An Introduction to Computational Geometry

Contents:

1. General introduction, application areas, literature

2. Survey of typical problems in computational geometry

3. Problem solution technique Plane Sweep

    3.1 Computing the tightest pair of n points in the plane
    3.2 Intersection of n line segments in the plane
    3.3 Intersection of two polygons

COMPUTATIONAL GEOMETRY

(since ca. 1975)

- Development of efficient and practical algorithms
  for the solution of geometric problems
- Determining the algorithmic complexity of geometric problems

Applications

- Robotics
  e.g. motion planning, orientation in unknown environment
- Computer aided geometric design
  e.g. computing intersection / union of geometric objects
- Geographic information systems (GIS)
  e.g. combining maps, queries with combinations of geometric properties
- Computer graphics
  e.g. visibility of 3D objects, ray tracing, radiosity
- Others
  e.g. molecular modeling, pattern/character recognition

Focus

- algorithmic core problems
  e.g. convex hull of n points in the plane,
  finding the closest of n points; ...
- data structures for efficient retrieval of geometric data
  e.g. k-dimensional search trees
- algorithmic techniques
  e.g. plane sweep, divide-and-conquer, randomized incremental construction,
  geometric transformation, domain decomposition
- degeneracies and robustness
  e.g. collinear points, roundoff-errors, ...

Literature (1)

- de Berg, van Kreveld, Overmars, Schwarzkopf:
  Computational Geometry, Algorithms and Applications, Second Edition.
  Springer, 2001.
  `http://www.cs.uu.nl/geobook/`
- J. Goodman and J. O'Rourke (eds.):
  The Handbook of Discrete and Computational Geometry.
  CRC Press, 1997
- J. Sack, J. Urrutia:
  Handbook of Computational Geometry.
  Elsevier, 1997
- M. Laszlo:
  Computational Geometry and Computer Graphics in C++.
  Prentice Hall, 1996

Literature (2)

The classic textbooks on computational geometry:

- F. Preparata, M. Shamos: Computational Geometry. Springer, 1985
- K. Mehlhorn: Multi-dimensional Searching and Computational Geometry. Springer, 1984
- R. Edelsbrunner: Algorithms in Combinatorial Geometry. Springer, 1987.
- K. Mulmuley: Computational Geometry: An Introduction through Randomized Algorithms. Prentice Hall, 1993.

In german language:

- Rolf Klein: Algorithmische Geometrie. Addison Wesley, 1997

Survey papers:

- J. Matousek: Geometric Range Searching. ACM Computing Surveys 26(4), 1994.

Literature (3)

Journals

Discrete Comp. Geometry, Comp. Geom. Theory Appl.,
J. Algorithms, Algorithmica, Acta Informatica, J.ACM, SIAM J. Comput.,
...

Conferences

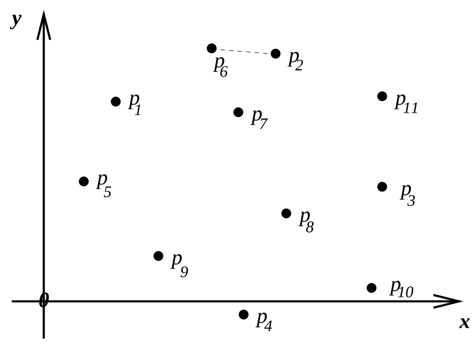
ACM Symp. on Comput. Geom., ACM/SIAM Symp. on Discrete Algorithms,
...

Web resources

- LEDA library of efficient data structures and algorithms, Univ. Saarbrücken
- CGAL computational geometry algorithms library, Univ. Saarbrücken
- ...

Some Typical Problems in Computational Geometry (1)

Tightest pair of n points in the plane

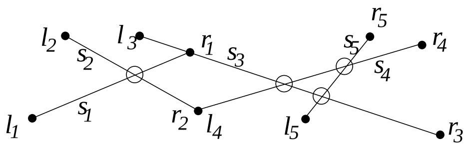


given n points $p_1, ..., p_n$ in the plane $\mathbb{R}^2$

determine minimum distance of two points p_i, p_j, $1 \le i < j \le n$,
and (maybe) pair (p_i, p_j)

Some Typical Problems in Computational Geometry (2)

Intersection of n line segments in the plane

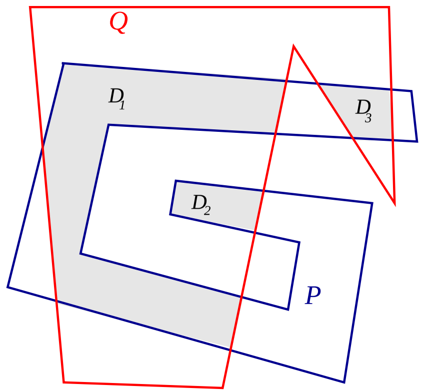


given: n line segments $S = \{s_i = \overline{l_i r_i}, i = 1,\dots,n\}$ (incl. endpoints) in the plane

compute: all k *proper* intersection points (no end point of a segment)

