DEPARTMENT OF COMPUTER GRAPHICS AND INTERACTION

GEOMETRIC SEARCHING
PART 1: POINT LOCATION in 2D

PETR FELKEL

FEL CTU PRAGUE

felkel@fel.cvut.cz

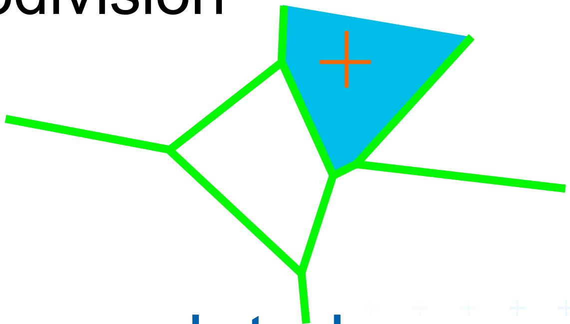
https://cw.felk.cvut.cz/doku.php/courses/a4m39vg/start

Based on [Berg] and [Mount]

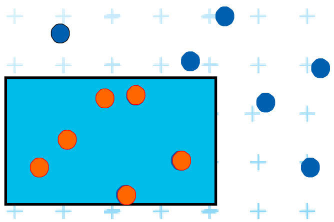
Version from 02.10.2024

Geometric searching problems

1. Point location (static) – Where am I?
   - (Find the name of the state, pointed by mouse cursor)
   - Search space S: a planar (spatial) subdivision
   - Query: point Q
   - Answer: region containing Q



2. Orthogonal range searching – Query a data base
   (Find points, located in d-dimensional axis-parallel box)
   - Search space S: a set of points
   - Query: set of orthogonal intervals q
   - Answer: subset of points in the box



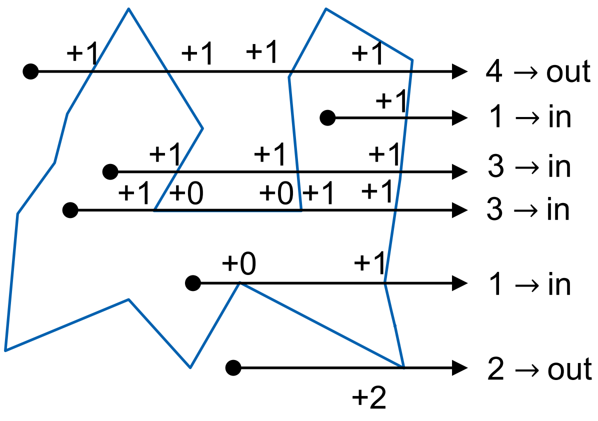
Both studied in DPG

Part 1: Point location in 2D

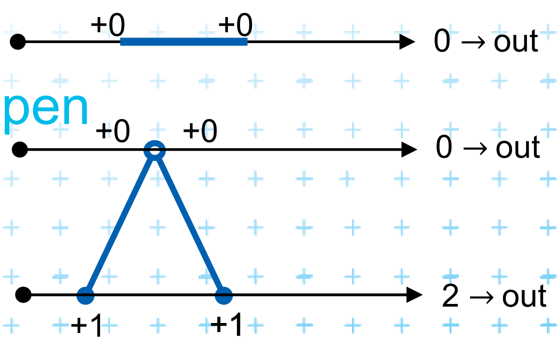
- Point location in polygon
- Planar subdivision
- DCEL data structure
- Point location in planar subdivision
  - slabs
  - monotone sequence
  - trapezoidal map

Point location in polygon by ray crossing

1. Ray crossing - $O(n)$
   - Compute number t of ray intersections with polygon edges (e.g., ray x_+ after point moved to origin)
   - $\text{odd}(t) \Rightarrow$ in
     $\text{even}(t) \Rightarrow$ out

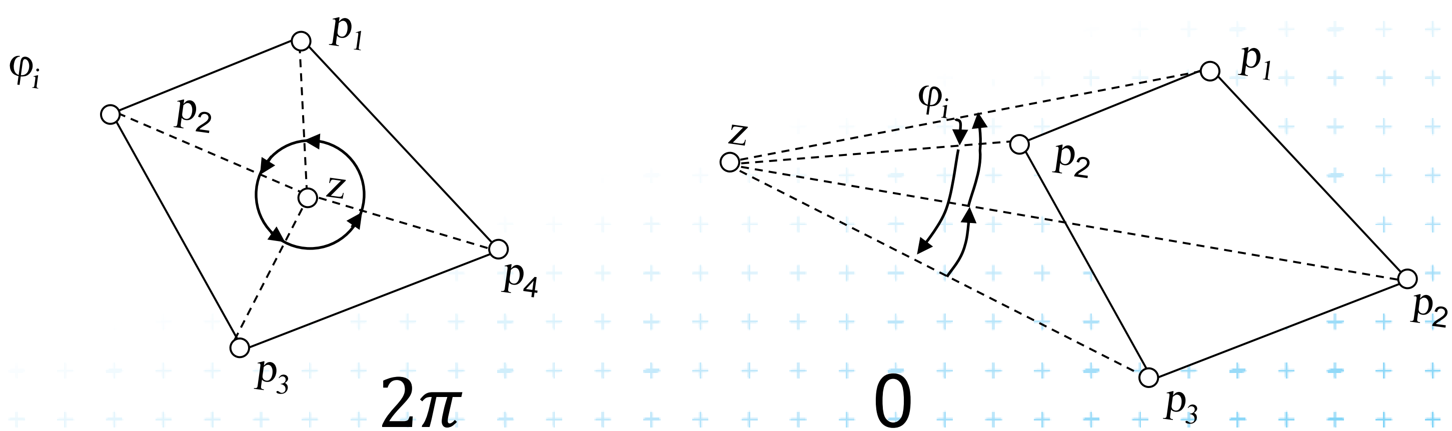


   - Singular cases must be handled!
     - Do not count horizontal line segments
     - Take non-horizontal segments as half-open (upper point not part of the segment)



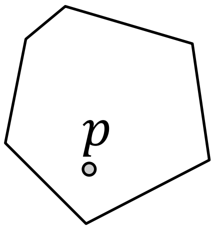
Point location in polygon

2. Winding number - $O(n)$
   (number of turns around the point)
   - Sum oriented angles $\varphi_i = \sphericalangle(p_i, z, p_{i+1})$
   - If $(\sum \varphi_i = 2\pi)$ then inside (1 turn)
   - If $(\sum \varphi_i = 0)$ then outside (no turn)
   - About 20-times slower than ray crossing

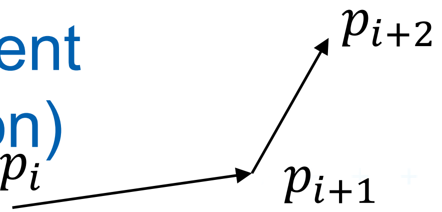


Point location in convex polygon

3. Position relative to all edges
   - For convex polygons
   - If (p is left from all edges) then inside



- Position of point in relation to the line segment (Determination of convex polygon orientation)



Convex polygon, non-collinear points

$p_i = [x_i, y_i, 1], \;\; p_{i+1} = [x_{i+1}, y_{i+1}, 1], \;\; p_{i+2} = [x_{i+2}, y_{i+2}, 1]$

$$\begin{vmatrix} x_i & y_i & 1 \\ x_{i+1} & y_{i+1} & 1 \\ x_{i+2} & y_{i+2} & 1 \end{vmatrix}$$

> 0 => point left from edge (for CCW polygon)

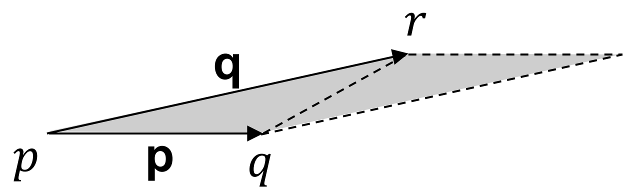
< 0 => point right from edge (for CW polygon)

Area of Triangle

$$T = \frac{1}{2}|\mathbf{p} \times \mathbf{q}|$$

$$\mathbf{p} = q - p$$

$$\mathbf{q} = r - p$$



$$2T = \mathbf{p}_x\mathbf{q}_y - \mathbf{p}_y\mathbf{q}_x$$ using vector product $\mathbf{p} \times \mathbf{q}$

$$2T = \begin{vmatrix} p_x & p_y & 1 \\ q_x & q_y & 1 \\ r_x & r_y & 1 \end{vmatrix}$$ using coordinates of points

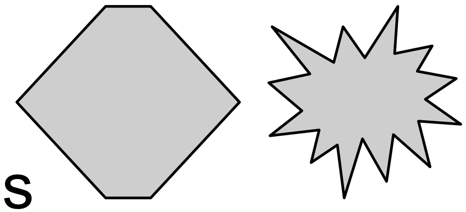
Orientation is computed as $\text{sign}(2\text{T}) =$

$$= \text{sign}(p_x q_y + q_x r_y + r_x p_y - p_x r_y - q_x p_y - r_x q_y)$$

$$= \text{sign}\left((q_x - p_x)(r_y - p_y) - (q_y - p_y)(r_x - p_x)\right)$$ for pivot p

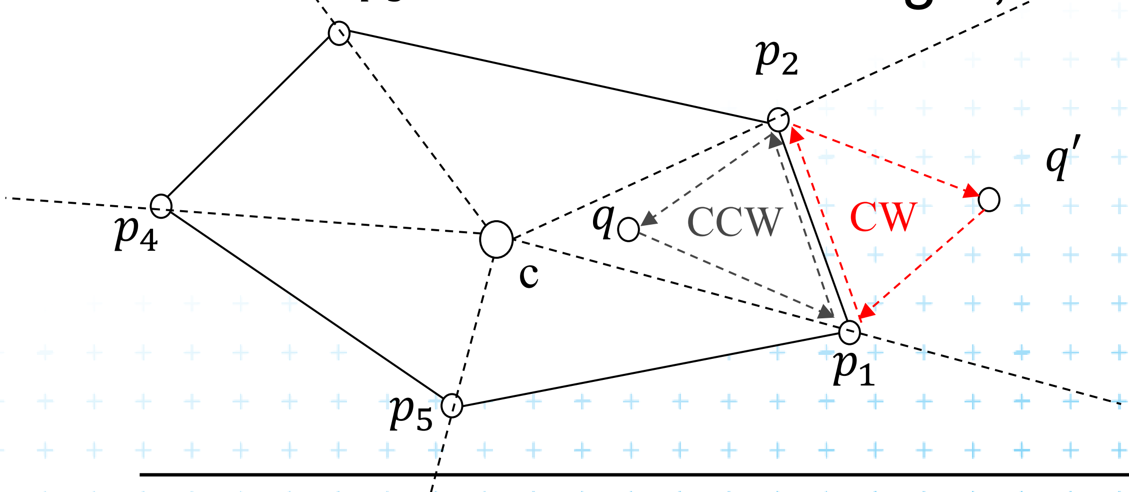
Point location in polygon

4. Binary search in angles



Works for convex and star-shaped polygons

1. Choose any point c inside / in the polygon core
2. c forms wedges with polygon edges, create BST
3. Binary search $q \rightarrow$ wedge výseč, based on left-right test
4. Finally compare with one edge (left, CCW => in, right, CW => out)

