17

Computational Geometry

Computational geometry is the algorithmic study of geometric problems. Its emergence coincided with application areas such as computer graphics, computer-aided design/manufacturing, and scientific computing, which together provide much of the motivation for geometric computing. The past twenty years have seen enormous maturity in computational geometry, resulting in a significant body of useful algorithms, software, textbooks, and research results.

Good books on computational geometry include:

- *de Berg, et al.* [dBvKOS00] – The "three Mark's" book is the best general introduction to the theory of computational geometry and its fundamental algorithms.

- *O'Rourke* [O'R01] – This is the best practical introduction to computational geometry. The emphasis is on careful and correct implementation of geometric algorithms. C and Java code are available from *http://maven.smith.edu/∼orourke/books/compgeom.html*.

- *Preparata and Shamos* [PS85] – Although somewhat out of date, this book remains a good general introduction to computational geometry, stressing algorithms for convex hulls, Voronoi diagrams, and intersection detection.

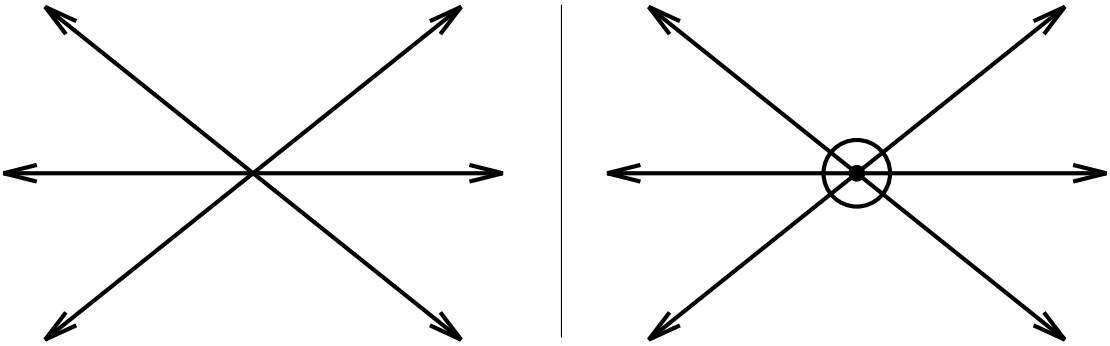
- *Goodman and O'Rourke* [GO04] – This recent collection of survey articles provides a detailed overview of what is known in most every subfield of discrete and computational geometry.

The leading conference in computational geometry is the *ACM Symposium on Computational Geometry*, held annually in late May or early June. Although the primary results presented at the conference are theoretical, there has been a

S.S. Skiena, *The Algorithm Design Manual*, 2nd ed., DOI: 10.1007/978-1-84800-070-4_17,

concerted effort on the part of the research community to increase the presence of applied, experimental work through video reviews and poster sessions.

There is a growing body of implementations of geometric algorithms. We point out specific implementations where applicable in the catalog, but the reader should be particularly aware of `CGAL` (Computational Geometry Algorithms Library)—a comprehensive library of geometric algorithms in C++ produced as a result of a joint European project. Anyone with a serious interest in geometric computing should check it out at *http://www.cgal.org/*.



INPUT OUTPUT

17.1 Robust Geometric Primitives

Input description: A point p and line segment l, or two line segments l_1, l_2.

Problem description: Does p lie over, under, or on l? Does l_1 intersect l_2?

Discussion: Implementing basic geometric primitives is a task fraught with peril, even for such simple tasks as returning the intersection point of two lines. It is more complicated than you may think. What should you return if the two lines are parallel, meaning they don't intersect at all? What if the lines are identical, so the intersection is not a point but the entire line? What if one of the lines is horizontal, so that in the course of solving the equations for the intersection point you are likely to divide by zero? What if the two lines are almost parallel, so that the intersection point is so far from the origin as to cause arithmetic overflows? These issues become even more complicated for intersecting line segments, since there are many other special cases that must be watched for and treated specially.

If you are new to implementing geometric algorithms, I suggest that you study O'Rourke's *Computational Geometry in C* [O'R01] for practical advice and complete implementations of basic geometric algorithms and data structures. You are likely to avoid many headaches by following in his footsteps.

There are two different issues at work here: geometric degeneracy and numerical stability. *Degeneracy* refers to annoying special cases that must be treated in substantially different ways, such as when two lines intersect in more or less than a single point. There are three primary approaches to dealing with degeneracy:

- *Ignore it* – Make as an operating assumption that your program will work correctly only if no three points are collinear, no three lines meet at a point, no intersections happen at the endpoints of line segments, etc. This is probably the most common approach, and what I might recommend for short-term

projects if you can live with frequent crashes. The drawback is that interesting data often comes from points sampled on a grid, and so is inherently very degenerate.

- *Fake it* – Randomly or symbolically perturb your data so that it becomes nondegenerate. By moving each of your points a small amount in a random direction, you can break many of the existing degeneracies in the data, hopefully without creating too many new problems. This probably should be the first thing to try once you decide that your program is crashing too often. A problem with random perturbations is that they can change the shape of your data in subtle ways, which may be intolerable for your application. There also exist techniques to "symbolically" perturb your data to remove degeneracies in a consistent manner, but these require serious study to apply correctly.
- *Deal with it* – Geometric applications can be made more robust by writing special code to handle each of the special cases that arise. This can work well if done with care at the beginning, but not so well if kludges are added whenever the system crashes. Expect to expend significant effort if you are determined to do it right.

Geometric computations often involve floating-point arithmetic, which leads to problems with overflows and numerical precision. There are three basic approaches to the issue of numerical stability:

- *Integer arithmetic* – By forcing all points of interest to lie on a fixed-size integer grid, you can perform exact comparisons to test whether any two points are equal or two line segments intersect. The cost is that the intersection point of two lines may not be exactly representable as a grid point. This is likely to be the simplest and best method, if you can get away with it.
- *Double precision reals* – By using double-precision floating point numbers, you may get lucky and avoid numerical errors. Your best bet might be to keep all the data as single-precision reals, and use double-precision for intermediate computations.
- *Arbitrary precision arithmetic* – This is certain to be correct, but also to be slow. This approach seems to be gaining favor in the research community. Careful analysis can minimize the need for high-precision arithmetic and thus the performance penalty. Still, you should expect high-precision arithmetic to be several orders of magnitude slower than standard floating-point arithmetic.

The difficulties associated with producing robust geometric software are still under attack by researchers. The best practical technique is to base your applications on a small set of geometric primitives that handle as much of the low-level geometry as possible. These primitives include:

- *Area of a triangle* – Although it is well known that the area $A(t)$ of a triangle $t = (a, b, c)$ is half the base times the height, computing the length of the base and altitude is messy work with trigonometric functions. It is better to use the determinant formula for *twice* the area:

$$2 \cdot A(t) = \begin{vmatrix} a_x & a_y & 1 \\ b_x & b_y & 1 \\ c_x & c_y & 1 \end{vmatrix} = a_x b_y - a_y b_x + a_y c_x - a_x c_y + b_x c_y - c_x b_y$$

  This formula generalizes to compute $d!$ times the volume of a simplex in d dimensions. Thus, $3! = 6$ times the volume of a tetrahedron $t = (a, b, c, d)$ in three dimensions is

$$6 \cdot A(t) = \begin{vmatrix} a_x & a_y & a_z & 1 \\ b_x & b_y & b_z & 1 \\ c_x & c_y & c_z & 1 \\ d_x & d_y & d_z & 1 \end{vmatrix}$$

  Note that these are signed volumes and can be negative, so take the absolute value first. Section 13.4 (page 404) explains how to compute determinants.

  The conceptually simplest way to compute the area of a polygon (or polyhedron) is to triangulate it and then sum up the area of each triangle. Implementations of a slicker algorithm that avoids triangulation are discussed in [O'R01, SR03].

- *Above-below-on test* – Does a given point c lie above, below, or on a given line l? A clean way to deal with this is to represent l as a directed line that passes through point a before point b, and ask whether c lies to the left or right of the directed line l. It is up to you to decide whether left means above or below.

  This primitive can be implemented using the sign of the triangle area as computed above. If the area of $t(a, b, c) > 0$, then c lies to the left of $\overline{ab}$. If the area of $t(a, b, c) = 0$, then c lies on $\overline{ab}$. Finally, if the area of $t(a, b, c) < 0$, then c lies to the right of $\overline{ab}$. This generalizes naturally to three dimensions, where the sign of the area denotes whether d lies above or below the oriented plane (a, b, c).

- *Line segment intersection* – The above-below primitive can be used to test whether a line intersects a line segment. It does iff one endpoint of the segment is to the left of the line and the other is to the right. Segment intersection is similar but more complicated. We refer you to implementations described below. The decision as to whether two segments intersect if they share an endpoint depends upon your application and is representative of the problems with degeneracy.

- *In-circle test* – Does point d lie inside or outside the circle defined by points a, b, and c in the plane? This primitive occurs in all Delaunay triangulation algorithms, and can be used as a robust way to do distance comparisons. Assuming that a, b, c are labeled in counterclockwise order around the circle, compute the determinant

$$\text{incircle}(a,b,c,d) = \begin{vmatrix} a_x & a_y & a_x^2+a_y^2 & 1 \\ b_x & b_y & b_x^2+b_y^2 & 1 \\ c_x & c_y & c_x^2+c_y^2 & 1 \\ d_x & d_y & d_x^2+d_y^2 & 1 \end{vmatrix}$$

  In-circle will return 0 if all four points are cocircular—a positive value if d is inside the circle, and negative if d is outside.

Check out the Implementations section before you build your own.

Implementations: CGAL (*www.cgal.org*) and LEDA (see Section 19.1.1 (page 658)) both provide very complete sets of geometric primitives for planar geometry written in C++. LEDA is easier to learn and to work with, but CGAL is more comprehensive and freely available. If you are starting a significant geometric application, you should at least check them out before you try to write your own.

O'Rourke [O'R01] provides implementations in C of most of the primitives discussed in this section. See Section 19.1.10 (page 662). These primitives were implemented primarily for exposition rather than production use, but they should be reliable and appropriate for small applications.

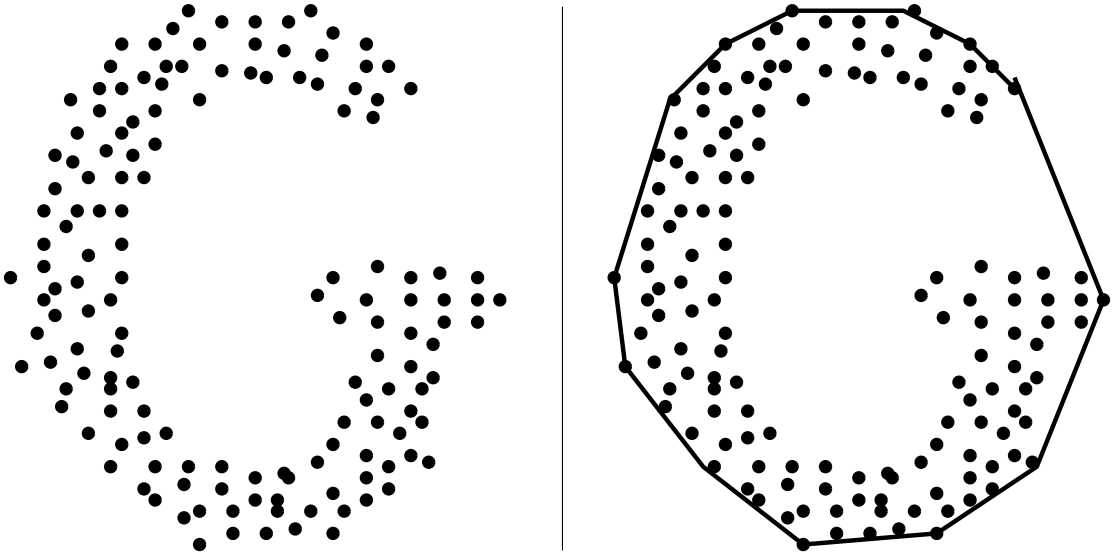
The *Core Library* (see *http://cs.nyu.edu/exact/*) provides an API, which supports the Exact Geometric Computation (EGC) approach to numerically robust algorithms. With small changes, any C/C++ program can use it to readily support three levels of accuracy: machine-precision, arbitrary-precision, and guaranteed.

Shewchuk's [She97] robust implementation of basic geometric primitives in C++ is available at *http://www.cs.cmu.edu/∼quake/robust.html*.

Notes: O'Rourke [O'R01] provides an implementation-oriented introduction to computational geometry which stresses robust geometric primitives. It is recommended reading. LEDA [MN99] provides another excellent role model.

Yap [Yap04] gives a excellent survey on techniques for achieving robust geometric computation, with a book [MY07] in preparation. Kettner, et al. [KMP+04] provides graphic evidence of the troubles that arise when employing real arithmetic in geometric algorithms such as convex hull. Controlled perturbation [MOS06] is a new approach for robust computation that is receiving considerable attention. Shewchuk [She97] and Fortune and van Wyk [FvW93] present careful studies on the costs of using arbitrary-precision arithmetic for geometric computation. By being careful about when to use it, reasonable efficiency can be maintained while achieving complete robustness.

Related Problems: Intersection detection (see page 591), maintaining arrangements (see page 614).



INPUT OUTPUT

17.2 Convex Hull

Input description: A set S of n points in d-dimensional space.

Problem description: Find the smallest convex polygon (or polyhedron) containing all the points of S.

Discussion: Finding the convex hull of a set of points is *the* most important elementary problem in computational geometry, just as sorting is the most important elementary problem in combinatorial algorithms. It arises because constructing the hull captures a rough idea of the shape or extent of a data set.

Convex hull serves as a first preprocessing step to many, if not most, geometric algorithms. For example, consider the problem of finding the diameter of a set of points, which is the pair of points that lie a maximum distance apart. The diameter must be between two points on the convex hull. The $O(n \lg n)$ algorithm for computing diameter proceeds by first constructing the convex hull, then for each hull vertex finding which other hull vertex lies farthest from it. The so-called "rotating-calipers" method can be used to move efficiently from one-diametrically opposed hull vertex to another by always proceeding in a clockwise fashion around the hull.

There are almost as many convex hull algorithms as sorting algorithms. Answer the following questions to help choose between them:

- *How many dimensions are you working with?* – Convex hulls in two and even three dimensions are fairly easy to work with. However, certain valid assumptions in lower dimensions break down as the dimensionality increases. For example, any n-vertex polygon in two dimensions has exactly n edges. However, the relationship between the numbers of faces and vertices becomes more complicated even in three dimensions. A cube has eight vertices and six faces, while an octahedron has eight faces and six vertices. This has implications for data structures that represent hulls—are you just looking for the hull points or do you need the defining polyhedron? Be aware of these complications of high-dimensional spaces if your problem takes you there.

  Simple $O(n \log n)$ convex-hull algorithms are available for the important special cases of two and three dimensions. In higher dimensions, things get more complicated. *Gift-wrapping* is the basic algorithm for constructing higher-dimensional convex hulls. Observe that a three-dimensional convex polyhedron is composed of two-dimensional faces, or *facets*, that are connected by one-dimensional lines or *edges*. Each edge joins exactly two facets together. Gift-wrapping starts by finding an initial facet associated with the lowest vertex and then conducting a breadth-first search from this facet to discover new, additional facets. Each edge e defining the boundary of a facet must be shared with one other facet. By running through each of the n points, we can identify which point defines the next facet with e. Essentially, we "wrap" the points one facet at a time by bending the wrapping paper around an edge until it hits the first point.

  The key to efficiency is making sure that each edge is explored only once. Implemented properly in d dimensions, gift-wrapping takes $O(n\phi_{d-1} + \phi_{d-2} \lg \phi_{d-2})$, where ϕ_{d-1} is the number of facets and ϕ_{d-2} is the number of edges in the convex hull. This can be as bad as $O(n^{\lfloor d/2 \rfloor + 1})$ when the convex hull is very complex. I recommend that you use one of the codes described below rather than roll your own.