Some Typical Problems in Computational Geometry (3)

Intersection of two polygons



Intersection $P \cap Q$ in general not contiguous

$\Rightarrow$ set of polygons D_i, $i = 1,\dots,r$

Some Typical Problems in Computational Geometry (4)

Multidimensional search structures

e.g. for interval queries, rectangular range queries

- k-dimensional search tree / BSP tree
- quadtree, octree
- priority search tree
- segment tree

static / dynamic

Some Typical Problems in Computational Geometry (5)

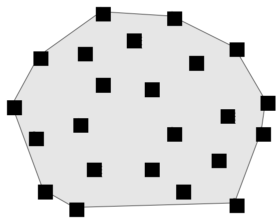
Convex Hull of n points in $\mathbb{R}^2$

given: set S of n points $p_i = (x_i, y_i) \in \mathbb{R}^2$, $i = 1,\dots,n$

compute the convex hull $ch(S)$ of S:

$$ch(S) = \bigcap_{K \supset S, K \text{ convex}} K$$

= the smallest convex set containing S.



Analogy: nails and rubberband

Some Typical Problems in Computational Geometry (6)

Lower bound for computing the convex hull of n points in $\mathbb{R}^2$

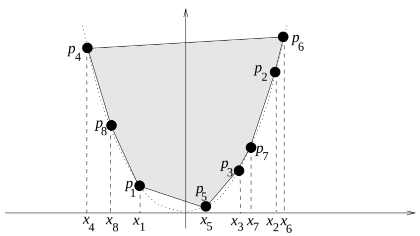
Computing $ch(S)$ needs time $\Omega(n \log n)$.

Reduction to sorting of n real numbers:

Let A be an arbitrary algorithm that computes the convex hull.

Given n real numbers $x_1, ..., x_n$

Set $S = \{p_i := (x_i, x_i^2) : i = 1, ..., n\}$



With A construct $ch(S)$: all p_i appear as vertices!

Linear traversal of the vertices of $ch(S)$, starting at the p_i with least x-coordinate, yields a sorted sequence of the x_i in linear time.

If A were faster than $O(n \log n)$ we could accordingly sort faster, contradiction!

Some Typical Problems in Computational Geometry (7)

Triangulation of a simple polygon

given: simple polygon P with n vertices

compute: decomposition of P into triangles

Triangulation T of P

= $\partial P \cup$ maximal set of non-intersecting diagonals in P

Existence proof by induction.

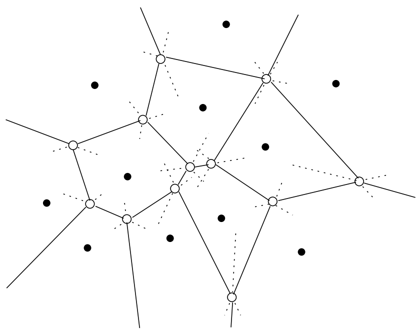
A triangulation of a convex polygon can be computed in time $O(n)$.

A triangulation of a simple polygon can be computed in time $O(n \log n)$, $O(n \log^* n)$, $O(n)$

Delaunay-Triangulation: a special triangulation T where for each edge $d = (v_1, v_2)$ in T the smallest circle around d contains no further vertex of P.

Some Typical Problems in Computational Geometry (8)

Voronoi diagram



simplest case:
VD for a set S of n points $p_1, ..., p_n$ in the plane $\mathbb{R}^2$

Voronoi region of a point $p_i \in S$:
subset of points in $\mathbb{R}^2$ that are closer to p_i than to any other $p_j \in S$

Voronoi diagram $VD(S)$ is a graph (Voronoi nodes, Voronoi edges)

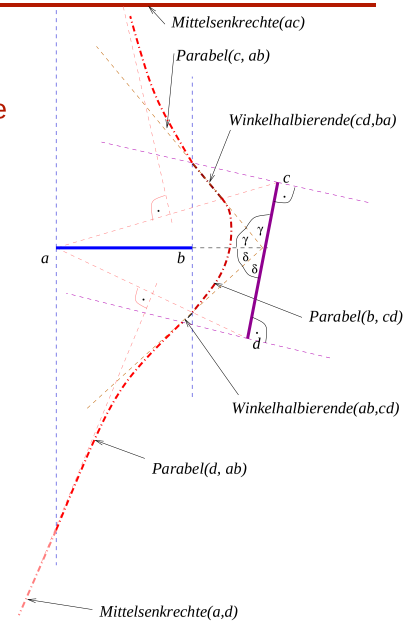
Voronoi nodes: points in $\mathbb{R}^2$ that have minimum distance to > 2 points of S

Voronoi edges: points in $\mathbb{R}^2$ that have minimum dist. to exactly 2 points of S

For $|S| = n$ has $VD(S)$ $O(n)$ Voronoi nodes and $O(n)$ Voronoi edges.

