem reduces to: given a subdivision of the plane in convex regions, store this subdivision such

that for an arbitrary point p we can efficiently determine the region p lies in. Storing with each region the nearest neighbor, this immediately solves the post office problem.

Searching in a planar subdivision, also called *point location*, is a very general problem and has many other applications as well. For example, to determine whether a point lies inside a non-convex polygon one can triangulate the polygon and perform point location on the set of triangles. Many techniques have been developed to solve the problem. The simplest one, called the slab method, draws a vertical line through each vertex of the subdivision. In this way we obtain $O(n)$ vertical slabs. We construct a balanced tree on the x-coordinates of the vertical lines to be able to determine for each point in which slab it lies. Next, for each slab, we construct a balanced tree storing all the edges that intersect the slab from top to bottom. Searching in this tree we can determine between which two edges a point lies and this tells us in which region the point lies. The query time will be $O(\log n)$ but the preprocessing time and amount of space required are $O(n^2)$.

The first optimal point location technique requiring $O(n)$ storage with a query time of $O(\log n)$ was given by Kirkpatrick [44]. The method is based on *triangular refinement.* First we triangulate all the regions. Hence, we are left with a subdivision of triangles. Now we remove a number of independent vertices and their edges and retriangulate the holes that appear. In this way we get a new triangulation with less triangles. We continue in the same way until we are left with one big triangle. In [44] it is show that this can be achieved in $O(\log n)$ steps. The different levels of triangulation look a lot alike. As a result, if one knows the position of a point in one level one can find in constant time the position of the point in the next level. To perform a query with a point p we start at the smallest triangulation. From here we step to the next triangulation and continue until we reach the original triangulation. Because each step takes $O(1)$ time and there are $O(\log n)$ levels the query time becomes $O(\log n)$. For details see [44] or [67] chapter 2.2.2.

Theorem 3.4 *Given a planar subdivision consisting of convex polygonal cells with a total of n edges, one can store the subdivision using $O(n)$ storage such that, given a point p, one can determine in time $O(\log n)$ the cell p lies in.*

From this the following result immediately follows.

Theorem 3.5 *One can store a set of points in the plane using $O(n)$ storage such that a nearest neighbor query can be answered in time $O(\log n)$.*

Although theoretically optimal the constants in the triangular refinement method are very high which makes the algorithm unpractical. Hence, people have worked on designing other techniques that are also fast in practice. The best method known today is probably the technique of Edelsbrunner, Guibas and Stolfi [23] (see also [22] chapter 11).

3.4 Further reading

Many different types of Voronoi diagrams do exist. The diagram presented in section 3.1 is just the simplest one. First of all one can usedifferent distance functions. Lee and Wong [52] showed how to compute Voronoi diagrams for the L_1 and L_∞ metric. Lee [48] generalized this to L_p metrics. Chew and Drysdale [13] generalized this further to *convex distance functions.*

A second generalization is to allow other types of objects instead of points. Lee and Drysdale [51] introduced Voronoi diagrams of line segments. Yap [82] extended this further to curved segments.

Another direction of research involved *higher order Voronoi diagrams.* A kth-order Voronoi diagram divides the plane into regions in which the point of the set that is the kth nearest is the same. So the first-order Voronoi diagram is the standard Voronoi diagram. On the other hand, in the nth-order Voronoi diagram we split the plane in regions such that the furthest point is the same. (This diagram is also called the *furthest point Voronoi diagram*). First algorithms to construct kth-order Voronoi diagrams were given by Lee [49]. Better results where obtained by Chazelle and Edelsbrunner [11] (see also [22]).

A survey on Voronoi diagrams together with a large bibliography can be found in Aurenhammer [1].

4 Windowing

The *windowing problem* asks for computing the part of a picture (or 2-dimensional scene) that lies inside a given axis-parallel rectangle. In database applications the problem is often refered to as the *range searching problem.* Many data structures have been proposed for storing a scene such that windowing with different windows can be performed efficiently. The type of data structure and efficiency highly depend on the type of objects in the scene. We will first look at the (in graphics unrealistic) situation where the scene consists of a collection of n points only. Later we will consider scenes of line segments and other objects.

An important property of data structures is whether they are static or dynamic. *Static* data structures are once constructed for a fixed set of objects and can be used only for queries with different windows. *Dynamic* data structures allow for both queries and insertions and deletions of objects. Dynamic data structures are much more flexible and are often required in graphics applications.

4.1 k-d trees

One of the data structures used very often in graphics is the *quad tree.* This structure, originally proposed by Finkel and Bentley [29] but in fact already known for a long time in computer graphics, recursively splits the set of objects in four subsets, corresponding to four quadrants of the plane. In graphics the structure is mostly use as an image space data structure. Here the splitting continues until the regions have the size of one pixel. Quad trees can also be used as object space data structures. In this case one continues splitting the plane (and, hence, the set of object) until the subsets get some small size.

In object space applications quad trees tend to become quite unbalanced. To avoid this another structures, called a *k-d tree* was introduced by Bentley [2] (see also [3]).

A 2-dimensional k-d tree is a binary tree that is constructed as follows: Let V be a set of points in the plane. Choose a point $p \in V$. $p = (p_1, p_2)$ will be stored in the root of the tree. Now split the set V in two subsets V_1 of points with x-coordinate $\leq p_1$ and V_2 of points with x-coordinate $> p_1$. V_1 will be stored in the left subtree of p and V_2 in the right subtree. Now in each subset again take a point. This will become the root of the subtree. Again split the set in two halves but this time with respect to the y-coordinate. On the next level of the tree we again split with respect to x-coordinate, etc. We continue this way until a subset contains only one point. This point we store in a leaf. See figure 7 for an example of a k-d tree.

The main advantage of k-d trees over quad-trees is that we always can split the set in an optimal way by taking as a splitting point the median for the particular coordinate of

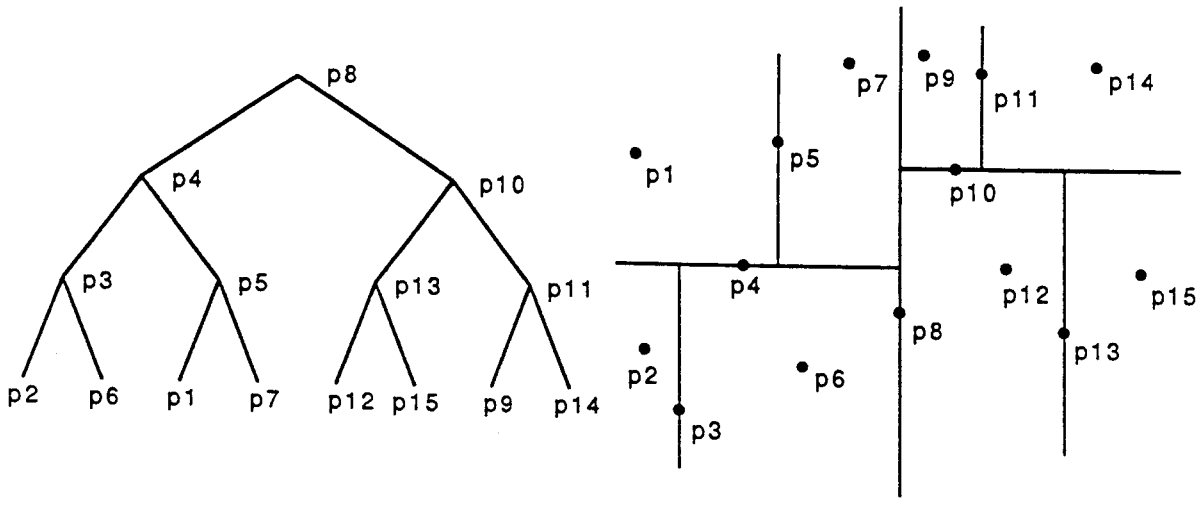

Figure 7: A k-d tree.

all the points. This means that we can always construct k-d trees of depth $\lceil \log n \rceil$. The construction can easily be carried out in time $O(n \log n)$.

To perform a windowing query on a k-d tree we search with the window in the tree. Note that each node in the k-d tree corresponds to a part of the plane as shown in figure 7. Now assume we are at some node v in the tree. First we check whether the point stored at v lies in the set. If so we report it. Next we look at the cell C of the plane corresponding to v. If the window completely covers C all points in the subtree below v lie in the window and we report them all (by traversing the subtree). If the window does not intersect C we don't have to continue in this subtree. Otherwise we continue the search at both sons of v (unless v is a leaf).

It can be shown that such a windowing query will take time at most $O(\sqrt{n}+k)$ where k is the number of answers found. This is based on the fact that any horizontal or vertical line intersects at most $O(\sqrt{n})$ cells of nodes. This leads to the following result:

Theorem 4.1 *There exists a structure, the k-d tree, for storing n points that can be constructed in time $O(n \log n)$, such that a windowing query takes time $O(\sqrt{n}+k)$. The structure uses $O(n)$ storage.*

One of the problems of k-d trees (and of quad trees) is that they are static. Insertion and deletion routines such as the ones for normal binary trees do not work on k-d trees because it is impossible to perform rotations in the tree. A way to avoid this is to redefine k-d trees in a different way.

