structure arises in many applications of geometry. The dual structure, called a *Delaunay triangulation* also has many interesting properties.

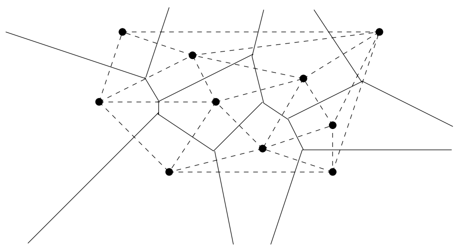

Figure 3: Voronoi diagram and Delaunay triangulation.

Search: Geometric search problems are of the following general form. Given a data set (e.g. points, lines, polygons) which will not change, preprocess this data set into a data structure so that some type of query can be answered as efficiently as possible. For example, a *nearest neighbor* search query is: determine the point of the data set that is closest to a given query point. A *range query* is: determine the set of points (or count the number of points) from the data set that lie within a given region. The region may be a rectangle, disc, or polygonal shape, like a triangle.

Lecture 2: Fixed-Radius Near Neighbors and Geometric Basics

Reading: The material on the Fixed-Radius Near Neighbor problem is taken from the paper: "The complexity of finding fixed-radius near neighbors," by J. L. Bentley, D. F. Stanat, and E. H. Williams, *Information Processing Letters*, 6(6), 1977, 209–212. The material on affine and Euclidean geometry is covered in many textbooks on basic geometry and computer graphics.

Fixed-Radius Near Neighbor Problem: As a warm-up exercise for the course, we begin by considering one of the oldest results in computational geometry. This problem was considered back in the mid 70's, and is a fundamental problem involving a set of points in dimension d. We will consider the problem in the plane, but the generalization to higher dimensions will be straightforward.

We assume that we are given a set P of n points in the plane. As will be our usual practice, we assume that each point p is represented by its (x, y) coordinates, denoted (p_x, p_y). The Euclidean distance between two points p and q, denoted $|pq|$ is

$$|pq| = \sqrt{(p_x - q_x)^2 + (p_y - q_y)^2}.$$

Given the set P and a distance $r > 0$, our goal is to report all pairs of distinct points $p, q \in P$ such that $|pq| \leq r$. This is called the *fixed-radius near neighbor (reporting) problem*.

Reporting versus Counting: We note that this is a *reporting* problem, which means that our objective is to report all such pairs. This is in contrast to the corresponding *counting* problem, in which the objective is to return a count of the number of pairs satisfying the distance condition.

It is usually easier to solve reporting problems optimally than counting problems. This may seem counterintuitive at first (after all, if you can report, then you can certainly count). The reason is that we know that any algorithm that reports some number k of pairs must take at least $\Omega(k)$ time. Thus if k is large, a reporting algorithm has the luxury of being able to run for a longer time and still claim to be optimal. In contrast, we cannot apply such a lower bound to a counting algorithm.

Simple Observations: To begin, let us make a few simple observations. This problem can easily be solved in $O(n^2)$ time, by simply enumerating all pairs of distinct points and computing the distance between each pair. The number of distinct pairs of n points is

$$\binom{n}{2} = \frac{n(n-1)}{2}.$$

Letting k denote the number of pairs that reported, our goal will be to find an algorithm whose running time is (nearly) linear in n and k, ideally $O(n+k)$. This will be optimal, since any algorithm must take the time to read all the input and print all the results. (This assumes a naive representation of the output. Perhaps there are more clever ways in which to encode the output, which would require less than $O(k)$ space.)

To gain some insight to the problem, let us consider how to solve the 1-dimensional version, where we are just given a set of n points on the line, say, $x_1, x_2, \ldots, x_n$. In this case, one solution would be to first sort the values in increasing order. Let suppose we have already done this, and so:

$$x_1 < x_2 < \ldots < x_n.$$

Now, for i running from 1 to n, we consider the successive points x_{i+1}, x_{i+2}, x_{i+3}, and so on, until we first find a point whose distance exceeds r. We report x_i together with all succeeding points that are within distance r.

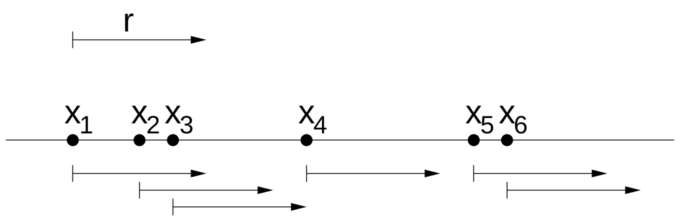

Figure 4: Fixed radius nearest neighbor on the line.

The running time of this algorithm involves the $O(n \log n)$ time needed to sort the points and the time required for distance computations. Let k_i denote the number of pairs generated when we visit p_i. Observe that the processing of p_i involves $k_i + 1$ distance computations (one additional computation for the points whose distance exceeds r). Thus, up to constant factors, the total running time is:

$$\begin{aligned} T(n,k) &= n \log n + \sum_{i=1}^{n} (k_i + 1) \;=\; n \log n + n + \sum_{i=1}^{n} k_i \\ &= n \log n + n + k \;=\; O(k + n \log n). \end{aligned}$$

This is close to the $O(k+n)$ time we were hoping for. It seems that any approach based on sorting is doomed to take at least $\Omega(n \log n)$ time. So, if we are to improve upon this, we cannot sort. But is sorting really necessary? Let us consider an approach based on bucketing.