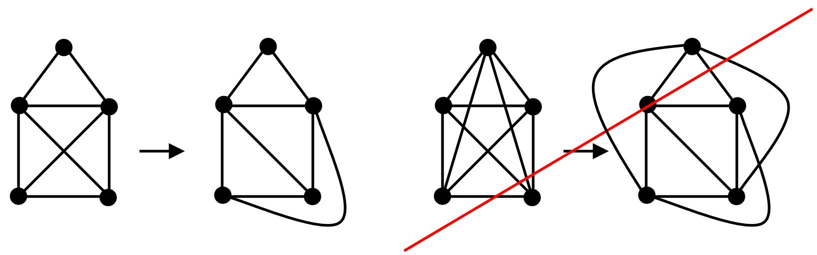


Planar graph

Planar graph

U=set of nodes, H=set of arcs

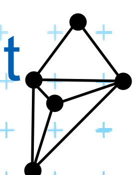
= Graph $G = (U, H)$ is planar, if it can be embedded into plane without crossings



Planar embedding of planar graph $G = (U, H)$

= mapping of each *node in U to vertex* in the plane and each *arc in H into simple curve (edge)* between the two images of end-nodes of the arc,
so that **no edges intersect** except at the vertices

Every planar graph can be embedded in such a way that arcs map to straight line segments [Fáry 1948]



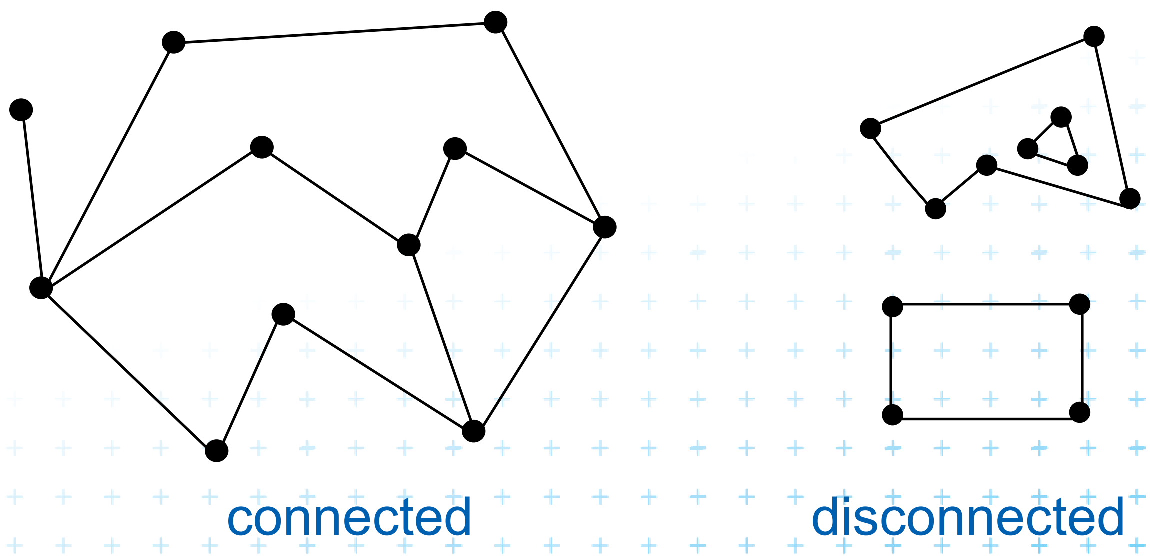
=> Planar Straight-line Graph

Planar subdivision

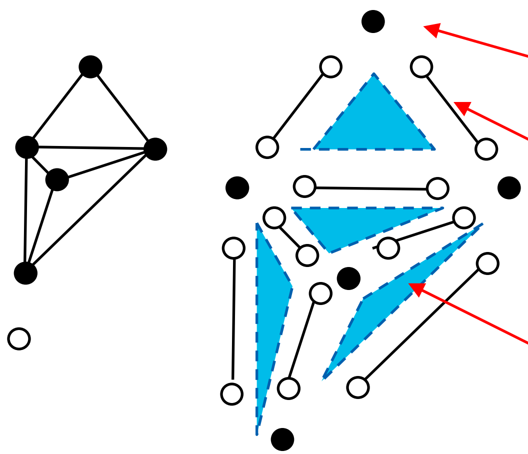
Also called PSLG – Planar Straight-line Graph

= Partition of the plane determined by *straight line planar embedding* of a planar graph.

= embedding of a planar graph in the plane such that its arcs are mapped into *straight line segments*



Planar subdivision



Vertex = embedding of graph node

Edge = embedding of graph arc (open – without vertices)

Face = maximal connected subset of a plane that doesn't contain points on edges nor vertices (open polygonal region whose boundary is formed by edges and vertices from the subdivision)

Complexity (size) of a subdivision = sum of number of vertices +
+ number of edges +
+ number of faces it consists of

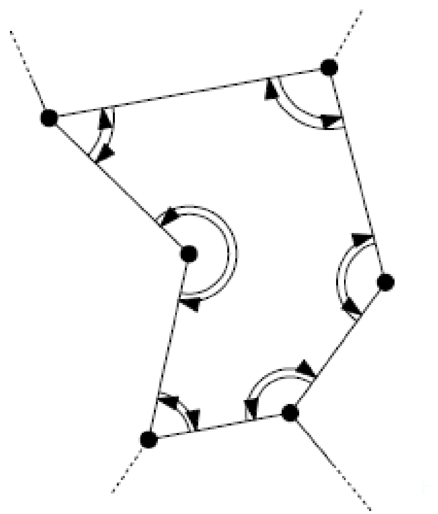
Euler's formula: $|V| - |E| + |F| \geq 2$
$|V| = n,\ |E| \leq 3n - 6,\ |F| \leq 2n - 4$

} $O(n)$ data structure

DCEL = Double Connected Edge List [Eastman 1982]

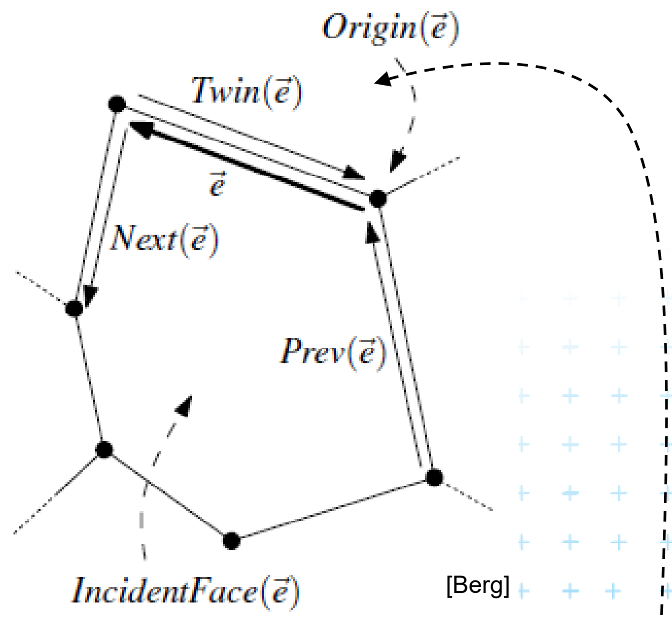
- A structure for storage of planar subdivision
- Operations like:

Walk around a face



Uses pointers to the *Next(e)* and *Prev(e)* edge

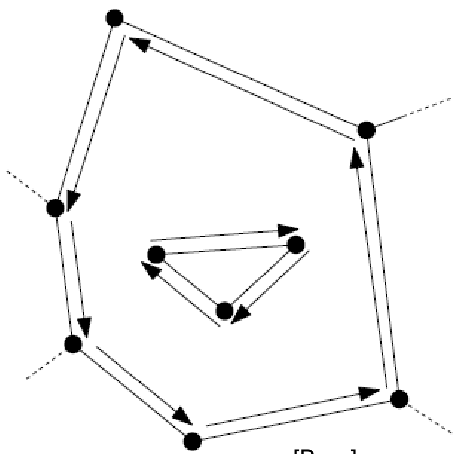
Get incident face



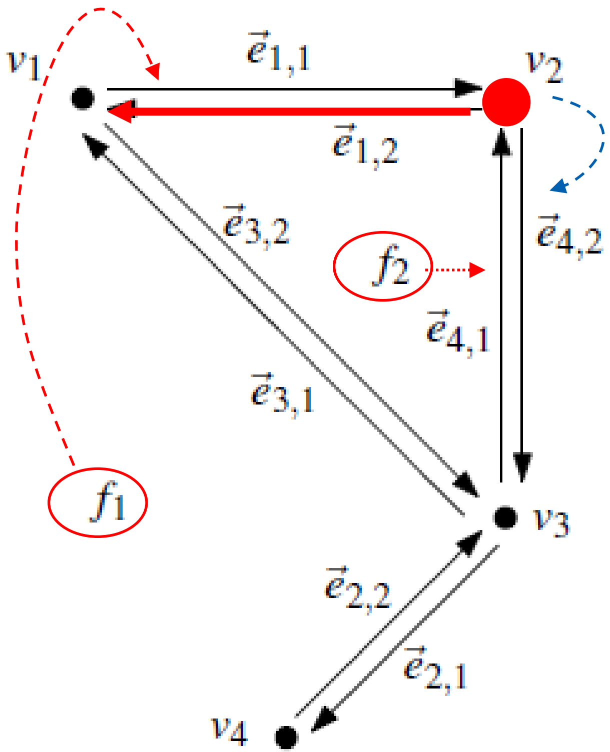
IncidentFace(e)
IncidentFace(Twin(e))

DCEL = Double Connected Edge List

- Vertex record v
  - Coordinates(v) and pointer to one IncidentEdge(v)
- Face record f
  - OuterComponent(f) pointer (boundary)
  - List of holes – InnerComponent(f)
- Half-edge record e
  - Origin(e), Twin(e), IncidentFace(e)
  - Next(e), Prev(e)
  - [Dest(e) = Origin(Twin(e))]
- Possible attribute data for each