Some Typical Problems in Computational Geometry (9)

Voronoi diagram of line segments in the plane



Bisector point – point:
straight line (Mittelsenkrechte)

Bisector point – straight line:
parabel

Bisector point set – straight line:
wave front of parabel arcs

Bisector straight line – straight line:
straight line (Winkelhalbierende)

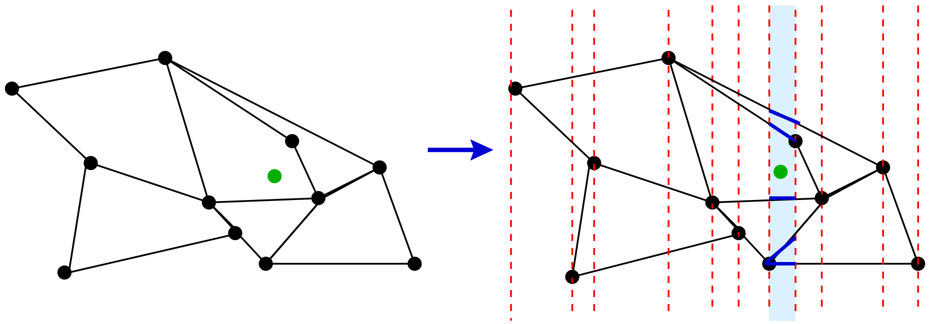
Bisector of two line segments:
composed from these

Some Typical Problems in Computational Geometry (10)

Point location

Given: a map = a planar subdivision of the plane into regions (e.g., polygons)
given also: a point q in the plane

compute the region that contains q (query)



preprocessing: partitioning (e.g. in slabs), build balanced search structures

Some Typical Problems in Computational Geometry (11)

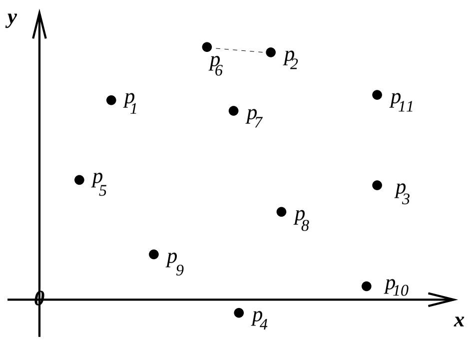
Robotics – Motion planning

Determine a collision-free path in the plane
(any, or the shortest path, or the most power-consuming path, ...)
for a robot (point, circle, polygon, ladder)
from point A to B in a scene of polygonal obstacles.

For circular robot: use point location and Voronoi diagram

PLANE SWEEP

Example problem: tightest pair of n points in the plane



given n points $p_1, \ldots, p_n$ in the plane $\mathbb{R}^2$
determine minimum distance of two points p_i, p_j, $1 \leq i < j \leq n$,
and (maybe) pair (p_i, p_j)

Tightest pair of n points in the plane (1)

naive method: enumerate all pairs

$currMinD \leftarrow \infty$
for each point p_i, $i = 1, \ldots, n-1$
  for each point p_j, $j = i+1, \ldots, n$
    if $|p_i - p_j| < currMinD$
      then $currMinD \leftarrow |p_i - p_j|$
output $currMinD$;

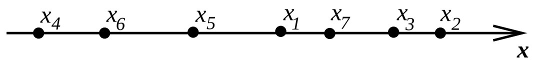
run time: $\Theta(n^2)$

Improvement?

Tightest pair of n points in the plane (2)

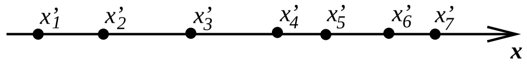
Consider the one-dimensional case:

given: n real numbers $x_1, \ldots, x_n$



determine: tightest pair (x_i, x_j) with $|x_i - x_j|$ minimal, $i \neq j$

Step 1: sort the x_i in increasing order $\rightarrow \quad x'_1 \leq x'_2 \leq \ldots \leq x'_n$



Step 2: scan the x'_i in increasing order
keeping track of the position $PosCurrDP$ of the current tightest pair $(x'_{PosCurrDP-1}, x'_{PosCurrDP})$ and its distance $CurrMinD$

Tightest pair of n points in the plane (3)

Pseudocode:

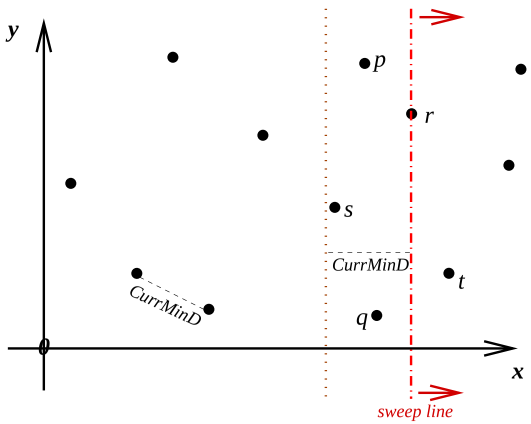
$Sort(\ x\ )$
$PosCurrMinD \leftarrow 2;$
$CurrMinD \leftarrow |x'_2 - x'_1|;$
for $i = 3, \ldots, n$
  if $CurrMinD > |x'_j - x'_{j-1}|$
    then $CurrMinD \leftarrow |x'_j - x'_{j-1}|;$
      $PosCurrMinD \leftarrow j;$
output $CurrMinD$, $PosCurrMinD$;