- *Is your data given as vertices or half-spaces?* – The problem of finding the intersection of a set of n half-spaces in d dimensions (each containing the origin) is dual to that of computing convex hulls of n points in d dimensions. Thus, the same basic algorithm suffices for both problems. The necessary duality transformation is discussed in Section 17.15 (page 614). The problem of half-plane intersection differs from convex hull when no interior point is given, since the instance may be infeasible (i.e., the intersection of the half-planes empty).

- *How many points are likely to be on the hull?* – If your point set was generated "randomly," it is likely that most points lie within the interior of the hull. Planar convex-hull programs can be made more efficient in practice using the observation that the leftmost, rightmost, topmost, and bottommost points all must be on the convex hull. This usually gives a set of either three or

four distinct points, defining a triangle or quadrilateral. Any point inside this region *cannot* be on the convex hull and so can be discarded in a linear sweep through the points. Ideally, only a few points will then remain to run through the full convex-hull algorithm.

This trick can also be applied beyond two dimensions, although it loses effectiveness as the dimension increases.

- *How do I find the shape of my point set?* – Although convex hulls provide a gross measure of shape, any details associated with concavities are lost. The convex hull of the G from the example input would be indistinguishable from the convex hull of O. *Alpha-shapes* are a more general structure that can be parameterized so as to retain arbitrarily large concavities. Implementations and references on alpha-shapes are included below.

The primary convex-hull algorithm in the plane is the *Graham scan*. Graham scan starts with one point p known to be on the convex hull (say the point with the lowest x-coordinate) and sorts the rest of the points in angular order around p. Starting with a hull consisting of p and the point with the smallest angle, we proceed counterclockwise around v adding points. If the angle formed by the new point and the last hull edge is less than 180 degrees, we add this new point to the hull. If the angle formed by the new point and the last "hull" edge is greater than 180 degrees, then a chain of vertices starting from the last hull edge must be deleted to maintain convexity. The total time is $O(n \lg n)$, since the bottleneck is the cost of sorting the points around v.

The basic Graham scan procedure can also be used to construct a nonself-intersecting (or *simple*) polygon passing through all the points. Sort the points around v, but instead of testing angles, simply connect the points in angular order. Connecting this to v gives a polygon without self-intersection, although it typically has many ugly skinny protrusions.

The gift-wrapping algorithm becomes especially simple in two dimensions, since each "facet" becomes an edge, each "edge" becomes a vertex of the polygon, and the "breadth-first search" simply walks around the hull in a clockwise or counterclockwise order. The 2D gift-wrapping (or *Jarvis march*) algorithm runs in $O(nh)$ time, where h is the number of vertices on the convex hull. I recommend sticking with Graham scan unless you *really* know in advance that there cannot be too many vertices on the hull.

Implementations: The CGAL library (*www.cgal.org*) offers C++ implementations of an extensive variety of convex-hull algorithms for two, three, and arbitrary numbers of dimensions. Alternate C++ implementations of planar convex hulls includes LEDA (see Section 19.1.1 (page 658)).

Qhull [BDH97] is a popular low-dimensional, convex-hull code, optimized for from two to about eight dimensions. It is written in C and can also construct Delaunay triangulations, Voronoi vertices, furthest-site Voronoi vertices, and

half-space intersections. Qhull has been widely used in scientific applications and has a well-maintained homepage at *http://www.qhull.org/*.

O'Rourke [O'R01] provides a robust implementation of the Graham scan in two dimensions and an $O(n^2)$ implementation of an incremental algorithm for convex hulls in three dimensions. C and Java implementations are both available. See Section 19.1.10 (page 662).

For excellent alpha-shapes code, originating from the work of Edelsbrunner and Mucke [EM94], check out *http://biogeometry.duke.edu/software/alphashapes/*. Ken Clarkson's higher-dimensional convex-hull code *Hull* also does alpha-shapes, and is available at *http://www.netlib.org/voronoi/hull.html*.

Different codes are needed for enumerating the vertices of intersecting half-spaces in higher dimensions. Avis's *lhs* (*http://cgm.cs.mcgill.ca/~avis/C/lrs.html*) is an arithmetically-robust ANSI C implementation of the Avis-Fukuda reverse search algorithm for vertex enumeration/convex-hull problems. Since the polyhedra is implicitly traversed but not explicitly stored in memory, even problems with very large output sizes can sometimes be solved.

Notes: Planar convex hulls play a similar role in computational geometry as sorting does in algorithm theory. Like sorting, convex hull is a fundamental problem for which many different algorithmic approaches lead to interesting or optimal algorithms. Quickhull and mergehull are examples of hull algorithms inspired by sorting algorithms [PS85]. A simple construction involving points on a parabola presented in Section 9.2.4 (page 322) reduces sorting to convex hull, so the information-theoretic lower bound for sorting implies that planar convex hull requires $\Omega(n \lg n)$ time to compute. A stronger lower bound is established in [Yao81].

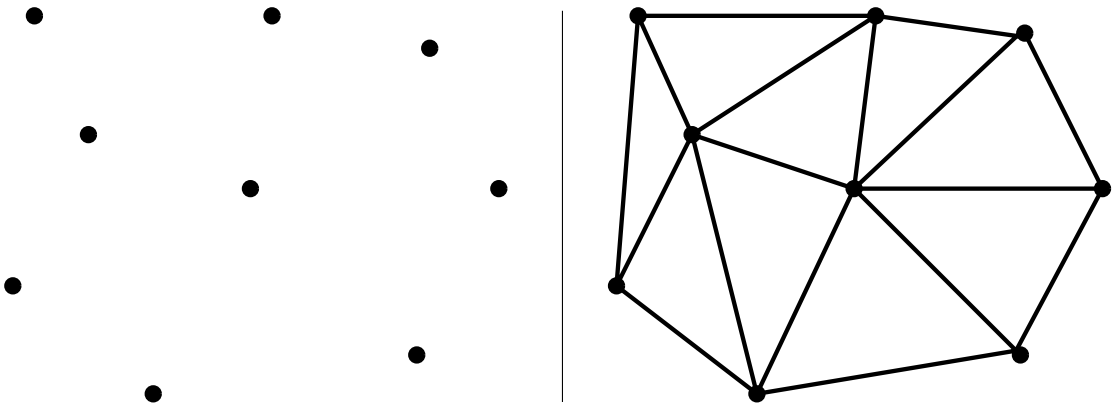
Good expositions of the Graham scan algorithm [Gra72] and the Jarvis march [Jar73] include [dBvKOS00, CLRS01, O'R01, PS85]. The optimal planar convex-hull algorithm [KS86] takes $O(n \lg h)$ time, where h is the number of hull vertices that captures the best performance of both Graham scan and gift wrapping and is (theoretically) better in between. Planar convex hull can be efficiently computed *in-place*, meaning without requiring additional memory in [BIK+04]. Seidel [Sei04] gives an up-to-date survey of convex hull algorithms and variants, particularly for higher dimensions.

Alpha-hulls, introduced in [EKS83], provide a useful notion of the shape of a point set. A generalization to three dimensions, with an implementation, is presented in [EM94].

Reverse-search algorithms for constructing convex hulls are effective in higher dimensions [AF96]. Through a clever lifting-map construction [ES86], the problem of building Voronoi diagrams in d-dimensions can be reduced to constructing convex hulls in $(d+1)$-dimensions. See Section 17.4 (page 576) for more details.

Dynamic algorithms for convex-hull maintenance are data structures that permit inserting and deleting arbitrary points while always representing the current convex hull. The first such dynamic data structure [OvL81] supported insertions and deletions in $O(\lg^2 n)$ time. More recently, Chan [Cha01] reduced the cost of such operation of near-logarithmic amortized time.

Related Problems: Sorting (see page 436), Voronoi diagrams (see page 576).



INPUT OUTPUT

17.3 Triangulation

Input description: A set of points or a polyhedron.

Problem description: Partition the interior of the point set or polyhedron into triangles.

Discussion: The first step in working with complicated geometric objects is often to break them into simple geometric objects. This makes triangulation a fundamental problem in computational geometry. The simplest geometric objects are triangles in two dimensions, and tetrahedra in three. Classical applications of triangulation include finite element analysis and computer graphics.

A particularly interesting application of triangulation is surface or function interpolation. Suppose that we have sampled the height of a mountain at a certain number of points. How can we estimate the height at any point q in the plane? We can project the sampled points on the plane, and then triangulate them. This triangulation partitions the plane into triangles, so we can estimate height by interpolating between the heights of the three points of the triangle that contain q. Furthermore, this triangulation and the associated height values define a mountain surface suitable for graphics rendering.

A triangulation in the plane is constructed by adding nonintersecting chords between the vertices until no more such chords can be added. Specific issues arising in triangulation include:

- *Are you triangulating a point set or a polygon?* – Often we are given a set of points to triangulate, as in the surface interpolation problem discussed

earlier. This involves first constructing the convex hull of the point set and then carving up the interior into triangles.

The simplest such $O(n \lg n)$ algorithm first sorts the points by x-coordinate. It then inserts them from left to right as per the convex-hull algorithm of page 105, building the triangulation by adding a chord to each point newly cut off from the hull.

- *Does the shape of the triangles in your triangulation matter?* – There are usually many different ways to partition your input into triangles. Consider a set of n points in convex position in the plane. The simplest way to triangulate them would be to add to the convex-hull diagonals from the first point to all of the others. However, this has the tendency to create skinny triangles.

  Many applications seek to to avoid skinny triangles, or equivalently, minimize small angles in the triangulation. The *Delaunay triangulation* of a point set minimizes the maximum angle over all possible triangulations. This isn't exactly what we are looking for, but it is pretty close, and the Delaunay triangulation has enough other interesting properties (including that it is dual to the Voronoi diagram) to make it the quality triangulation of choice. Further, it can be constructed in $O(n \lg n)$ time using implementations described below.

- *How can I improve the shape of a given triangulation?* – Each internal edge of any triangulation is shared between two triangles. The four vertices defining these two triangles form either (1) a convex quadrilateral, or (2) a triangle with a triangular bite taken out of it. The beauty of the convex case is that exchanging the internal edge with a chord linking the other two vertices yields a different triangulation.

  This gives us a local "edge-flip" operation for changing and possibly improving a given triangulation. Indeed, a Delaunay triangulation can be constructed from any initial triangulation by removing skinny triangles until no locally-improving exchange remains.

- *What dimension are we working in?* – Three-dimensional problems are usually harder than two-dimensional problems. The three-dimensional generalization of triangulation involves partitioning the space into four-vertex tetrahedra by adding nonintersecting faces. One important difficulty is that there is no way to tetrahedralize the interior of certain polyhedra without adding extra vertices. Furthermore, it is NP-complete to decide whether such a tetrahedralization exists, so we should not feel afraid to add extra vertices to simplify our problem.

- *What constraints does the input have?* – When we are triangulating a polygon or polyhedra, we are restricted to adding chords that do not intersect any of the boundary facets. In general, we may have a set of obstacles or

constraints that cannot be intersected by inserted chords. The best such triangulation is likely to be the so-called *constrained Delaunay triangulation*. Implementations are described next.

- *Are you allowed to add extra points, or move input vertices?* – When the shape of the triangles does matter, it might pay to strategically add a small number of extra "Steiner" points to the data set to facilitate the construction of a triangulation (say) with no small angles. As discussed above, *no* triangulation may exist for certain polyhedra without adding Steiner points.

To construct a triangulation of a convex polygon in linear time, just pick an arbitrary starting vertex v and insert chords from v to each other vertex in the polygon. Because the polygon is convex, we can be confident that none of the boundary edges of the polygon will be intersected by these chords, and that all of them lie within the polygon. The simplest algorithm for constructing general polygon triangulations tries each of the $O(n^2)$ possible chords and inserts them if they do not intersect a boundary edge or previously inserted chord. There are practical algorithms that run in $O(n \lg n)$ time and theoretically interesting algorithms that run in linear time. See the Implementations and Notes section for details.

Implementations: Triangle, by Jonathan Shewchuk, is an award-winning C language code that generates Delaunay triangulations, constrained Delaunay triangulations (forced to have certain edges), and quality-conforming Delaunay triangulations (which avoid small angles by inserting extra points). It has been widely used for finite element analysis and is fast and robust. Triangle is the first thing I would try if I needed a two-dimensional triangulation code. It is available at *http://www.cs.cmu.edu/~quake/triangle.html*.

Fortune's Sweep2 is a widely used 2D code for Voronoi diagrams and Delaunay triangulations, written in C. This code may be simpler to work with if all you need is the Delaunay triangulation of points in the plane. It is based on his sweepline algorithm [For87] for Voronoi diagrams and is available from Netlib (see Section 19.1.5 (page 659)) at *http://www.netlib.org/voronoi/*.

Mesh generation for finite element methods is an enormous field. Steve Owen's Meshing Research Corner (*http://www.andrew.cmu.edu/user/sowen/mesh.html*) provides a comprehensive overview of the literature, with links to an enormous variety of software. QMG (*http://www.cs.cornell.edu/Info/People/vavasis/qmg-home.html*) is a particularly recommended code.

Both the CGAL (*www.cgal.org*) and LEDA (see Section 19.1.1 (page 658)) libraries offer C++ implementations of an extensive variety of triangulation algorithms for two and three dimensions, including constrained and furthest site Delaunay triangulations.

Higher-dimensional Delaunay triangulations are a special case of higher-dimensional convex hulls. Qhull [BDH97] is a popular low-dimensional convex hull code, say for from two to about eight dimensions. It is written in C and can also construct Delaunay triangulations, Voronoi vertices, furthest-site Voronoi

vertices, and half-space intersections. Qhull has been widely used in scientific applications and has a well-maintained homepage at *http://www.qhull.org/*. Another choice is Ken Clarkson's higher-dimensional convex-hull code, *Hull*, available at *http://www.netlib.org/voronoi/hull.html*.

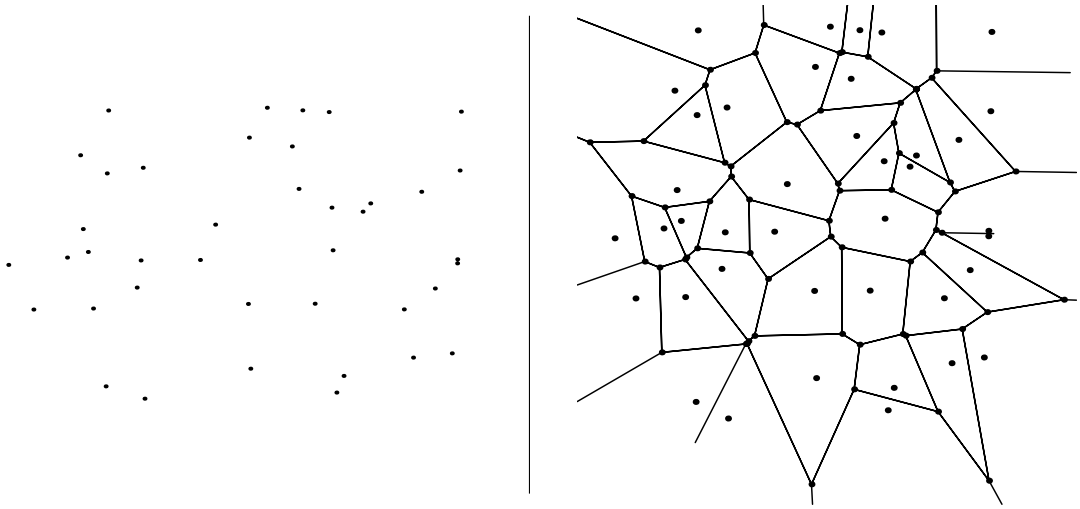
Notes: After a long search, Chazelle [Cha91] discovered a linear-time algorithm for triangulating a simple polygon. This algorithm is sufficiently hopeless to implement that it qualifies more as an existence proof. The first $O(n \lg n)$ algorithm for polygon triangulation was given by [GJPT78]. An $O(n \lg \lg n)$ algorithm by Tarjan and Van Wyk [TW88] followed before Chazelle's result. Bern [Ber04a] gives a recent survey on polygon and point-set triangulation.

The *International Meshing Roundtable* is an annual conference for people interested in mesh and grid generation. Excellent surveys on mesh generation include [Ber02, Ede06].