[Berg]

DCEL = Double Connected Edge List



G – geometry
T – topology

G

Vertex	Coordinates	IncidentEdge
v_1	$(0,4)$	$\vec{e}_{1,1}$
v_2	$(2,4)$	$\vec{e}_{4,2}$
v_3	$(2,2)$	$\vec{e}_{2,1}$
v_4	$(1,1)$	$\vec{e}_{2,2}$

One of edges

List of holes (edges)

T

Face	OuterComponent	InnerComponents
f_1	**nil**	$\vec{e}_{1,1}$
f_2	$\vec{e}_{4,1}$	**nil**

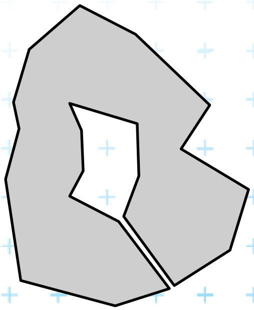
T

Half-edge	Origin	Twin	IncidentFace	Next	Prev
$\vec{e}_{1,1}$	v_1	$\vec{e}_{1,2}$	f_1	$\vec{e}_{4,2}$	$\vec{e}_{3,1}$
$\vec{e}_{1,2}$	v_2	$\vec{e}_{1,1}$	f_2	$\vec{e}_{3,2}$	$\vec{e}_{4,1}$
$\vec{e}_{2,1}$	v_3	$\vec{e}_{2,2}$	f_1	$\vec{e}_{2,2}$	$\vec{e}_{4,2}$
$\vec{e}_{2,2}$	v_4	$\vec{e}_{2,1}$	f_1	$\vec{e}_{3,1}$	$\vec{e}_{2,1}$
$\vec{e}_{3,1}$	v_3	$\vec{e}_{3,2}$	f_1	$\vec{e}_{1,1}$	$\vec{e}_{2,2}$
$\vec{e}_{3,2}$	v_1	$\vec{e}_{3,1}$	f_2	$\vec{e}_{4,1}$	$\vec{e}_{1,2}$
$\vec{e}_{4,1}$	v_3	$\vec{e}_{4,2}$	f_2	$\vec{e}_{1,2}$	$\vec{e}_{3,2}$
$\vec{e}_{4,2}$	v_2	$\vec{e}_{4,1}$	f_1	$\vec{e}_{2,1}$	$\vec{e}_{1,1}$

[Berg]

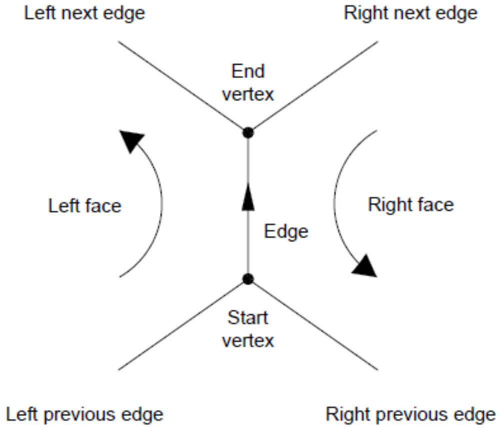
DCEL simplifications

- If no operations with vertices and no attributes
  - No vertex table (no separate vertex records)
  - Store vertex coords in half-edge origin (in the half-edge table)
- If no need for faces (e.g. river network)
  - No face record and no IncidentFace() field (in the half-edge table)
- If only connected subdivision allowed
  - Join holes with rest by dummy edges
  - Visit all half-edges by simple graph traversal
  - No InnerComponent() list for faces

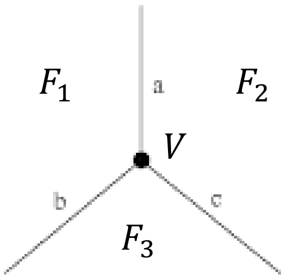


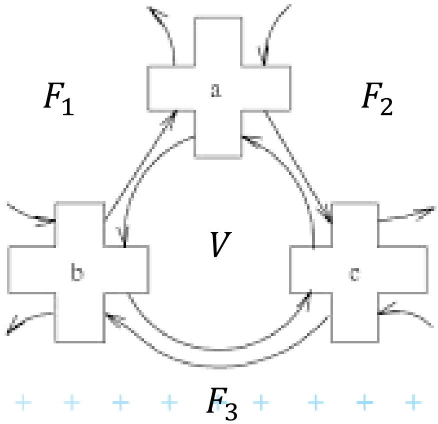
Other structures for representing PSLG

https://i.stack.imgur.com/yadGo.png



- Winged edge [Baumgart 1975]
  - The oldest, complicated manipulation
  - Randomly stored edge direction around faces
- Quad edge [Guibas & Stolfi 1985]
  - Stores PSLG and its dual
  - Pointers to edges
    - Around vertex
    - Around face
  - E.g., for Voronoi diagrams & Delaunay triangulations



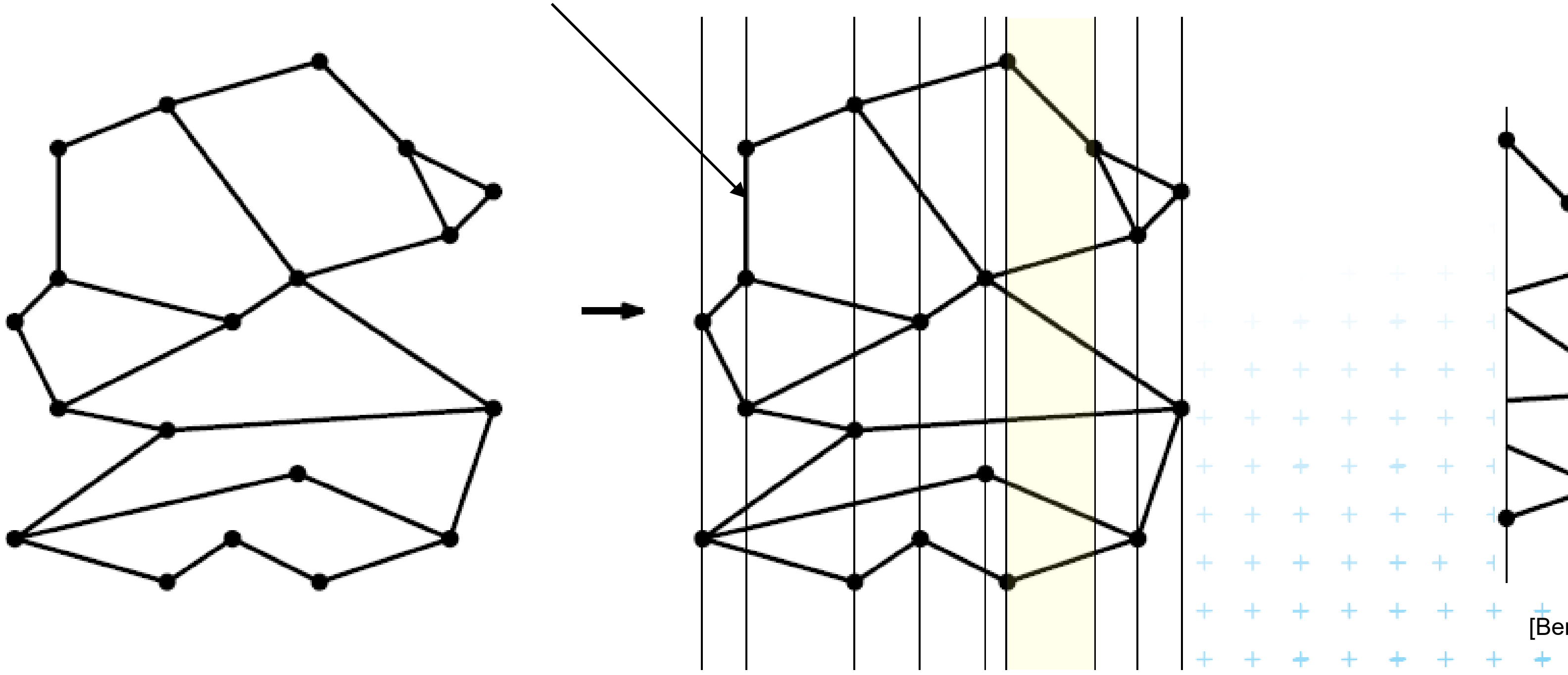


Point location in planar subdivision

- Using special search structures an optimal algorithm can be made with
  - $O(n)$ preprocessing,
  - $O(n)$ memory and
  - $O(\log n)$ query time.
- Simpler methods
  1. Slabs $O(\log n)$ query, $O(n^2)$ memory
  2. monotone chain tree $O(\log^2 n)$ query, $O(n^2)$ memory
  3. trapezoidal map $O(\log n)$ query expected time $O(n)$ expected memory