1-dimensional Solution with Bucketing: Rather than sorting the points, suppose that we subdivide the line into intervals of length r. In particular, we can take the line to be composed of an infinite collection of half-open intervals:

$$\ldots, [-3r, -2r),\ [-2r, -r),\ [-r, 0),\ [0, r),\ [r, 2r),\ [2r, 3r), \ldots$$

In general, interval b is $[br, (b+1)r)$.

A point x lies in the interval $b = \lfloor x/r \rfloor$. Thus, in $O(n)$ time we can associate the n points of P with a set of n integer bucket indices b_i. Although there are an infinite number of intervals, at most n will be *occupied*, meaning that they contain a point of P.

There are a number of ways to organize the occupied buckets. They could be sorted, but then we are back to $O(n \log n)$ time. Since bucket indices are integers, a better approach is to store the occupied bucket indices in a hash table. Thus in $O(1)$ expected time, we can determine which bucket contains a given point and look this bucket up in the hash table. We can store all the points lying with any given bucket in an (unsorted) list associated with its bucket, in $O(n)$ total time.

The fact that the running time is in the expected case, rather than worst case is a bit unpleasant. However, it can be shown that by using a good randomized hash function, the probability that the total running time is worse than $O(n)$ can be made arbitrarily small. If the algorithm performs significantly more than the expected number of computations, we can simply chose a different random hash function and try again. This will lead to a very practical solution.

How does bucketing help? Observe that if point x lies in bucket b, then any successors that lie within distance r must lie either in bucket b or in $b+1$. This suggests the straightforward solution shown below.

Fixed-Radius Near Neighbor on the Line by Bucketing

```
(1) Store the points of $P$ into buckets of size $r$, stored in a hash table.
(2) For each $x \in P$ do the following:
    (a) Let $b$ be the bucket containing $x$.
    (b) Search all the points of buckets $b$ and $b+1$, report $x$ along with all those points $x'$ that lie within distance $r$
        of $x$.
```

To avoid duplicates, we need only report pairs (x, x') where $x' > x$. The key question is determining the time complexity of this algorithm is how many distance computations are performed in step (2b). We compare each point in bucket b with all the points in buckets b and $b+1$. However, not all of these distance computations will result in a pair of points whose distance is within r. Might it be that we waste a great deal of time in performing computations for which we receive no benefit? The lemma below shows that we perform no more than a constant factor times as many distance computations and pairs that are produced.

It will simplify things considerably if, rather than counting distinct pairs of points, we simply count all (ordered) pairs of points that lie within distance r of each other. Thus each pair of points will be counted twice, (p, q) and (q, p). Note that this includes reporting each point as a pair (p, p) as well, since each point is within distance r of itself. This does not affect the asymptotic bounds, since the number of distinct pairs is smaller by a factor of roughly $1/2$.

Lemma: Let k denote the number of (not necessarily distinct) pairs of points of P that are within distance r of each other. Let D denote the total number distance computations made in step (2b) of the above algorithm. Then $D \leq 2k$.

Proof: We will make use of the following inequality in the proof:

$$xy \leq \frac{x^2 + y^2}{2}.$$

This follows by expanding the obvious inequality $(x - y)^2 \geq 0$.

Let B denote the (infinite) set of buckets. For any bucket $b \in B$, let $b+1$ denote its successor bucket on the line, and let n_b denote the number of points of P in b. Define

$$S = \sum_{b \in B} n_b^2.$$

First we bound the total number of distance computations D as a function of S. Each point in bucket b computes the distance to every other point in bucket b and every point in bucket $b+1$, and hence

$$D \;=\; \sum_{b \in B} n_b(n_b + n_{b+1}) \;=\; \sum_{b \in B} n_b^2 + n_b n_{b+1} \;=\; \sum_{b \in B} n_b^2 + \sum_{b \in B} n_b n_{b+1}$$

$$
\begin{aligned}
&\le \sum_{b\in B} n_b^2 + \sum_{b\in B} \frac{n_b^2 + n_{b+1}^2}{2} \\
&= \sum_{b\in B} n_b^2 + \sum_{b\in B} \frac{n_b^2}{2} + \sum_{b\in B} \frac{n_{b+1}^2}{2} \;=\; S + \frac{S}{2} + \frac{S}{2} \;=\; 2S.
\end{aligned}
$$

Next we bound the number of pairs reported from below as a function of S. Since each pair of points lying in bucket b is within distance r of every other, there are n_b^2 pairs in bucket b alone that are within distance r of each other, and hence (considering just the pairs generated within each bucket) we have $k \geq S$. Therefore we have

$$D \leq 2S \leq 2k,$$

which completes the proof.