Linear-time algorithms for triangulating monotone polygons have been long known [GJPT78], and are the basis of algorithms for triangulating simple polygons. A polygon is monotone when there exists a direction d such that any line with slope d intersects the polygon in at most two points.

A heavily studied class of optimal triangulations seeks to minimize the total length of the chords used. The computational complexity of constructing this *minimum weight triangulation* was resolved when Rote [MR06] proved it NP-complete. Thus interest has shifted to provably good approximation algorithms. The minimum weight triangulation of a convex polygon can be found in $O(n^3)$ time using dynamic programming, as discussed in Section 8.6.1 (page 300).

Related Problems: Voronoi diagrams (see page 576), polygon partitioning (see page 601).



INPUT OUTPUT

17.4 Voronoi Diagrams

Input description: A set S of points $p_1, \ldots, p_n$.

Problem description: Decompose space into regions around each point such that all points in the region around p_i are closer to p_i than any other point in S.

Discussion: Voronoi diagrams represent the region of influence around each of a given set of sites. If these sites represent the locations of McDonald's restaurants, the Voronoi diagram partitions space into cells around each restaurant. For each person living in a particular cell, the defining McDonald's represents the closest place to get a Big Mac.

Voronoi diagrams have a surprising variety of applications:

- *Nearest neighbor search* – Finding the nearest neighbor of query point q from among a fixed set of points S is simply a matter of determining the cell in the Voronoi diagram of S that contains q. See Section 17.5 (page 580) for more details.

- *Facility location* – Suppose McDonald's wants to open another restaurant. To minimize interference with existing McDonald's, it should be located as far away from the closest restaurant as possible. This location is always at a vertex of the Voronoi diagram, and can be found in a linear-time search through all the Voronoi vertices.

- *Largest empty circle* – Suppose you needed to obtain a large, contiguous, undeveloped piece of land on which to build a factory. The same condition used to select McDonald's locations is appropriate for other undesirable facilities, namely that they be as far as possible from any relevant sites of interest. A Voronoi vertex defines the center of the largest empty circle among the points.

- *Path planning* – If the sites of S are the centers of obstacles we seek to avoid, the edges of the Voronoi diagram define the possible channels that maximize the distance to the obstacles. Thus the "safest" path among the obstacles will stick to the edges of the Voronoi diagram.

- *Quality triangulations* – In triangulating a set of points, we often desire nice, fat triangles that avoid small angles and skinny triangles. The *Delaunay triangulation* maximizes the minimum angle over all triangulations. Furthermore, it is easily constructed as the dual of the Voronoi diagram. See Section 17.3 (page 572) for details.

Each edge of a Voronoi diagram is a segment of the perpendicular bisector of two points in S, since this is the line that partitions the plane between the points. The conceptually simplest method to construct Voronoi diagrams is randomized incremental construction. To add another site to the diagram, we locate the cell that contains it and add the perpendicular bisectors separating this new site from all sites defining impacted regions. If the sites are inserted in random order, only a small number of regions are likely to be impacted with each insertion.

However, the method of choice is Fortune's sweepline algorithm, especially since robust implementations of it are readily available. The algorithm works by projecting the set of sites in the plane into a set of cones in three dimensions such that the Voronoi diagram is defined by projecting the cones back onto the plane. Advantages of Fortune's algorithm include that (1) it runs in optimal $\Theta(n \log n)$ time, (2) it is reasonable to implement, and (3) we need not store the entire diagram if we can use it as we sweep over it.

There is an interesting relationship between convex hulls in $d+1$ dimensions and Delaunay triangulations (or equivalently Voronoi diagrams) in d-dimensions. By projecting each site in E^d $(x_1, x_2, \ldots, x_d)$ into the point $(x_1, x_2, \ldots, x_d, \sum_{i=1}^{d} x_i^2)$, taking the convex hull of this $(d+1)$-dimensional point set and then projecting back into d dimensions, we obtain the Delaunay triangulation. Details are given in the Notes section, but this provides the best way to construct Voronoi diagrams in higher dimensions. Programs that compute higher-dimensional convex hulls are discussed in Section 17.2 (page 568).

Several important variations of standard Voronoi diagrams arise in practice. See the references below for details:

- *Non-Euclidean distance metrics* – Recall that Voronoi diagrams decompose space into regions of influence around each of the given sites. We have assumed that Euclidean distance measures influence, but this is inappropriate

for certain applications. If people drive to McDonald's, the time it takes to get there depends upon where the major roads are. Efficient algorithms are known for constructing Voronoi diagrams under a variety of different metrics, and for curved or constrained objects.

- *Power diagrams* – These structures decompose space into regions of influence around the sites, where the sites are no longer constrained to have all the same power. Imagine a map of radio stations operating at a given frequency. The region of influence around a station depends both on the power of its transmitter and the position/power of neighboring transmitters.

- *Kth-order and furthest-site diagrams* – The idea of decomposing space into regions sharing some property can be taken beyond closest-point Voronoi diagrams. Any point within a single cell of the kth-order Voronoi diagram shares the same set of k's closest points in S. In furthest-site diagrams, any point within a particular region shares the same furthest point in S. Point location (see Section 17.7 (page 587)) on these structures permits fast retrieval of the appropriate points.

Implementations: Fortune's Sweep2 is a widely used 2D code for Voronoi diagrams and Delaunay triangulations, written in C. This code is simple to work with if all you need is the Voronoi diagram. It is based on his sweepline algorithm [For87] for Voronoi diagrams and is available from Netlib (see Section 19.1.5 (page 659)) at *http://www.netlib.org/voronoi/*.

Both the CGAL (*www.cgal.org*) and LEDA (see Section 19.1.1 (page 658)) libraries offer C++ implementations of a variety of Voronoi diagram and Delaunay triangulation algorithms in two and three dimensions.

Higher-dimensional and furthest-site Voronoi diagrams can be constructed as a special case of higher-dimensional convex hulls. Qhull [BDH97] is a popular low-dimensional convex-hull code, useful for from two to about eight dimensions. It is written in C and can also construct Delaunay triangulations, Voronoi vertices, furthest-site Voronoi vertices, and half-space intersections. Qhull has been widely used in scientific applications and has a well-maintained homepage at *http://www.qhull.org/*. Another choice is Ken Clarkson's higher-dimensional convex-hull code, *Hull*, available at *http://www.netlib.org/voronoi/hull.html*.

Notes: Voronoi diagrams were studied by Dirichlet in 1850 and are occasionally referred to as *Dirichlet tessellations*. They are named after G. Voronoi, who discussed them in a 1908 paper. In mathematics, concepts get named after the last person to discover them.

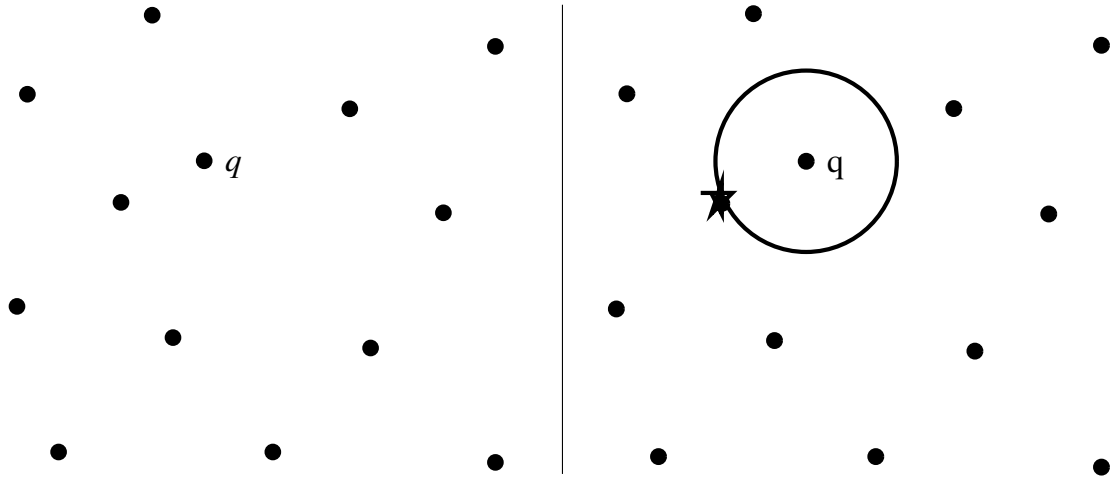
The book by Okabi, et al. [OBSC00] is the most complete treatment of Voronoi diagrams and their applications. Aurenhammer [Aur91] and Fortune [For04] provide excellent surveys on Voronoi diagrams and associated variants such as power diagrams. The first $O(n \lg n)$ algorithm for constructing Voronoi diagrams was based on divide-and-conquer and is due to Shamos and Hoey [SH75]. Good expositions of both Fortune's sweepline algorithm [For87] for constructing Voronoi diagrams in $O(n \lg n)$ and the relationship

between Delaunay triangulations and $(d+1)$-dimensional convex hulls [ES86] include [dBvKOS00, O'R01].

In a kth-order Voronoi diagram, we partition the plane such that each point in a region is closest to the same set of k sites. Using the algorithm of [ES86], the complete set of kth-order Voronoi diagrams can be constructed in $O(n^3)$ time. By doing point location on this structure, the k nearest neighbors to a query point can be found in $O(k + \lg n)$. Expositions on kth-order Voronoi diagrams include [O'R01, PS85].

The smallest enclosing circle problem can be solved in $O(n \lg n)$ time using $(n-1)$st order Voronoi diagrams [PS85]. In fact, there exists a linear-time algorithm based on low-dimensional linear programming [Meg83]. A linear algorithm for computing the Voronoi diagram of a convex polygon is given by [AGSS89].

Related Problems: Nearest neighbor search (see page 580), point location (see page 587), triangulation (see page 572).



INPUT OUTPUT

17.5 Nearest Neighbor Search

Input description: A set S of n points in d dimensions; a query point q.

Problem description: Which point in S is closest to q?

Discussion: The need to quickly find the nearest neighbor of a query point arises in a variety of geometric applications. The classic example involves designing a system to dispatch emergency vehicles to the scene of a fire. Once the dispatcher learns the location of the fire, she uses a map to find the firehouse closest to this point to minimize transportation delays. This situation occurs in any application mapping customers to service providers.

A nearest-neighbor search is also important in classification. Suppose we are given a collection of numerical data about people (say age, height, weight, years of education, and income level) each of whom has been labeled as Democrat or Republican. We seek a classifier to decide which way a given person is likely to vote. Each of the people in our data set is represented by a party-labeled point in d-dimensional space. A simple classifier can be built by assigning the new point to the party affiliation of its nearest neighbor.

Such nearest-neighbor classifiers are widely used, often in high-dimensional spaces. The vector-quantization method of image compression partitions an image into 8×8 pixel regions. This method uses a predetermined library of several thousand 8×8 pixel tiles and replaces each image region by the most similar library tile. The most similar tile is the point in 64-dimensional space that is closest to

the image region in question. Compression is achieved by reporting the identifier of the closest library tile instead of the 64 pixels, at some loss of image fidelity.

Issues arising in nearest-neighbor search include:

- *How many points are you searching?* – When your data set contains only a small number of points (say $n \leq 100$), or if only a few queries are ever destined to be performed, the simple approach is best. Compare the query point q against each of the n data points. Only when fast queries are needed for large numbers of points does it pay to consider more sophisticated methods.

- *How many dimensions are you working in?* – A nearest neighbor search gets progressively harder as the dimensionality increases. The *kd*-tree data structure, presented in Section 12.6 (page 389), does a very good job in moderate-dimensional spaces—even the plane. Still, above 20 dimensions or so, kd-tree search degenerates pretty much to a linear search through the data points. Searches in high-dimensional spaces become hard because a sphere of radius r, representing all the points with distance $\leq r$ from the center, progressively fills up less volume relative to a cube as the dimensionality increases. Thus, any data structure based on partitioning points into enclosing volumes will become progressively less effective.

  In two dimensions, Voronoi diagrams (see Section 17.4 (page 576)) provide an efficient data structure for nearest-neighbor queries. The Voronoi diagram of a point set in the plane decomposes the plane into regions such that the cell containing data point p consists of all points that are nearer to p than any other point in S. Finding the nearest neighbor of query point q reduces to finding which Voronoi diagram cell contains q and reporting the data point associated with it. Although Voronoi diagrams can be built in higher dimensions, their size rapidly grows to the point of unusability.

- *Do you really need the exact nearest neighbor?* – Finding the exact nearest neighbor in a very high dimensional space is hard work; indeed, you probably won't do better than a linear (brute force) search. However, there are algorithms/heuristics that can give you a reasonably close neighbor of your query point fairly quickly.

  One important technique is *dimension reduction*. Projections exist that map any set of n points in d-dimensions into a $d' = O(\lg n/\epsilon^2)$-dimensional space such that distance to the nearest neighbor in the low-dimensional space is within $(1+\epsilon)$ times that of the actual nearest neighbor. Projecting the points onto a random hyperplane of dimension d' in E^d will likely do the trick.

  Another idea is adding some randomness when you search your data structure. A *kd*-tree can be efficiently searched for the cell containing the query point q—a cell whose boundary points are good candidates to be close neighbors. Now suppose we search for a point q', which is a small random perturbations of q. It should land in a different but nearby cell, one of whose

boundary points might prove to be an even closer neighbor of q. Repeating such random queries gives us a way to productively use exactly as much computing time as we are willing to spend to improve the answer.

- *Is your data set static or dynamic?* – Will there be occasional insertions or deletions of new data points in your application? If these are very rare events, it might pay to rebuild your data structure from scratch each time. If they are frequent, select a version of the kd-tree that supports insertions and deletions.

The nearest neighbor graph on a set S of n points links each vertex to its nearest neighbor. This graph is a subgraph of the Delaunay triangulation, and so can be computed in $O(n \log n)$. This is quite a bargain, since it takes $\Theta(n \log n)$ time just to discover the closest pair of points in S.

As a lower bound, the closest-pair problem reduces to sorting in one dimension. In a sorted set of numbers, the closest pair corresponds to two numbers that lie next to each other, so we need only check that which is the minimum gap between the $n - 1$ adjacent pairs. The limiting case occurs when the closest pair are zero distance apart, meaning that the elements are not unique.

Implementations: *ANN* is a C++ library for both exact and approximate nearest neighbor searching in arbitrarily high dimensions. It performs well for searches over hundreds of thousands of points in up to about 20 dimensions. It supports all l_p distance norms, including Euclidean and Manhattan distance, and is available at *http://www.cs.umd.edu/~mount/ANN/*. It is the first code I would turn to for nearest neighbor search.

Samet's spatial index demos (*http://donar.umiacs.umd.edu/quadtree/*) provide a series of Java applets illustrating many variants of *kd*-trees, in association with [Sam06]. *KDTREE 2* contains C++ and Fortran 95 implementations of *kd*-trees for efficient nearest nearest neighbor search in many dimensions. See *http://arxiv.org/abs/physics/0408067*.

Ranger [MS93] is a tool for visualizing and experimenting with nearest-neighbor and orthogonal-range queries in high-dimensional data sets, using multidimensional search trees. Four different search data structures are supported by *Ranger*: naive kd-trees, median kd-trees, nonorthogonal kd-trees, and the vantage point tree. It is available in the algorithm repository *http://www.cs.sunysb.edu/~algorith*.

Nearpt3 is a special-purpose code for nearest-neighbor search of extremely large point sets in three dimensions. See *http://wrfranklin.org/Research/nearpt3*.

See Section 17.4 (page 576) for a complete collection of Voronoi diagram implementations. In particular, CGAL (*www.cgal.org*) and LEDA (see Section 19.1.1 (page 658)) provide implementations of Voronoi diagrams in C++, as well as planar point location to make effective use of them for nearest-neighbor search.