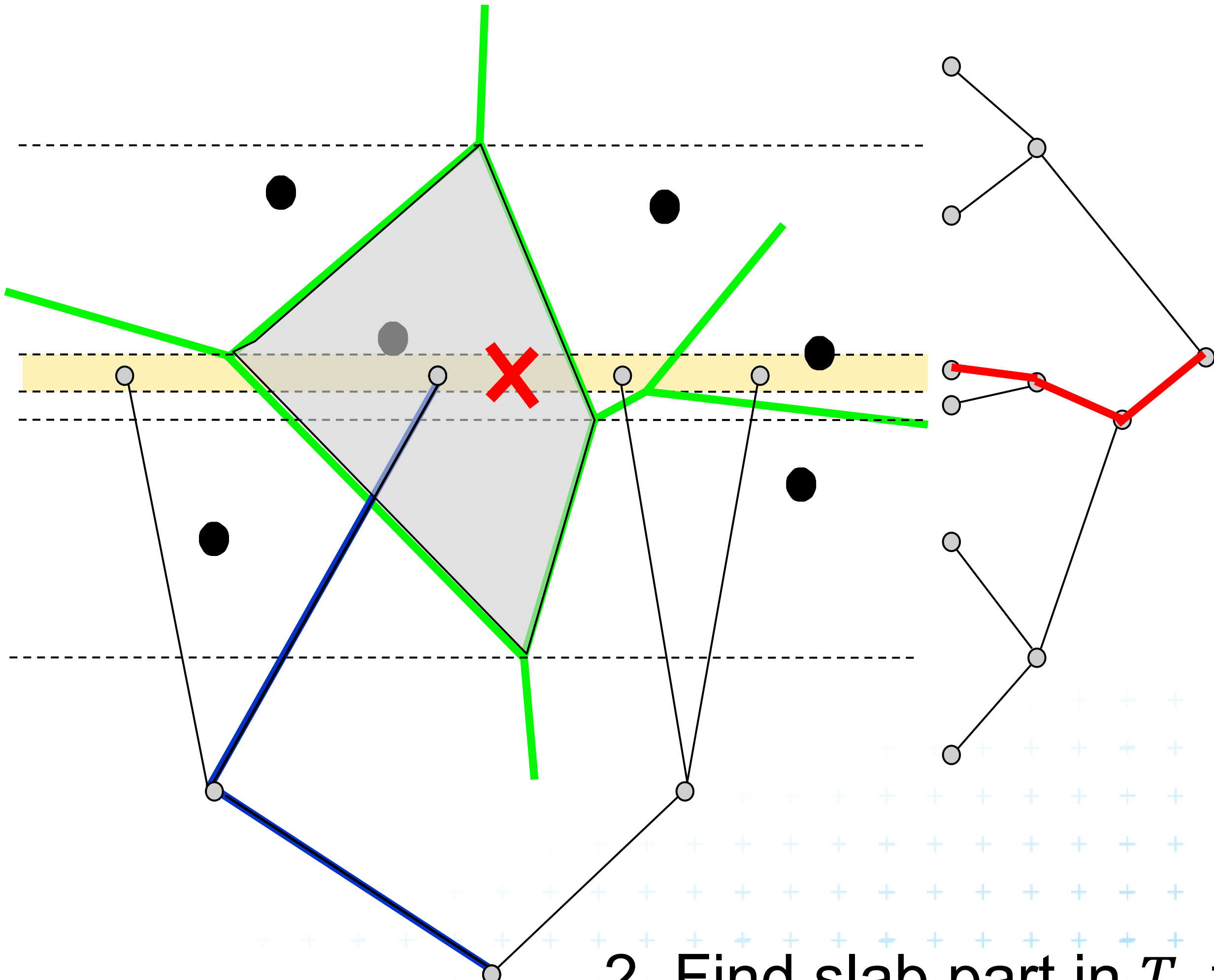
1. Vertical (horizontal) slabs [Dobkin and Lipton, 1976]

- Draw vertical or horizontal lines through vertices
- It partitions the plane into vertical slabs
  - Avoid points with same x coordinate (solved later)



[Berg]

Horizontal slabs example



1. Find slab in T_y for y

T_x and T_y are arrays

2. Find slab part in T_x for x

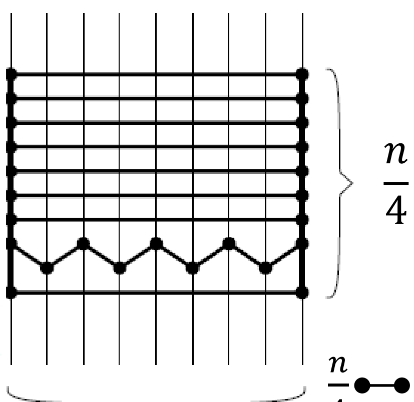
Horizontal slabs complexity

- Query time $O(\log n)$

  $O(\log n)$ time in slab array T_y (size max $2n$ endpoints)

  + $O(\log n)$ time in slab array T_x (slab crossed max by n edges)

- Memory $O(n^2)$
  - Slabs: Array with y-coordinates of vertices … $O(n)$
  - For each slab $O(n)$ edges intersecting the slab



$\frac{n}{4}$ slabs [Berg]

$O(n^2)$ construction

$O(\log n)$ query

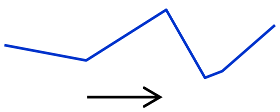
$O(n^2)$ memory

$\frac{n}{4}$ •—• $+ 2\frac{n}{4}$ (vertical edges) $+ \frac{n}{4}$ (diagonal edges) $= O(n)$ edges $\quad \frac{n}{4} * \frac{n}{4} = O(n^2)$ faces

2. Monotone chain tree

[Lee and Preparata, 1977]

- Construct monotone planar subdivision
  - The edges are all monotone in the same direction

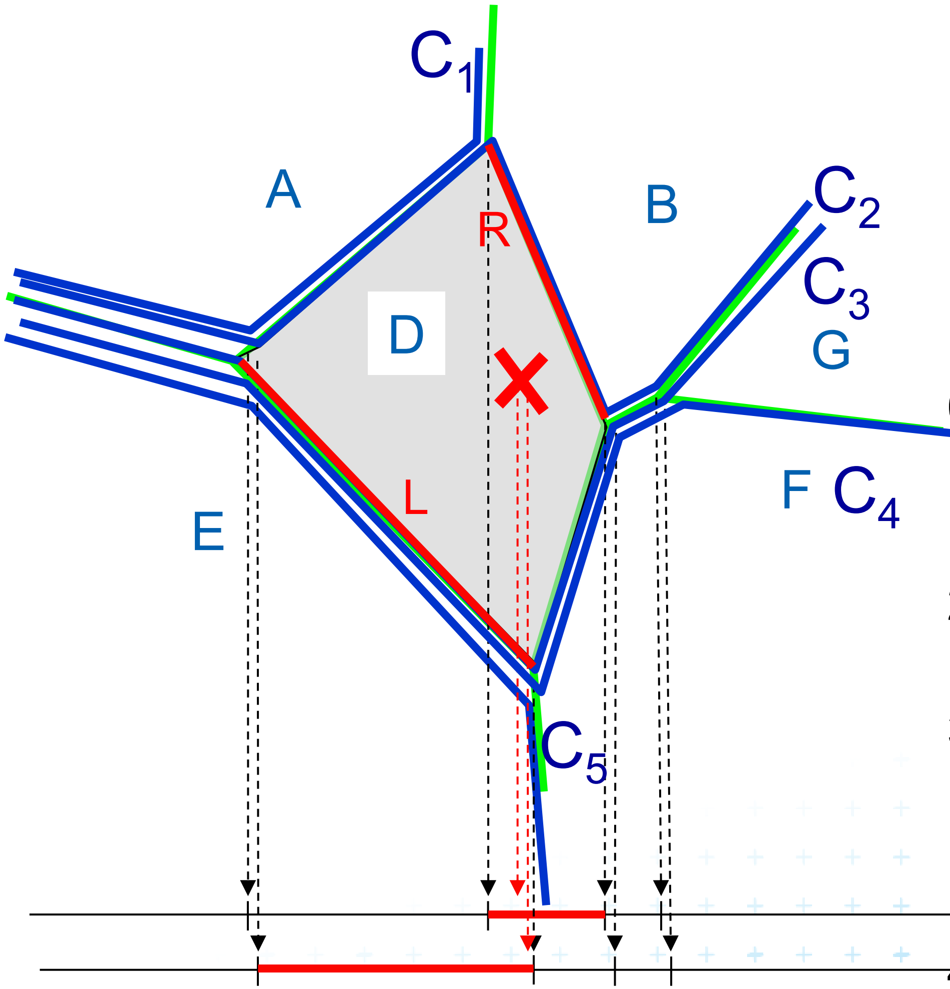


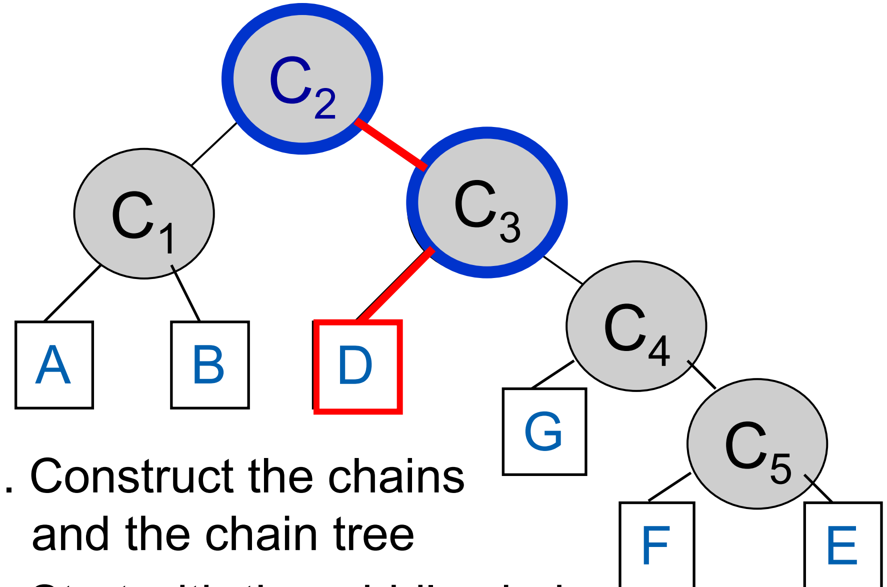
- Each separator chain
  - is monotone (can be projected to line and searched)
  - splits the plane into two parts – allows binary search
- Algorithm
  - Preprocess: Find the separators (e.g., horizontal)
  - Search:
    Binary search among separators (Y) … $O(\log n)$ times
    Binary search along the separator (X) … $O(\log n)$

    $O(\log^2 n)$ query

    $O(n^2)$ memory
  - Not optimal, but simple
  - Can be made optimal, but the algorithm and data structures are complicated

Monotone chain tree example

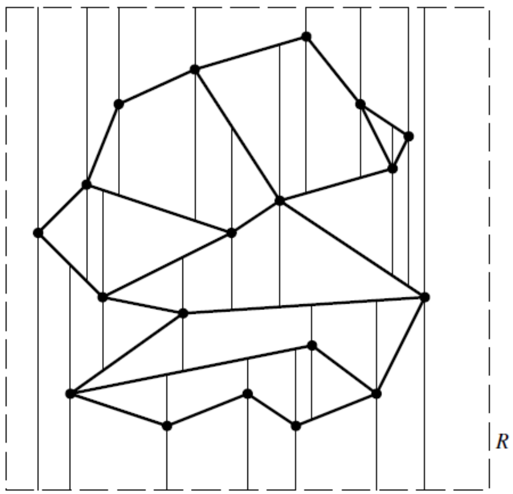




0. Construct the chains and the chain tree
1. Start with the middle chain
2. Find projection of *x* in the projection of the chain – determine the segment
3. Identify position of x in relation to the segment – Left or Right (This is the position of *x* relatively to the whole chain)
4. Continue in L or R chain -> goto 2. or stop if in the leaf