By combining this with the $O(n)$ expected time needed to bucket the points, it follows that the total expected running time is $O(n + k)$.

Generalization to d dimensions: This bucketing algorithm is easy to extend to multiple dimensions. For example, in dimension 2, we bucket points into a square grid in which each grid square is of side length r. The bucket index of a point (x, y) is a pair $(\lfloor x/r \rfloor, \lfloor y/r \rfloor)$. We apply a hash function that accepts two arguments. To generalize the algorithm, for each point we consider the points in its surrounding 3×3 subgrid of buckets. By generalizing the above arguements, it can be shown that the algorithm's expected running time is $O(n + k)$. The details are left as an exercise.

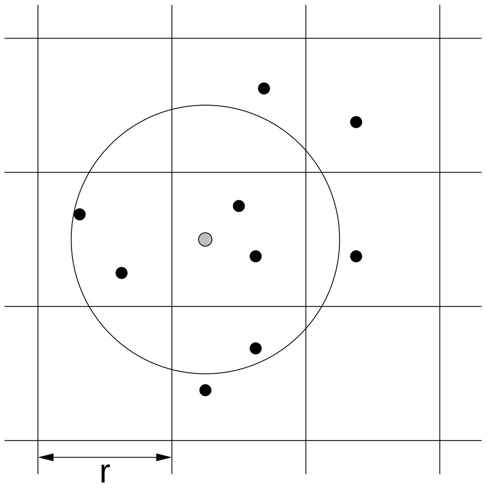

Figure 5: Fixed radius nearest neighbor on the plane.

This example problem serves to illustrate some of the typical elements of computational geometry. Geometry itself did not play a significant role in the problem, other than the relatively easy task of computing distances. We will see examples later this semester where geometry plays a much more important role. The major emphasis was on accounting for the algorithm's running time. Also note that, although we discussed the possibility of generalizing the algorithm to higher dimensions, we did not treat the dimension as an asymptotic quantity. In fact, a more careful analysis reveals that this algorithm's running time increases exponentially with the dimension. (Can you see why?)

Geometry Basics: As we go through the semester, we will introduce much of the geometric facts and computational primitives that we will be needing. For the most part, we will assume that any geometric primitive involving a constant number of elements of constant complexity can be computed in $O(1)$ time, and we will not concern ourselves with how this computation is done. (For example, given three noncolinear points in the plane, compute the unique circle passing through these points.) Nonetheless, for a bit of completeness, let us begin with a quick review of the basic elements of affine and Euclidean geometry.

There are a number of different geometric systems that can be used to express geometric algorithms: affine geometry, Euclidean geometry, and projective geometry, for example. This semester we will be working almost exclusively with affine and Euclidean geometry. Before getting to Euclidean geometry we will first define a somewhat more basic geometry called *affine geometry*. Later we will add one operation, called an inner product, which extends affine geometry to Euclidean geometry.

Affine Geometry: An affine geometry consists of a set of *scalars* (the real numbers), a set of *points*, and a set of *free vectors* (or simply *vectors*). Points are used to specify position. Free vectors are used to specify direction and magnitude, but have no fixed position in space. (This is in contrast to linear algebra where there is no real distinction between points and vectors. However this distinction is useful, since the two are really quite different.)

The following are the operations that can be performed on scalars, points, and vectors. Vector operations are just the familiar ones from linear algebra. It is possible to subtract two points. The difference $p - q$ of two points results in a free vector directed from q to p. It is also possible to add a point to a vector. In point-vector addition $p + v$ results in the point which is translated by v from p. Letting S denote an generic scalar, V a generic vector and P a generic point, the following are the legal operations in affine geometry:

$S \cdot V$	$\rightarrow$	V	scalar-vector multiplication
$V + V$	$\rightarrow$	V	vector addition
$P - P$	$\rightarrow$	V	point subtraction
$P + V$	$\rightarrow$	P	point-vector addition

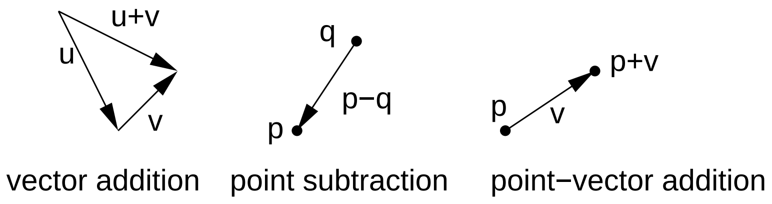

Figure 6: Affine operations.

A number of operations can be derived from these. For example, we can define the subtraction of two vectors $\vec{u} - \vec{v}$ as $\vec{u} + (-1) \cdot \vec{v}$ or scalar-vector division $\vec{v}/\alpha$ as $(1/\alpha) \cdot \vec{v}$ provided $\alpha \neq 0$. There is one special vector, called the *zero vector*, $\vec{0}$, which has no magnitude, such that $\vec{v} + \vec{0} = \vec{v}$.