Kreveld and Overmars [47] define a so-called *divided k-d tree*. Let V be a set of points in the plane. Sort the points on x-coordinate. Now divide this sorted set in $\sqrt{n}$ groups of each $\sqrt{n}$ points. We construct a simple balanced binary search tree on x-coordinate with each leaf corresponding to one of these groups. This tree is called the top tree. For each group we construct a balanced binary search tree on y-coordinate. These are called the bottom trees. So in the top tree we split on x-coordinate and in the bottom trees on y-coordinate (in contrast to a normal k-d tree where we split on each level according to a different coordinate). See figure 8 for an example of a divided k-d tree.

Clearly the structure takes $O(n)$ storage and can be constructed in time $O(n \log n)$. To perform a window query, let the query window be $([A_1 : B_1], [A_2 : B_2])$, i.e., we want to find all points with x-coordinate between A_1 and B_1 and y-coordinate between A_2 and

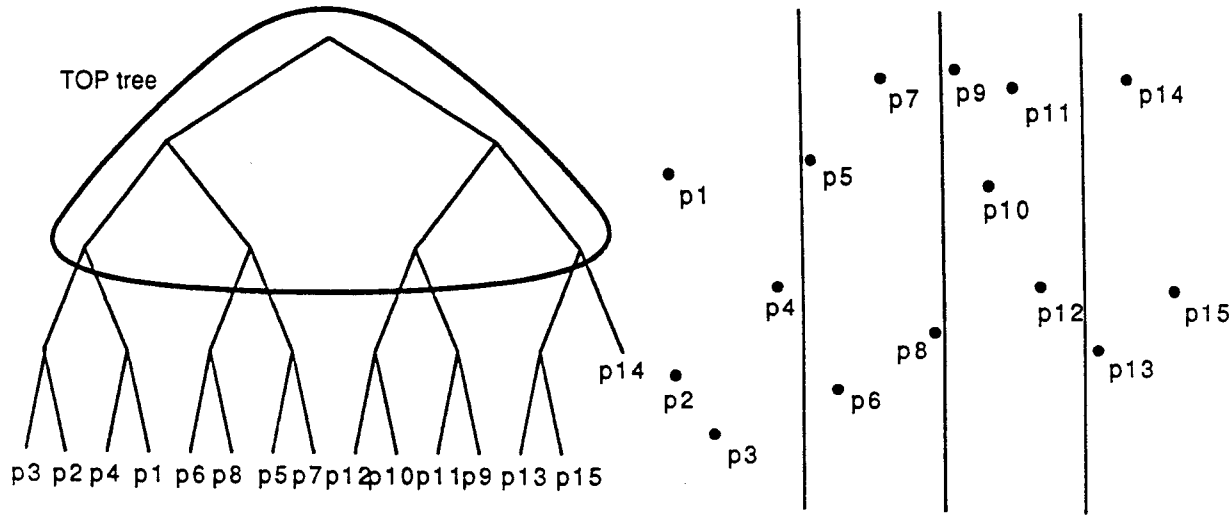

Figure 8: A divided k-d tree

B_2. We search with both A_1 and B_1 in the top tree. Let the leaves we end in be δ_1 and δ_2. For each leaf between δ_1 and δ_2 we know that all points stored in the corresponding bottom tree have x-coordinate between A_1 and B_1. Hence, in each of these bottom trees we simply search with A_2 and B_2 and report all points with y-coordinate between them (remember that the bottom trees are sorted on y-coordinate). Now we are left with the subtrees below δ_1 and δ_2. For the points in these subtrees we do not know whether their x-coordinate lies in the range. So we search in them with A_2 and B_2 but for each point found we also check whether it lies between A_1 and B_1 with respect to its x-coordinate.

Theorem 4.2 *Divided k-d trees of n points can be constructed in time $O(n \log n)$ and use $O(n)$ storage. Windowing queries can be performed on them in time $O(k + \sqrt{n} \log n)$, where k is the number of answers found.*

Proof. Searching with A_1 and B_1 in the top tree takes $O(\log n)$ time. We find at most $O(\sqrt{n})$ bottom trees in between the two leaves in which we have to search. Searching such a bottom tree takes time $O(k' + \log n)$ where k' is the number of answers found in this bottom tree. Searching the trees below δ_1 and δ_2 takes at most time $O(\sqrt{n})$ because this is a bound on the size of the bottom trees. □

To be able to perform insertions and deletions efficiently we relax the conditions on the divided k-d tree slightly. We allow the bottom trees to grow as long as their size remains smaller than $2\sqrt{n}$ and also the top tree may get a size of $2\sqrt{n}$. This does not increase the query time in order of magnitude. To insert a point p we search in the top tree to find the correct bottom tree and insert p in this bottom tree. This takes time $O(\log n)$. If the bottom tree remains smaller than $2\sqrt{n}$ we are done. Otherwise we split the bottom tree in two bottom trees each of size $\sqrt{n}$ and insert a new leaf in the top tree. This takes time $O(\sqrt{n})$ but this time can be charged to $\sqrt{n}$ preceeding insertions in the bottom tree at which it did not need to be split. Hence, this is an average of $O(1)$ per insertion. When the top tree gets too big we rebuid the complete tree. This takes time $O(n \log n)$ but this can be charged to n updates that must have taken place since the last rebuilding. Hence, the average insertion time is $O(\log n)$. Deleting a point p is even simpler. We search in the top tree to find the bottom tree that contains p. Next we delete p from this bottom tree and we are done. (When the number of deletions gets too big we have to rebuild the

whole tree to maintain the balance conditions. See [47] for details.) Hence, the deletion time is also $O(\log n)$.

The structure can in fact be improved slightly. Rather than splitting into $\sqrt{n}$ groups we had better split into $\sqrt{n/\log n}$ groups, each of size $\sqrt{n \log n}$. It can easily be seen that this improves the query time to $O(k + \sqrt{n \log n})$.

Theorem 4.3 *In a divided k-d tree updates can be performed in amortized time $O(\log n)$. A windowing query can be performed in time $O(k + \sqrt{n \log n})$ where k is the number of answers found. The structure can be constructed in time $O(n \log n)$ and uses $O(n)$ storage.*

When the window is small the query time is normally a lot better because the number of bottom trees we have to search will be small as well.

4.2 Range trees

We will now concentrate on another structure, called a *range tree*, that has a much faster query time at the cost of a slight increase in storage. The structure was designed independently by a number of researchers (see e.g. [4, 53, 78, 80]).

Let us first consider the one-dimensional problem. So we are given a set of points on the real line and we want to store them such that for a given "window" $[A_1 : B_1]$ we can efficiently determine the points lying in between A_1 and B_1. To this end we store the points in the leaves of a balanced binary search tree, in sorted order. Moreover we link the leaves in this order in a double linked list. The structure clearly uses $O(n)$ storage. We call this structure a one-dimensional range tree.

To perform a query with window $[A_1 : B_1]$ we search with A_1 in the tree to locate the smallest point $\geq A_1$. Let this point be K. If K does not exist or $K > B_1$ we are done. Otherwise, we report K and look at the right neighbor of K in the tree and repeat the test here. It is clear that such a query takes time $O(\log n)$ to search with A_1 plus the number of answers found.

Theorem 4.4 *Given a set of n points on the real line, the one-dimensional windowing problem can be solved in a query time of $O(k + \log n)$ using $O(n)$ storage, where k is the number of answers found.*

Now let us look at the two-dimensional problem. So V is a set of points in the plane. A two-dimensional range tree of V has the following form. We sort all point with respect to their x-coordinate and store them in the leaves of a balanced binary search tree in this order. To each internal node δ we associate a one-dimensional range tree T_δ of the points in the subtree rooted at δ, on their y-coordinate (see figure 9). Hence, in the main tree points are sorted with respect to their x-coordinate and in the associated structures points are sorted with respect to their y-coordinate.

To perform a query with window $([A_1 : B_1], [A_2 : B_2])$ we search with both A_1 and B_1 in the main tree. For some time A_1 and B_1 will follow the same search path but at some internal node δ A_1 will go to the left son and B_1 will go to the right son. Let $\beta_1 \ldots \beta_i$ be the nodes that are rightson of a node on the search path of A_1 below δ and do not lie on the search path themselves. Silmilar, let $\gamma_1 \ldots \gamma_j$ be the nodes that are leftson of a node on the search path of B_1 below δ that do not lie on the search path themselves. See figure 10 for the situation.