Run time: $O(n \log n)$

Tightest pair of n points in the plane (4)

back to the 2D case:

Method: Sweep over the plane in the direction of the x-axis



Tightest pair of n points in the plane (5)

We observe:

If the sweep line meets a new point (e.g. r), all potential partners of r to form a pair with distance $< CurrMinD$ are located in the interior of a stripe of width $CurrMinD$ behind the sweep line.

This stripe "moves" with the sweep line to the right
(where its width may be adapted if necessary)

$\rightarrow$ It is sufficient to consider, at any point of time, only the points located in the interior of the stripe.

Tightest pair of n points in the plane (6)

Data structure for the “stripe behind the sweep line”:

Sweep-status-structure (SSS, also called Y-structure)

requires efficient support of the following operations:

- insert point into SSS
- remove point from SSS
- find point in SSS with minimum distance

Tightest pair of n points in the plane (7)

Events that require an update of the SSS:

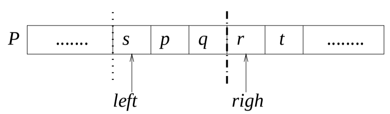
1. left border of stripe moves across a point p
   $\rightarrow$ remove p from SSS
2. sweep line meets a new point r
   $\rightarrow$ insert r into SSS
   check whether some point p in the SSS
   has distance $< CurrMinD$ from r
   if yes:
   update $CurrMinD$ (= stripe width)
   remove from the SSS all points p that now have a distance $\geq CurrMinD$

Tightest pair of n points in the plane (7)

Data structure to determine the order of processing the points:
Event structure or X-structure

Observation:
Points enter and leave the SSS in order of increasing x-coordinates

$\rightarrow$ Preprocessing:
sort points in order of increasing x coordinates in an array $P[1:n]$:



During the sweep keep two indices:
index *left* points to leftmost point in the stripe
index *right* points to leftmost point to the right of the sweep line

Tightest pair of n points in the plane (8)

Order of $P[left]$ versus $P[right]$:

If $P[left].x + CurrMinD \leq P[right].x$
then $P[left]$ is processed first (to be removed from SSS)
else $P[right]$ is processed first (to be inserted in SSS)

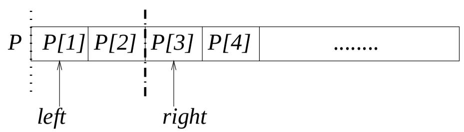
Tightest pair of n points in the plane (9)

Putting things together: the sweep algorithm

// Initialisation:

sort the n points by increasing x-coordinates
and insert them into array P

SSS.init(); // initially SSS is empty
SSS.insert(P[1]);
SSS.insert(P[2]);
$CurrMinD \leftarrow |P[2] - P[1]|$;
$left \leftarrow 1$;
$right \leftarrow 3$;



Tightest pair of n points in the plane (10)

// Sweep:

while /* I_1 holds */ $(right < n)$ **do**
  if $P[left].x + CurrMinD \leq P[right].x$
    then // old point *P[left]* leaves stripe
      SSS.remove(P[left]);
      $left \leftarrow left + 1$;
    else // new point *P[right]* enters stripe; I_2 holds
      SSS.insert(P[right]);
      $right \leftarrow right + 1$;
      $CurrMinD \leftarrow$ *SSS.MinDist(P[right], CurrMinD)*;
output *CurrMinD*;

still to be specified: routine *MinDist*

Tightest pair of n points in the plane (11)

Correctness of the algorithm:

Lemma 1: At the program points denoted by I_1 und I_2
the following invariants hold:

I_1: The minimum distance among the points *P[1]...P[right–1]* is *CurrMinD*.
I_2: The SSS contains exactly the points *P[i]*, $1 \leq i \leq right-1$
with $P[i].x > P[right].x - CurrMinD$

Proof: by induction (Exercise)

Tightest pair of n points in the plane (11) – Run time of the algorithm

The preprocessing takes time $O(n \log n)$ (sorting).

Sweep: Each point is inserted into the SSS exactly once and removed at most once.

Inserting a point into the SSS and removing a point from the SSS
can be done in time $O(\log n)$
if the SSS is implemented as a balanced search tree.

A call to *MinDist(P[i],m)* takes ($\rightarrow$) time $O(\log n + k_i)$
where k_i = number of potential partners of $P[i]$ in the SSS at that time.