Note that it is *not* possible to multiply a point times a scalar or to add two points together. However there is a special operation that combines these two elements, called an *affine combination*. Given two points p_0 and p_1 and two scalars α_0 and α_1, such that $\alpha_0 + \alpha_1 = 1$, we define the affine combination

$$\mathrm{aff}(p_0, p_1; \alpha_0, \alpha_1) \; = \; \alpha_0 p_0 + \alpha_1 p_1 \; = \; p_0 + \alpha_1 (p_1 - p_0).$$

Note that the middle term of the above equation is not legal given our list of operations. But this is how the affine combination is typically expressed, namely as the weighted average of two points. The right-hand side (which is easily seen to be algebraically equivalent) is legal. An important observation is that, if $p_0 \neq p_1$, then

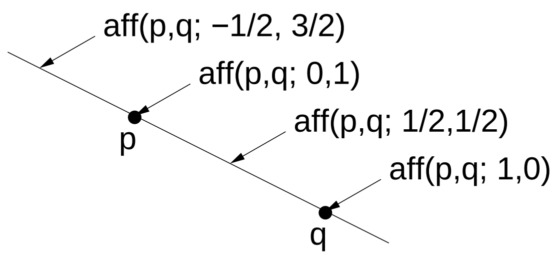

Figure 7: Affine combination.

the point $\mathrm{aff}(p_0, p_1; \alpha_0, \alpha_1)$ lies on the line joining p_0 and p_1. As α_1 varies from $-\infty$ to $+\infty$ it traces out all the points on this line.

In the special case where $0 \le \alpha_0, \alpha_1 \le 1$, $\mathrm{aff}(p_0, p_1; \alpha_0, \alpha_1)$ is a point that subdivides the line segment $\overline{p_0p_1}$ into two subsegments of relative sizes α_1 to α_0. The resulting operation is called a *convex combination*, and the set of all convex combinations traces out the line segment $\overline{p_0p_1}$.

It is easy to extend both types of combinations to more than two points, by adding the condition that the sum $\alpha_0 + \alpha_1 + \alpha_2 = 1$.

$$\mathrm{aff}(p_0, p_1, p_2; \alpha_0, \alpha_1, \alpha_2) \;=\; \alpha_0p_0 + \alpha_1p_1 + \alpha_2p_2 \;=\; p_0 + \alpha_1(p_1 - p_0) + \alpha_2(p_2 - p_0).$$

The set of all affine combinations of three (noncolinear) points generates a plane. The set of all convex combinations of three points generates all the points of the triangle defined by the points. These shapes are called the *affine span* or *affine closure*, and *convex closure* of the points, respectively.

Euclidean Geometry: In affine geometry we have provided no way to talk about angles or distances. Euclidean geometry is an extension of affine geometry which includes one additional operation, called the *inner product*, which maps two real vectors (not points) into a nonnegative real. One important example of an inner product is the *dot product*, defined as follows. Suppose that the d-dimensional vector $\vec{u}$ is represented by the (nonhomogeneous) coordinate vector $(u_1, u_2, \ldots, u_d)$. Then define

$$\vec{u} \cdot \vec{v} = \sum_{i=0}^{d-1} u_i v_i,$$

The dot product is useful in computing the following entities.

Length: of a vector $\vec{v}$ is defined to be $|\vec{v}| = \sqrt{\vec{v} \cdot \vec{v}}$.

Normalization: Given any nonzero vector $\vec{v}$, define the *normalization* to be a vector of unit length that points in the same direction as $\vec{v}$. We will denote this by $\hat{v}$:

$$\hat{v} = \frac{\vec{v}}{|\vec{v}|}.$$

Distance between points: Denoted either dist(p, q) or $|pq|$ is the length of the vector between them, $|p - q|$.

Angle: between two nonzero vectors $\vec{u}$ and $\vec{v}$ (ranging from 0 to π) is

$$\mathrm{ang}(\vec{u}, \vec{v}) = \cos^{-1}\left(\frac{\vec{u} \cdot \vec{v}}{|\vec{u}||\vec{v}|}\right). = \cos^{-1}(\hat{u} \cdot \hat{v}).$$

This is easy to derive from the law of cosines.

Lecture 3: Orientations and Convex Hulls

Reading: Chapter 1 in the 4M's (de Berg, van Kreveld, Overmars, Schwarzkopf). The divide-and-conquer algorithm is given in Joseph O'Rourke's, "Computational Geometry in C." O'Rourke's book is also a good source for information about orientations and some other geometric primitives.

Orientation: In order to make discrete decisions, we would like a geometric operation that operates on points in a manner that is analogous to the relational operations $(<, =, >)$ with numbers. There does not seem to be any natural intrinsic way to compare two points in d-dimensional space, but there is a natural relation between ordered $(d+1)$-tuples of points in d-space, which extends the notion of binary relations in 1-space, called *orientation*.

Given an ordered triple of points $\langle p, q, r\rangle$ in the plane, we say that they have *positive orientation* if they define a counterclockwise oriented triangle, *negative orientation* if they define a clockwise oriented triangle, and *zero orientation* if they are collinear (which includes as well the case where two or more of the points are identical). Note that orientation depends on the order in which the points are given.