All points with x-coordinate between A_1 and B_1 lie in exactly one of the subtrees rooted at the β and γ nodes. Of these points we have to find the ones whose y-coordinate

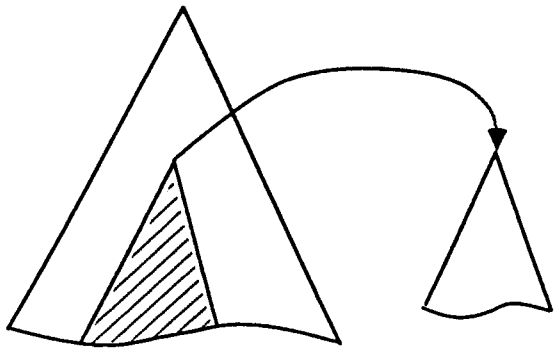

Figure 9: Associated trees.

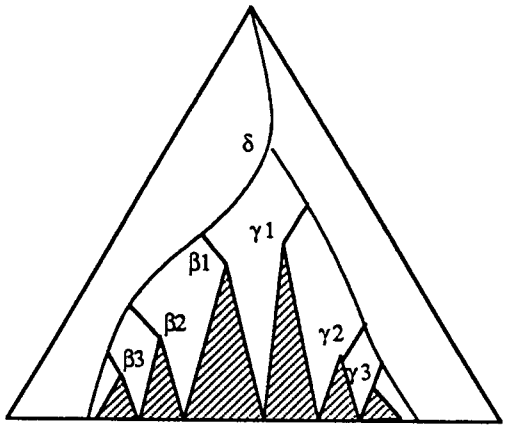

Figure 10: The β and γ nodes.

lies between A_2 and B_2. For this we use the structures $T_{\beta_1} \ldots T_{\beta_i}$ and $T_{\gamma_1} \ldots T_{\gamma_j}$. All points in the x-range are stored in exactly one such structure and all points in these structures lie in the x-range. On each of these T-structures we perform a one-dimensional windowing query with window $[A_2 : B_2]$.

Theorem 4.5 *Given a set of n points in the plane, the two-dimensional windowing problem can be solved in a query time of $O(k + \log^2 n)$ using $O(n \log n)$ storage, where k is the number of answers found.*

Proof. Let us first consider the amount of storage required. The main tree uses only $O(n)$ storage. On each level of the main tree each point in V occurs exacly once in an associated structures. As an associated structure of n' points uses storage $O(n')$, the associated structures on one level together use $O(n)$ storage. As the tree has $O(\log n)$ levels the storage bound follows.

To perform a query we first seach in the main tree with A_1 and B_1 which takes time $O(\log n)$. In this way we locate $O(\log n)$ T_β and T_γ structures that each have a query time of at most $O(\log n)$ plus the number of answers found. Hence, the total query time is bounded by $O(\log^2 n)$ plus the total number of answers found. □

The two-dimensional range tree can be made dynamic but the update algorithms are quite complicated (see e.g [78, 80]).

4.3 Segment trees

Range trees are only capable of storing points. In many applications, especially in computer graphics, objects are not points but, for example, rectangles, line segments, etc. In this section we will treat a second data structure, the segment tree, that can, in combination with e.g. range trees, be used for storing other geometric objects.

The *segment tree* was introduced by Bentley and Wood[8] to solve some rectangle problems. Although used for solving many two-dimensional problems, the segment tree is a one-dimensional data structure.

Let $V = \{[a_1 : b_1], [a_2 : b_2], \ldots, [a_n : b_n]\}$ be a set of n intervals (segments) on the real line. (Intervals are allowed to intersect, overlap, etc.) All left and right endpoints of the intervals we sort. Let the sorted list be $x_1, x_2, \ldots, x_{2n}$. These endpoints split the real line in a number of so-called elementary intervals $(-\infty : x_1), [x_1 : x_2), [x_2 : x_3), \ldots, [x_{2n} : \infty)$. These elementary intervals we store in the leaves of a balanced binary search tree. With each internal node δ we associate the interval I_δ that is the union of the intervals associated to the sons of δ. (In other words, I_δ in the interval spanned by the subtree rooted at δ.) With each node δ we store a structure T_δ containing all intervals in V that contain I_δ but do not contain $I_{\text{father}(\delta)}$. The form of T_δ depends on the problem we want to solve using the segment tree. See figure 11 for an example of a segment tree. With each node δ the intervals stored in T_δ are indicated.

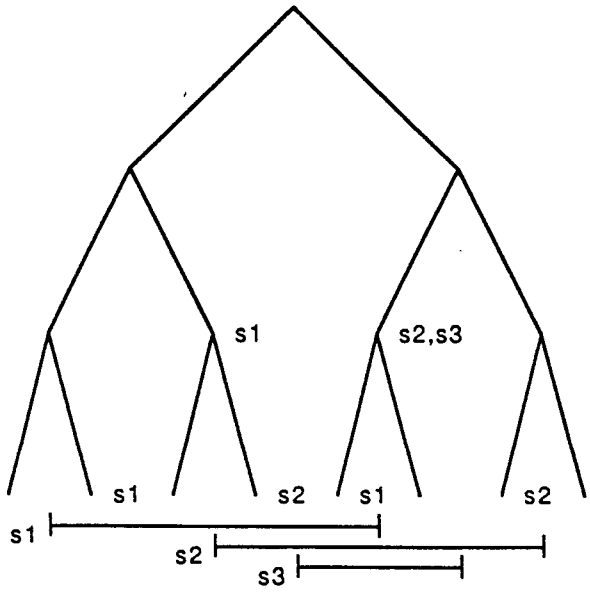

Figure 11: A segment tree.

The segment tree is mostly used for so-called *stabbing queries*. Let p be a point on the line, we want to report all intervals that contain p. To solve this problem we structure T_δ as a simple list of the intervals. To find all intervals containing p, we search with p in the tree, starting at the root. Assume we are at some node δ. We report all intervals stored in T_δ. Next we look at the two sons δ_1 and δ_2 of δ. When p lies in I_{δ_1} we continue the search in δ_1. Otherwise we continue the search in δ_2.

The correctness of the algorithm can be seen as follows. Clearly all intervals reported contain p, because for each node δ visited, p lies in I_δ and the reported intervals contain

I_δ. No interval is reported more than once, because on each path from the root to a leaf an interval can occur at most once. Finally, if interval i contains p it clearly contains the elementary interval stored in the leaf p ends in during the search. As i does not contain I_{root} there must be some node δ on the search path towards this leaf such that i contains I_δ but not $I_{\text{father}(\delta)}$. i will, hence, be reported when the search passes this node. (In fact, this is not completely correct. When p is the right endpoint of i, i will not be found. This case can easily be solved by either treating endpoints in a special way, or by adding some extra information in the segment tree.)

Theorem 4.6 *Given a set of n intervals, they can be stored in a segment tree, using $O(n \log n)$ storage, such that stabbing queries can be answered in time $O(k + \log n)$ where k is the number of reported answers.*

Proof. The query time follows immediately from the above discussion. It remains to prove the bound on storage required. On each level of the segment tree, each interval can be stored at at most 2 nodes. (This follows from the definition.) Hence, at each level the T-structures together use at most $O(n)$ storage. The bound follows. □

Segment trees are (depending on the choice of T_δ) dynamic. See e.g. [8] for details.

In the same way as in range trees, we can construct two-dimensional segment trees. Let V be a set n of axis-parallel rectangles in the plane. (For example, bounding boxes of a set of graphical objects.) To store them, we project all rectangles on the x-axis. In this way we obtain a set of intervals. These intervals we store in a segment tree. For each node δ we have a number of intervals that have to be stored in T_δ. These intervals correspond to rectangles. These corresponding rectangles we project on the y-axis and we make T_δ into a segment tree that stores these y-intervals. In T_δ we store at each internal node a list of rectangles to which the y-intervals, that should be stored there, belong. It is easy to see that the structure obtained requires $O(n \log^2 n)$ storage.

A two-dimensional stabbing query asks for all rectangles that contain a given point p. In graphics, this is for example important when picking an object. To perform a stabbing query we use the two-dimensional segment tree. Let $p = (p_1, p_2)$. We search with p_1 in the main segment tree. The nodes on the search path together contain all rectangles that contain p_1 in the x-projection. Among them we have to find those whose y-projection contains p_2. To this end we use the T-structures. For each node δ on the search path of p_1 we search with p_2 in T_δ, reporting all rectangles stored at nodes on the search path of p_2. As we have to query $O(\log n)$ T-structures, each requiring time $O(\log n)$ plus the number of answers found, we obtain the following result:

Theorem 4.7 *Given a set of n axis-parallel rectangles, they can be stored in a two-dimensional segment tree, using $O(n \log^2 n)$ storage, such that stabbing queries can be answered in time $O(k + \log^2 n)$ where k is the number of answers found.*

Now let us return to our windowing problem. But this time let us assume that our picture consists of (non-intersecting) line segments rather than points. For example, a large map stored in a database. We will show how to store the picture in a two-dimensional data structure based on the segment tree such that for each given window we can efficiently determine what part of the picture is visible in the window. (These results are based on work by Overmars [60, 61]).