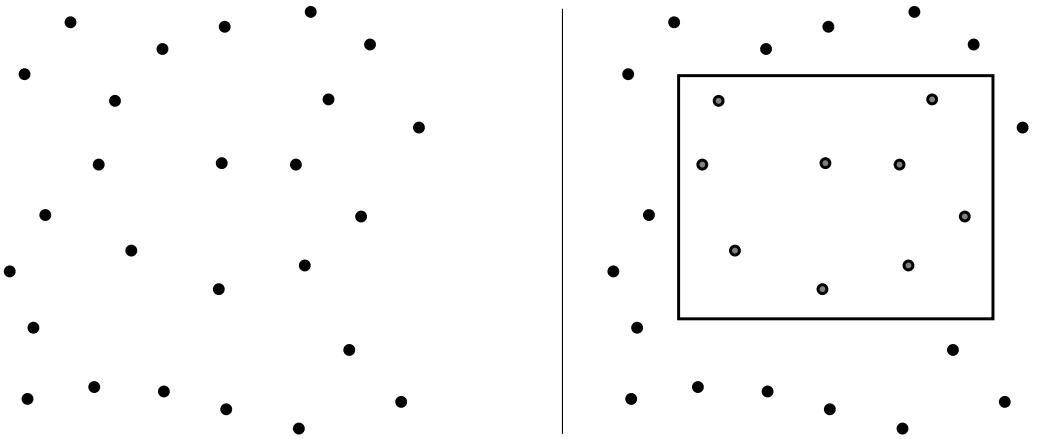
Notes: Indyk [Ind04] ably surveys recent results in approximate nearest neighbor search in high dimensions based on random projection methods. Both theoretical [IM04] and empirical [BM01] results indicate that the method preserves distances quite nicely.

The theoretical guarantees underlying Ayra and Mount's approximate nearest neighbor code *ANN* [AM93, AMN+98] are somewhat different. A sparse weighted graph structure is built from the data set, and the nearest neighbor is found by starting at a random point and greedily walking towards the query point in the graph. The closest point found over several random trials is declared the winner. Similar data structures hold promise for other problems in high-dimensional spaces. Nearest-neighbor search was a subject of the Fifth DIMACS challenge, as reported in [GJM02].

Samet [Sam06] is the best reference on kd-trees and other spatial data structures. All major (and many minor) variants are developed in substantial detail. A shorter survey [Sam05] is also available. The technique of using random perturbations of the query point is due to [Pan06].

Good expositions on finding the closest pair of points in the plane [BS76] include [CLRS01, Man89]. These algorithms use a divide-and-conquer approach instead of just selecting from the Delaunay triangulation.

Related Problems: Kd-trees (see page 389), Voronoi diagrams (see page 576), range search (see page 584).



INPUT OUTPUT

17.6 Range Search

Input description: A set S of n points in E^d and a query region Q.

Problem description: What points in S lie within Q?

Discussion: Range-search problems arise in database and geographic information system (GIS) applications. Any data object with d numerical fields, such as person with height, weight, and income, can be modeled as a point in d-dimensional space. A *range query* describes a region in space and asks for all points or the number of points in the region. For example, asking for all people with income between \$0 and \$10,000, with height between 6'0" and 7'0", and weight between 50 and 140 lbs., defines a box containing people whose wallet and body are both thin.

The difficulty of range search depends on several factors:

- *How many range queries are you going to perform?* – The simplest approach to range search tests each of the n points one by one against the query polygon Q. This works just fine when the number of queries will be small. Algorithms to test whether a point is within a given polygon are presented in Section 17.7 (page 587).

- *What shape is your query polygon?* – The easiest regions to query against are *axis-parallel rectangles*, because the inside/outside test reduces to verifying whether each coordinate lies within a prescribed range. The input-output figure illustrates such an *orthogonal range query*.

When querying against a nonconvex polygon, it will pay to partition your polygon into convex pieces or (even better) triangles and then query the point set against each one of the pieces. This works because it is quick and easy to test whether a point lies inside a convex polygon. Algorithms for such convex decompositions are discussed in Section 17.11 (page 601).

- *How many dimensions?* – A general approach to range queries builds a kd-tree on the point set, as discussed in Section 12.6 (page 389). A depth-first traversal of the kd-tree is performed for the query, with each tree node expanded only if the associated rectangle intersects the query region. The entire tree might have to be traversed for sufficiently large or misaligned query regions, but in general kd-trees lead to an efficient solution. Algorithms with more efficient worst-case performance are known in two dimensions, but kd-trees are likely to work just fine in the plane. In higher dimensions, they provide the only viable solution to the problem.

- *Is your point set static, or might there be insertions/deletions?* – A clever practical approach to range search and many other geometric searching problems is based on Delaunay triangulations. Delaunay triangulation edges connect each point p to nearby points, including its nearest neighbor. To perform a range query, we start by using planar point location (see Section 17.7 (page 587)) to quickly identify a triangle within the region of interest. We then do a depth-first search around a vertex of this triangle, pruning the search whenever it visits a point too distant to have interesting undiscovered neighbors. This should be efficient, because the total number of points visited should be roughly proportional to the number within the query region.

  One nice thing about this approach is that it is relatively easy to employ "edge-flip" operations to fix up a Delaunay triangulation following a point insertion or deletion. See Section 17.3 (page 572) for more details.

- *Can I just count the number of points in a region, or do I have to identify them?* – For many applications, it suffices to count the number of points in a region instead of returning them. Harkening back to the introductory example, we may want to know whether there are more thin/poor people or rich/fat ones. The need to find the densest or emptiest region in space often arises, and this problem can be solved by counting range queries.

  A nice data structure for efficiently answering such aggregate range queries is based on the dominance ordering of the point set. A point x is said to *dominate* point y if y lies both below and to the left of x. Let $DOM(p)$ be a function that counts the number of points in S that are dominated by p. The number of points m in the orthogonal rectangle defined by $x_{\min} \leq x \leq x_{\max}$ and $y_{\min} \leq y \leq y_{\max}$ is given by

$$m = DOM(x_{\max}, y_{\max}) - DOM(x_{\max}, y_{\min}) - DOM(x_{\min}, y_{\max}) + DOM(x_{\min}, y_{\min})$$

The second additive term corrects for the points for the lower left-hand corner that have been subtracted away twice.

To answer arbitrary dominance queries efficiently, partition the space into n^2 rectangles by drawing a horizontal and vertical line through each of the n points. The set of dominated points is identical for each point in any rectangle, so the dominance count of the lower left-hand corner of each rectangle can be precomputed, stored, and reported for any query point within it. Range queries reduce to binary search and thus take $O(\lg n)$ time. Unfortunately, this data structure takes quadratic space. However, the same idea can be adapted to kd-trees to create a more space-efficient search structure.

Implementations: Both CGAL (*www.cgal.org*) and LEDA (see Section 19.1.1 (page 658)) use a dynamic Delaunay triangulation data structure to support circular, triangular, and orthogonal range queries. Both libraries also provide implementations of range tree data structures, which support orthogonal range queries in $O(k + \lg^2 n)$ time where n is the complexity of the subdivision and k is the number of points in the rectangular region.

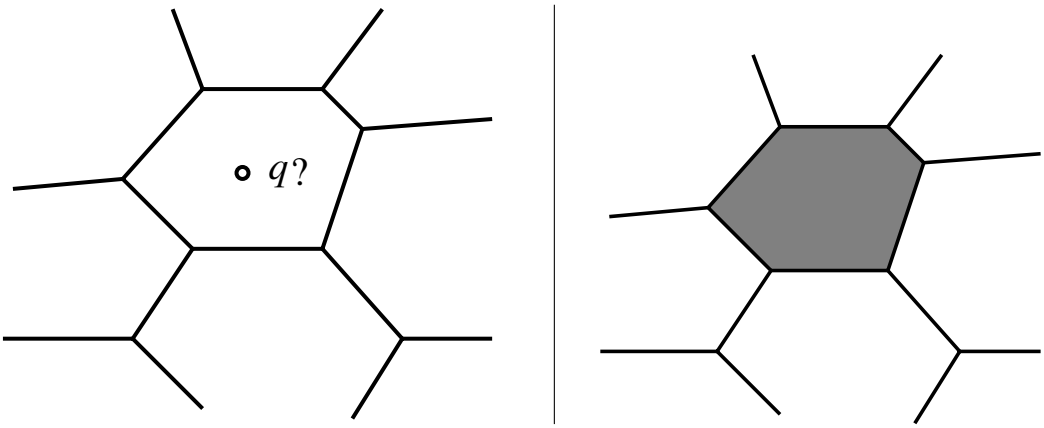
ANN is a C++ library for both exact and approximate nearest neighbor searching in arbitrarily high dimensions. It performs well for searches over hundreds of thousands of points in up to about 20 dimensions. It supports fixed-radius, nearest-neighbor queries over all l_p distance norms, which can be used to approximate circular and orthogonal range queries under the l_2 and l_1 norms, respectively. *ANN* is available at *http://www.cs.umd.edu/∼mount/ANN/*.

Ranger is a tool for visualizing and experimenting with nearest-neighbor and orthogonal-range queries in high-dimensional data sets, using multidimensional search trees. Four different search data structures are supported by *Ranger*: naive kd-trees, median kd-trees, nonorthogonal kd-trees, and the vantage point tree. For each of these, *Ranger* supports queries in up to 25 dimensions under any Minkowski metric. It is available in the algorithm repository.

Notes: Good expositions on data structures with worst-case $O(\lg n + k)$ performance for orthogonal-range searching [Wil85] include [dBvKOS00, PS85]. Exposition on *kd*-trees for orthogonal range queries in two dimensions appear in [dBvKOS00, PS85]. Their worst-case performance can be very bad; [LW77] describes an instance in two dimensions requiring $O(\sqrt{n})$ time to report that a rectangle is empty.

The problem becomes considerably more difficult for nonorthogonal range queries, where the query region is not an axis-aligned rectangle. For half-plane intersection queries, $O(\lg n)$ time and linear space suffice [CGL85]; for range searching with simplex query regions (such as a triangle in the plane), lower bounds preclude efficient worst-case data structures. See Agrawal [Aga04] for a survey and discussion.

Related Problems: Kd-trees (see page 389), point location (see page 587).



INPUT OUTPUT

17.7 Point Location

Input description: A decomposition of the plane into polygonal regions and a query point q.

Problem description: Which region contains the query point q?

Discussion: Point location is a fundamental subproblem in computational geometry, usually needed as an ingredient to solve larger geometric problems. In a typical police dispatch system, the city will be partitioned into different precincts or districts. Given a map of regions and a query point (the crime scene), the system must identify which region contains the point. This is exactly the problem of planar point location. Variations include:

- *Is a given point inside or outside of polygon P?* – The simplest version of point location involves only two regions, inside-P and outside-P, and asks which contains a given query point. For polygons with lots of narrow spirals, this can be surprisingly difficult to tell by inspection. The secret to doing it both by eye or machine is to draw a ray starting from the query point and ending beyond the furthest extent of the polygon. Count the number of times the polygon crosses through an edge. The query point will lie within the polygon iff this number is odd. The case of the line passing through a vertex instead of an edge is evident from context, since we are counting the number of times we pass through the boundary of the polygon. Testing each of the n edges for intersection against the query ray takes $O(n)$ time. Faster

algorithms for convex polygons are based on binary search, and take $O(\lg n)$ time.

- *How many queries must be performed?* – It is always possible to perform this inside-polygon test separately on each region in a given planar subdivision. However, it would be wasteful to perform many such point location queries on the same subdivision. It would be much better to construct a grid-like or tree-like data structure on top of our subdivision to get us near the correct region quickly. Such search structures are discussed in more detail below.

- *How complicated are the regions of your subdivision?* – More sophisticated inside-outside tests are required when the regions of your subdivision are arbitrary polygons. By triangulating all polygonal regions first, each inside-outside test reduces to testing whether a point is in a triangle. Such tests can be made particularly fast and simple, at the minor cost of recording the full polygon name for each triangle. An added benefit is that the smaller your regions are, the better grid-like or tree-like superstructures are likely to perform. Some care should be taken when you triangulate to avoid long skinny triangles, as discussed in Section 17.3 (page 572).

- *How regularly sized and spaced are your regions?* – If all resulting triangles are about the same size and shape, the simplest point location method imposes a regularly-spaced $k \times k$ grid of horizontal and vertical lines over the entire subdivision. For each of the k^2 rectangular regions, we maintain a list of all the regions that are at least partially contained within the rectangle. Performing a point location query in such a *grid file* involves a binary search or hash table lookup to identify which rectangle contains query point q, and then searching each region in the resulting list to identify the right one.

  Such grid files can perform very well, provided that each triangular region overlaps only a few rectangles (thus minimizing storage space) and each rectangle overlaps only a few triangles (thus minimizing search time). Whether it performs well depends on the regularity of your subdivision. Some flexibility can be achieved by spacing the horizontal lines irregularly, depending upon where the regions actually lie. The *slab method*, discussed below, is a variation on this idea that guarantees worst-case efficient point location at the cost of quadratic space.

- *How many dimensions will you be working in?* – In three or more dimensions, some flavor of kd-tree will almost certainly be the point-location method of choice. They may also be the right answer for planar subdivisions that are too irregular for grid files.

  Kd-trees, described in Section 12.6 (page 389), decompose the space into a hierarchy of rectangular boxes. At each node in the tree, the current box is split into a small number (typically 2, 4, or 2^d for dimension d) of smaller boxes. Each leaf box is labeled with the (small) set regions that are at least

partially contained in the box. The point location search starts at the root of the tree and keeps traversing down the child whose box contains the query point q. When the search hits a leaf, we test each of the relevant regions to see which one of them contains q. As with grid files, we hope that each leaf contains a small number of regions and that each region does not cut across too many leaf cells.

- *Am I close to the right cell?* – Walking is a simple point-location technique that might even work well beyond two dimensions. Start from an arbitrary point p in an arbitrary cell, hopefully close to the query point q. Construct the line (or ray) from p to q and identify which face of the cell this hits (a so-called *ray shooting query*). Such queries take constant time in triangulated arrangements.

  Proceeding to the neighboring cell through this face gets us one step closer to the target. The expected path length will be $O(n^{1/d})$ for sufficiently regular d-dimensional arrangements, although linear in the worst case.

The simplest algorithm to guarantee $O(\lg n)$ worst-case access is the *slab* method, which draws horizontal lines through each vertex, thus creating $n + 1$ "slabs" between the lines. Since the slabs are defined by horizontal lines, finding the slab containing a particular query point can be done using a binary search on the y-coordinate of q. Since there can be no vertices within any slab, the region containing a point within a slab can be identified by a second binary search on the edges that cross the slab. The catch is that a binary search tree must be maintained for each slab, for a worst-case of $O(n^2)$ space. A more space-efficient approach based on building a hierarchy of triangulations over the regions also achieves $O(\lg n)$ for search and is discussed in the notes below.

Worst-case efficient computational geometry methods either require a lot of storage or are fairly complicated to implement. We identify implementations of worst-case methods below, which are worth at least experimenting with. However, we recommend kd-trees for most general point-location applications.

Implementations: Both CGAL (*www.cgal.org*) and LEDA (see Section 19.1.1 (page 658)) provide excellent support for maintaining planar subdivisions in C++. CGAL favors a jump-and-walk strategy, although a worst-case logarithmic search is also provided. LEDA implements expected $O(\lg n)$ point location using partially persistent search trees.

ANN is a C++ library for both exact and approximate nearest-neighbor searching in arbitrarily high dimensions. It can be used to quickly identify a nearby cell boundary point to begin walking from. Check it out at *http://www.cs.umd.edu/~mount/ANN/*.

Arrange is a package for maintaining arrangements of polygons in either the plane or on the sphere. Polygons may be degenerate, and hence represent arrangements of lines. A randomized incremental construction algorithm is used, and efficient point location on the arrangement is supported.

Arrange is written in C by Michael Goldwasser and is available from *http://euler.slu.edu/∼goldwasser/publications/*.

Routines (in C) to test whether a point lies in a simple polygon have been provided by [O'R01, SR03].