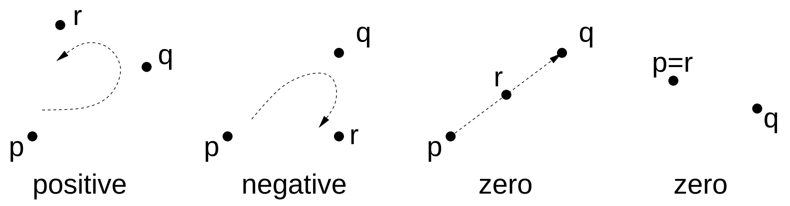

Figure 8: Orientations of the ordered triple (p, q, r).

Orientation is formally defined as the sign of the determinant of the points given in homogeneous coordinates, that is, by prepending a 1 to each coordinate. For example, in the plane, we define

$$\text{Orient}(p, q, r) = \det \begin{pmatrix} 1 & p_x & p_y \\ 1 & q_x & q_y \\ 1 & r_x & r_y \end{pmatrix}.$$

Observe that in the 1-dimensional case, Orient(p, q) is just $q - p$. Hence it is positive if $p < q$, zero if $p = q$, and negative if $p > q$. Thus orientation generalizes $<, =, >$ in 1-dimensional space. Also note that the sign of the orientation of an ordered triple is unchanged if the points are translated, rotated, or scaled (by a positive scale factor). A reflection transformation, e.g., $f(x, y) = (-x, y)$, reverses the sign of the orientation. In general, applying any affine transformation to the point alters the sign of the orientation according to the sign of the matrix used in the transformation.

In general, given an ordered 4-tuple points in 3-space, we can also define their orientation as being either positive (forming a right-handed screw), negative (a left-handed screw), or zero (coplanar). This can be generalized to any ordered $(d+1)$-tuple points in d-space.

Areas and Angles: The orientation determinant, together with the Euclidean norm can be used to compute angles in the plane. This determinant Orient(p, q, r) is equal to twice the signed area of the triangle $\triangle pqr$ (positive if CCW and negative otherwise). Thus the area of the triangle can be determined by dividing this quantity by 2. In general in dimension d the area of the simplex spanned by $d + 1$ points can be determined by taking this determinant and dividing by $(d!)$. Once you know the area of a triangle, you can use this to compute the area of a polygon, by expressing it as the sum of triangle areas. (Although there are other methods that may be faster or easier to implement.)

Recall that the dot product returns the cosine of an angle. However, this is not helpful for distinguishing positive from negative angles. The sine of the angle $\theta = \angle pqr$ (the signed angled from vector $p - q$ to vector $r - q$) can be computed as

$$\sin\theta = |p-q||r-q|\text{Orient}(q,p,r).$$

(Notice the order of the parameters.) By knowing both the sine and cosine of an angle we can unambiguously determine the angle.

Convexity: Now that we have discussed some of the basics, let us consider a fundamental structure in computational geometry, called the *convex hull*. We will give a more formal definition later, but the convex hull can be defined intuitively by surrounding a collection of points with a rubber band and letting the rubber band snap tightly around the points.

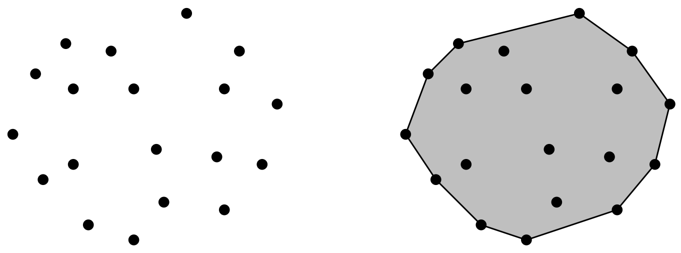

Figure 9: A point set and its convex hull.

There are a number of reasons that the convex hull of a point set is an important geometric structure. One is that it is one of the simplest shape approximations for a set of points. It can also be used for approximating more complex shapes. For example, the convex hull of a polygon in the plane or polyhedron in 3-space is the convex hull of its vertices. (Perhaps the most common shape approximation used in the minimum axis-parallel bounding box, but this is trivial to compute.)

Also many algorithms compute the convex hull as an initial stage in their execution or to filter out irrelevant points. For example, in order to find the smallest rectangle or triangle that encloses a set of points, it suffices to first compute the convex hull of the points, and then find the smallest rectangle or triangle enclosing the hull.

Convexity: A set S is *convex* if given any points $p, q \in S$ any convex combination of p and q is in S, or equivalently, the line segment $\overline{pq} \subseteq S$.

Convex hull: The *convex hull* of any set S is the intersection of all convex sets that contains S, or more intuitively, the smallest convex set that contains S. Following our book's notation, we will denote this $\mathcal{CH}(S)$.

