

# COMPUTATIONAL GEOMETRY

## Review for Final

### 1 Convex Hulls in 3 or More Dimensions

Know that 3D convex hull of  $n$  points can be constructed in  $O(n \log n)$  time (by divide and conquer).

The convex hull is stored as a planar graph (its vertices and edges), in a “winged edge” data structure or similar, which allows us to walk around faces, step around the edges incident on a vertex, etc, in constant time per step.

The convex hull of  $n$  points in  $d$  dimensions, for  $d \geq 4$ , can be computed in time  $O(n^{\lfloor d/2 \rfloor})$  and has worst-case size  $O(n^{\lfloor d/2 \rfloor})$ . Thus, in 4D, the convex hull is computed in time  $O(n^2)$  and may have quadratic complexity (i.e., there could be  $\Omega(n^2)$  edges and faces of dimension 2 or 3; obviously, there can be at most  $n$  vertices).

### 2 Voronoi and Delaunay

See the handout on Voronoi and Delaunay properties and facts!

You should understand the basic combinatorics of planar subdivisions and triangulations:  $f - e + v = 2$  (Euler),  $e \leq 3v - 6$ , sum of the degrees equals  $2e$  (true for all graphs), sum of the number of sides on each face of a planar graph is  $2e$ . In a triangulation of the convex hull of  $n$  points ( $h$  of which are on the hull), these facts imply that the number of triangles is  $t = f = 2n - h - 2$ , and that  $e = 3n - h - 3$ . You should be able to derive and use such relationships.

You should be able to draw the diagrams for small sets of points. You should be able to draw the NNG, the RNG, the MST, the approximate TSP tour, etc.

You should understand what the furthest-point Delaunay/Voronoi diagram is, as in HW6. You should be able to draw a small example (I would assist you by showing some relevant circles, so it is useful to realize that the furthest-point Delaunay edges are defined by the “full circle” property).

### 3 Arrangements

Understand the concepts of vertices, edges, and faces (cells) in an arrangement of  $n$  lines. We consider edges to be *open*, not including endpoints (if any), and faces also to be *open* (not including their boundary). Sign vectors: We can represent any vertex/edge/face by a *sign vector*,  $(a_1, \dots, a_n)$ , where  $a_i \in \{0, +, -\}$  indicates if the vertex/edge/face lies on, above, or below (respectively) the line  $\ell_i$ .

You should understand how many vertices, edges, faces there can be (upper and lower bounds). In particular, for a *simple*<sup>1</sup> arrangement of  $n$  lines in the plane, there are exactly  $v = \binom{n}{2}$  vertices,  $e = n^2$  edges, and  $f = \binom{n}{2} + n + 1$  faces. (For non-simple arrangements, replace “=” with “ $\leq$ ”.) You should understand these relationships.

Thus, the complexity of the whole arrangement is  $O(n^2)$  (for any arrangement, not just simple arrangements). The complexity of a single cell (2-dimensional face) is  $O(n)$ . The complexity of a set of  $m$  faces is at most  $O(m^{2/3}n^{2/3} + n + m)$ .

Understand the definition of a “zone” (the union of the set of (open) cells that intersect a line  $\ell$ ).

**Zone Theorem:** The sum of the complexities of the  $\leq n + 1$  cells that are stabbed by a line  $\ell$  in an arrangement of  $n$  lines is at most  $6n$ . (The proof is by induction.) This leads to an  $O(n^2)$  simple algorithm for constructing an arrangement of  $n$  lines (building a “winged-edge” data structure for the arrangement).

The  $k$ -level in an arrangement of  $n$  lines is the set of edges for which there are exactly  $k - 1$  lines strictly above the edge, together with the endpoints of these edges. The  $k$ -level is a useful concept in speaking about the “Ham Sandwich Theorem”: Any set of red points and blue points in the plane has a ham sandwich cut,  $\ell$ , that simultaneously splits in half (as nearly as possible in case of an odd number of points) both the red set and the blue set.

### 4 Duality

**Definition:** The “standard” (or “usual”) duality for us maps the point  $(a, b)$  to the line  $y = 2ax - b$ . Understand the relationship of the points and dual lines to the standard parabola ( $y = x^2$ ), as in Figure 6.6. You should also

<sup>1</sup>Meaning that no three lines pass through a common vertex and no two lines are parallel.

have basic knowledge of polarity, as you did on the homework (HW7):  $(a, b)$  gets mapped to the line  $ax + by = 1$ . Understand the relationship with the unit circle centered on the origin, as in the homework.