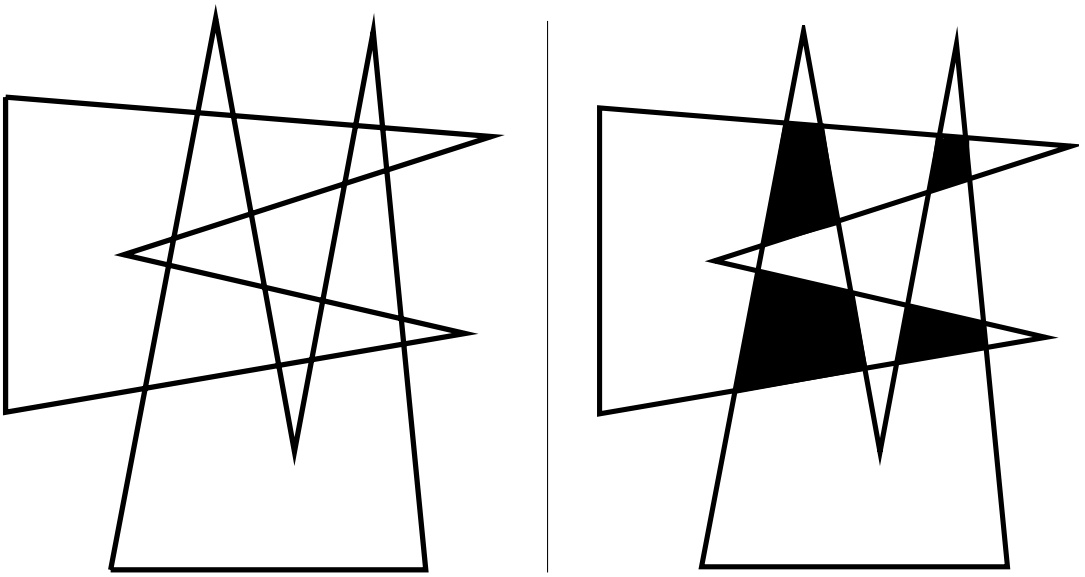
Notes: Snoeyink [Sno04] gives an excellent survey of the state-of-the-art in point location, both theoretical and practical. Very thorough treatments of deterministic planar-point location data structures are provided by [dBvKOS00, PS85].

Tamassia and Vismara [TV01] use planar point location as a case study of geometric algorithm engineering, in Java. An experimental study of algorithms for planar point location is described in [EKA84]. The winner was a bucketing technique akin to the grid file.

The elegant triangle refinement method of Kirkpatrick [Kir83] builds a hierarchy of triangulations above the actual planar subdivision such that each triangle on a given level intersects only a constant number of triangles on the following level. Since each triangulation is a fraction of the size of the subsequent one, the total space is obtained by summing up a geometric series and hence is linear. Furthermore, the height of the hierarchy is $O(\lg n)$, ensuring fast query times. An alternative algorithm realizing the same time bounds is [EGS86]. The slab method described above is due to [DL76] and is presented in [PS85]. Expositions on the inside-outside test for simple polygons include [Hai94, O'R01, PS85, SR03].

More recently, there has been interest in dynamic data structures for point location, which support fast incremental updates of the planar subdivision (such as insertions and deletions of edges and vertices) as well as fast point location. Chiang and Tamassia's [CT92] survey is an appropriate place to begin, with updated references in [Sno04].

Related Problems: Kd-trees (see page 389), Voronoi diagrams (see page 576), nearest neighbor search (see page 580).



INPUT OUTPUT

17.8 Intersection Detection

Input description: A set S of lines and line segments $l_1, \ldots, l_n$ or a pair of polygons or polyhedra P_1 and P_2.

Problem description: Which pairs of line segments intersect each other? What is the intersection of P_1 and P_2?

Discussion: Intersection detection is a fundamental geometric primitive with many applications. Picture a virtual-reality simulation of an architectural model for a building. The illusion of reality vanishes the instant the virtual person walks through a virtual wall. To enforce such physical constraints, any such intersection between polyhedral models must be immediately detected and the operator notified or constrained.

Another application of intersection detection is design rule checking for VLSI layouts. A minor design defect resulting in two crossing metal strips could short out the chip, but such errors can be detected before fabrication, using programs to find all intersections between line segments.

Issues arising in intersection detection include:

- *Do you want to compute the intersection or just report it?* – We distinguish between intersection detection and computing the actual intersection.

Detecting that an intersection exists can be substantially easier, and often suffices. For the virtual-reality application, it might not be important to know exactly where we hit the wall—just that we hit it.

- *Are you intersecting lines or line segments?* – The big difference here is that any two lines with different slopes intersect in at exactly one point. All the points of intersections can be found in $O(n^2)$ time by comparing each pair of lines. Constructing the arrangement of the lines provides more information than just the intersection points, and is discussed in Section 17.15 (page 614).

  Finding all the intersections between n line segments is considerably more challenging. Even the basic primitive of testing whether two line segments intersect is not as trivial, as discussed in Section 17.1 (page 564). Of course, we could explicitly test each line segment pair and thus find all intersections in $O(n^2)$ time, but faster algorithms exist when there are few intersection points.

- *How many intersection points do you expect?* – In VLSI design-rule checking, we expect the set of line segments to have few if any intersections. What we seek is an algorithm whose time is *output sensitive*, taking time proportional to the number of intersection points.

  Such output-sensitive algorithms exist for line-segment intersection. The fastest algorithm takes $O(n \lg n + k)$ time, where k is the number of intersections. These algorithms are based on the planar sweepline approach.

- *Can you see point x from point y?* – Visibility queries ask whether vertex x can see vertex y unobstructed in a room full of obstacles. This can be phrased as the following line-segment intersection problem: does the line segment from x to y intersect any obstacle? Such visibility problems arise in robot motion planning (see Section 17.14) and in hidden-surface elimination for computer graphics.

- *Are the intersecting objects convex?* – Better intersection algorithms exist when the line segments form the boundaries of polygons. The critical issue here becomes whether the polygons are convex. Intersecting a convex n-gon with a convex m-gon can be done in $O(n + m)$ time, using the sweepline algorithm discussed next. This is possible because the intersection of two convex polygons must form another convex polygon with at most $n + m$ vertices.

  However, the intersection of two nonconvex polygons is not so well behaved. Consider the intersection of two "combs" generalizing the Picasso-like frontspiece to this section. As illustrated, the intersection of nonconvex polygons may be disconnected and have quadratic size in the worst case.

Intersecting polyhedra is more complicated than polygons, because two polyhedra can intersect even when no edges do. Consider the example of a needle piercing the interior of a face. In general, however, the same issues arise for both polygons and polyhedra.

- *Are you searching for intersections repeatedly with the same basic objects?* – In the walk-through application just described, the room and the objects in it don't change between one scene and the next. Only the person moves, and intersections are rare.

  One common technique is to approximate the objects in the scene by simpler objects that enclose them, such as boxes. Whenever two enclosing boxes intersect, then the underlying objects *might* intersect, and so further work is necessary to decide the issue. However, it is much more efficient to test whether simple boxes intersect than more complicated objects, so we win if collisions are rare. Many variations on this theme are possible, but this idea leads to large performance improvements for complicated environments.

Planar sweep algorithms can be used to efficiently compute the intersections among a set of line segments, or the intersection/union of two polygons. These algorithms keep track of interesting changes as we sweep a vertical line from left to right over the data. At its leftmost position, the line intersects nothing, but as it moves to the right, it encounters a series of events:

- *Insertion* – The leftmost point of a line segment may be encountered, and it is now available to intersect some other line segment.

- *Deletion* – The rightmost point of a line segment is encountered. This means that we have completely swept over the segment, and so it can be deleted from further consideration.

- *Intersection* – If the active line segments that intersect the sweep line are maintained as sorted from top to bottom, the next intersection must occur between neighboring line segments. After this intersection, these two line segments swap their relative order.

Keeping track of what is going on requires two data structures. The future is maintained by an *event queue*, or a priority queue ordered by the x-coordinate of all possible future events of interest: insertion, deletion, and intersection. See Section 12.2 (page 373) for priority queue implementations. The present is represented by the *horizon*—an ordered list of line segments intersecting the current position of the sweepline. The horizon can be maintained using any dictionary data structure, such as a balanced tree.

To adapt this approach to compute the intersection or union of polygons, we modify the processing of the three basic event types. This algorithm can be considerably simplified for pairs of convex polygons, since (1) at most four polygon

edges intersect the sweepline, so no horizon data structure is needed, and (2) no event-queue sorting is needed, since we can start from the leftmost vertex of each polygon and proceed to the right by following the polygonal ordering.

Implementations: Both LEDA (see Section 19.1.1 (page 658)) and CGAL (*www.cgal.org*) offers extensive support for line segment and polygonal intersection. In particular, they provide a C++ implementation of the Bentley-Ottmann sweepline algorithm [BO79], finding all k intersection points between n line segments in the plane in $O((n+k)\lg n)$ time.

O'Rourke [O'R01] provides a robust program in C to compute the intersection of two convex polygons. See Section 19.1.10 (page 662).

The University of North Carolina GAMMA group has produced several efficient collision detection libraries, of which *SWIFT++* [EL01] is the most recent member. It can detect intersection, compute approximate/exact distances between objects, and determine object-pair contacts in scenes composed of rigid polyhedral models. See *http://www.cs.unc.edu/∼geom/collide/* for pointers to all of these libraries.

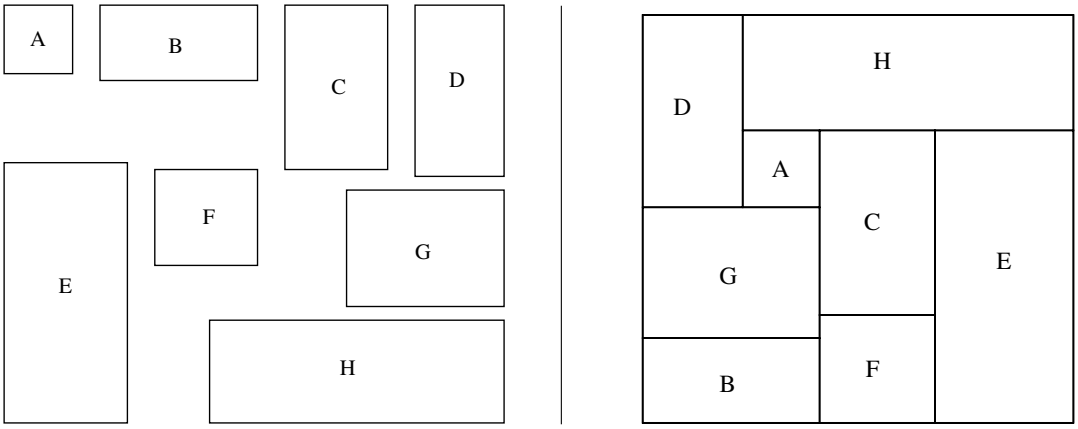
Finding the mutual intersection of a collection of half-spaces is a special case of the convex-hulls, and Qhull [BDH97] is convex hull code of choice for general dimensions. Qhull has been widely used in scientific applications and has a well-maintained homepage at *http://www.qhull.org/*.

Notes: Mount [Mou04] is an excellent survey of algorithms for computing intersections of geometric objects such as line segments, polygons, and polyhedra. Books with chapters discussing such problems include [dBvKOS00, CLRS01, PS85]. Preparata and Shamos [PS85] provide a good exposition on the special case of finding intersections and unions of axis-oriented rectangles—a problem that arises often in VLSI design.

An optimal $O(n \lg n + k)$ algorithm for computing line segment intersections is due to Chazelle and Edelsbrunner [CE92]. Simpler, randomized algorithms achieving the same time bound are thoroughly presented by Mulmuley [Mul94].

Lin and Manocha [LM04] survey techniques and software for collision detection.

Related Problems: Maintaining arrangements (see page 614), motion planning (see page 610).



17.9 Bin Packing

Input description: A set of n items with sizes $d_1, \ldots, d_n$. A set of m bins with capacity $c_1, \ldots, c_m$.

Problem description: Store all the items using the smallest number of bins.

Discussion: Bin packing arises in a variety of packaging and manufacturing problems. Suppose that you are manufacturing widgets cut from sheet metal or pants cut from cloth. To minimize cost and waste, we seek to lay out the parts so as to use as few fixed-size metal sheets or bolts of cloth as possible. Identifying which part goes on which sheet in which location is a bin-packing variant called the *cutting stock* problem. Once our widgets have been successfully manufactured, we are faced with another bin-packing problem—namely how best to fit the boxes into trucks to minimize the number of trucks needed to ship everything.

Even the most elementary-sounding bin-packing problems are NP-complete; see the discussion of integer partition in Section 13.10 (page 427). Thus, we are doomed to think in terms of heuristics instead of worst-case optimal algorithms. Fortunately, relatively simple heuristics tend to work well on most bin-packing problems. Further, many applications have peculiar, problem-specific constraints that tend to frustrate sophisticated algorithms for bin packing. The following factors will affect the choice of heuristic:

- *What are the shapes and sizes of the objects?* – The character of a bin-packing problem depends greatly on the shapes of the objects to be packed. Solving a standard jigsaw puzzle is a much different problem than packing squares

into a rectangular box. In *one-dimensional bin packing*, each object's size is given simply as an integer. This is equivalent to packing boxes of equal width into a chimney of that width, and makes it a special case of the knapsack problem of Section 13.10 (page 427).

When all the boxes are of identical size and shape, repeatedly filling each row gives a reasonable, but not necessarily optimal, packing. Consider trying to fill a 3×3 square with 2×1 bricks. You can only pack three bricks using one orientation, while four bricks suffice with two.

- *Are there constraints on the orientation and placement of objects?* – Many boxes are labeled "this side up" (imposing an orientation on the box) or "do not stack" (requiring them sit on top of any box pile). Respecting these constraints restricts our flexibility in packing and hence will increase in the number of trucks needed to send out certain shipments. Most shippers solve the problem by ignoring the labels. Indeed, your task will be simpler if you don't have to worry about the consequences of them.

- *Is the problem on-line or off-line?* – Do we know the complete set of objects to pack at the beginning of the job (an *off-line* problem)? Or will we get them one at a time and deal with them as they arrive (an *on-line* problem)? The difference is important, because we can do a better job packing when we can take a global view and plan ahead. For example, we can arrange the objects in an order that will facilitate efficient packing, perhaps by sorting them from biggest to smallest.

The standard off-line heuristics for bin packing order the objects by size or shape and then insert them into bins. Typical insertion rules are (1) select the first or leftmost bin the object fits in, (2) select the bin with the most room, (3) select the bin that provides the tightest fit, or (4) select a random bin.

Analytical and empirical results suggest that *first-fit decreasing* is the best heuristic. Sort the objects in decreasing order of size, so that the biggest object is first and the smallest last. Insert each object one by one into the first bin that has room for it. If no bin has room, we must start another bin. In the case of one-dimensional bin packing, this can never require more than 22% more bins than necessary and usually does much better. First-fit decreasing has an intuitive appeal to it, for we pack the bulky objects first and hope that little objects can fill up the cracks.

First-fit decreasing is easily implemented in $O(n \lg n + bn)$ time, where $b \leq \min(n, m)$ is the number of bins actually used. Simply do a linear sweep through the bins on each insertion. A faster $O(n \lg n)$ implementation is possible by using a binary tree to keep track of the space remaining in each bin.

We can fiddle with the insertion order in such a scheme to deal with problem-specific constraints. For example, it is reasonable to take "do not stack" boxes last (perhaps after artificially lowering the height of the bins to leave some room up

top to work with) and to place fixed-orientation boxes at the beginning (so we can use the extra flexibility later to stick boxes on top).

Packing boxes is much easier than packing arbitrary geometric shapes, enough so that a general approach packs each part into its own box and then packs the boxes. Finding an enclosing rectangle for a polygonal part is easy; just find the upper, lower, left, and right tangents in a given orientation. Finding the orientation and minimizing the area (or volume) of such a box is more difficult, but can be done in both two and three dimensions [O'R85]. See the Implementations section for a fast approximation to minimum enclosing box.

In the case of nonconvex parts, considerable useful space can be wasted in the holes created by placing the part in a box. One solution is to find the *maximum empty rectangle* within each boxed part and use this to contain other parts if it is sufficiently large. More advanced solutions are discussed in the references.

Implementations: Martello and Toth's collection of Fortran implementations of algorithms for a variety of knapsack problem variants are available at *http://www.or.deis.unibo.it/kp.html*. An electronic copy of [MT90a] has also been generously made available.

David Pisinger maintains a well-organized collection of C-language codes for knapsack problems and related variants like bin packing and container loading. These are available at *http://www.diku.dk/∼pisinger/codes.html*.