An equivalent definition of convex hull is the set of points that can be expressed as convex combinations of the points in S. (A proof can be found in any book on convexity theory.) Recall that a convex combination of three or more points is an affine combination of the points in which the coefficients sum to 1 and all the coefficients are in the interval $[0,1]$.

Some Terminology: Although we will not discuss topology with any degree of formalism, we will need to use some terminology from topology. These terms deserve formal definitions, but we are going to cheat and rely on intuitive definitions, which will suffice for the sort simple, well behaved geometry objects that we will be dealing with. Beware that these definitions are not fully general, and you are refered to a good text on topology for formal definitions. For our purposes, define a *neighborhood* of a point p to be the set of points whose distance to p is strictly less than some positive r, that is, it is the set of points lying within an open ball of radius r centered about p. Given a set S, a point p is an *interior point* of S if for some radius r the neighborhood about p of radius r is contained within S. A point is an *exterior point* if it an interior point for the complement of S.

Points that are neither interior or exterior are boundary points. A set is *open* if it contains none of its boundary points and *closed* if its complement is open. If p is in S but is not an interior point, we will call it a *boundary point*. We say that a geometric set is *bounded* if it can be enclosed in a ball of finite radius. A *compact set* is one that is both closed and bounded.

In general, convex sets may have either straight or curved boundaries and may be bounded or unbounded. Convex sets may be topologically open or closed. Some examples are shown in the figure below. The convex hull of a finite set of points in the plane is a bounded, closed, convex polygon.

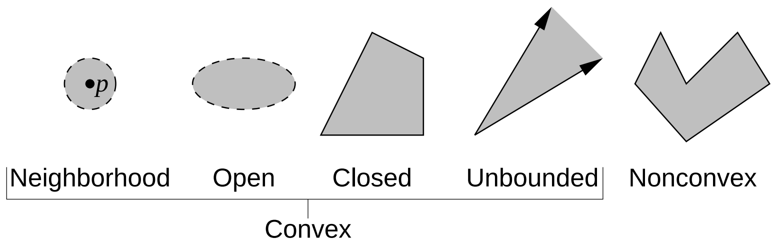

Figure 10: Terminology.

Convex hull problem: The (planar) *convex hull problem* is, given a set of n points P in the plane, output a representation of P's convex hull. The convex hull is a closed convex polygon, the simplest representation is a counterclockwise enumeration of the vertices of the convex hull. (A clockwise is also possible. We usually prefer counterclockwise enumerations, since they correspond to positive orientations, but obviously one representation is easily converted into the other.) Ideally, the hull should consist only of *extreme points*, in the sense that if three points lie on an edge of the boundary of the convex hull, then the middle point should not be output as part of the hull.

There is a simple $O(n^3)$ convex hull algorithm, which operates by considering each ordered pair of points (p, q), and the determining whether all the remaining points of the set lie within the half-plane lying to the right of the directed line from p to q. (Observe that this can be tested using the orientation test.) The question is, can we do better?

Graham's scan: We will present an $O(n \log n)$ algorithm for convex hulls. It is a simple variation of a famous algorithm for convex hulls, called *Graham's scan*. This algorithm dates back to the early 70's. The algorithm is based on an approach called *incremental construction*, in which points are added one at a time, and the hull is updated with each new insertion. If we were to add points in some arbitrary order, we would need some method of testing whether points are inside the existing hull or not. To avoid the need for this test, we will add points in increasing order of x-coordinate, thus guaranteeing that each newly added point is outside the current hull. (Note that Graham's original algorithm sorted points in a different way. It found the lowest point in the data set and then sorted points cyclically around this point.)

Since we are working from left to right, it would be convenient if the convex hull vertices were also ordered from left to right. The convex hull is a cyclically ordered sets. Cyclically ordered sets are somewhat messier to work with than simple linearly ordered sets, so we will break the hull into two hulls, an *upper hull* and *lower hull*. The break points common to both hulls will be the leftmost and rightmost vertices of the convex hull. After building both, the two hulls can be concatenated into a single cyclic counterclockwise list.

Here is a brief presentation of the algorithm for computing the upper hull. We will store the hull vertices in a stack U, where the top of the stack corresponds to the most recently added point. Let $\text{first}(U)$ and $\text{second}(U)$ denote the top and second element from the top of U, respectively. Observe that as we read the stack from top to bottom, the points should make a (strict) left-hand turn, that is, they should have a positive orientation. Thus, after adding the last point, if the previous two points fail to have a positive orientation, we pop them off the stack. Since the orientations of remaining points on the stack are unaffected, there is no need to check any points other than the most recent point and its top two neighbors on the stack.