To solve the windowing problem, we split it in two parts. The first part asks for all line segments that lie completely in the window. The second part asks for all line segments that intersect the boundary of the window.

To solve the first problem we can use the solution for range searching described in the previous section. When a line segment s lies completely in the window surely its left endpoint lies in the window. So we can store all left endpoints in a range tree and perform a range query with the window. This will take time $O(\log^2 n)$. The structure uses $O(n \log n)$ storage.

So we only have to consider the second subproblem: find those line segments that intersect the boundary of the window. We will only look at the top boundary. The other boundaries follow in a similar way.

First we project all line segments on a vertical line. In this way each line segment becomes an interval. Of these intervals we construct a segment tree. All intervals stored at some internal node δ completely cover the interval I_δ associated to δ. I_δ corresponds to a horizontal slab in the plane. All the line segment stored at δ intersect the slab completely and, because they do not intersect each other, appear ordered in the slab. See figure 12. We store them in this order in the internal nodes of a balanced binary search tree T_δ. It is easy to see that the structure requires $O(n \log n)$ storage.

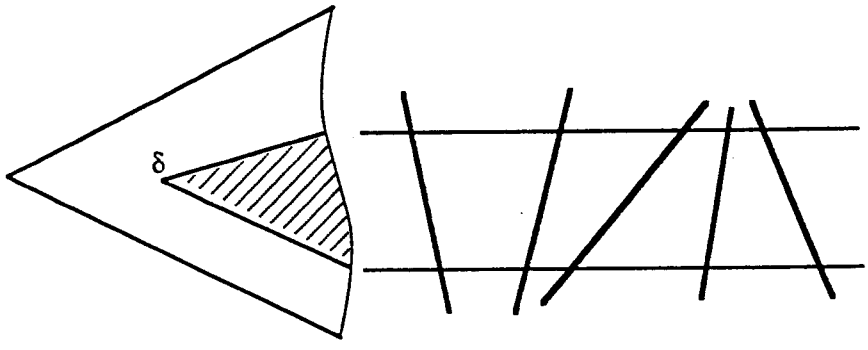

Figure 12: Line segments stored at δ.

To perform a query with the top boundary $\overline{(x_1, y)(x_2, y)}$ we search with y in the segment tree. Clearly, line segments intersecting the top boundary must be stored at one of the nodes on the search path. For each such node δ we search with x_1 and x_2 in T_δ, locating all the segments in between. It is easy to see that this can be done in time $O(\log n)$ plus the number of answers found. As we have to query $O(\log n)$ structures T_δ the total query time is bounded by $O(\log^2 n)$ plus the total number of answers. For details see [60]. It is also shown there that insertions and deletion can be performed in time $O(\log^2 n)$. Combining this with the result for range queries we obtain:

Theorem 4.8 *Given a set of n non-intersecting line segments in the plane, we can store them using $O(n \log n)$ storage such that those k segments visible in a given window can be determined in $O(k + \log^2 n)$ time. The structure is dynamic and can be updated in $O(\log^2 n)$ time.*

Using this method, some of the line segments might be reported more than once. This can easily be avoided by checking whether segments have already been reported before. This can be done in $O(1)$ time per reported segment.

4.4 General scenes

The restriction that the picture consists of line segments only can easily be dropped. The method of the previous section can simply be extended to more general objects. In fact, the only restriction is that they are orthogonal convex. (I.e., any horizontal or vertical line segment with both endpoints inside the object must lie completely inside the object.) And almost all objects can be decomposed in orthogonal convex objects. (Of course, the objects should not overlap or intersect but they are allowed to touch.)

The method now works completely the same. To find the objects that lie completely inside the window we choose a representation point inside each object and perform a windowing query on those points. To find the objects intersecting the top boundary we again project all objects on a vertical line, construct a segment tree on the projections and associate with each node δ a search tree T_δ storing the objects in the slab in their x-order. Queries are now performed in essentially the same way, yielding the same time bounds.

4.5 Further reading

Quad trees and k-d trees have been used in many different applications. For an overview of quad trees and their application see Samet [70]. See [69] for a bibliography of over 250 papers on quad trees and related structures. For dynamization techniques for ordinary quad and k-d trees see [64]. See [59] for an overview of general techniques for dynamizing data structures.

Other types of structures for windowing can be found in [5, 6, 25]. A very general (and tunable) technique can be found in [72]. An interesting related structure is the *priority search tree* introduced by McCreight [55] that is a combination of a search tree and a priority queue and can be used for many types of half-open range queries. In [20] this structure is used as a basis to obtain a very efficient and dynamic structure for range searching.

Many other applications of segments trees exist. See e.g. Edelsbrunner [21]. Here also a related structure, the *interval tree* is introduced. (See also [67].)

5 Intersection problems

Intersection problems form the basis of many types of applications. For example, hidden surface removal algorithms are often based on computing intersections (see the next section). In this section we will discuss some different intersection problems. First we consider the simple case in which all objects are axis-parallel line segments. Next we extend this result to arbitrarily oriented line segments. Finally we show how to treat more general types of objects as well.

5.1 Axis-parallel line segments

Let V be a set of horizontal and vertical line segments in the plane. We are interested in finding all intersections between segments. As the number of intersections can be as large as $(n/2)^2$ any algorithm for this problem will take time $\Omega(n^2)$ in the worst case. But in many applications the number of intersections will be much smaller. Hence, we would like to have an algorithm that runs fast when the number of intersections is small. Such an algorithm is called *output sensitive*.

We will use the scanline technique to solve this problem. We move a vertical scanline from left to right over the set of line segments. At any moment there will be a collection of horizontal line segments that intersect the scanline. These line segments we store, ordered by y-coordinate in a simple balanced search tree T (for example, an AVL-tree). Hence, T allows for insertions and deletions of line segments in time $O(\log n)$. The scanline will stop at each x-coordinate of an endpoint of a line segment. Hence, we first sort all endpoints by x-coordinate to obtain a list of halting points. Now each time the scanline halts there are three possible cases:

- A left endpoint of a horizontal line segment s. In this case we insert s in T.
- A right endpoint of a horizontal line segment s. In this case we delete s from T.
- A vertical line segment s. Let the endpoint be (x, y_1) and (x, y_2). We search with both y_1 and y_2 in T and report all horizontal segments that lie between them. In this way all horizontal segments intersecting s are found in time $O(k' + \log n)$ where k' is the number of answers found.

(When more than one event happens at the same moment some care has to be taken. We first have to perform the insertions, next the queries on T and finally the deletions.)

Theorem 5.1 *Given a set V of n horizontal and vertical line segments, all intersections in V can be determined in time $O(k + n \log n)$ where k is the number of intersections found.*

When we are only interested in the question whether there are intersections the method clearly takes time $O(n \log n)$ because we can stop after the first intersection found.

A similar technique can be used to determine intersections among axis-parallel rectangles in the same time bounds. Only some care has to be taken that rectangles that completely contain some other rectangle are also reported. (See e.g. [67] chapter 8 for details.)

5.2 Arbitrary line segments

We will now drop the restriction that line segments must be axis-parallel. Hence, we have a set of arbitrarily oriented line segments. Again we will try to obtain an output-sensitive method. This algorithm is due to Bentley and Ottmann [7].

The method is also based on the scanline technique. Again we move a scanline from left to right over the plane. At each moment the scanline is intersecting a number of line segments. These line segments we store in the order of intersection (from top to bottom) in a balanced binary search tree T. (Note that we do not store the actual intersection points. They change continuously when the scanline moves so they cannot be maintained.) The tree T changes whenever the scanline encounters an endpoint of a line segment. At a left endpoint the segment must be inserted and at a right endpoint the segment must be deleted. But T also changes at an intersection because two segments will change order in T at such a point. Hence, the scanline must stop at each endpoint and at each intersection. But we don't know the intersections!!

To avoid this problem note that, when the next halting point for the scanline is an intersection between segments s and s', then s and s' must be neighbors in T. Hence, it suffices to keep track of intersection points between neighbors in T. To this end we use a

priority queue Q that stores all positions where the scanline must stop. We initialize Q to contain all endpoints of the line segments. T is initialized to be empty. The algorithm now is a simple loop that, as long as Q is not empty, takes the first element from Q and treats it according to the following cases:

- The point is a left endpoint of a segment s. Insert s in T. Let s_1 and s_2 be the two neighbors of s. If s and s_1 intersect to the right of the scanline, insert the intersection in Q. Similar for s and s_2. If s_1 and s_2 intersect to the right of the scanline, remove the intersection from Q. (They are no longer neighbors in T.)
- The point is a right endpoint of a segment s. Remove s from T. Let s_1 and s_2 be the two neighbors of s. If s_1 and s_2 intersect to the right of the scanline, insert the intersection in Q. They have become neighbors in T.
- The point is an intersection of s and s'. See figure 13 for the situation. Report the intersection. Let s_1 be the other neighbor of s and s_2 the other neighbor of s'. Switch s and s' in T. If s and s_2 intersect to the right of the scanline, insert the intersection in Q. Similar for s' and s_1. (These are the new neighbors.) If s and s_1 intersect to the right of the scanline, remove the intersection from Q. Similar for s' and s_2. (These are no longer neighbors.)