A first step towards packing arbitrary shapes packs each in its own minimum volume box. For a code to find an approximation to the optimal packing, see *http://valis.cs.uiuc.edu/∼sariel/research/papers/00/diameter/diam_prog.html*. This algorithm runs in near-linear time [BH01].

Notes: See [CFC94, CGJ96, LMM02] for surveys of the extensive literature on bin packing and the cutting stock problem. Keller, Pferschy, and Psinger [KPP04] is the most current reference on the knapsack problem and variants. Experimental results on bin-packing heuristics include [BJLM83, MT87].

Efficient algorithms are known for finding the largest empty rectangle in a polygon [DMR97] and point set [CDL86].

Sphere packing is an important and well-studied special case of bin packing, with applications to error-correcting codes. Particularly notorious was the "Kepler conjecture"—the problem of establishing the densest packing of unit spheres in three dimensions. This conjecture was finally settled by Hales and Ferguson in 1998; see [Szp03] for an exposition. Conway and Sloane [CS93] is the best reference on sphere packing and related problems.

Milenkovic has worked extensively on two-dimensional bin-packing problems for the apparel industry, minimizing the amount of material needed to manufacture pants and other clothing. Reports of this work include [DM97, Mil97].

Related Problems: Knapsack problem (see page 427), set packing (see page 625).

INPUT OUTPUT

17.10 Medial-Axis Transform

Input description: A polygon or polyhedron P.

Problem description: What are the set of points within P that have more than one closest point on the boundary of P?

Discussion: The medial-axis transformation is useful in *thinning* a polygon, or as is sometimes said, finding its *skeleton*. The goal is to extract a simple, robust representation of the shape of the polygon. The thinned versions of the letters A and B capture the essence of their shape, and would be relatively unaffected by changing the thickness of strokes or by adding font-dependent flourishes such as serifs. The skeleton also represents the center of the given shape, a property that leads to other applications like shape reconstruction and motion planning.

The medial-axis transformation of a polygon is always a tree, making it fairly easy to use dynamic programming to measure the "edit distance" between the skeleton of a known model and the skeleton of an unknown object. Whenever the two skeletons are close enough, we can classify the unknown object as an instance of our model. This technique has proven useful in computer vision and in optical character recognition. The skeleton of a polygon with holes (like the A and B) is not a tree but an embedded planar graph, but it remains easy to work with.

There are two distinct approaches to computing medial-axis transforms, depending upon whether your input is arbitrary geometric points or pixel images:

- *Geometric data* – Recall that the Voronoi diagram of a point set S (see Section 17.4 (page 576)) decomposes the plane into regions around each point $s_i \in S$ such that points within the region around s_i are closer to s_i than to any other site in S. Similarly, the Voronoi diagram of a set of line segments L decomposes the plane into regions around each line segment $l_i \in L$ such that all points within the region around l_i are closer to l_i than to any other site in L.

Any polygon is defined by a collection of line segments such that l_i shares a vertex with l_{i+1}. The medial-axis transform of a polygon P is simply the portion of the line-segment Voronoi diagram that lies within P. Any line-segment Voronoi diagram code thus suffices to do polygon thinning.

The *straight skeleton* is a structure related to the medial axis of a polygon, except that the bisectors are not equidistant to its defining edges but instead to the supporting lines of such edges. The straight skeleton, medial axis and Voronoi diagram are all identical for convex polygons, but in general skeleton bisectors may not be located in the center of the polygon. However, the straight skeleton is quite similar to a proper medial axis transform but is easier to compute. In particular, all edges in a straight skeleton are polygonal. See the Notes section for references with more details on how to compute it.

- *Image data* – Digitized images can be interpreted as points sitting at the lattice points on an integer grid. Thus, we could extract a polygonal description from boundaries in an image and feed it to the geometric algorithms just described. However, the internal vertices of the skeleton will most likely not lie at grid points. Geometric approaches to image processing problems often flounder because images are pixel-based and not continuous.

  A direct pixel-based approach for constructing a skeleton implements the "brush fire" view of thinning. Imagine a fire burning along all edges of the polygon, racing inward at a constant speed. The skeleton is marked by all points where two or more fires meet. The resulting algorithm traverses all the boundary pixels of the object, identifies those vertices as being in the skeleton, deletes the rest of the boundary, and repeats. The algorithm terminates when all pixels are extreme, leaving an object only one or two pixels thick. When implemented properly, this takes linear time in the number of pixels in the image.

  Algorithms that explicitly manipulate pixels tend to be easy to implement, because they avoid complicated data structures. However, the geometry doesn't work out exactly right in such pixel-based approaches. For example, the skeleton of a polygon is no longer always a tree or even necessarily connected, and the points in the skeleton will be close-to-but-not-quite equidistant to two boundary edges. Since you are trying to do continuous geometry in a discrete world, there is no way to solve the problem completely. You just have to live with it.

Implementations: CGAL (*www.cgal.org*) includes a package for computing the straight skeleton of a polygon P. Associated with it are routines for constructing offset contours defining the polygonal regions within P whose points are at least distance d from the boundary.

VRONI [Hel01] is a robust and efficient program for computing Voronoi diagrams of line segments, points, and arcs in the plane. It can readily compute

medial-axis transforms of polygons since it can construct Voronoi diagrams of arbitrary line segments. *VRONI* has been tested on thousands of synthetic and real-world data sets, some with over a million vertices. For more information, see *http://www.cosy.sbg.ac.at/~held/projects/vroni/vroni.html*. Other programs for constructing Voronoi diagrams are discussed in Section 17.4 (page 576).

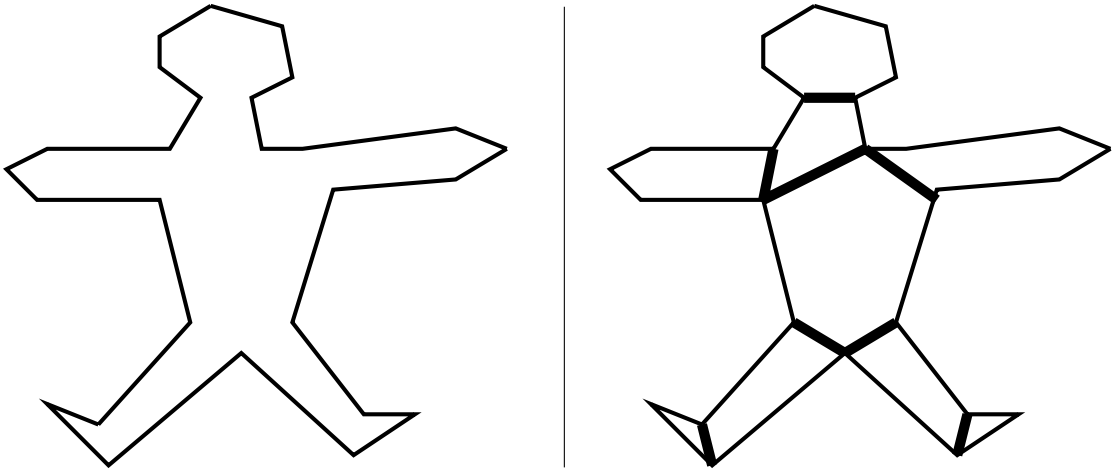
Programs that reconstruct or interpolate point clouds often are based on medial axis transforms. *Cocone* (*http://www.cse.ohio-state.edu/~tamaldey/cocone.html*) constructs an approximate medial-axis transform of the polyhedral surface it interpolates from points in E^3. See [Dey06] for the theory behind *Cocone*. *Powercrust* [ACK01a, ACK01b] constructs a discrete approximation to the medial-axis transform, and then reconstructs the surface from this transform. When the point samples are sufficiently dense, the algorithm is guaranteed to produce a geometrically and topologically correct approximation to the surface. It is available at *http://www.cs.utexas.edu/users/amenta/powercrust/*.

Notes: For a comprehensive survey of thinning approaches in image processing, see [LLS92, Ogn93]. The medial axis transformation was introduced for shape similarity studies in biology [Blu67]. Applications of the medial-axis transformation in pattern recognition are discussed in [DHS00]. The medial axis transformation is fundamental to the power crust algorithm for surface reconstruction from sampled points; see [ACK01a, ACK01b]. Good expositions on the medial-axis transform include [dBvKOS00, O'R01, Pav82].

The medial-axis of a polygon can be computed in $O(n \lg n)$ time for arbitrary n-gons [Lee82], although linear-time algorithms exist for convex polygons [AGSS89]. An $O(n \lg n)$ algorithm for constructing medial-axis transforms in curved regions was given by Kirkpatrick [Kir79].

Straight skeletons were introduced in [AAAG95], with a subquadratic algorithm due to [EE99]. See [LD03] for an interesting application of straight skeletons to defining the roof structures in virtual building models.

Related Problems: Voronoi diagrams (see page 576), Minkowski sum (see page 617).



INPUT OUTPUT

17.11 Polygon Partitioning

Input description: A polygon or polyhedron P.

Problem description: Partition P into a small number of simple (typically convex) pieces.

Discussion: Polygon partitioning is an important preprocessing step for many geometric algorithms, because geometric problems tend to be simpler on convex objects than on nonconvex ones. It is often easier to work with a small number of convex pieces than with a single nonconvex polygon.

Several flavors of polygon partitioning arise, depending upon the particular application:

- *Should all the pieces be triangles?* – Triangulation is the mother of all polygon partitioning problems, where we partition the interior of the polygon completely into triangles. Triangles are convex and have only three sides, making them the most elementary possible polygon.

  All triangulations of an n-vertex polygon contain exactly $n - 2$ triangles. Therefore, triangulation cannot be the answer if we seek a small number of convex pieces. A "nice" triangulation is judged by the shape of the triangles, not the count. See Section 17.3 (page 572) for a thorough discussion of triangulation.

- *Do I want to cover or partition my polygon?* – *Partitioning* a polygon means completely dividing the interior into nonoverlapping pieces. *Covering* a polygon means that our decomposition is permitted to contain mutually overlapping pieces. Both can be useful in different situations. In decomposing a complicated query polygon in preparation for a range search (Section 17.6 (page 584)), we seek a partitioning, so that each point we locate occurs in exactly one piece. In decomposing a polygon for painting purposes, a covering suffices, since there is no difficulty with filling in a region twice. We will concentrate here on partitioning, since it is simpler to do right, and any application needing a covering will accept a partitioning. The only drawback is that partitions can be larger than coverings.

- *Am I allowed to add extra vertices?* – A final issue is whether we are allowed to add Steiner vertices to the polygon, either by splitting edges or adding interior points. Otherwise, we are restricted to adding chords between two existing vertices. The former may result in a smaller number of pieces, at the cost of more complicated algorithms and perhaps messier results.

The Hertel-Mehlhorn heuristic for convex decomposition using diagonals is simple and efficient. It starts with an arbitrary triangulation of the polygon and then deletes any chord that leaves only convex pieces. A chord deletion creates a nonconvex piece only if it creates an internal angle that is greater than 180 degrees. The decision of whether such an angle will be created can be made locally from the chords and edges surrounding the chord, in constant time. The result always contains no more than four times the minimum number of convex pieces.

I recommend using this heuristic unless it is critical for you to minimize the number of pieces. By experimenting with different triangulations and various deletion orders, you may be able to obtain somewhat better decompositions.

Dynamic programming may be used to find the absolute minimum number of diagonals used in the decomposition. The simplest implementation, which maintains the number of pieces for all $O(n^2)$ subpolygons split by a chord, runs in $O(n^4)$. Faster algorithms use fancier data structures, running in $O(n+r^2 \min(r^2, n))$ time, where r is the number of reflex vertices. An $O(n^3)$ algorithm that further reduces the number of pieces by adding interior vertices is cited below, although it is complex and presumably difficult to implement.

An alternate decomposition problem partitions polygons into *monotone* pieces. The vertices of a y-monotone polygon can be divided into two chains such that any horizontal line intersects either chain at most once.

Implementations: Many triangulation codes start by finding a trapezoidal or monotone decomposition of polygons. Further, a triangulation is a simple form of convex decomposition. Check out the codes in Section 17.3 (page 572) as a starting point.

CGAL (*www.cgal.org*) contains a polygon-partitioning library that includes (1) the Hertel-Mehlhorn heuristic for partitioning a polygon into convex pieces,

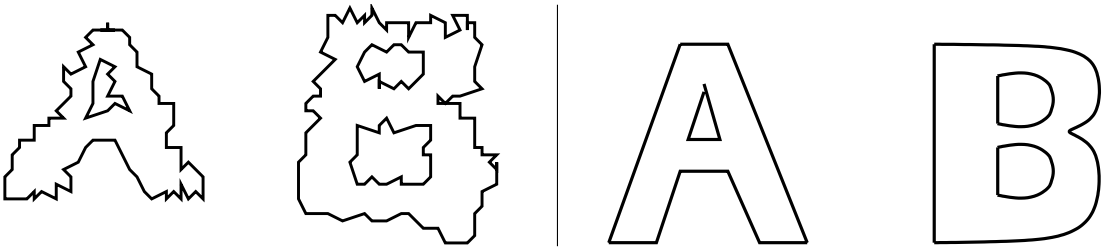
(2) finding a optimal convex partitioning using the $O(n^4)$ dynamic programming algorithm, and (3) an $O(n \log n)$ sweepline heuristic for partitioning into monotone polygons.

A triangulation code of particular relevance here is GEOMPACK—a suite of Fortran 77 codes by Barry Joe for 2- and 3-dimensional triangulation and convex decomposition problems. In particular, it does both Delaunay triangulation and convex decompositions of polygonal and polyhedral regions, as well as arbitrary-dimensional Delaunay triangulations.

Notes: Recent survey articles on polygon partitioning include [Kei00, OS04]. Keil and Sack [KS85] give an excellent survey on what is known about partitioning and covering polygons. Expositions on the Hertel-Mehlhorn heuristic [HM83] include [O'R01]. The $O(n+r^2 \min(r^2, n))$ dynamic programming algorithm for minimum convex decomposition using diagonals is due to Keil and Snoeyink [KS02]. The $O(r^3 + n)$ algorithm minimizing the number of convex pieces with Steiner points appears in [CD85]. Lien and Amato [LA06] provide an efficient heuristic for decomposing polygons with holes into "almost convex" polygons in $O(nr)$ time, with later work generalizing this to polyhedra.

Art gallery problems are an interesting topic related to polygon covering, where we seek to position the minimum number of guards in a given polygon such that every point in the interior of the polygon is watched by at least one guard. This corresponds to covering the polygon with a minimum number of star-shaped polygons. O'Rourke [O'R87] is a beautiful (although sadly out of print) book that presents the art gallery problem and its many variations.

Related Problems: Triangulation (see page 572), set cover (see page 621).



INPUT OUTPUT

17.12 Simplifying Polygons

Input description: A polygon or polyhedron p, with n vertices.

Problem description: Find a polygon or polyhedron p' containing only n' vertices, such that the shape of p' is as close as possible to p.

Discussion: Polygon simplification has two primary applications. The first involves cleaning up a noisy representation of a shape, perhaps obtained by scanning a picture of an object. Simplifying it can remove the noise and reconstruct the original object. The second involves data compression, where we seek to reduce detail on a large and complicated object yet leave it looking essentially the same. This can be a big win in computer graphics, where the smaller model might be significantly faster to render.

Several issues arise in shape simplification:

- *Do you want the convex hull?* – The simplest simplification is the convex hull of the object's vertices (see Section 17.2 (page 568)). The convex hull removes all internal concavities from the polygon. If you were simplifying a robot model for motion planning, this is almost certainly a good thing. But using the convex hull in an OCR system would be disastrous, because the concavities of characters provide most of the interesting features. An 'X' would be identical to an 'I', since both hulls are boxes. Another problem is that taking the convex hull of a convex polygon does nothing to simplify it further.

- *Am I allowed to insert or just delete points?* – The typical goal of simplification is to to represent the object as well as possible using a given number of vertices. The simplest approaches do local modifications to the boundary in order to reduce the vertex count. For example, if three consecutive vertices form a small-area triangle or define an extremely large angle, the center

vertex can be deleted and replaced with an edge without severely distorting the polygon.