Let us consider the upper hull, since the lower hull is symmetric. Let $\langle p_1, p_2, \ldots, p_n \rangle$ denote the set of points, sorted by increase x-coordinates. As we walk around the upper hull from left to right, observe that each consecutive triple along the hull makes a right-hand turn. That is, if p, q, r are consecutive points along the upper hull, then $\text{Orient}(p, q, r) < 0$. When a new point p_i is added to the current hull, this may violate the right-hand turn

```
(1) Sort the points according to increasing order of their $x$-coordinates, denoted $\langle p_1, p_2, \ldots, p_n\rangle$.
(2) Push $p_1$ and then $p_2$ onto $U$.
(3) for $i = 3$ to $n$ do:
    (a) while size$(U) \geq 2$ and Orient$(p_i, \text{first}(U), \text{second}(U)) \leq 0$, pop $U$.
    (b) Push $p_i$ onto $U$.
```

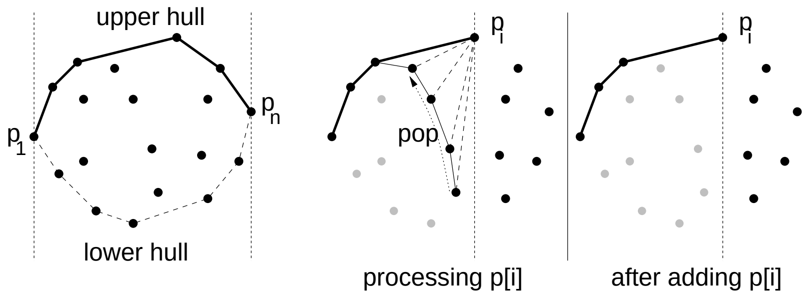

Figure 11: Convex hulls and Graham's scan.

invariant. So we check the last three points on the upper hull, including p_i. They fail to form a right-hand turn, then we delete the point prior to p_i. This is repeated until the number of points on the upper hull (including p_i) is less than three, or the right-hand turn condition is reestablished. See the text for a complete description of the code. We have ignored a number of special cases. We will consider these next time.

Analysis: Let us prove the main result about the running time of Graham's scan.

Theorem: Graham's scan runs in $O(n \log n)$ time.

Proof: Sorting the points according to x-coordinates can be done by any efficient sorting algorithm in $O(n \log n)$ time. Let D_i denote the number of points that are popped (deleted) on processing p_i. Because each orientation test takes $O(1)$ time, the amount of time spent processing p_i is $O(D_i + 1)$. (The extra $+1$ is for the last point tested, which is not deleted.) Thus, the total running time is proportional to

$$\sum_{i=1}^{n}(D_i + 1) \;=\; n + \sum_{i=1}^{n} D_i.$$

To bound $\sum_i D_i$, observe that each of the n points is pushed onto the stack once. Once a point is deleted it can never be deleted again. Since each of n points can be deleted at most once, $\sum_i D_i \leq n$. Thus after sorting, the total running time is $O(n)$. Since this is true for the lower hull as well, the total time is $O(2n) = O(n)$.

Convex Hull by Divide-and-Conquer: As with sorting, there are many different approaches to solving the convex hull problem for a planar point set P. Next we will consider another $O(n \log n)$ algorithm, which is based on the divide-and-conquer design technique. It can be viewed as a generalization of the famous MergeSort sorting algorithm (see Cormen, Leiserson, and Rivest). Here is an outline of the algorithm. It begins by sorting the points by their x-coordinate, in $O(n \log n)$ time.

The asymptotic running time of the algorithm can be expressed by a recurrence. Given an input of size n, consider the time needed to perform all the parts of the procedure, ignoring the recursive calls. This includes the time to partition the point set, compute the two tangents, and return the final result. Clearly the first and third of these steps can be performed in $O(n)$ time, assuming a linked list representation of the hull vertices. Below we

```
(1) If $|P| \leq 3$, then compute the convex hull by brute force in $O(1)$ time and return.
(2) Otherwise, partition the point set $P$ into two sets $A$ and $B$, where $A$ consists of half the points with the lowest
    $x$-coordinates and $B$ consists of half of the points with the highest $x$-coordinates.
(3) Recursively compute $H_A = \mathcal{CH}(A)$ and $H_B = \mathcal{CH}(B)$.
(4) Merge the two hulls into a common convex hull, $H$, by computing the upper and lower tangents for $H_A$ and $H_B$
    and discarding all the points lying between these two tangents.
```

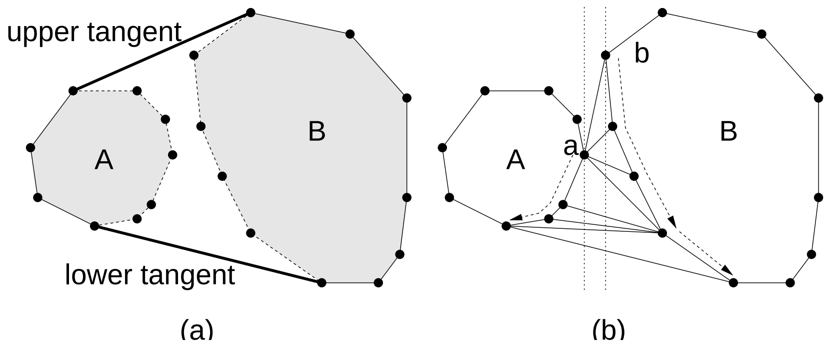

Figure 12: Computing the lower tangent.

will show that the tangents can be computed in $O(n)$ time. Thus, ignoring constant factors, we can describe the running time by the following recurrence.

$$T(n) = \begin{cases} 1 & \text{if } n \leq 3 \\ n + 2T(n/2) & \text{otherwise.} \end{cases}$$

This is the same recurrence that arises in Mergesort. It is easy to show that it solves to $T(n) \in O(n \log n)$ (see CLR). All that remains is showing how to compute the two tangents.