3. Trapezoidal map (TM) search

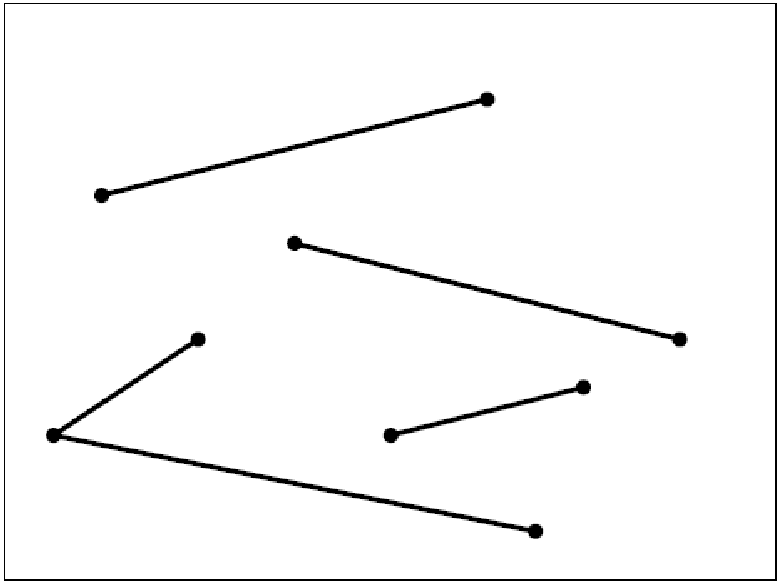
- The simplest and most practical known optimal algorithm
- Randomized algorithm with $O(n)$ expected storage and $O(\log n)$ expected query time
- Expectation depends on the random order of segments during construction, not on the position of the segments
- TM is refinement of original subdivision
- Converts complex shapes into simple ones
- Weaker assumption on input:
  - Input individual segments, not polygons
  - $S = \{s_1, s_2, \dots, s_n\}$
  - S_i subset of first i segments
  - Answer: segment below the pointed trapezoid (Δ)



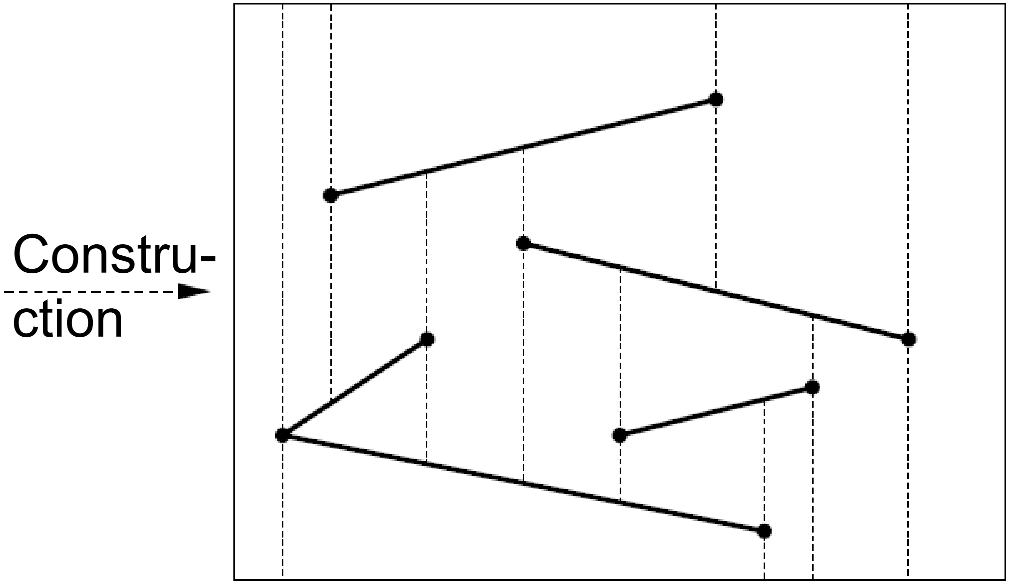
[Berg]

Trapezoidal map of line segments in general position

Input: individual segments *S*



Trapezoidal map *T*



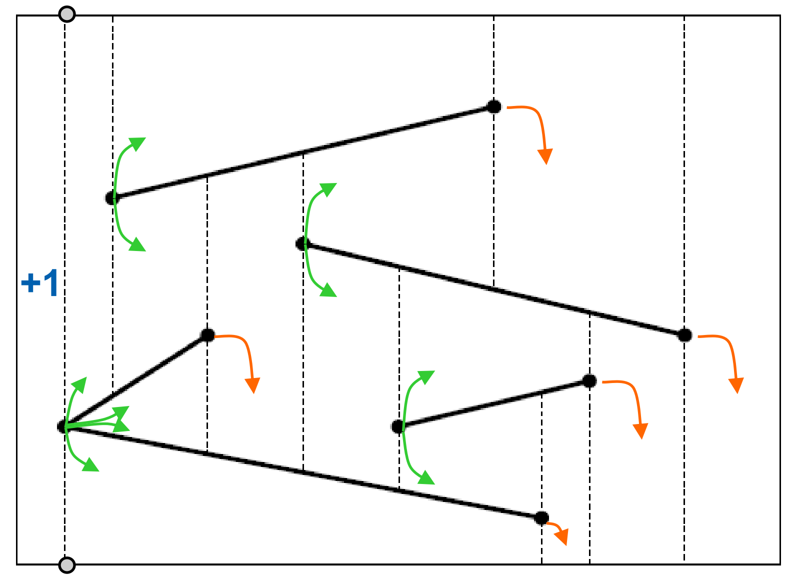
[Mount]

- They do not intersect, except in endpoints
- No vertical segments
- No 2 distinct endpoints with the same x-coordinate

- Bounding rectangle
- 4 Bullets up and down
- Stop on input segment or on bounding rectangle

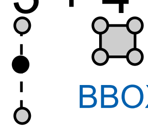
Trapezoidal map of line segments in general position

- Faces are trapezoids Δ with vertical sides
- Given n segments, TM has
  - at most 6n+4 vertices
  - at most 3n+1 trapezoids



[Mount]

- Proof:
  - each endpoint 2 bullets -> 1+2 points
  - 2n endpoints * 3 + 4 = 6n+4 vertices



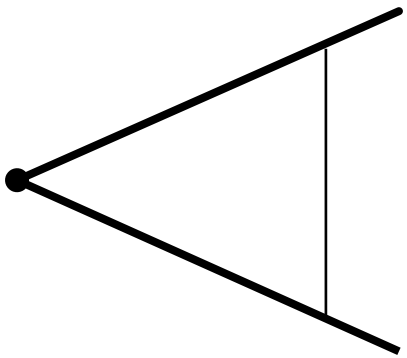
BBOX

  - start point –> max 2 trapezoids Δ
  - end point –> 1 trapezoid Δ
  - 3 * (n segments) + 1 left Δ => max 3n+1 Δ

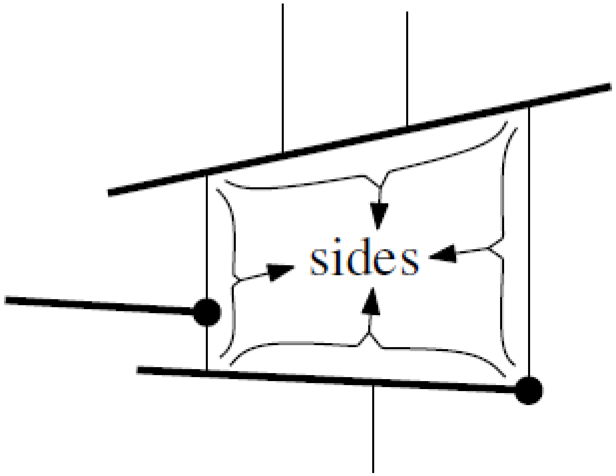
Trapezoidal map of line segments in general position

Each face has

- one or two vertical sides (trapezoid or triangle) and
- exactly two non-vertical sides



One vertical side



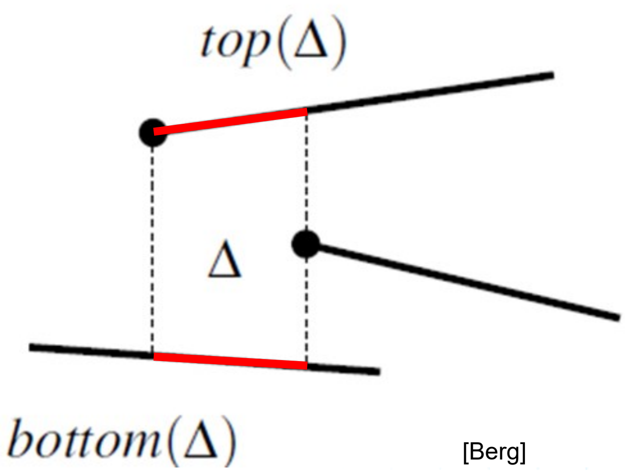
Two vertical sides

[Berg]

Two non-vertical sides

Non-vertical side ‾‾‾ or ___

- is contained in one of the segments of set *S*
- or in the horizontal edge of bounding rectangle *R*



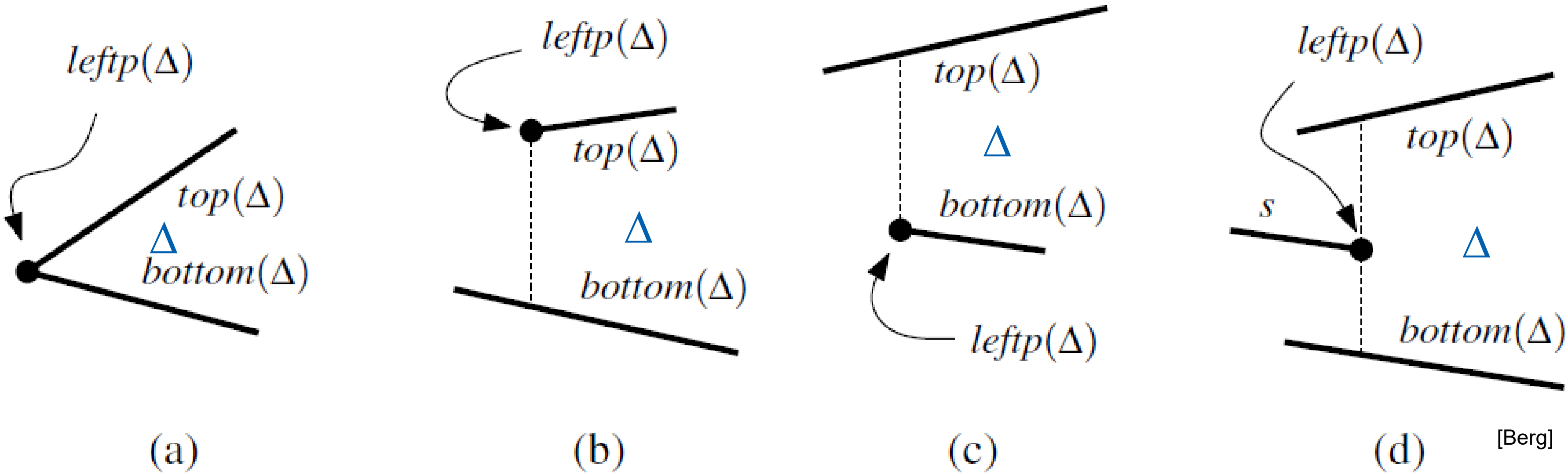
[Berg]

segments:

top(Δ) - bounds from above

bottom(Δ) - bounds from below

Vertical sides – left vertical side of Δ



Left vertical side is defined by the segment end-point p=*leftp*(Δ)