$\rightarrow$ total run time: $O(n \log n + \sum_{i=3}^{n} k_i)$

remains to be done: upper bound for k_i, $i = 1, ..., n$

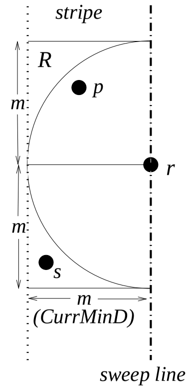
(We shall see: $k_i \leq 10$, i.e. constant)

Tightest pair of n points in the plane (12) – Routine MinDist()

For a given point r and minimum distance m,
SSS.MinDist(r, m) determines the minimum $\min_{p \in SSS} |\overline{pr}|$

Only a point located in the interior of the rectangle R (more precisely: in the half-circle around r with radius m) may have a distance $< m$ from r.

$\rightarrow$ only the y-coordinates of the points in SSS are of interest.



Implementation of the SSS

e.g. as AVL tree whose leaves (points) are linearly linked in order of increasing y-coordinates

$\rightarrow$ Insert / Remove in time $O(\log n)$

$\rightarrow$ *MinDist()* in time $O(\log n + k_i)$
where k_i = number of leaves within rectangle

Tightest pair of n points in the plane (13) – Upper bound for k_i, $i = 1, \ldots, n$

Lemma 2:
Given a set P of points in the plane
that have (pairwise) at least distance $m > 0$.
Then a rectangle R with edge lengths M und $2M$ contains ≤ 10 points of P.

Proof: pairwise minimum distance m
$\rightarrow$ circles around the points with radius $m/2$ do not overlap.

For each point of R at least a quarter of its circle's area is contained in R
$\rightarrow$ R may contain at most

$$\frac{\text{Area}(R)}{\text{Area(Quarter circle sector)}} = \frac{2m^2}{\frac{1}{4}\pi\left(\frac{m}{2}\right)^2} = \frac{32}{\pi} < 11$$

points of P

$\rightarrow k_i \leq 10.$

Remark: a sharper bound yields $k_i \leq 6$.

Tightest pair of n points in the plane (13) – Summary

The minimum distance of n points in the plane
can be computed in time $O(n \log n)$.
This is asymptotically optimal. [Hinrichs, Nievergelt, Schorn, IPL 26, 1988]

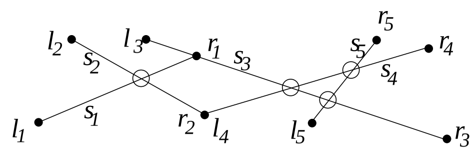
Problem solution method "plane sweep":

- basic idea: exploit locality
- data structures: SSS (Y-structure), X-structure
- update rules for events
  must preserve invariants $\rightarrow$ correctness
- transforms a static 2D problem into a dynamic 1D problem
- X-structure may also be dynamic

in $\mathbb{R}^3$: similar, with a sweep plane

Next example: Intersection of n line segments in the plane

Intersection of n line segments in the plane



given: n line segments $S = \{s_i = \overline{l_i r_i}, i = 1, \ldots, n\}$ (incl. endpoints) in the plane $\mathbb{R}^2$

compute: all k *proper* intersection points (no end point of a segment)

Lower bound for run time: $O(n \log n + k)$ see [Klein'97]

Naive method: Enumerating all pairs

for all pairs of segments s_i, s_j $i \neq j$:
  if there is a proper intersection point $p = s_i \cap s_j$
  then output p;

Run time: $\Theta(n^2)$ $\Rightarrow$ only acceptable if $k = \Theta(n^2)$

Intersection of n line segments with Plane Sweep (1)

[J. Bentley, T. Ottmann: Algorithms for reporting and counting geometric intersections. IEEE Trans. Comp. C-28, 1979]

Preliminary assumptions:

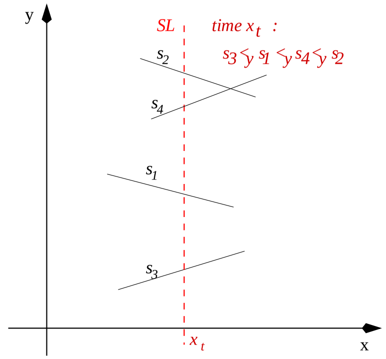
1. x-coordinates of all segment endpoints are distinct
   ($\rightarrow$ no segment is parallel to the y–axis, thus notation l_i (left endpoint), r_i (right endpoint) is well-defined)
2. any 2 line segments $s_i \neq s_j$ intersect in at most one point
3. in each intersection point intersect at most 2 segments.

Intersection of n line segments with Plane Sweep (2)

Move Sweep-Line SL from left to right over plane.