Properties: Understand the 5 basic properties of duality: Section 6.5.2. What is the dual of the set of points that make up a line segment? a ray? How is the convex hull of  $n$  points related to the upper and lower envelopes of their dual lines? (see HW7)

Applications: We saw a few – finding a line to stab the most segments (duality and arrangements methods solve in  $O(n^2)$ , versus a more naive  $O(n^3)$ ); computing the intersection of  $n$  halfplanes in time  $O(n \log n)$  (since duality tells us it is the same problem as computing convex hulls of point sets); angular sorting (sort by angle all the points around each point  $p_i$ ; duality gives  $O(n^2)$  instead of  $O(n^2 \log n)$ ); Ham Sandwich Theorem. Additional applications include degeneracy testing (given  $n$  points in the plane, determine if any three of them are collinear), in  $O(n^2)$  time instead of the naive  $O(n^3)$ ; hidden surface elimination for  $n$  triangles in 3D (done in  $O(n^2)$  time by sweeping the arrangement of the lines through the projections of the triangle edges in the viewing plane; “topological sweep” allows this to be done using only  $O(n)$  memory (very important in graphics applications!)); applications to “ $k$ -levels” in arrangements and connections to the  $k$ th order Voronoi diagram (discussed in text), and many others.

## 5 Intersection Detection/Reporting Among Segments

Bentley-Ottmann sweep: Understand the algorithm and its complexity ( $O(n \log n)$  to detect if there is an intersection,  $O((k + n) \log n)$  to compute all  $k$  of the intersections among the  $n$  segments). Note that the Bentley-Ottmann sweep is optimal (best possible (proof is from Element Uniqueness)),  $\Theta(n \log n)$ , for *detection*; however, it is not optimal for *reporting*. For reporting all intersections, there is in fact an algorithm of optimal time  $\Theta(k + n \log n)$  (Chazelle-Edelsbrunner, and then Balaban improved the memory usage to  $O(n)$ ).

Be able to apply the algorithm to a small example, building a chart as we did in HW8 and on the Bentley-Ottmann handout. Understand both the “usual” and the “modified” versions of the algorithm. (The modified version removes events  $x_{i,j}$  whenever  $s_i$  and  $s_j$  stop being adjacent in the sweep line status,  $\mathcal{L}$ , and then reinserts them later when adjacency is re-established. This allows us to keep the total memory usage for the queue  $Q$  and  $\mathcal{L}$  at only  $O(n)$ .)

## 6 Point Location Search

A polygonal subdivision  $\mathcal{S}$  is a drawing of a planar graph with straight, noncrossing edges.

It is *simple* if every face is a simple polygon (no floating “island” or “holes”). For example, a triangulation is a simple polygonal subdivision, as is a Voronoi diagram.

If  $\mathcal{S}$  has  $n$  vertices, we know it must have (at most)  $O(n)$  edges and  $O(n)$  faces.

In the point location problem, we want to *preprocess*  $\mathcal{S}$ , building a data structure, so that we can quickly answer *point location queries* for any query point  $q = (x, y)$ : Which face of  $\mathcal{S}$  contains  $q$ ?

We gave an optimal point location method based on Kirkpatrick’s algorithm. It takes a subdivision  $\mathcal{S}$ , triangulates all the faces that are not already triangles (put a big triangle surrounding the whole thing, if the outer face (at infinity) is not a triangle), and then builds a hierarchical data structure (a DAG) that permits  $O(\log n)$  time point location queries. The preprocessing is  $O(n)$  if the input is a triangulation; if the input is a *simple* polygonal subdivision, then we know we can triangulate it (face by face) in time  $O(n)$  (using Chazelle’s triangulation algorithm); if some of the faces have holes, then it may take  $O(n \log n)$  time to triangulate it, after which the rest of the preprocessing algorithm takes only  $O(n)$  time.

You should be able to build the Kirkpatrick hierarchy and understand how to use it to do point location on small examples.

The general method applies to polygonal subdivisions; as special cases, we can also use it to determine in  $O(\log n)$  time if a point  $q$  lies inside or outside a simple polygon  $P$ , etc. (For convex polygons (or even monotone polygons), you saw on HW8 that even simpler methods are possible, without the Kirkpatrick hierarchy, for doing point location, based on binary search.) Of course, if we just want to test one or two points for containment in a simple polygon, we may prefer not building a hierarchical data structure that would allow us to do them in  $O(\log n)$  per query; the point-in-polygon test based on shooting a ray and counting intersections (to see if the number is even or odd) may be the method of choice (see `InPoly1` of Section 7.4).