(a) common left point p itself

(b) by the lower vert. extension of left point p ending at bottom()

(c) by the upper vert. extension of left point p ending at top()

(d) by both vert. extensions of the right point p

(e) the left edge of the bounding rectangle R (leftmost Δ only)

Vertical sides - summary

Vertical edges are defined by segment endpoints

- *leftp*(Δ) = the end point defining the left edge of Δ
- *rightp*(Δ) = the end point defining the right edge of Δ

leftp(Δ) is

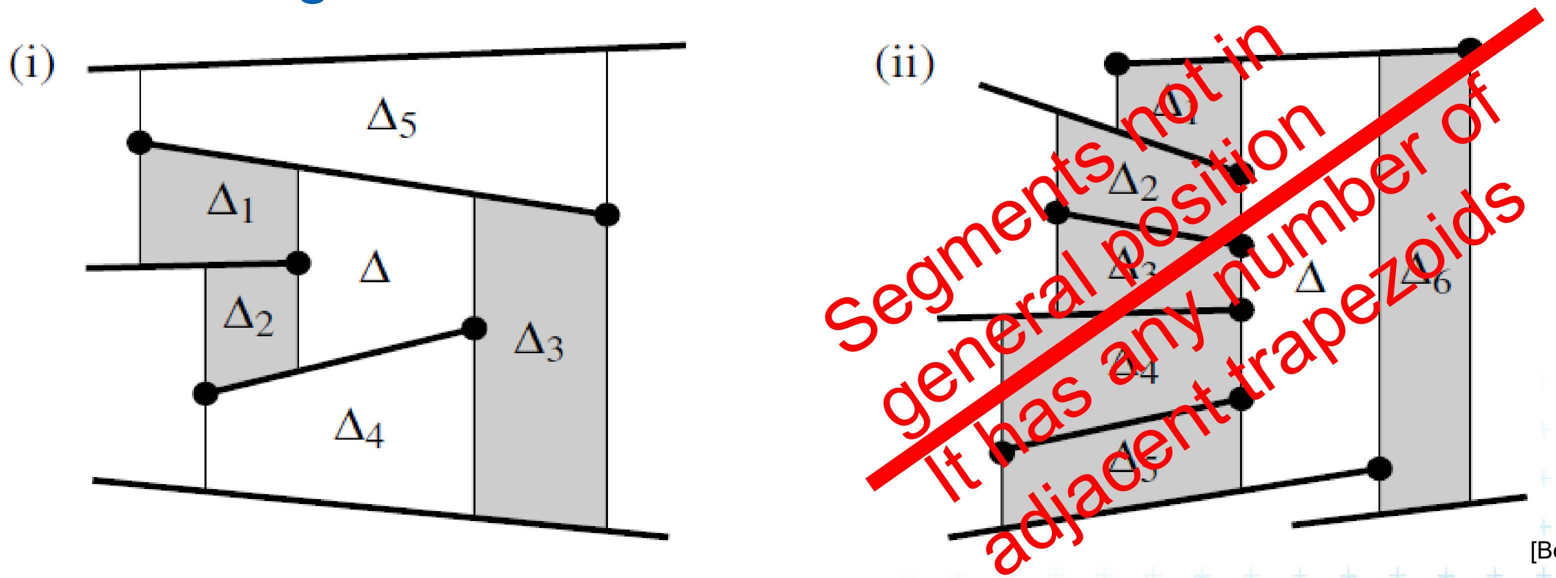
- the left endpoint of *top*() or *bottom*() or both (b, c, a)
- the right point of a third segment (d)
- the lower left corner of the bounding rectangle R (e)

Trapezoid Δ

- Trapezoid Δ is uniquely defined by
  - the segments *top*(Δ), *bottom*(Δ)
  - And by the endpoints *leftp*(Δ), *rightp*(Δ)

Adjacency of trapezoids segments in general position

- Trapezoids Δ and Δ' are adjacent, if they meet along a vertical edge



[Berg]

- Δ_1= upper left neighbor of Δ (common *top*(Δ) edge)
- Δ_2 = lower left neighbor of Δ (common *bottom*(Δ))
- Δ_3 is a right neighbor of Δ (common *top*(Δ) or *bottom*(Δ))

Representation of the trapezoidal map *T*

Special trapezoidal map structure $T(S)$ stores:

- Records for all line segments and end points
- Records for each trapezoid $\Delta \in T(S)$
  - Definition of Δ - pointers to segments *top*(Δ), *bottom*(Δ),
    - pointers to points *leftp*(Δ), *rightp*(Δ)
  - Pointers to its max four neighboring trapezoids (L and R)
  - Pointer to the leaf A in the search structure *D* (see below)

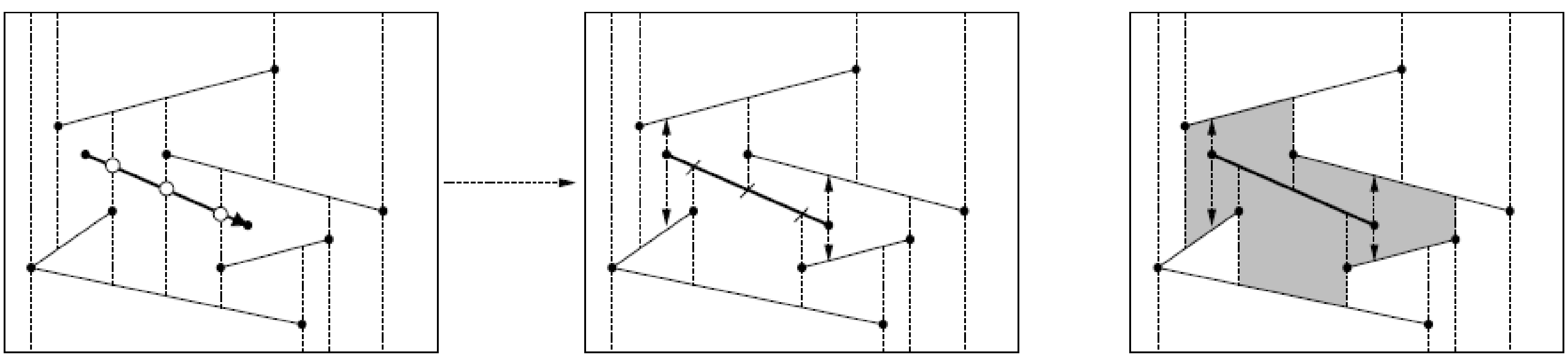
- Does not store the geometry explicitly!
- Geometry of trapezoids is computed in O(1)

Construction of trapezoidal map

- Randomized incremental algorithm
  1. Create the initial bounding rectangle ($T_0 = 1\Delta$) … $O(n)$
  2. Randomize the order of segments in S
  3. for i = 1 to n do
  4. Add segment S_i to trapezoidal map T_i
  5. locate left endpoint of S_i in T_{i-1} $\Rightarrow$ start trapezoid
  6. find intersected trapezoids
  7. shoot 4 bullets from endpoints of S_i
  8. trim intersected vertical bullet paths

  $\Rightarrow$ create new trapezoids

[Mount]



Locate left endpoint and determine intersections

Shoot new bullet paths and trim intersecting rays

Newly created trapezoids

Trapezoidal map point location

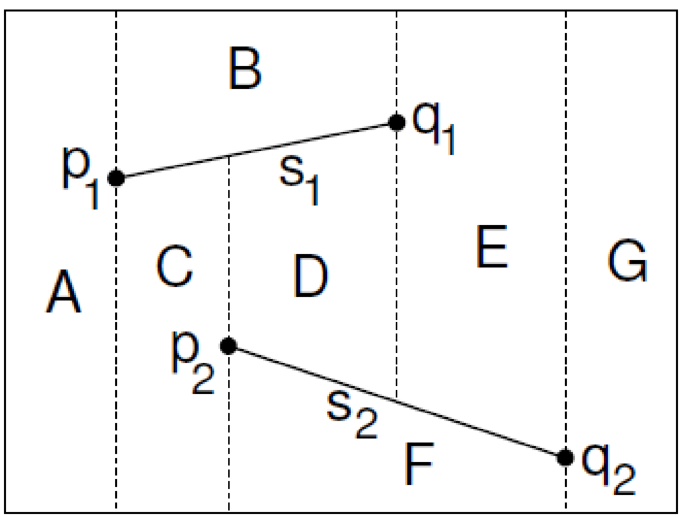
- While creating the trapezoidal map *T* construct the *Point location data structure D*
- Query this data structure

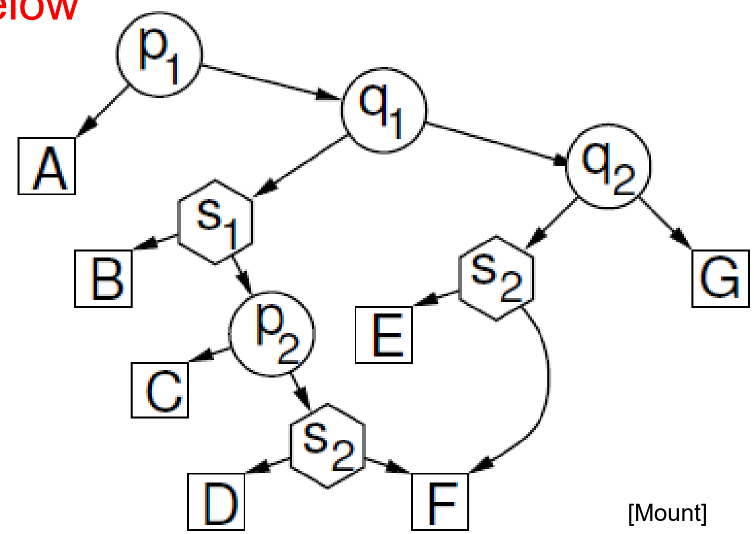
Point location data structure D

- Rooted directed acyclic graph (not a tree!!)
  - Leaves A – trapezoids, each appears exactly once
  - Internal nodes – 2 outgoing edges, guide the search

p_1 x-node – x-coord x_0 of segment start- or end-point
left child lies left of vertical line $x=x_0$
right child lies right of vertical line $x=x_0$
– used first to detect the vertical slab