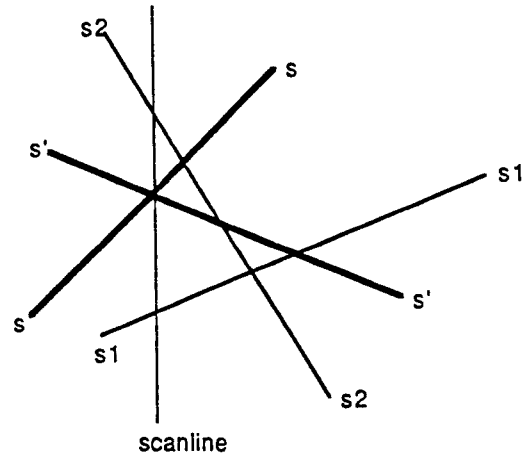

Figure 13: Treating an intersection.

So for each point passed by the scanline we have to do one insertion or deletion on T and at most 2 insertions and 2 deletions on Q. As all these operations take $O(\log n)$ time, each stop of the scanline takes time $O(\log n)$. The number of stops of the scanline is $2n + k$ where k is the number of intersections found. Hence, the total amount of time spend is $O((n + k) \log n)$. Note that the algorithm correctly maintains all intersections between neighbors in Q and, hence, correctly finds all answers. (Some care has to be taken when more points lie on a vertical line or segments are vertical. The second problem can be avoided by choosing a slightly different direction for the scanline to move. The first problem can be solved by first treating the left endpoints, next the intersections and finally the right endpoints, or the other way round if you don't want touching segments to be reported.)

Theorem 5.2 *Given a set V of n arbitrarily oriented line segments in the plane, all k intersections can be determined in time $O((n + k) \log n)$.*

5.3 Other objects

In some applications the objects between which one wants to compute intersection are no line segments. For example they are pieces of curves or triangles. When the objects are curves the method presented above can easily be adapted. The only restriction is that the intersection of the pieces with any vertical line (the scanline) consists of one point only and that intersections between two pieces can be calculated efficiently. If the curves do not satisfy the first requirement they can normally be cut up into smaller pieces that do satisfy the requirement. The method now works in exactly the same way, obtaining the same time bounds.

When the objects are solid, e.g. triangles or circles, some care has to be taken that objects that contain one another are also reported as intersecting. If we assume that the intersection between an object and a vertical line consists of one interval (i.e., the objects are convex with respect to the y-axis) again the same idea can be used. Only this time the tree T must store intervals. For this an *interval tree* can be used (see e.g. [67], section 8.8.1) and we again obtain the same time bounds.

We get a quite different type of problems when we ask for the intersection between two large objects, e.g. two polygons. When the polygons are convex this intersection can easily be computed in time $O(n)$ where n is the total number of vertices. (Note that the intersection will be a convex m-gon with $m \leq n$.) The idea is the following. Let P and P' be the two polygons. We sort all points by x-coordinate. (This can easy be done in time $O(n)$ by splitting the polygons in an upper and a lower half.) Now we move a scanline from left to right. At any moment the scanline will intersect both P and P' in one interval. Let s_1 be the top and s_2 be the bottom segment of P intersected by the scan line (if they exist). Similar s_1' and s_2' for P'. Clearly, the next position where the scanline should halt is either the next vertex in the x-order or the intersection of s_1 and s_1' or the intersection of s_2 and s_2' with the obvious modifications in the segments. (For details see e.g. [67] section 7.2.1.)

Theorem 5.3 *Given two convex polygons with n vertices, their intersection can be determined in time $O(n)$.*

Finding the intersection between two non-convex polygons is a lot harder. Note that the intersection can consist of $\Omega(n^2)$ parts. Hence, we again would like an output sensitive solution. One way of solving the problem is to simply compute all intersections between the boundaries using the technique in section 5.2 and using this information to compute the actual intersection. This would give an $O((n+k)\log n)$ solution for the problem. This can be improved to $O(k + n\log n)$ using so-called red-blue intersection. Mairson and Stolfi [54] show that, given one collection of blue line segments that do not intersect each other and another collection of red line segments that do not intersect each other, the intersections between the two sets can be computed in time $O(k + n\log n)$. Now we can colour the boundary of one polygon blue and of the other polygon red and we used this algorithm to compute the intersections.

Theorem 5.4 *Given two non-convex polygons with n vertices, their intersection can be determined in time $O(k + n\log n)$, where k is the complexity of the intersection.*

5.4 Further reading

For a large number of results on intersecting rectangles and similar object see Edelsbrunner [21]. Recently, Chazelle and Edelsbrunner [12] have improved the bound for finding intersections among arbitrarily oriented line segments to $O(k + n \log n)$.

6 Hidden surface removal

A fundamental problem in 3-dimensional graphics is the *hidden surface removal* problem. We are given a scene of objects in 3-dimensional space and a point or direction of view and want to know which parts of objects are visible. Many algorithms are known for this problem, e.g. the z-buffer algorithm, the depth sort algorithm, the area subdivision method, etc. See Sutherland et. al. [74] for an overview of many different techniques or any textbook on graphics. Unfortunately, all of these techniques are image space methods, i.e., they use the resolution of the screen and are unable to come up with a structural description of the visible scene.

In this chapter we will give two object space algorithms for the hidden surface removal problem. In both algorithms we assume that the objects consist of polygons and that the polygons are not penetrating each other. We assume that the point from which we are looking is along the z-axis at $-\infty$. In other words, the projection is a parallel projection on the xy-plane. Note that this does not impose restrictions because any type of projection and point of view can be reduced to this simple case by applying an appropriate transformation to the objects.

6.1 Projection method

The first method is by Schmitt [71]. It is based on the line segment intersection method presented in the preceeding section. As a first step we project all polygons on the xy-plane and compute all intersections between the boundaries. In this way we obtain a straight-line planar graph in which each node corresponds to an intersection or a vertex of a polygon. See figure 14 for an example. The thick lines form the graph. This graph might not be connected. To this end, and to be able to traverse the graph, we add one long edge *bot* below all the others and for each node that has only edges going to the right we add a so-called drain-edge that goes vertically down until it hits another edge. (These are the dotted lines in figure 14.)

We will only describe how to obtain the visible edges. Visible surfaces can be obtained in a similar way. We traverse the graph, starting at the left endpoint of *bot* and ending at the right endpoint of *bot*. At any moment we keep a tree T that contains all the polygons at the current position, sorted by relative depth. Moreover, we keep a value I that indicates the number of polygons that is covering the current edge. So the edge is visible when $I = 0$. In the beginning $I = 0$. Now we traverse the whole graph from left to right. Assume we follow an edge $e = \overline{pq}$. If $I = 0$ and e is not a drain edge and $e \neq bot$ then we report e as being visible. Now there are a few cases. (We assume for simplicity that no intersection point lies on a vertex or on another intersection point. Hence, every node has degree 2 for a vertex, degree 4 for an intersection and degree 3 when a drain-edge starts or ends here. The method can easily be adapted when this is not the case.)

- q is a vertex (has degree 2). If the other edge runs to the right simply continue along this edge. Don't change I or T. If the other edge runs to the left, stop here.

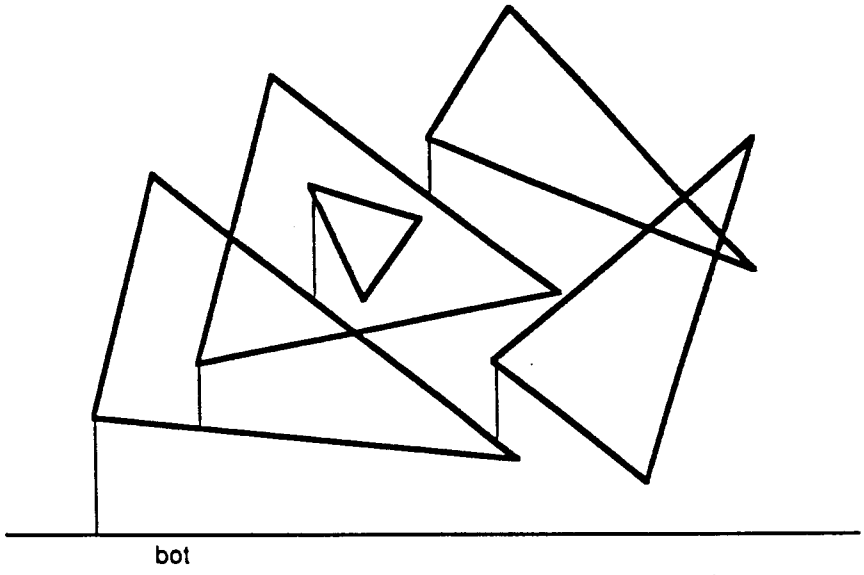

Figure 14: The planar graph with drain-edges.