At time x_t, $-\infty < x_t < \infty$, is SL the straight line $x = x_t$

At any time x_t let $P_t = \{s_j \in S : s_j \cap \{x = x_t\} \neq \emptyset\}$



On the segments in P_t there is a total order $<_y$ according to increasing y-coordinates of intersection points with SL

$\Rightarrow$ keep track of the order of the $s_j \in P_t$ in the *Sweep Status Structure SSS*

Intersection of n line segments with Plane Sweep (3)

Events that change the order $<_y$ of the $s_j \in P_t$:

1. SL meets left endpoint l_i of a segment s_i
2. SL meets right endpoint r_j of a segment s_j
3. SL meets (proper) intersection point p of two segments s_i, s_j

Order of processing events of type (1.) and (2.) is clear:
sort all endpoints l_i and r_i in increasing x-coordinates Time $O(n \log n)$

for events of type (3.) ???

are computed only during the computation
$\Rightarrow$ requires *dynamic* event data structure ES

Intersection of n line segments with Plane Sweep (4)

Events are represented as triplets:

$(l_i, s_i, 0)$ Item for left endpoint of s_i
$(r_j, 0, s_j)$ Item for right endpoint of s_j
(p, s_i, s_j) Item for intersection point of s_i and s_j

where the time of an event is the x-coordinate of the point.

Intersection of n line segments with Plane Sweep (5)

Operations on ES:

$(p, s_i, s_j) = ES.deleteMin()$; // dequeue next event from ES

$ES.insert(p, s_i, s_j)$; // insert event in ES at position $p.x$

$ES.remove(p, s_i, s_j)$; // remove event from ES

$\Rightarrow$ Priority Queue

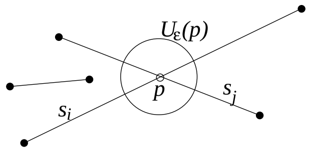
implemented e.g. as balanced binary tree

$\rightarrow$ all operations perform in time $O(\log |ES|) = O(\log(2n+k)) = O(\log n)$

Intersection of n line segments with Plane Sweep (6)

Lemma 3: If two line segments s_i, s_j, $i \neq j$, have a proper intersection point p then they are direct neighbors (wrt. the order along the SL, i.e. in SSS)) immediately before the event (p, s_i, s_j)

Proof:



Let $\varepsilon > 0$, such that $U_\varepsilon(p)$ is intersected only by s_i and s_j.
$\Rightarrow$ for all x_t with $p.x - \varepsilon < x_t < p.x$ are s_i, s_j direct neighbors w.r.t. the order along the SL.

Invariant: Intersections of segments directly neighbored in SSS are computed and inserted in ES.

$\Rightarrow$ no intersection point is missed when computing $s_i \cap s_j$ as soon as s_i and s_j become direct neighbors in SSS.

Intersection of n line segments with Plane Sweep (7)

Actions for events

Type 1: left endpoint l_i of a segment s_i:

```
leftEndpoint(l_i, s_i) {
 s ← SSS.insert (s_i, key = l_i.y);
 s_u ← SSS.predecessor(s);
 s_o ← SSS.successor(s);
 if s_u exists:
   compute p = s_u ∩ s_i;
   if p ex.: ES.insert(p, s_u, s_i);
 if s_o exists:
   compute p = s_o ∩ s_i;
   if p ex.: ES.insert(p, s_i, s_o);
}
```


Intersection of n line segments with Plane Sweep (8)

Type 2: right endpoint r_j of a segment s_j:

```
rightEndpoint(r_j, s_j) {
 s ← SSS.find (s_j, key = r_j.y);
 s_u ← SSS.predecessor(s);
 s_o ← SSS.successor(s);
 if s_u and s_o exist:
   compute p = s_u ∩ s_o;
   if p ex.: ES.insert(p, s_u, s_o);
 SSS.remove (s);
}
```

Intersection of n line segments with Plane Sweep (9)

Type 3: intersection point p of two segments s_i, s_j:

$intersectionPoint(p, s_i, s_j)$ {
output ("Intersection point:", p, s_i, s_j);
$s \leftarrow SSS.find\ (s_i, key = p.y)$;
$s' \leftarrow SSS.successor(s)$;
$s_u \leftarrow SSS.predecessor(s)$;
$s_o \leftarrow SSS.successor(s')$;
if s_o exists:
  compute $q = s_o \cap s_i$;   if q ex.: $ES.insert(q, s_i, s_o)$;
if s_u exists:
  compute $r = s_u \cap s_j$;   if r ex.: $ES.insert(r, s_u, s_j)$;
$SSS.exchange\ (s, s')$;
// **if** s_u and s_o exist:
//   compute $t = s_u \cap s_o$;   if t ex.: $ES.remove(t)$;
}

Intersection of n line segments with Plane Sweep (10)

Space requirements: dominated by $\max |ES| \leq 2n + k$

may be limited to $3n$ if ES stores only the intersection points of segments that are *directly* neighbored in SSS (commented lines)