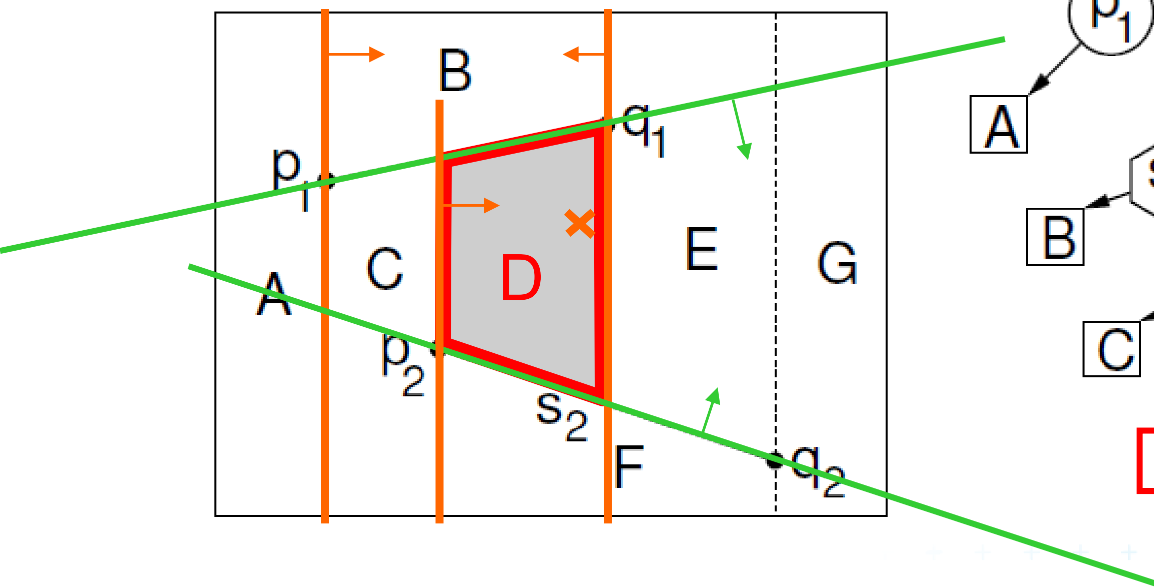
s_1 y-node – pointer to the line segment of the subdivision (not only its y!!!)
left – above, right – below

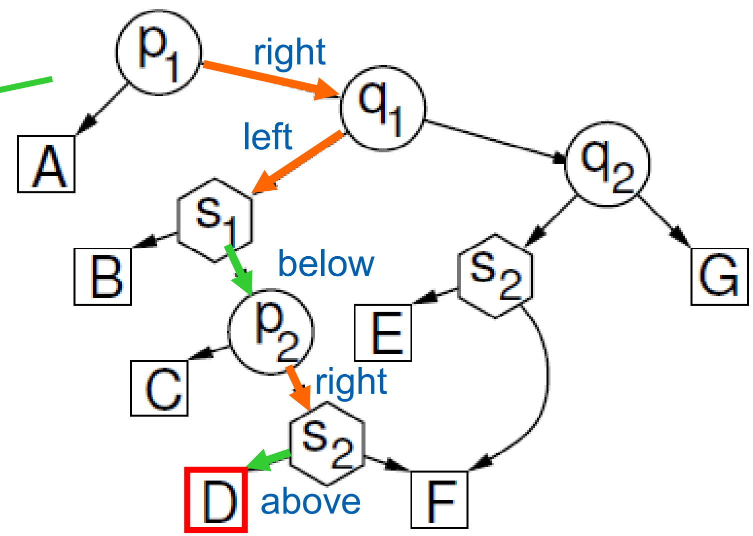




[Mount]

TM search example

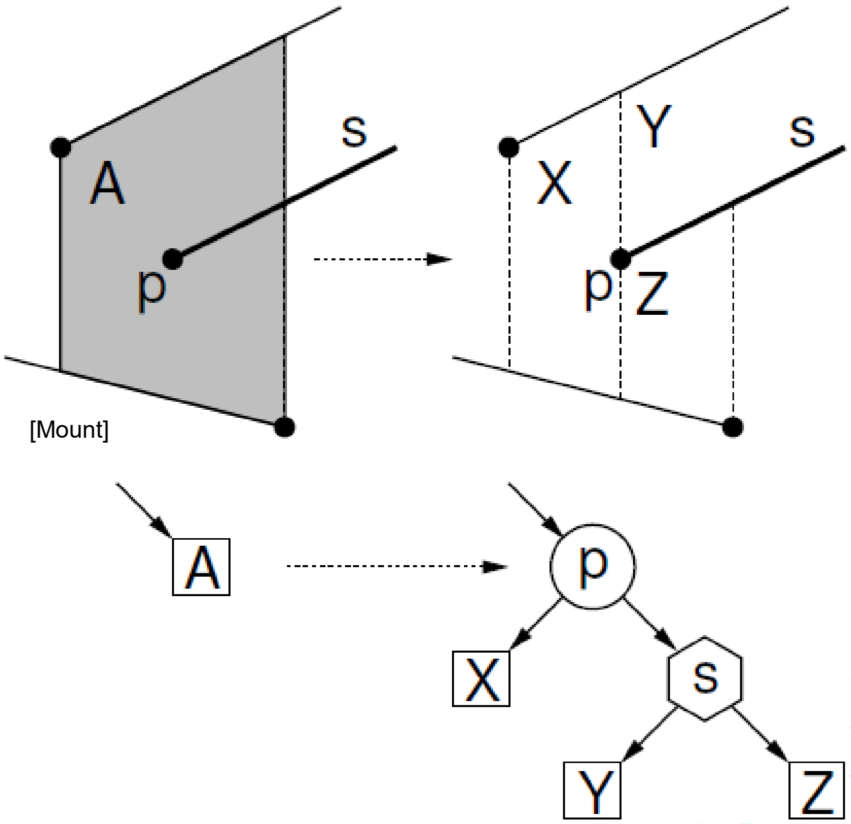




[Mount]

Construction – addition of a segment

a) Single (left or right) endpoint - 3 new trapezoids

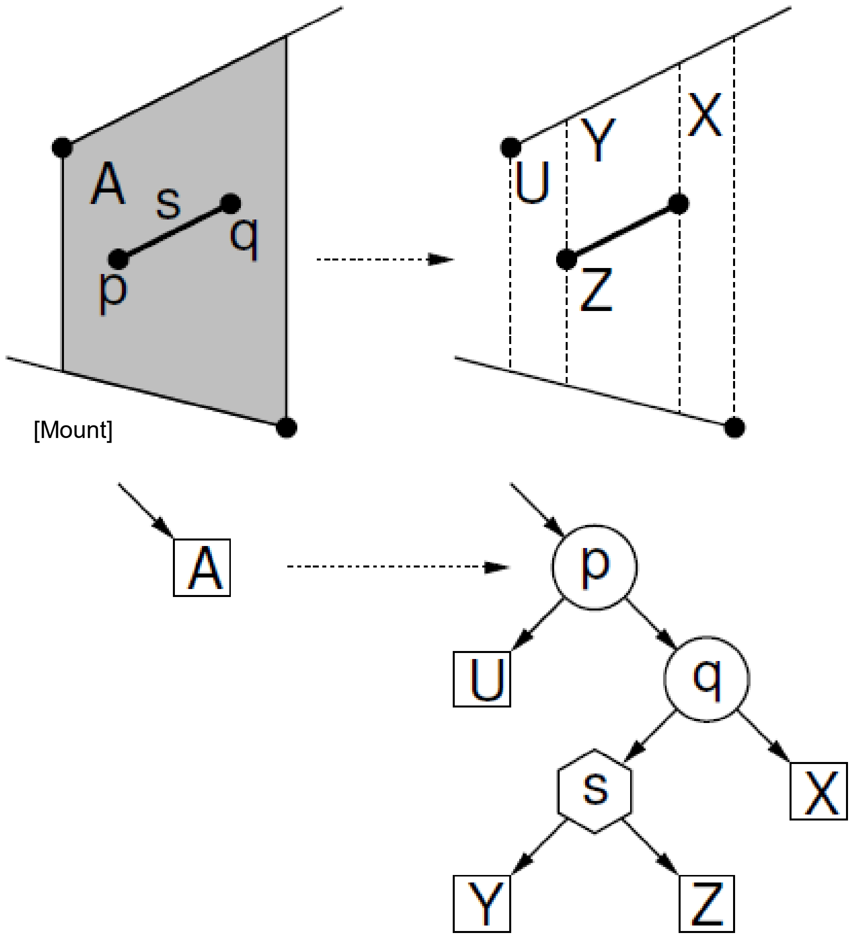


Trapezoid A replaced by

- * x-node for point *p*
- add left leaf for X Δ
- add right subtree
- * y-node for segment *s*
- add left leaf for Y Δ above
- add right leaf Z Δ below

Construction – addition of a segment

b) Two segment endpoints – 4 new trapezoids

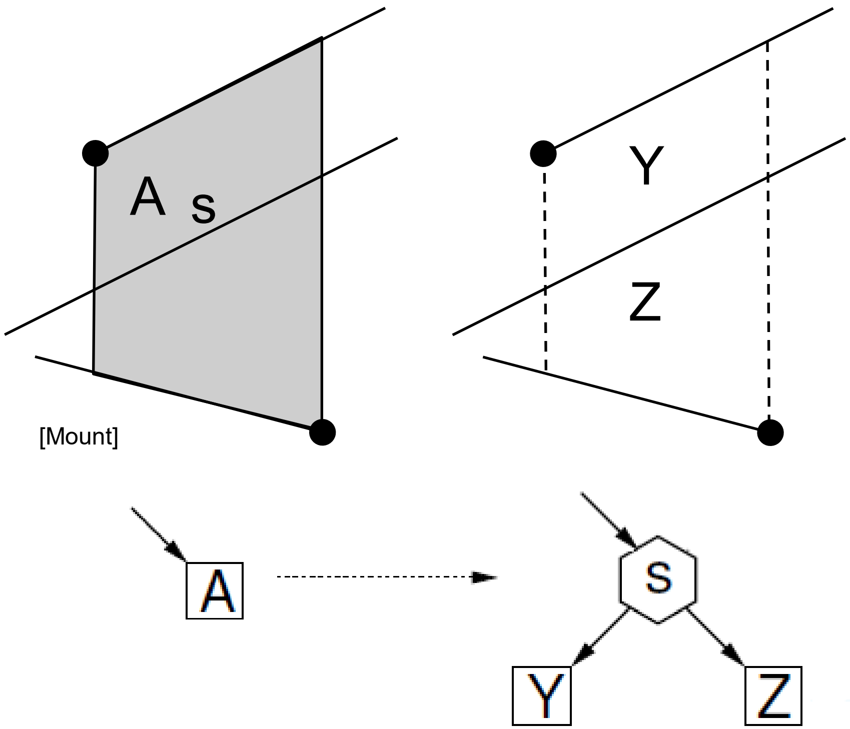


Trapezoid A replaced by

- * x-node for point *p*
- * x-node for point *q*
- * y-node for segment *s*
- add leaves for U, X, Y, Z