One thing that simplifies the process of computing the tangents is that the two point sets A and B are separated from each other by a vertical line (assuming no duplicate x-coordinates). Let's concentrate on the lower tangent, since the upper tangent is symmetric. The algorithm operates by a simple "walking" procedure. We initialize a to be the rightmost point of H_A and b is the leftmost point of H_B. (These can be found in linear time.) Lower tangency is a condition that can be tested locally by an orientation test of the two vertices and neighboring vertices on the hull. (This is a simple exercise.) We iterate the following two loops, which march a and b down, until they reach the points lower tangency.

Finding the Lower Tangent

```
**LowerTangent**$(H_A, H_B)$ :
    (1) Let $a$ be the rightmost point of $H_A$.
    (2) Let $b$ be the leftmost point of $H_B$.
    (3) While $ab$ is not a lower tangent for $H_A$ and $H_B$ do
        (a) While $ab$ is not a lower tangent to $H_A$ do $a = a - 1$ (move $a$ clockwise).
        (b) While $ab$ is not a lower tangent to $H_B$ do $b = b + 1$ (move $b$ counterclockwise).
    (4) Return $ab$.
```

Proving the correctness of this procedure is a little tricky, but not too bad. Check O'Rourke's book out for a careful proof. The important thing is that each vertex on each hull can be visited at most once by the search, and hence its running time is $O(m)$, where $m = |H_A| + |H_B| \leq |A| + |B|$. This is exactly what we needed to get the overall $O(n \log n)$ running time.

Lecture 4: More Convex Hull Algorithms

Reading: Today's material is not covered in the 4M's book. It is covered in O'Rourke's book on Computational Geometry. Chan's algorithm can be found in T. Chan, "Optimal output-sensitive convex hull algorithms in two and three dimensions", *Discrete and Computational Geometry*, 16, 1996, 361–368.

QuickHull: If the divide-and-conquer algorithm can be viewed as a sort of generalization of MergeSort, one might ask whether there is corresponding generalization of other sorting algorithm for computing convex hulls. In particular, the next algorithm that we will consider can be thought of as a generalization of the QuickSort sorting procedure. The resulting algorithm is called QuickHull.

Like QuickSort, this algorithm runs in $O(n \log n)$ time for favorable inputs but can take as long as $O(n^2)$ time for unfavorable inputs. However, unlike QuickSort, there is no obvious way to convert it into a randomized algorithm with $O(n \log n)$ expected running time. Nonetheless, QuickHull tends to perform very well in practice.

The intuition is that in many applications most of the points lie in the interior of the hull. For example, if the points are uniformly distributed in a unit square, then it can be shown that the expected number of points on the convex hull is $O(\log n)$.

The idea behind QuickHull is to discard points that are not on the hull as quickly as possible. QuickHull begins by computing the points with the maximum and minimum, x- and y-coordinates. Clearly these points must be on the hull. Horizontal and vertical lines passing through these points are support lines for the hull, and so define a bounding rectangle, within which the hull is contained. Furthermore, the convex quadrilateral defined by these four points lies within the convex hull, so the points lying within this quadrilateral can be eliminated from further consideration. All of this can be done in $O(n)$ time.

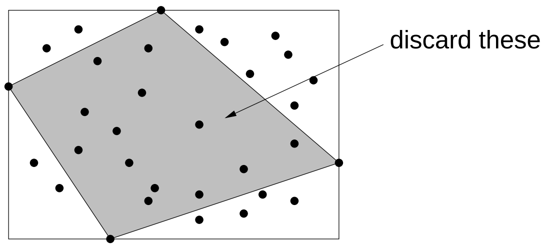

Figure 13: QuickHull's initial quadrilateral.

To continue the algorithm, we classify the remaining points into the four corner triangles that remain. In general, as this algorithm executes, we will have an inner convex polygon, and associated with each edge we have a set of points that lie "outside" of that edge. (More formally, these points are witnesses to the fact that this edge is not on the convex hull, because they lie outside the half-plane defined by this edge.) When this set of points is empty, the edge is a final edge of the hull. Consider some edge ab. Assume that the points that lie "outside" of this hull edge have been placed in a bucket that is associated with ab. Our job is to find a point c among these points that lies on the hull, discard the points in the triangle abc, and split the remaining points into two subsets, those that lie outside ac and those than lie outside of cb. We can classify each point by making two orientation tests.

How should c be selected? There are a number of possible selection criteria that one might think of. The method that is most often proposed is to let c be the point that maximizes the perpendicular distance from the line ab. (For example, another possible choice might be the point that maximizes the angle cba or cab. It turns out that