- q is an intersection (has degree 4). Let s be the edge of the polygon e is part of. Let s' be the edge that is intersecting s and P' be the polygon s' is a boundary of. We either enter P' or leave P'. If we enter P' insert it in T. If P' lies below the current edge, increase I. If we leave P', delete it from T. If P' lies below the current edge decrease I. Now continue along s.

- q is the bottom node of a drain edge. In this case we first continue along the drain edge. Next we continue along the normal edge. (I.e., we make two recursive calls of the procedure.) I and T are not changed.

- q is the top node of a drain edge. Let this node be a vertex of polygon P. We search with P in T to determine the number of polygons that lie below P at q. Set I to this number. Next we recursively call the procedure on both other edges starting at q.

See figure 15 for the four different cases. The arrows indicate the directions in which we continue. It is easy to see that the whole graph is traversed this way and that I and T are maintained correctly. At each node we have to do at most $O(\log n)$ work to insert, delete or search in T.

Theorem 6.1 *Given a set of non-penetrating polygons with a total of n edges in space, the hidden line removal problem can be solved in time $O((n+k)\log n)$ where k is the number of intersections between edges in the projection.*

6.2 Output-sensitive method

A disadvantage of the above method is that the time complexity can be very large even when the output size is small. For example, one big rectangle could be covering all the other objects. The method would still compute all intersections between the projections of all the invisible objects. Hence, again, we would like to have a true output-sensitive algorithm. We will now describe a recently designed algorithm by Overmars and Sharir

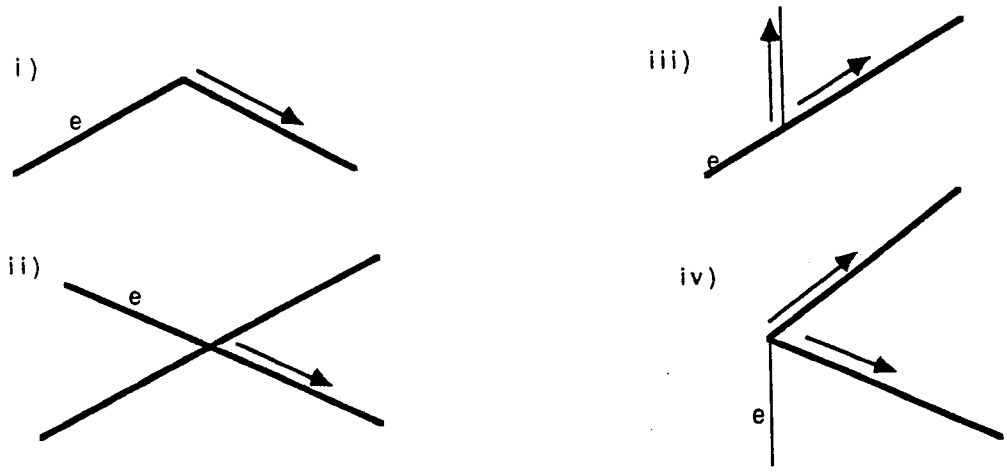

Figure 15: The four cases.

[66] that accomplishes this task. We make the restriction that all objects are triangles (remember that polygons can easily be triangulated), but the method can easily be extended to other polygons, as long as the number of vertices per polygon is limited. Moreover, we assume that the triangles can be ordered by depth, i.e., there is no cyclic overlap. If this is not the case, cyclic overlap has to be removed by cutting up some triangles.

The method is based on the following result:

Theorem 6.2 *Let A be a set of (non-overlapping) polygonal regions in the plane with a total of n_A bounding line segments. Let B be another such set with n_B bounding line segments. The parts of the polygons in A that do not lie inside polygons in B can be computed in time $O((n_A+n_B+k)\log(n_A+n_B))$ where k is the complexity of the resulting set of polygons.*

Proof. This can easily be done by computing all intersections between edges in A and in B and traversing the graph obtained. □

This result can be viewed in the following way. Assume we have two subsets of triangles in space, one V_A lying completely behind the other V_B (in depth). Now assume we compute all visible parts of V_A (not looking at V_B) and all visible parts of V_B. Let A consist of the visible parts of V_A and B the visible parts of V_B. The theorem now says that the visible parts in $A \cup B$ can be computed in the time bound stated.

Let k_0 be some constant. Take the nearest $\sqrt{k_0}$ triangles and computed which parts are visible. This can be done in time $O(k_0 \log k_0)$ time in a trivial way. Let k_1 be the complexity of the resulting scene. Take the next $\sqrt{k_1}$ triangles. Compute the parts that are visible among them. This takes time $O(k_1 \log k_1)$. The complexity of the scene is at most $O(k_1)$. Merge this with the first part in the way described above in time $O((k_1 + k_2)\log k_1)$ where k_2 is the complexity of the new scene. Now take the next $\sqrt{k_2}$ triangles and repeat the process. In this way we continue until no triangles are left.

Theorem 6.3 *Given a set of n triangle in space. Their visible parts can be computed in time $O(n\sqrt{k}\log n)$ where k is the complexity of the visible scene.*

Proof. Note that the total amount of time spend in the above algorithm is bounded by $O(\sum k_i \log n)$ while $\sum \sqrt{k_i} = n$ and $k_i \leq k$ for all i. Now

$$\sum k_i = \sum(\sqrt{k_i}\sqrt{k_i}) \leq \sum(\sqrt{k_i})\sqrt{k} = n\sqrt{k}$$

from which the theorem follows. □

In fact the bound can be improved to $O(k + n\sqrt{c}\log n)$ where c is the maximal contour size during the execution of the algorithm.

6.3 Further reading

A number of other object space algorithms are known for hidden surface removal, e.g. Devai [19], Goodrich [32] and McKenna [56]. Also a number of algorithms exist that deal with restricted cases like e.g. polygonal terrains. See e.g. Reif and Sen [68].

An important special case occurs when the set of objects consists of axis-parallel rectangles, each at a particular depth. This situations occurs for example in applications where there is a large number of overlapping windows on the screen. The best solution for this problem known today is due to Bern [9] and runs in time $O(n \log n \text{loglog} n + k \log n)$ where k is the complexity of the output scene. See also Güting and Ottmann [37].

7 Other research

In this section we give a number of other subjects treated in computational geometry and mention some results obtained.

7.1 Grids

In computer graphics coordinates of geometric objects often are not arbitrary reals but choosen from some bounded universe. For example, the screen coordinates. Although the general algorithms presented above can handle such cases as well, in a number of situations one can obtain more efficient or simpler algorithms when working in a bounded universe. Let us assume the coordinates are integers between 0 and $u-1$ (i.e., u is the universe size). Hence, the plane has become a $u \times u$ grid. See Keil and Kirkpatrick [42], Karlsson [40] and Overmars [62] for overviews of such results. Most of them are based on the fact that one can use table look-up or perfect hashing techniques instead of searching. For example, in [62] (see also [41]) it is shown how to compute all intersections in a set of line segments with endpoints on a grid in time $O(k + u + n \log n)$ in a very simple way.

One of the basic tools used in algorithms is sorting. It is easy to see that, using bucket sort, a set of n values between 0 and $u-1$ can be sorted in time $O(n+u)$. Using repeated bucket sort Kirkpatrick and Reisch [45] have improved this to $O(n\text{loglog}_n u)$. Note that this is $O(n)$ as long as $u = O(n^c)$ for some constant c. Now consider the convex hull problem. Note that in the first method presented in section 2.1 the time consuming step was sorting. After sorting the method runs in $O(n)$ time. Hence, applying the efficient sorting method on a grid we obtain:

Theorem 7.1 *Given a set of n points on a $u \times u$ grid, the convex hull can be computed in time $O(n\text{loglog}_n u)$,*

A second basic tool is searching. Van Emde Boas [27], see also [28], has shown that one can store n values between 0 and $u-1$ using $O(u)$ storage such that values can be inserted, deleted and searched for in time $O(\text{loglog}\, u)$. Also 1-dimensional range queries can be answered this efficienty. Now reconsider the axis-parallel line segment intersection problem where we have a set of horizontal and vertical line segments and want to compute all intersection. In section 5.1 we have given a scanline algorithm to solve this problem. We moved a scanline from left to right and maintained the intersections with horizontal line segments in a balanced tree. When the line segments lie on a grid we can use a VanEmdeBoas tree for this purpose. Insertions and deletions now take time $O(\text{loglog}\, u)$. When the scanline reaches a vertical line segment we perform a range query on this structure which also takes time $O(\text{loglog}\, u)$. Sorting the halting points for the scanline can be done in time $O(n\text{loglog}_n u)$ as stated above. We conclude:

Theorem 7.2 *Given a set of n horizontal and vertical line segments on a $u \times u$ grid, all intersections can be found in time $O(k+n\text{loglog}\, u)$, where k is the number of intersections found.*

Many other problems can be solved more efficiently on a grid, including windowing, some proximity problems, etc.