Construction – addition of a segment

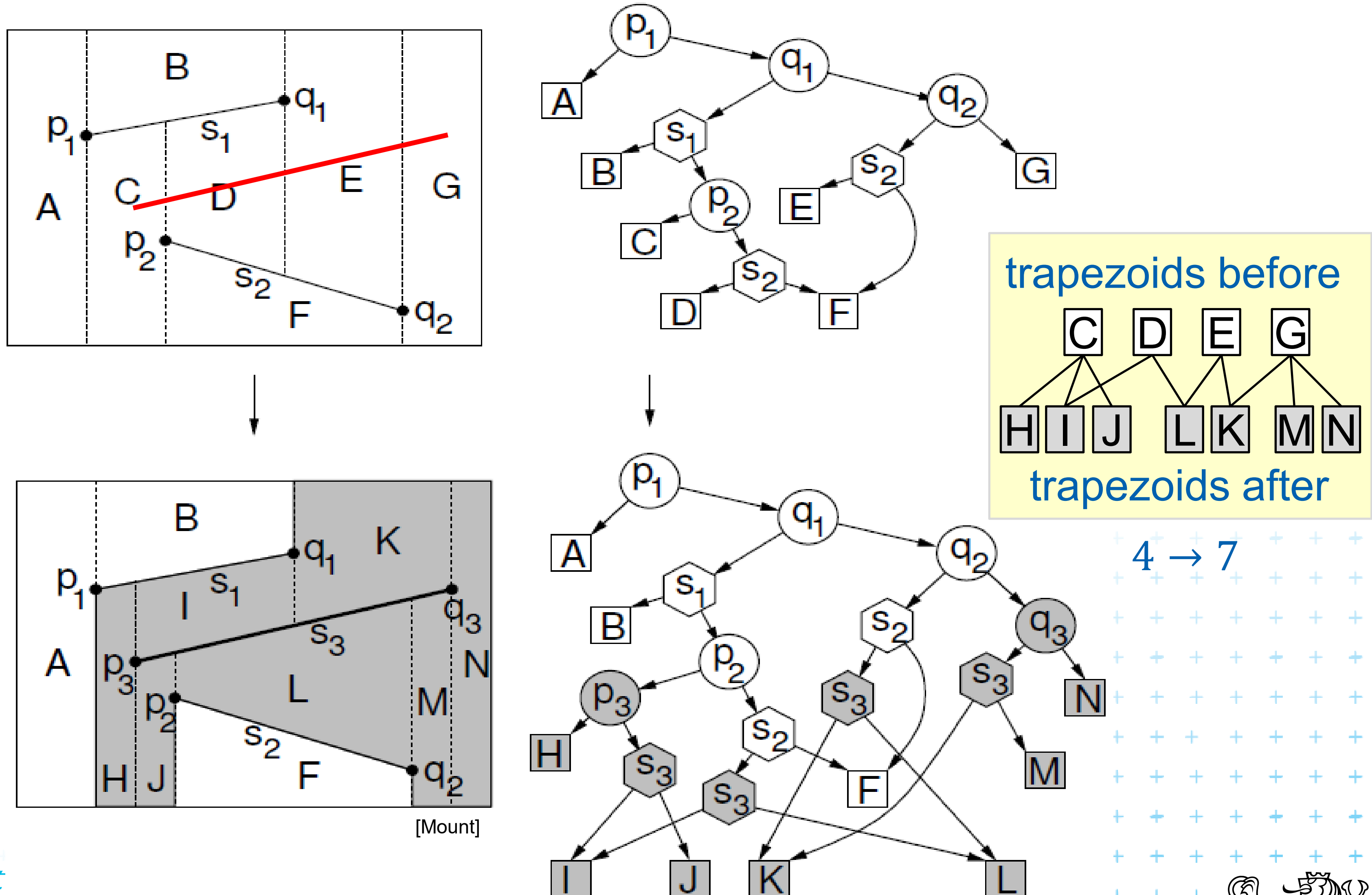
c) No segment endpoint – create 2 trapezoids



Trapezoid A replaced by

- * y-node for segment *s*
- add leaves for Y, Z

Segment insertion example

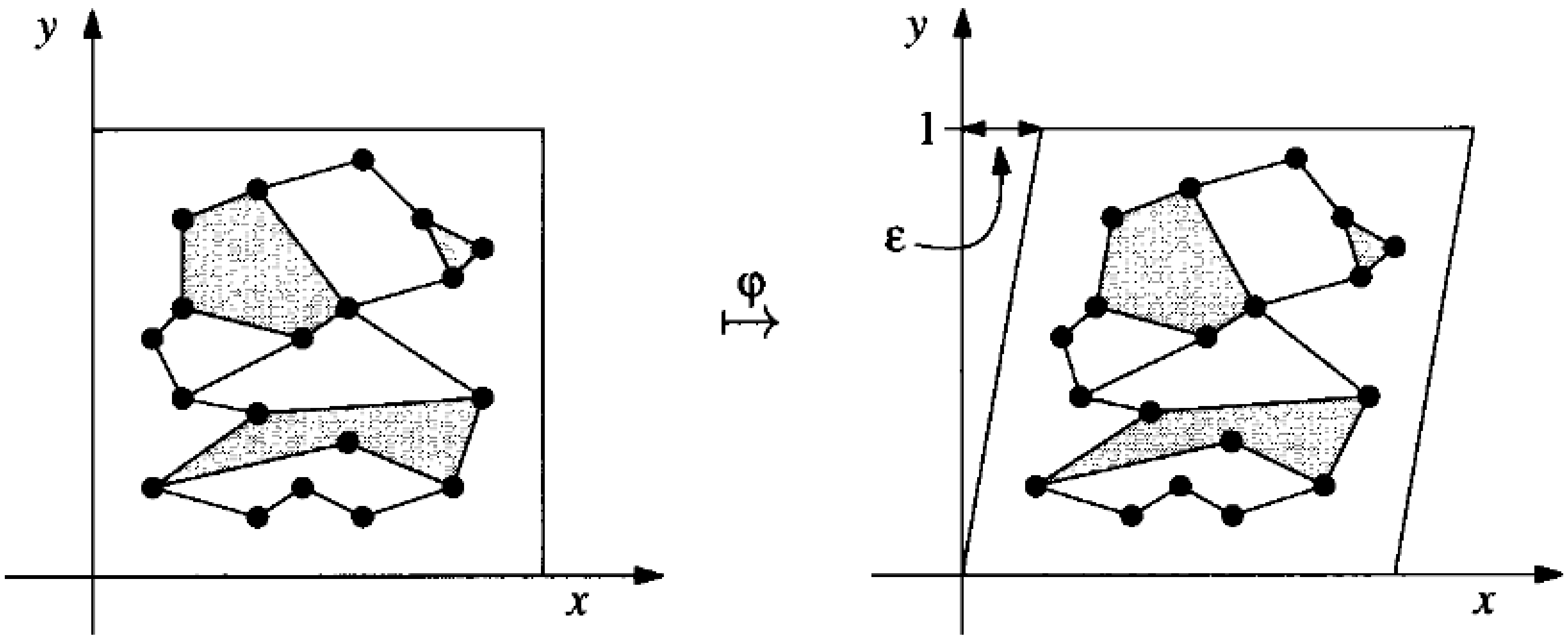


Analysis and proofs

- This holds:
  - Number of newly created Δ for inserted segment $O(1)$ (some added, some removed)
  - Search structure size is max $O(n^2)$, but $O(n)$ expected
  - Search point $O(\log n)$ in average
    => Expected construction $O(n(1 + \log n)) = O(n \log n)$
- For detailed analysis and proofs see
  - [Berg] or [Mount]

Handling of degenerate cases - principle

- No distinct endpoints lie on common vertical line
  - Rotate or shear the coordinates $x' = x + \varepsilon y,\ y' = y$



[Berg]

Handling of degenerate cases - realization

- Trick
  - store original (x, y), not the sheared x', y'
  - we need to perform just 2 operations:

1. For two points p, q determine if transformed point q is to the left, to the right or on vertical line through point p
   - If $x_p = x_q$ then compare y_p and y_q (on only for $y_p = y_q$)
   - => use the original coords (x, y) and **lexicographic order**
2. For segment given by two points decide if 3rd point q lies above, below, or on the segment $p_1\, p_2$
   - Mapping preserves this relation
   - => use the original coords (x, y)

Point location summary

- Slab method [Dobkin and Lipton, 1976]
  - $O(n^2)$ memory $O(\log n)$ time
- Monotone chain tree in planar subdivision [Lee and Preparata,77]
  - $O(n^2)$ memory $O(\log^2 n)$ time
- Layered directed acyclic graph (Layered DAG) in planar subdivision [Chazelle , Guibas, 1986] [Edelsbrunner, Guibas, and Stolfi, 1986]
  - $O(n)$ memory $O(\log n)$ time => optimal algorithm of planar subdivision search (optimal but complex alg. => see elsewhere)
- Trapezoidal map
  - $O(n)$ expected memory $O(\log n)$ expected time
  - $O(n \log n)$ expected preprocessing (simple alg.)

References

- [Berg] Mark de Berg, Otfried Cheong, Marc van Kreveld, Mark Overmars: Computational Geometry: *Algorithms and Applications*, Springer-Verlag, 3rd rev. ed. 2008. 386 pages, 370 fig. ISBN: 978-3-540-77973-5 http://www.cs.uu.nl/geobook/
- [Mount] Mount, D.: *Computational Geometry Lecture Notes for Fall 2016*, University of Maryland, Lectures 9, 10 http://www.cs.umd.edu/class/fall2016/cmsc754/Lects/cmsc754-fall16-lects.pdf