Methods that only delete vertices quickly melt the shape into unrecognizability, however. More robust heuristics move vertices around to cover up the gaps that are created by deletions. Such "split-and-merge" heuristics can do a decent job, although nothing is guaranteed. Better results are likely by using the Douglas-Peucker algorithm, described next.

- *Must the resulting polygon be intersection-free?* – A serious drawback of incremental procedures is that they fail to ensure *simple* polygons, meaning they are without self-intersections. Thus "simplified" polygons may have ugly artifacts that may cause problems for subsequent routines. If simplicity is important, you should test all the line segments of your polygon for any pairwise intersections, as discussed in Section 17.8 (page 591).

  A polygon simplification approach that guarantees a simple approximation involves computing minimum-link paths. The *link distance* of a path between points s and t is the number of straight segments on the path. An as-the-crow-flies path has a link distance of one, while in general the link distance is one more than the number of turns on the path. The link distance between points s and t in a scene with obstacles is defined by the minimum link distance over all paths from s to t.

  The link distance approach "fattens" the boundary of the polygon by some acceptable error window ϵ (see Section 17.16 (page 617)) in order to construct a channel around the polygon. The minimum-link cycle in this channel represents the simplest polygon that never deviates from the original boundary by more than ϵ. An easy-to-compute approximation to link distance reduces it to breadth-first search, by placing a discrete set of possible turn points within the channel and connecting each pair of mutually visible points by an edge.

- *Are you given an image to clean up instead of a polygon to simplify?* – The conventional approach to cleaning up noise from a digital image is to take the Fourier transform of the image, filter out the high-frequency elements, and then take the inverse transform to recreate the image. See Section 13.11 (page 431) for details on the fast Fourier transform.

The Douglas-Puecker algorithm for shape simplification starts with a simple approximation and then refines it, instead of starting with a complicated polygon and trying to simplify it. Start by selecting two vertices v_1 and v_2 of polygon P, and propose the degenerate polygon v_1, v_2, v_1 as a simple approximation P'. Scan through each of the vertices of P, and select the one that is farthest from the corresponding edge of the polygon P'. Inserting this vertex adds the triangle to P' to minimize the maximum deviation from P. Points can be inserted until satisfactory results are achieved. This takes $O(kn)$ to insert k points when $|P| = n$.

Simplification becomes considerably more difficult in three dimensions. Indeed, it is NP-complete to find the minimum-size surface separating two polyhedra. Higher-dimensional analogies of the planar algorithms discussed here can be used to heuristically simplify polyhedra. See the Notes section.

Implementations: The Douglas-Peucker algorithm is readily implemented. Snoeyink provides a C implementation of his algorithm with efficient worst-case performance [HS94] at *http://www.cs.unc.edu/∼snoeyink/papers/DPsimp.arch*.

Simplification envelopes are a technique for automatically generating level-of-detail hierarchies for polygonal models. The user specifies a single error tolerance, and the maximum surface deviation of the simplified model from the original model, and a new, simplified model is generated. An implementation is available from *http://www.cs.unc.edu/∼geom/envelope.html*. This code preserves holes and prevents self-intersection.

QSlim is a quadric-based simplification algorithm that can produce high quality approximations of triangulated surfaces quite rapidly. It is available at *http://graphics.cs.uiuc.edu/∼garland/software.html*.

Yet another approach to polygonal simplification is based on simplifying and expanding the medial-axis transform of the polygon. The medial-axis transform (see Section 17.10 (page 598)) produces a skeleton of the polygon, which can be trimmed before inverting the transform to yield a simpler polygon. *Cocone* (*http://www.cse.ohio-state.edu/∼tamaldey/cocone.html*) constructs an approximate medial-axis transform of the polyhedral surface it interpolates from points in E^3. See [Dey06] for the theory behind *Cocone*. *Powercrust* [ACK01a, ACK01b] constructs a discrete approximation to the medial-axis transform, and then reconstructs the surface from this transform. When the point samples are sufficiently dense, the algorithm is guaranteed to produce a geometrically and topologically correct approximation to the surface. It is available at *http://www.cs.utexas.edu/users/amenta/powercrust/*.

CGAL (*www.cgal.org*) provides support for the most extreme polygon/polyhedral simplification, finding the smallest enclosing circle/sphere.

Notes: The Douglas-Peucker incremental refinement algorithm [DP73] is the basis for most shape simplification schemes, with faster implementations due to [HS94, HS98]. The link distance approach to polygon simplification is presented in [GHMS93]. Shape simplification problems become considerably more complex in three dimensions. Even finding the minimum-vertex convex polyhedron lying between two nested convex polyhedra is NP-complete [DJ92], although approximation algorithms are known [MS95b].

Heckbert and Garland [HG97] survey algorithms for shape simplification. Shape simplification using medial-axis transformations (see 17.10) are presented in [TH03].

Testing whether a polygon is simple can be performed in linear time, at least in theory, as a consequence of Chazelle's linear-time triangulation algorithm [Cha91].

Related Problems: Fourier transform (see page 431), convex hull (see page 568).

INPUT OUTPUT

17.13 Shape Similarity

Input description: Two polygonal shapes, P_1 and P_2.

Problem description: How similar are P_1 and P_2?

Discussion: Shape similarity is a problem that underlies much of pattern recognition. Consider a system for optical character recognition (OCR). We are given a library of shape models representing letters, and unknown shapes obtained by scanning a page. We seek to identify the unknown shapes by matching them to the most similar shape models.

Shape similarity is an inherently ill-defined problem, because what "similar" means is application dependent. Thus, no single algorithmic approach can solve all shape-matching problems. Whichever method you select, expect to spend a large chunk of time tweaking it to achieve maximum performance.

Among your possible approaches are

- *Hamming distance* – Assume that your two polygons have been overlaid one on top of the other. *Hamming distance* measures the area of symmetric difference between the two polygons—in other words, the area lying within one of the two polygons but not both of them. When two polygons are identical and properly aligned, the Hamming distance is zero. If the polygons differ only in a little noise at the boundary, then the Hamming distance of properly aligned polygons will be small.

  Computing the area of the symmetric difference reduces to finding the intersection or union of two polygons (discussed in Section 17.8 (page 591)) and then computing areas (discussed in Section 17.1). But the difficult part of

computing Hamming distance is finding the right alignment of the two polygons. This overlay problem is simplified in applications such as OCR because the characters are inherently aligned within lines on the page and are not free to rotate. Efficient algorithms for optimizing the overlap of convex polygons without rotation are cited below. Simple but reasonably effective heuristics are based on identifying reference landmarks on each polygon (such as the centroid, bounding box, or extremal vertices) and then matching a subset of these landmarks to define the alignment.

Hamming distance is particularly simple and efficient to compute on bit-mapped images, since after alignment all we do is sum the differences of the corresponding pixels. Although Hamming distance makes sense conceptually and can be simple to implement, it captures only a crude notion of shape and is likely to be ineffective in most applications.

- *Hausdorff distance* – An alternative distance metric is *Hausdorff distance*, which identifies the point on P_1 that is the maximum distance from P_2 and returns this distance. The Hausdorff distance is not symmetrical, for the tip of a long but thin protrusion from P_1 can imply a large Hausdorff distance P_1 to P_2, even though every point on P_2 is close to some point on P_1. A fattening of the entire boundary of one of the models (as is liable to happen with boundary noise) by a small amount may substantially increase the Hamming distance yet have little effect on the Hausdorff distance.

  Which is better, Hamming or Hausdorff? It depends upon your application. As with Hamming distance, computing the right alignment between the polygons can be difficult and time-consuming.

- *Comparing Skeletons* – A more powerful approach to shape similarity uses thinning (see Section 17.10 (page 598)) to extract a tree-like skeleton for each object. This skeleton captures many aspects of the original shape. The problem now reduces to comparing the shape of two such skeletons, using such features as the topology of the tree and the lengths/slopes of the edges. This comparison can be modeled as a form of subgraph isomorphism (see Section 16.9 (page 550)), with edges allowed to match whenever their lengths and slopes are sufficiently similar.

- *Support Vector Machines* – A final approach for pattern recognition/matching problems uses a learning-based technique such as neural networks or the more powerful *support vector machines*. These prove a reasonable approach to recognition problems when you have a lot of data to experiment with and no particular ideas of what to do with it. First, you must identify a set of easily computed features of the shape, such as area, number of sides, and number of holes. Based on these features, a black-box program (the support vector machine training algorithm) takes your training data and produces a classification function. This classification function accepts as input the values

of these features and returns a measure of what the shape is, or how close it is to a particular shape.

How good are the resulting classifiers? It depends upon the application. Like any ad hoc method, SVMs usually take a fair amount of tweaking and tuning to realize their full potential.

There is one caveat. If you don't know how/why black-box classifiers are making their decisions, you can't know when they will fail. An interesting case was a system built for the military to distinguish between images of cars and tanks. It performed very well on test images but disastrously in the field. Eventually, someone realized that the car images had been filmed on a sunnier day than the tanks, and the program was classifying solely on the presence of clouds in the background of the image!

Implementations: A Hausdorff-based image comparison implementation in C is available at *http://www.cs.cornell.edu/vision/hausdorff/hausmatch.html*. An alternate distance metric between polygons can be based on its angle-turning function [ACH+91]. An implementation in C of this turning function metric by Eugene K. Ressler is provided at *http://www.cs.sunysb.edu/∼algorith*.

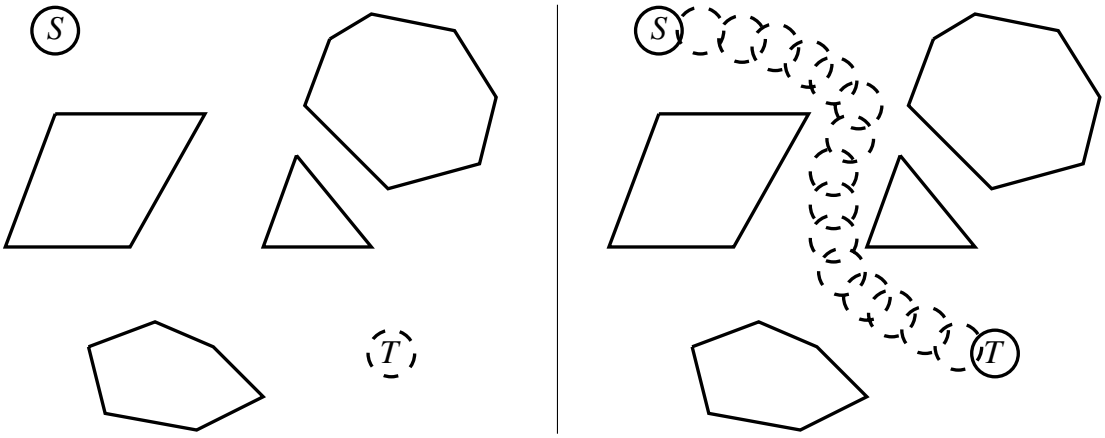
Several excellent support vector machine classifiers are available. These include the kernal-machine library (*http://www.terborg.net/research/kml/*), SVM^{light} (*http://svmlight.joachims.org/*) and the widely used and well-supported LIBSVM (*http://www.csie.ntu.edu.tw/∼cjlin/libsvm/*).

Notes: General books on pattern classification algorithms include [DHS00, JD88]. A wide variety of computational geometry approaches to shape similarity testing have been proposed, including [AMWW88, ACH+91, Ata84, AE83, BM89, OW85]. See the survey by Alt and Guibas [AG00].

The optimal alignment of n and m-vertex convex polygons subject to translation (but not rotation) can be computed in $O((n+m)\log(n+m))$ time [dBDK+98]. An approximation of the optimal overlap under translation and rotation is due to Ahn, et al. [ACP+07].

A linear-time algorithm for computing the Hausdorff distance between two convex polygons is given in [Ata83], with algorithms for the general case reported in [HK90].

Related Problems: Graph isomorphism (see page 550), thinning (see page 598).



INPUT OUTPUT

17.14 Motion Planning

Input description: A polygonal-shaped robot starting in a given position s in a room containing polygonal obstacles, and a goal position t.

Problem description: Find the shortest route taking s to t without intersecting any obstacles.

Discussion: That motion planning is a complex problem is obvious to anyone who has tried to move a large piece of furniture into a small apartment. It also arises in systems for molecular docking. Many drugs are small molecules that act by binding to a given target model. Identifying which binding sites are accessible to a candidate drug is clearly an instance of motion planning. Plotting paths for mobile robots is another canonical motion-planning application.

Finally, motion planning provides a tool for computer animation. Given the set of object models and where they appear in scenes s_1 and s_2, a motion planning algorithm can construct a short sequence of intermediate motions to transform s_1 to s_2. These motions can serve to fill in the intermediate scenes between s_1 and s_2, with such scene interpolation greatly reducing the workload on the animator.

Many factors govern the complexity of motion planning problems:

- *Is your robot a point?* – For point robots, motion planning becomes finding the shortest path from s to t around the obstacles. This is also known as geometric shortest path. The most readily implementable approach constructs the *visibility graph* of the polygonal obstacles, plus the points s and t. This

visibility graph has a vertex for each obstacle vertex and an edge between two obstacle vertices iff they "see" each other without being blocked by some obstacle edge.

The visibility graph can be constructed by testing each of the $\binom{n}{2}$ vertex-pair edge candidates for intersection against each of the n obstacle edges, although faster algorithms are known. Assign each edge of this visibility graph with weight equal to its length. Then the shortest path from s to t can be found using Dijkstra's shortest-path algorithm (see Section 15.4 (page 489)) in time bounded by the time required to construct the visibility graph.

- *What motions can your robot perform?* – Motion planning becomes considerably more difficult when the robot becomes a polygon instead of a point. Now all of the corridors that we use must be wide enough to permit the robot to pass through.

  The algorithmic complexity depends upon the number of *degrees of freedom* that the robot can use to move. Is it free to rotate as well as to translate? Does the robot have links that are free to bend or to rotate independently, as in an arm with a hand? Each degree of freedom corresponds to a dimension in the search space of possible configurations. Additional freedom makes it more likely that a short path exists from start to goal, although it also becomes harder to find this path.

- *Can you simplify the shape of your robot?* – Motion planning algorithms tend to be complex and time-consuming. Anything you can do to simplify your environment is a win. In particular, consider replacing your robot in an enclosing disk. If there is a start-to-goal path for this disk, it defines such a path for the robot inside of it. Furthermore, any orientation of a disk is equivalent to any other orientation, so rotation provides no help in finding a path. All movements can thus be limited to the simpler case of translation.

- *Are motions limited to translation only?* – When rotation is not allowed, the *expanded obstacles* approach can be used to reduce the problem of polygonal motion planning to the previously-resolved case of a point robot. Pick a reference point on the robot, and replace each obstacle by its Minkowski sum with the robot polygon (see Section 17.16 (page 617)). This creates a larger, fatter obstacle, defined by the shadow traced as the robot walks a loop around the object while maintaining contact with it. Finding a path from the initial reference position to the goal amidst these fattened obstacles defines a legal path for the polygonal robot in the original environment.

- *Are the obstacles known in advance?* – We have assumed that the robot starts out with a map of its environment. But this can't be true, say, in applications where the obstacles move. There are two approaches to solving motion-planning problems without a map. The first approach explores the

environment, building a map of what has been seen, and then uses this map to plan a path to the goal. A simpler strategy proceeds like a sightless man with a compass. Walk in the direction towards the goal until progress is blocked by an obstacle, and then trace a path along the obstacle until the robot is again free to proceed directly towards the goal. Unfortunately, this will fail in environments of sufficient complexity.

The most practical approach to general motion planning involves randomly sampling the *configuration space* of the robot. The configuration space defines the set of legal positions for the robot using one dimension for each degree of freedom. A planar robot capable of translation and rotation has three degrees of freedom, namely the x- and y-coordinates of a reference point on the robot and the angle θ relative to this point. Certain points in this space represent legal positions, while others intersect obstacles.

Construct a set of legal configuration-space points by random sampling. For each pair of points p_1 and p_2, decide whether there exists a direct, nonintersecting path between them. This defines a graph with vertices for each legal point and edges for each such traversable pair. Motion planning now reduces to finding a direct path from the initial/final position to some vertex in the graph, and then solving a shortest-path problem between the two vertices.