7.2 Combinatorial geometry

Many result from combinatorial geometry are used in computational geometry. See the book of Edelsbrunner [22] for an overview of such techniques.

Let us just briefly mention one of these techniques, *duality*. Using dualization one can transform points into lines and lines into points maintaining the incidence relation. So a point p maps to a line $D(p)$ and a line l to a point $D(l)$ such that if p lies on l, $D(l)$ lies on $D(p)$. In fact not only incidence but vertical distance is maintained. (See [22] for details.) One can use dualization to turn problems into different problems. For example, the question whether in a set of points three points lie on a line dualizes to: given a set of lines are there three that pass through the same point. This problem can be solved in time $O(n^2)$ by traversing the subdivision of the plane induced by the lines (see [22] for details).

7.3 Partition trees

As we have seen in section 4 it is easy to store geometric objects in data structures as long as they are in some sense axis-parallel, or the query objects are axis-parallel. This is no longer the case when both objects and query objects are not axis-parallel. For example it is hard to store a set of line segments V such that for any other line segment s the elements in V intersecting s can be determined efficiently.

To solve such query problems so-called *partition trees* have been designed (see Willard [79], Edelbrunner and Welzl [26], Haussler and Welzl [38] and Welzl [77]). The easiest type of a partition tree is the so-called *conjungation tree* ([26]). Let V be a set of points in the plane. A conjungation tree is a binary tree in which each node corresponds to a part of the plane. The root corresponds to the whole plane. The leaves correspond to parts that contain just one point of V. Each node stores a line l that splits its part in the parts for its two sons such that each of the two parts contains the same number of points of V, i.e., l splits the subset of points in equal halves. Now the basic property of the conjungation tree is that the two sons of a node have the same splitting line. Such

a line, that splits two subsets at the same moment in equal halves always exists and is called a *conjungate*. The points in the set V are stored in the leaves.

The conjungation tree requires $O(n)$ storage. Moreover, it can be constructed in time $O(n \log n)$. One can show that in a conjungation tree any line l will cut through at most $O(n^{.695})$ parts stored at nodes. Hence, the large majority of the parts lie on one side of the line.

One type of query that can be performed using partition trees is the so-called *halfplanar range query*: Given a halfplane H (i.e., the part of the plane on one side of a line l_H) determine all (or the number) of points in H.

Theorem 7.3 *In a conjungation tree, halfplanar range queries can be solved in time $O(k + n^{.695})$ where k is the number of answers found.*

Proof. We simply search with l_H in the tree. Assume we are at some node δ. If δ is a leaf we check whether the point stored lies in the halfplane H. Otherwise we look at the part of the plane corresponding to δ. If l_H intersects this part we continue the search in both sons. If the part lies completely inside H we report all points in it. If the part lies completely outside H we don't search any further (at this node).

It is easy to see that the number of nodes visited is bounded by the number of parts l_H is intersecting and, as stated above, this is at most $O(n^{.695})$. □

Partition trees can also be used for storing line segments. Queries like: which line segments intersect a given query line segment, can then be answered in similar time bounds. (See e.g. Guibas et. al. [35].)

7.4 Random sampling

A technique that has recently received a lot of attention in computational geometry is *random sampling*. Algorithms using random sampling tend to be simple and have very good expected time bounds. The main difference with other expected-case algorithms (like the one for convex hulls presented in section 2.1) is that the time bounds depends on the random behaviour of the algorithm and not on some distribution of the set of objects.

The basic idea behind random sampling is that objects are added in a random order or that first a random subsets of the objects is taken. Such a random subset normally gives (at least in a probabilistic sense) a lot of information about the whole set. For example, if a small random subset V' of a set of points V is taken, the chance that another point in V falls inside the convex hull of V' is large. This information can be used to solve the problem more efficiently.

Many geometric problems can be solved using random sampling. See e.g. Clarkson [14, 15, 16], Haussler and Welzl [38] and Clarkson et. al. [17] for application of random sampling in proximity problems, point location, range searching, convex hulls, etc. For example, in [16] a simple method is given to compute all intersection in a set of line segments in expected time $O(k + n \log n)$ where k is the number of intersections.

7.5 Further reading

Many interesting papers on computational geometry have been published in the following journals (among others): Algoritmica, Computer Vision, Graphics and Image Processing, Information Processing Letters, Journal of Algorithms, Discrete and Computational

Geometry, SIAM J. of Computing, and ACM Transactions on Graphics. Another important source of publications form the proceedings of the ACM Symp. on Computational Geometry that is held yearly.

References

1. Aurenhammer, F., Voronoi diagrams – A survey. Techn. Rep. 263, Inst. f. Informationsverarbeitung, TU Graz, 1988

2. Bentley, J.L., Multidimensional binary search trees used for associated searching. C. ACM *18*, 509-517 (1975)

3. Bentley, J.L., Multidimensional binary search trees in database applications. IEEE Trans. Software Eng. *SE-5*, 333-340 (1979)

4. Bentley, J.L., Decomposable searching problems. Inform. Proc. Lett. *8*, 244-251 (1979)

5. Bentley, J.L., Friedman, J.H., Data structures for range searching. ACM Comput. Surveys *11*, 397-409 (1979)

6. Bentley, J.L., Maurer, H.A., Efficient worst-case data structures for range searching. Acta Inform. *13*, 155-168 (1980)

7. Bentley, J.L., Ottmann, T., Algorithms for reporting and counting geometric intersections. IEEE Trans. Comput. *C-28*, 643-647 (1979)

8. Bentley, J.L., Wood, D., An optimal algorithm for reporting intersections of rectangles. IEEE Trans. Comput. *C-29*, 571-577 (1980)

9. Bern, M., Hidden surface removal for rectangles. Proc. 4th ACM Symp. on Computational Geometry, 1988, pp. 183-192

10. Bykat, A., Convex hull of a finite set of points in two dimensions. Inform. Proc. Lett. *7*, 296-298 (1978)

11. Chazelle, B., Edelsbrunner, H., An improved algorithm for constructing k th-order Voronoi diagrams. IEEE Trans Comput. *C-36*, 1349-1354 (1987)

12. Chazelle, B., Edelsbrunner, H., An optimal algorithm for intersecting line segments in the plane. Proc. 29th IEEE Symp. on Foundations of Computer Science, 1988, pp. 590-600

13. Chew, L.P., Drysdale, R.L., Voronoi diagrams based on convex distance functions. Proc. 1st ACM Symp. on Computational Geometry, 1985, pp. 235-244

14. Clarkson, K.L., A probabilistic algorithm for the post office problem. Proc. 17th ACM Symp. on Theory of Computing, 1985, pp. 175-184

15. Clarkson, K.L., New applications of random sampling in computational geometry. Discrete Comput. Geom. *2*, 195-222 (1987)

16. Clarkson, K.L., Applications of random sampling in computational geometry II. Proc. 4th ACM Symp. on Computational Geometry, 1988, pp. 1-11

17. Clarkson, K.L., Tarjan, R.E., Van Wijk, C.J., A fast Las Vegas algorithm for triangulating a simple polygon. Proc. 4th ACM Symp. on Computational Geometry, 1988, pp. 18-22

18. Delaunay, B., Sur la sphère vide. Bull. Acad. Sci. USSR (VII), Classe Sci. Mat. Nat., 793-800 (1934)

19. Dévai, F., Quadratic bounds for hidden line elimination. Proc. 2nd ACM Symp. on Computational Geometry, 1986, pp. 269-275

20. Edelsbrunner, H., A note on dynamic range searching. Bull. of the EATCS *15*, 34-40 (1981)

21. Edelsbrunner, H., Intersection problems in computational geometry. Techn. Rep. F93, Inst. f. Information Processing, TU Graz, 1982

22. Edelsbrunner, H., Algorithms in combinatorial geometry. EATC Monographs on Theoretical Computer Science 10, Springer, Berlin 1987

23. Edelsbrunner, H., Guibas, L.J., Stolfi, J., Optimal point location in a monotone subdivision. SIAM J. Comput. *15*, 317-340 (1986)

24. Edelsbrunner, H., Kirkpatrick, D.G., Seidel, R., On the shape of a set of points in the plane. IEEE Trans. Information Theory *IT-29*, 551-559 (1983)

25. Edelsbrunner, H., Overmars, M.H., Seidel, R., Some methods of computational geometry applied to computer graphics. Comp. Vision, Graphics and Image Proc. *28*, 92-108 (1984)

26. Edelsbrunner, H., Welzl, E., Halfplanar range search in linear space and $O(n^{0.695})$ query time. Inform. Proc. Lett. *23*, 289-293 (1986)