Intersection of n line segments with Plane Sweep (11) – the entire algorithm

#include <LEDA/sortseq.h>
#include <LEDA/pqueue.h>
...
typedef struct { point p, segment s_i, segment s_j;} triplet;
sortseq <float, segment> SSS; // init.: empty
pqueue <float,triplet> ES; // init.: empty
...
for all segments s_i, $i = 1, ..., n$:
 $ES.insert\ (l_i, s_i, 0)$; $ES.insert\ (r_i, 0, s_j)$;
while $ES.nonempty()$ {
  $(p, s_i, s_j) \leftarrow ES.deleteMin()$;
  if $s_j = 0$ // left endpoint – type 1
   $leftEndpoint(p, s_i)$;
  if $s_i = 0$ // right endpoint – type 2
   $rightEndpoint(p, s_j)$;
  otherwise: $intersectionPoint(p, s_i, s_j)$; // – type 3

Intersection of n line segments with Plane Sweep (12)

Run time: $|ES| \leq 2n + k \Rightarrow$ time: $O((n+k)\log n)$

Remark 1: a rough bound. See [Pach/Sharir SIAM J. Comput. 20, 1991]: $|ES| = \Theta(n \log n)$

(Exercise: Lower bound $\Omega(n \log n)$)

Remark 2: Run time $O((n+k)\log n)$ not optimal:
[Chazelle/Edelsbrunner J.ACM 1992]: time $O(k + n\log n)$, space $O(n+k)$
[Balaban '95]: time $O(k + n\log n)$ optimal, space $O(n)$ optimal

Intersection of n line segments with Plane Sweep (13)

Removing the simplifying assumptions

1. Distinct x-coordinates of the endpoints

at insertion into ES: ($ES.insert()$)
use lexikographic order on pairs (x,y)
corresponds to a minimal counterclockwise rotation of the coordinate system

at sweep: ($ES.deleteMin()$)
with multiple events of equal x-coordinate, process
first all left endpoints,
then alle intersection points,
then all right endpoints,
in each category in increasing order of y-coordinates

Intersection of n line segments with Plane Sweep (14)

2. Multiple intersection points

keep (see above) in ES only intersection points of segments directly neighbored on the SL

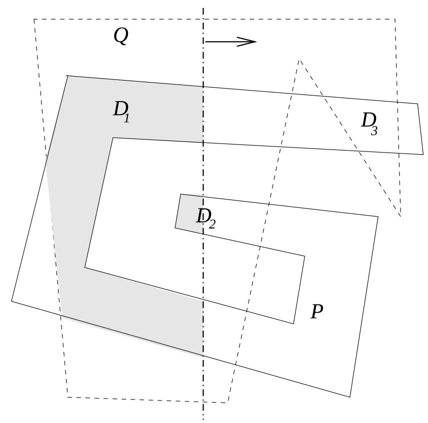
multiple intersection point p: immediately before SL reaches p,
ES contains a subsequence
$(p,s_1,s_2),\ (p,s_2,s_3),\ (p,s_3,s_4)$
such that p can be identified as multiple intersection point and the subsequence processed as a whole.

Intersection of n line segments with Plane Sweep (15)

3. Numerical problems (Accuracy of number representation / calculation)

Schorn '91
Burnikel/Mehlhorn/Schirra '94

Intersection of two polygons



Intersection $P \cap Q$ in general not contiguous
$\Rightarrow$ set of polygons D_i, $i = 1,\ldots,r$

Intersection of two polygons (2)

Intersection $P \cap Q$ is nonempty, if

(1) there ex. edges e of P and e' of Q with $e \cap e' \neq \emptyset$, <u>or</u>

(2) there ex. vertex v of P with v inside Q, or
there ex. vertex w of Q with w inside P

Test (1) with Plane-Sweep algorithm see above

Test (2) with ray test

$\Rightarrow$ Test for intersection of two polygons with n vertices altogether can be done in time $O(n \log n)$.

Intersection of two polygons with Plane Sweep (1)

Sweep-Line SL runs from left to right over scene

$SL \cap P$, $SL \cap Q$ define intervals of type $P \cap Q$, $P \cap \overline{Q}$, $\overline{P} \cap Q$, $\overline{P} \cap \overline{Q}$ on SL

Invariant: To the left of SL, partial intersection polygons D_i of $P \cap Q$ have been constructed.