There are many ways to enhance this basic technique, such as adding additional vertices to regions of particular interest. Building such a road map provides a nice, clean approach to solving problems that would otherwise get very messy.

Implementations: The *Motion Planning Toolkit* (MPK) is a C++ library and toolkit for developing single- and multi-robot motion planners. It includes SBL, a fast single-query probabilistic roadmap path planner, and is available at *http://robotics.stanford.edu/∼mitul/mpk/*.

The University of North Carolina GAMMA group has produced several efficient collision detection libraries (not really motion planning) of which *SWIFT++* [EL01] is the most recent member of this family. It can detect intersection, compute approximate/exact distances between objects, and determine object-pair contacts in scenes composed of rigid polyhedral models. See *http://www.cs.unc.edu/∼geom/collide/* for pointers to all of these libraries.

The computational geometry library CGAL (*www.cgal.org*) contains many algorithms related to motion planning including visibility graph construction and Minkowski sums. O'Rourke [O'R01] gives a toy implementation of an algorithm to plot motion for a two-jointed robot arm in the plane. See Section 19.1.10 (page 662).

Notes: Latombe's book [Lat91] describes practical approaches to motion planning, including the random sampling method described above. Two other worthy books on motion planning are available freely on line, by LaValle [LaV06] (*http://planning.cs.uiuc.edu/*) and Laumond [Lau98] (*http://www.laas.fr/∼jpl/book.html*).

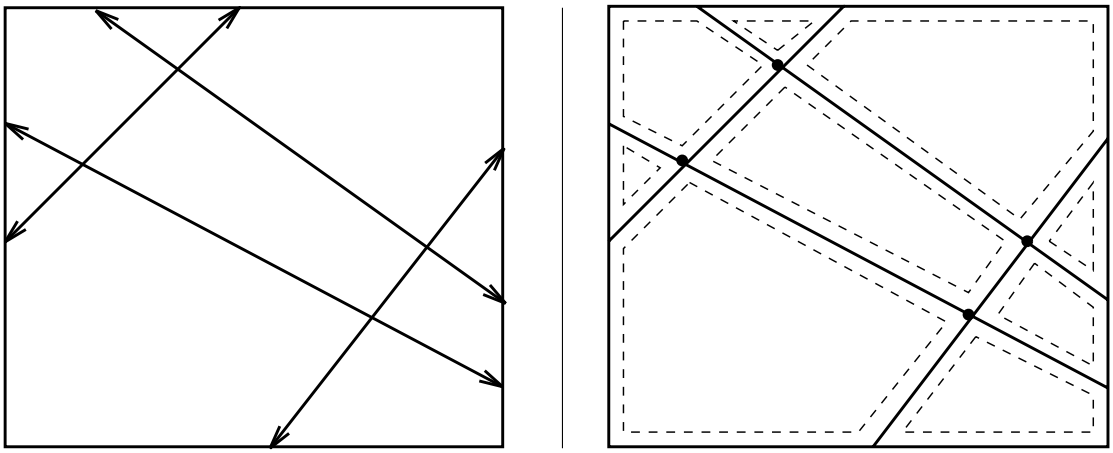
Motion planning was originally studied by Schwartz and Sharir as the "piano mover's problem." Their solution constructs the complete free space of robot positions that do not intersect obstacles, and then finds the shortest path within the proper connected component. These free space descriptions are very complicated, involving arrangements of higher-degree algebraic surfaces. The fundamental papers on the piano mover's problem appear in [HSS87], with [Sha04] a survey of current results.

The best general result for this free-space approach to motion planning is due to Canny [Can87], who showed that any problem with d degrees of freedom can be solved in $O(n^d \lg n)$, although faster algorithms exist for special cases of the general motion-planning problem. The expanded obstacle approach to motion planning is due to Lozano-Perez and Wesley [LPW79]. The heuristic, sightless man's approach to motion planning discussed previously has been studied by Lumelski [LS87].

The time complexity of algorithms based on the free-space approach to motion planning depends intimately on the combinatorial complexity of the arrangement of surfaces defining the free space. Algorithms for maintaining arrangements are presented in Section 17.15 (page 614). Davenport-Schinzel sequences often arise in the analysis of such arrangements. Sharir and Agarwal [SA95] provide a comprehensive treatment of Davenport-Schinzel sequences and their relevance to motion planning.

The visibility graph of n line segments with E pairs of visible vertices can be constructed in $O(n \lg n + E)$ time [GM91, PV96], which is optimal. Hershberger and Suri [HS99] have an $O(n \lg n)$ algorithm for finding shortest paths for point-robots with polygonal obstacles. Chew [Che85] provides an $O(n^2 \lg n)$ for finding shortest paths for a disk-robot in such a scene.

Related Problems: Shortest path (see page 489), Minkowski sum (see page 617).



INPUT OUTPUT

17.15 Maintaining Line Arrangements

Input description: A set of lines and line segments $l_1, \ldots, l_n$.

Problem description: What is the decomposition of the plane defined by $l_1, \ldots, l_n$?

Discussion: A fundamental problem in computational geometry is explicitly constructing the regions formed by the intersections of a set of n lines. Many problems reduce to constructing and analyzing such an arrangement of a specific set of lines. Examples include:

- *Degeneracy testing* – Given a set of n lines in the plane, do any three of them pass through the same point? Brute-force testing of all triples takes $O(n^3)$ time. Instead, we can construct the arrangement of the lines and then walk over each vertex and explicitly count its degree, all in quadratic time.
- *Satisfying the maximum number of linear constraints* – Suppose that we are given a set of n linear constraints, each of the form $y \leq a_i x + b_i$. Which point in the plane satisfies the largest number of them? Construct the arrangement of the lines. All points in any region or *cell* of this arrangement satisfy exactly the same set of constraints, so we need to test only one point per cell to find the global maximum.

Thinking of geometric problems in terms of features in an arrangement can be very useful in formulating algorithms. Unfortunately, it must be admitted that

arrangements are not as popular in practice as might be supposed. Primarily, this is because some depth of understanding is required to apply them correctly. The computational geometry library CGAL now provides a general and robust enough implementation to justify the effort to do so. Issues arising in arrangements include

- *What is the right way to construct a line arrangement?* – Algorithms for constructing arrangements are incremental. Begin with an arrangement of one or two lines. Subsequent lines are inserted into the arrangement one at a time, yielding larger and larger arrangements. To insert a new line, we start on the leftmost cell containing the line and walk over the arrangement to the right, moving from cell to neighboring cell and splitting into two pieces those cells that contain the new line.

- *How big will your arrangement be?* – A geometric fact called the *zone theorem* implies that the kth line inserted cuts through k cells of the arrangement, and further that $O(k)$ total edges form the boundary of these cells. This means that we can scan through each edge of every cell we encounter on our insertion walk, confident that only linear total work will be performed while inserting the line into the arrangement. Therefore, the total time to insert all n lines in constructing the full arrangement is $O(n^2)$.

- *What do you want to do with your arrangement?* – Given an arrangement and a query point, we are often interested in identifying which cell of the arrangement contains the point. This is the problem of point location, discussed in Section 17.7 (page 587). Given an arrangement of lines or line segments, we are often interested in computing all points of intersection of the lines. The problem of intersection detection is discussed in Section 17.8 (page 591).

- *Does your input consist of points instead of lines?* – Although lines and points seem to be different geometric objects, appearances can be misleading. Through the use of *duality transformations*, we can turn line L into point p and vice versa:

  $$L : y = 2ax - b \leftrightarrow p : (a, b)$$

  Duality is important because we can now apply line arrangements to point problems, often with surprising results.

  For example, suppose we are given a set of n points, and we want to know whether any three of them all lie on the same line. This sounds similar to the degeneracy testing problem discussed above. In fact it is *exactly the same*, with only the role of points and lines exchanged. We can dualize our points into lines as above, construct the arrangement, and then search for a vertex with three lines passing through it. The dual of this vertex defines the line on which the three initial vertices lie.

It often becomes useful to traverse each face of an existing arrangement exactly once. Such traversals are called *sweepline algorithms*, and are discussed in some detail in Section 17.8 (page 591). The basic procedure sorts the intersection points by x-coordinate and then walks from left to right while keeping track of all we have seen.

Implementations: CGAL (*www.cgal.org*) provides a generic and robust package for arrangements of curves (not just lines) in the plane. This should be the starting point for any serious project using arrangements.

A robust code for constructing and topologically sweeping an arrangement in C++ is provided at *http://www.cs.tufts.edu/research/geometry/other/sweep/*. An extension of topological sweep to deal with the visibility complex of a collection of pairwise disjoint convex planar sets has been provided in CGAL.

Arrange is a package for maintaining arrangements of polygons in either the plane or on the sphere. Polygons may be degenerate, and hence represent arrangements of lines. A randomized incremental construction algorithm is used, and efficient point location on the arrangement is supported. *Arrange* is written in C by Michael Goldwasser and is available from *http://euler.slu.edu/∼goldwasser/publications/*.

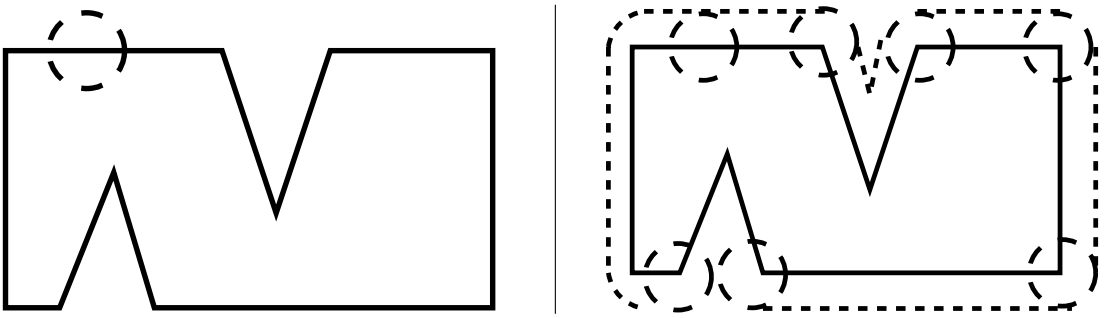
Notes: Edelsbrunner [Ede87] provides a comprehensive treatment of the combinatorial theory of arrangements, plus algorithms on arrangements with applications. It is an essential reference for anyone seriously interested in the subject. Recent surveys of combinatorial and algorithmic results include [AS00, Hal04]. Good expositions on constructing arrangements include [dBvKOS00, O'R01]. Implementation issues related to arrangements as implemented in CGAL are discussed in [FWH04, HH00].

Arrangements generalize naturally beyond two dimensions. Instead of lines, the space decomposition is defined by planes (or beyond 3-dimensions, *hyperplanes*). The zone theorem states that any arrangement of n d-dimensional hyperplanes has total complexity $O(n^d)$, and any single hyperplane intersects cells of complexity $O(n^{d-1})$. This provides the justification for the incremental construction algorithm for arrangements. Walking around the boundary of each cell to find the next cell that the hyperplane intersects takes time proportional to the number of cells created by inserting the hyperplane.

The history of the zone theorem has become somewhat muddled, because the original proofs were later found to be in error in higher dimensions. See [ESS93] for a discussion and a correct proof. The theory of Davenport-Schinzel sequences is intimately tied into the study of arrangements, which is presented in [SA95].

The naive algorithm for sweeping an arrangement of lines sorts the n^2 intersection points by x-coordinate and hence requires $O(n^2 \lg n)$ time. The *topological sweep* [EG89, EG91] eliminates the need to sort, and so traverses the arrangement in quadratic time. This algorithm is readily implementable and can be applied to speed up many sweepline algorithms. See [RSS02] for a robust implementation with experimental results.

Related Problems: Intersection detection (see page 591), point location (see page 587).



INPUT OUTPUT

17.16 Minkowski Sum

Input description: Point sets or polygons A and B, containing n and m vertices respectively.

Problem description: What is the convolution of A and B—i.e. , the Minkowski sum $A + B = \{x + y | x \in A, y \in B\}$?

Discussion: Minkowski sums are useful geometric operations that can *fatten* objects in appropriate ways. For example, a popular approach to motion planning for polygonal robots in a room with polygonal obstacles (see Section 17.14 (page 610)) fattens each of the obstacles by taking the Minkowski sum of them with the shape of the robot. This reduces the problem to the (more easily solved) case of point robots. Another application is in shape simplification (see Section 17.12 (page 604)). Here we fatten the boundary of an object to create a channel around it, and then let the minimum link path lying within this channel define the simplified shape. Finally, convolving an irregular object with a small circle will help smooth out the boundaries by eliminating minor nicks and cuts.

The definition of a Minkowski sum assumes that the polygons A and B have been positioned on a coordinate system:

$$A + B = \{x + y \mid x \in A, y \in B\}$$

where $x+y$ is the vector sum of two points. Thinking of this in terms of translation, the Minkowski sum is the union of all translations of A by a point defined within B. Issues arising in computing Minkowski sums include

- *Are your objects rasterized images or explicit polygons?* – The definition of Minkowski summation suggests a simple algorithm if A and B are rasterized images. Initialize a sufficiently large matrix of pixels by determining the size

of the convolution of the bounding boxes of A and B. For each pair of points in A and B, sum up their coordinates and darken the appropriate pixel. These algorithms get more complicated if an explicit polygonal representation of the Minkowski sum is needed.

- *Do you want to fatten your object by a fixed amount?* – The most common fattening operation expands a model M by a given tolerance t, known as *offsetting*. As shown in the figures above, this is accomplished by computing the Minkowski sum of M with a disk of radius t. The basic algorithms still work, although the offset is not a polygon. Its boundary is instead composed of circular arcs and line segments.

- *Are your objects convex or non-convex?* – The complexity of computing Minkowski sums depends in a serious way on the shape of the polygons. If both A and B are convex, the Minkowski sum can be found in $O(n+m)$ time by tracing the boundary of one polygon with another. If one of them is nonconvex, the *size* of the sum alone can be as large as $\Theta(nm)$. Even worse is when both A and B are nonconvex, in which case the *size* of the sum can be as large as $\Theta(n^2m^2)$. Minkowski sums of nonconvex polygons are often ugly in a majestic sort of way, with holes either created or destroyed in surprising fashion.

A straightforward approach to computing the Minkowski sum is based on triangulation and union. First, triangulate both polygons, then compute the Minkowski sum of each triangle of A against each triangle of B. The sum of a triangle against another triangle is easy to compute and is a special case of convex polygons, discussed below. The union of these $O(nm)$ convex polygons will be $A+B$. Algorithms for computing the union of polygons are based on plane sweep, as discussed in Section 17.8 (page 591).

Computing the Minkowski sum of two convex polygons is easier than the general case, because the sum will always be convex. For convex polygons it is easiest to slide A along the boundary of B and compute the sum edge by edge. Partitioning each polygon into a small number of convex pieces (see Section 17.11 (page 601)), and then unioning the Minkowski sum for each pair of pieces, will usually be much more efficient than working with two fully triangulated polygons.

Implementations: The CGAL (*www.cgal.org*) Minkowski sum package provides an efficient and robust code to find the Minkowski sums of two arbitrary polygons, as well as compute both exact and approximate offsets.

An implementation for computing the Minkowski sums of two convex polyhedra in $3D$ is described in [FH06] and available at *http://www.cs.tau.ac.il/∼efif/CD/*.

Notes: Good expositions on algorithms for Minkowski sums include [dBvKOS00, O'R01]. The fastest algorithms for various cases of Minkowski sums include [KOS91, Sha87].

The practical efficiency of Minkowski sum in the general case depends upon how the polygons are decomposed into convex pieces. The optimal solution is not necessarily the

partition into the fewest number of convex pieces. Agarwal et al. [AFH02] give a through study of decomposition methods for Minkowski sum.

The combinatorial complexity of the Minkowski sum of two convex polyhedra in three dimensions is completely resolved in [FHW07]. An implementation of Minkowski sum for such polyhedra is described in [FH06].

Kedem and Sharir [KS90] present an efficient algorithm for translational motion planning for polygonal robots, based on Minkowski sums.

Related Problems: Thinning (see page 598), motion planning (see page 610), simplifying polygons (see page 604).