27. van Emde Boas, P., Preserving order in a forest in less than logarithmic time and linear space. Inform. Proc. Lett. *6*, 80-82 (1977)

28. van Emde Boas, P., Kaas, R., Zijlstra, E., Design and implementation of an efficient priority queue. Math. Systems Theory *10*, 99-127 (1977)

29. Finkel, R.A., Bentley, J.L., Quad-trees; a data structure for retrieval on composite keys. Acta Inform. *4*, 1-9 (1974)

30. Fortune, S., A sweepline algorithm for Voronoi diagrams. Algorithmica *2*, 153-174 (1987)

31. Garey, M.R., Johnson, D.S., Preparata, F.P., Tarjan, R.E., Triangulating a simple polygon. Inform. Proc. Lett *7*, 175-180 (1978)

32. Goodrich, M.T., A polygonal approach to hidden line elimination. Proc. 25th Allerton Conf. on Communication, Control and Computing, 1987, pp. 849-858

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

34. Graham, R.L., Yao, F.F., Finding the convex hull of a simple polygon. J. Algorithms *4*, 324-331 (1983)

35. Guibas, L., Overmars, M.H., Sharir, M., Intersecting line segments, ray shooting, and other applications of geometric partitioning techniques. Proc. SWAT 1988, Lect. Notes in Comp. Science 318, Springer, Berlin 1988, pp. 64-73

36. Guibas, L., Stolfi, J., Primitives for the manipulation of general subdivisions and the computation of Voronoi diagrams. ACM Trans. Graphics *4*, 74-123 (1985)

37. Güting, R.H., Ottmann, T., New algorithms for special cases of the hidden line elimination problem. Comp. Vision, Graphics and Image Proc. *40*, 188-204 (1987)

38. Haussler, D., Welzl, E., ϵ-nets and simplex range queries. Discrete Comput. Geom. *2*, 127-151 (1987)

39. Jarvis, R.A., On the identification of the convex hull of a finite set of points in the plane. Inform. Proc. Lett. *2*, 18-21 (1973)

40. Karlsson, R.G., Algorithms in a restricted universe. PhD thesis, Techn. Rep. CS-84-50, Dept. of Computer Science, University of Waterloo, 1984

41. Karlsson, R.G., Overmars, M.H., Scanline algorithms on a grid. BIT *28*, 227-241 (1988)

42. Keil, J.M., Kirkpatrick, D.G., Computational geometry on integer grids. Proc. 19th Allerton Conf. on Communication, Control and Computing, 1981, pp. 41-50

43. Keil, J.M., Sack, J.R., Minimum decomposition of polygonal objects. in: G. Toussaint (ed.), Computational Geometry, North-Holland, Amsterdam 1985

44. Kirkpatrick, D.G., Optimal search in planar subdivisions. SIAM J. Comput. *12*, 28-35 (1983)

45. Kirkpatrick, D.G., Reisch, S., Upperbounds for sorting integers on random access machines. Theor. Comp. Science *28*, 263-276 (1984)

46. Kirkpatrick, D.G., Seidel, R., The ultimate planar convex hull algorithm? SIAM J. Comput. *15*, 287-299 (1986)

47. van Kreveld, M.J., Overmars, M.H., Divided k-d trees. Techn. Rep. RUU-CS-88-28, Dept. of Computer Science, University of Utrecht, 1988.

48. Lee, D.T., Two-dimensional Voronoi diagrams in the L_p metric. J. ACM *27*, 604-618 (1980)

49. Lee, D.T., On k-nearest neighbor Voronoi diagrams in the plane. IEEE Trans. Comput. *C-31*, 478-487 (1982)

50. Lee, D.T., On finding the convex hull of a simple polygon. J. Comput. Inform. Sciences *12*, 87-98 (1983)

51. Lee, D.T., Drysdale III, R.L., Generalisation of Voronoi diagrams in the plane. SIAM J. Comput. *10*, 73-87 (1981)

52. Lee, D.T., Wong, C.K., Voronoi diagrams in L_1 (L_∞) metrics with 2-dimensional storage applications. SIAM J. Comput. *9*, 200-211 (1980)

53. Lueker, G.S., A data structure for orthogonal range queries. Proc. 19th IEEE Symp. on Foundations of Computer Science, 1978, pp. 28-34

54. Mairson, H., Stolfi, J., Reporting and counting intersections between two sets of line segments. in: R. Earnshaw (ed.), Theoretical foundations for computer graphics and CAD, Springer, Berlin 1988, pp. 307-326

55. McCreight, E.M., Priority search trees. SIAM J. Comput. *14*, 257-276 (1985)

56. McKenna, M., Worst-case optimal hidden surface removal. ACM Trans. Graphics *6*, 19-28 (1987)

57. Mehlhorn, K., Data structures and algorithms 3: Multi-dimensional searching and computational geometry. EATCS Monographs on Theoretical Computer Science 3, Springer-Verlag, Berlin 1984

58. O'Rourke, J., Art gallery theorems and algorithms. Oxford Univ. Press, Oxford 1987

59. Overmars, M.H., The design of dynamic data structures. Lect. Notes in Computer Science 156, Springer, Berlin 1983

60. Overmars, M.H., Range searching in a set of line segments. Proc. 1st ACM Symp. on Computational Geometry, 1985, pp. 177-185

61. Overmars, M.H., Geometric data structures for computer graphics. in: R. Earnshaw (ed.), Fundamental algorithms for computer graphics, Springer, Berlin 1985, pp. 919-931

62. Overmars, M.H., Computational geometry on a grid: an overview. in: R. Earnshaw (ed.), Theoretical foundations for computer graphics and CAD, Springer, Berlin 1988, pp. 167-184

63. Overmars, M.H., van Leeuwen, J., Further comments on Bykat's convex hull algorithm, Inform. Proc. Lett. *10*, 209-212 (1980)

64. Overmars, M.H., van Leeuwen, J., Dynamic multi-dimensional data structures based on quad- and k-d trees. Acta Inform. *17*, 267-285 (1982)

65. Overmars, M.H., van Leeuwen, J., Maintenance of configurations in the plane. J. Comput. Syst. Sciences *23*, 166-204 (1981)

66. Overmars, M.H., Sharir, M., Output-sensitive hidden surface removal, Techn. Rep., Dept. of Computer Science, University of Utrecht, 1989, to appear

67. Preparata, F.P., Shamos, M.I. Computational geometry. Springer, New York 1985

68. Reif, J.H., Sen, S., An efficient output-sensitive hidden-surface removal algorithm and its parallelization. Proc. 4th ACM Symp. on Computational Geometry, 1988, pp. 193-200

69. Samet, H., Bibliography on quadtrees and related hierarchical data structures. in: L. Kessenaar, F. Peters and M. van Lierop (ed.), Data structures for raster graphics, Springer, Berlin 1986, pp. 181-201

70. Samet, H., An overview of quadtrees, octtrees, and related hierarchical data structures. in: R. Earnshaw (ed.), Theoretical foundations for computer graphics and CAD, Springer, Berlin 1988, pp. 51-68.

71. Schmitt, A., Time and space bounds for hidden line and hidden surface algorithms. Proc. Eurographics 1981, pp. 43-56

72. Scholten, H.W., Overmars, M.H., General methods for adding range restrictions to decomposable searching problems. J. Symbolic Computation *7*, 1-10 (1989)

73. Shamos, M.I., Hoey, D., Closest point problems. Proc. 16th IEEE Symp. on Foundations of Computer Science, 1975, pp. 151-162

74. Sutherland, I.E., Sproull, R.F., Schumacker, R.A., A characterization of ten hidden-surface algorithms. Computing Surveys *6*, 1-25 (1974)

75. Tarjan, R.E., Van Wijk, C.J., And $O(n\log\log n)$-time algorithm for triangulating a simple polygon. SIAM J. Comput. *17*, 143-178 (1988)

76. Voronoi, G., Nouvelles applications des parametres continus à la theorie des formes quadratiques. Deuxième Mémorie: Recherches sur les paralléloèdres primitifs, J. reine angew. Math. *134*, 198-287 (1908)

77. Welzl, E., Partition trees for triangle counting and other range searching problems. Proc. 4th ACM Symp. on Computational Geometry, 1988, pp. 23-33

78. Willard, D.E., Predicate-oriented database search algorithms. Garland Publishing Company, New York 1979

79. Willard, D.E., Polygon retrieval. SIAM J. Comput. *11*, 149-165 (1982)

80. Willard, D.E., Lueker, G.S., Adding range restriction capability to dynamic data structures. J. ACM *32*, 597-617 (1985)

81. Yao, A.C., A lowerbound to finding convex hulls. J. ACM *28*, 780-787 (1981)

82. Yap, C.K., An $O(n \log n)$ algorithm for the Voronoi diagram of a set of simple curve segments. Discrete Comput. Geom. *2*, 365-393 (1987)