Sweep Status Structure SSS: List of edges of P and Q that are currently intersected by SL ($\Rightarrow$ intervals)

For each of these edges e keep

- type of the interval $(e, SSS.successor(e))$
- reference e.edgeSequence to sequence F of edges (doubly linked list), representing the edges of the corresponding intersection polygon D_i to the left of SL

$\Rightarrow$ SSS-items are triplets (Edge e, EdgeSequence F, Type t)

Intersection of two polygons with Plane Sweep (2)

Implementation of SSS as balanced binary tree

$\Rightarrow$ *find()*, *insert()*, *remove()* in time $O(\log n)$

Events:

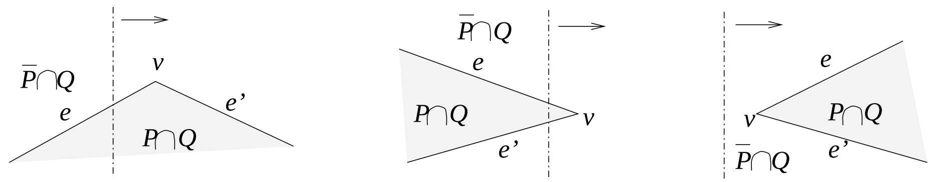
- vertices of P or Q
- intersection points of edges of P and Q

$\Rightarrow$ similar to "Intersection of line segments in the plane" with Priority Queue (or precompute if space doesn't matter)

Intersection of two polygons with Plane Sweep (3)

Updating the SSS:

1. Vertex v (e.g. of P): 3 cases:



(a) (b) (c)

(1a) one edge e von v to the left, one e' to the right:

item $i \leftarrow SSS.find(key=e)$;
i.edgeSequence.*append(e)*;
$i.edge \leftarrow e'$; // i.type remains the same

Intersection of two polygons with Plane Sweep (1)

(1b) both edges e, e' to the left:

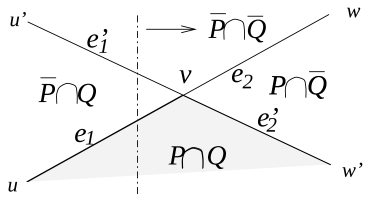
item $i_1 \leftarrow$ *SSS.find(key=e)*;
item $i_2 \leftarrow$ *SSS.find(key=e')*;
i_1.edgeSequence.*concatenate*(i_2.edgeSequence, e, e'); // make a loop
SSS.remove(i_1); *SSS.remove*(i_2); // remove interval

(1c) both edges e, e' to the right:

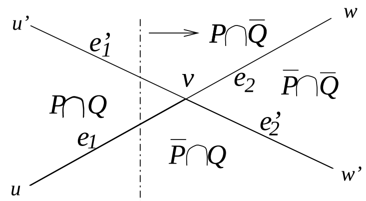
EdgeSequence $F_1(v)$, $F_2(v)$; // initialize empty chains from v
item $i_1(e, F_1, \overline{P} \cap Q)$; // new item
item $i_2(e', F_2, P \cap Q)$; // new item
$D_i \leftarrow$ *Solution.append*(F_1, F_2);// new partial inters.-pol.
SSS.insert(i_1, key=e); *SSS.insert(i_2, dir=before)*;

Intersection of two polygons with Plane Sweep (4)

2. Intersection point v of two edges e of P, e' of Q:

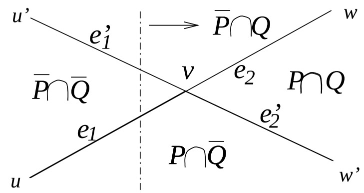


(2a)



(2b)

Number of intervals remains the same in (2a), (2b), (2c)



(2c)

Intersection of two polygons with Plane Sweep (5)

3 cases:

(2a)

item $i_1 \leftarrow$ *SSS.find(key=e)*;
item $i_2 \leftarrow$ *SSS.find(key=e')*;
Edge $e_1 \leftarrow (u, v)$, $e_2 \leftarrow (v, w)$; // split edge e
Edge $e'_1 \leftarrow (u', v)$, $e'_2 \leftarrow (v, w')$; // split edge e'
i_1.edgeSequence.*append*(e_1);
i_2.edgeSequence.*append*(e'_1);
i_1.Edge $\leftarrow (e'_2)$; i_1.type$\leftarrow P \cap \overline{Q}$;
i_2.Edge $\leftarrow (e_2)$; // i_2.type remains $\overline{P} \cap \overline{Q}$;

(2b), (2c): Exercise!

Intersection of two polygons with Plane Sweep (6)

The entire algorithm:

1. given: polygons P, Q with n vertices together
   SSS.init(); *ES.init()*; // initially empty
   solution.init(); // initially empty
2. sweep over scene as extension of the algorithm for line segment intersection by (1a), (1b),...,(2c)

   Time: $O((n+k)\log n)$

   ! When chaining edge sequences F_1, F_2, that previously belonged to different partial intersection polygons D_i, D_j, we must union D_i and D_j:

   F_2.polygon $\leftarrow F_1$.polygon;
   F_1.*append*(F_2);
   solution.remove (F_2.polygon);
3. For all remaining partial intersection polygons $D_i \in$ *solution*:
   solution.output(D_i);

Intersection of two polygons with Plane Sweep (7)

Theorem: The intersection of two simple polygons with n vertices and k edge intersection points can be computed in time $O((n+k)\log n)$ and space $O(n)$.

Special case: P, Q convex $\Rightarrow k = O(n)$ (Exercise)