



**DCGI**

DEPARTMENT OF COMPUTER GRAPHICS AND INTERACTION

# VORONOI DIAGRAM

**PETR FELKEL**

FEL CTU PRAGUE

felkel@fel.cvut.cz

<https://cw.fel.cvut.cz/wiki/courses/cg/>

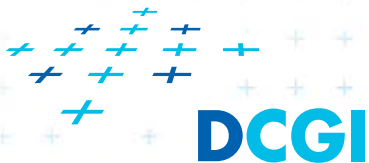
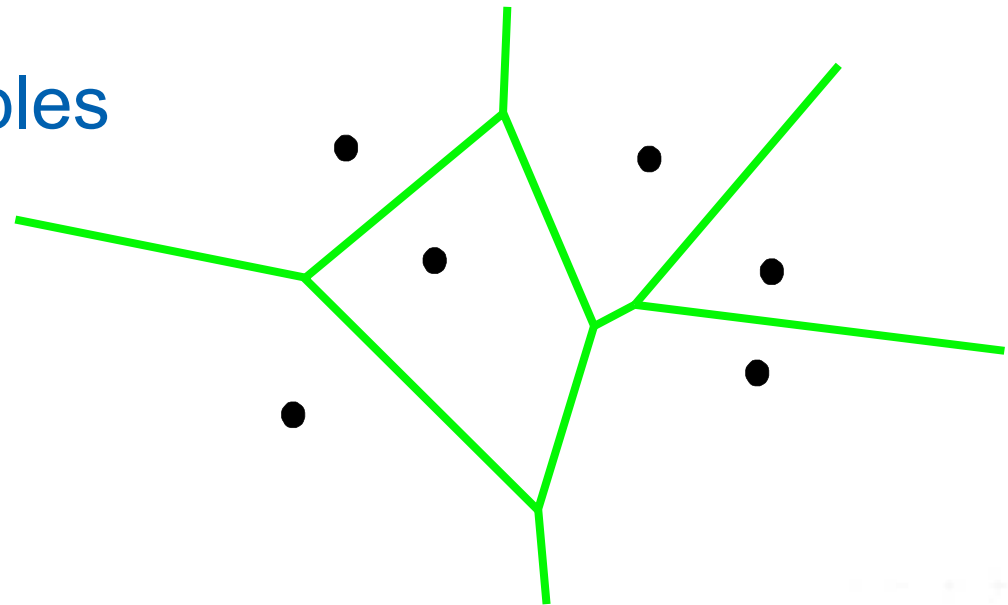
Based on [Berg] and [Mount]

Version from 8.11.2018

# Talk overview

---

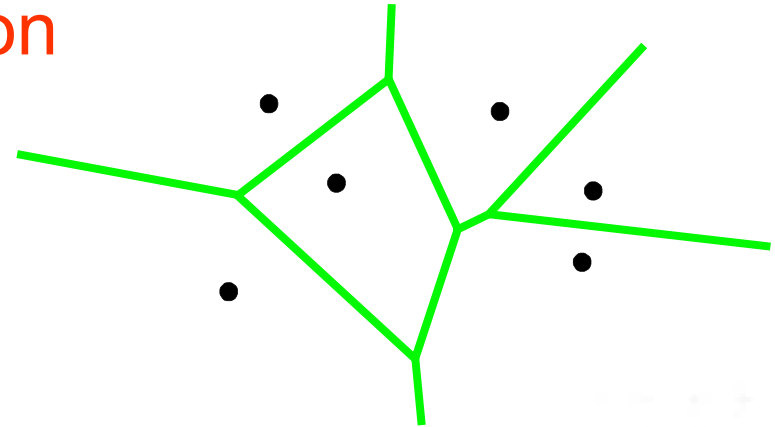
- Definition and examples
- Applications
- Algorithms in 2D
  - D&C  $O(n \log n)$
  - Sweep line  $O(n \log n)$



# Voronoi diagram (VD)

---

- One of the most important structure in Comp. geom.
- Encodes **proximity information**  
What is close to what?
- Standard VD – this lecture
  - Set of points - nDim
  - Euclidean space & metric
- Generalizations
  - Set of line segments or curves
  - Different metrics
  - Higher order VD's (furthest point)



# Voronoi cell (for points in plane)

---

- Let  $P = \{p_1, p_2, \dots, p_n\}$  be a set of points (*sites*) in dDim space ... 2D space (plane) here

- **Voronoi cell**  $V(p_i)$  – is open!  
= set of points  $q$  closer to  $p_i$  than to any other site:

$$V(p_i) = \{q, \|p_i q\| < \|p_j q\|, \forall j \neq i\}, \text{ where}$$

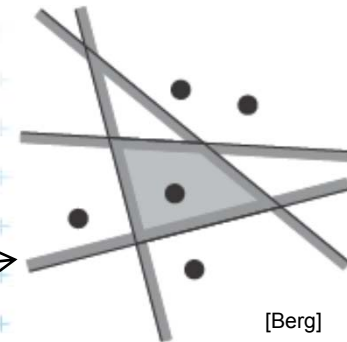
$\|pq\|$  is the Euclidean distance between  $p$  and  $q$

= intersection of open halfplanes

$$V(p_i) = \bigcap_{j \neq i} h(p_i, p_j)$$

$h(p_i, p_j)$  = open halfplane

= set of pts strictly closer to  $p_i$  than to  $p_j$



[Berg]



# Voronoi diagram (in plane)

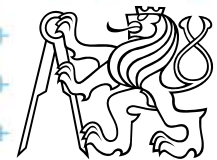
---

- **Voronoi diagram**  $\text{Vor}(P)$  of points  $P$ 
  - = what is left of the plane after removing all the open Voronoi cells
  - = collection of line segments (possibly unbounded)

Site (given point)



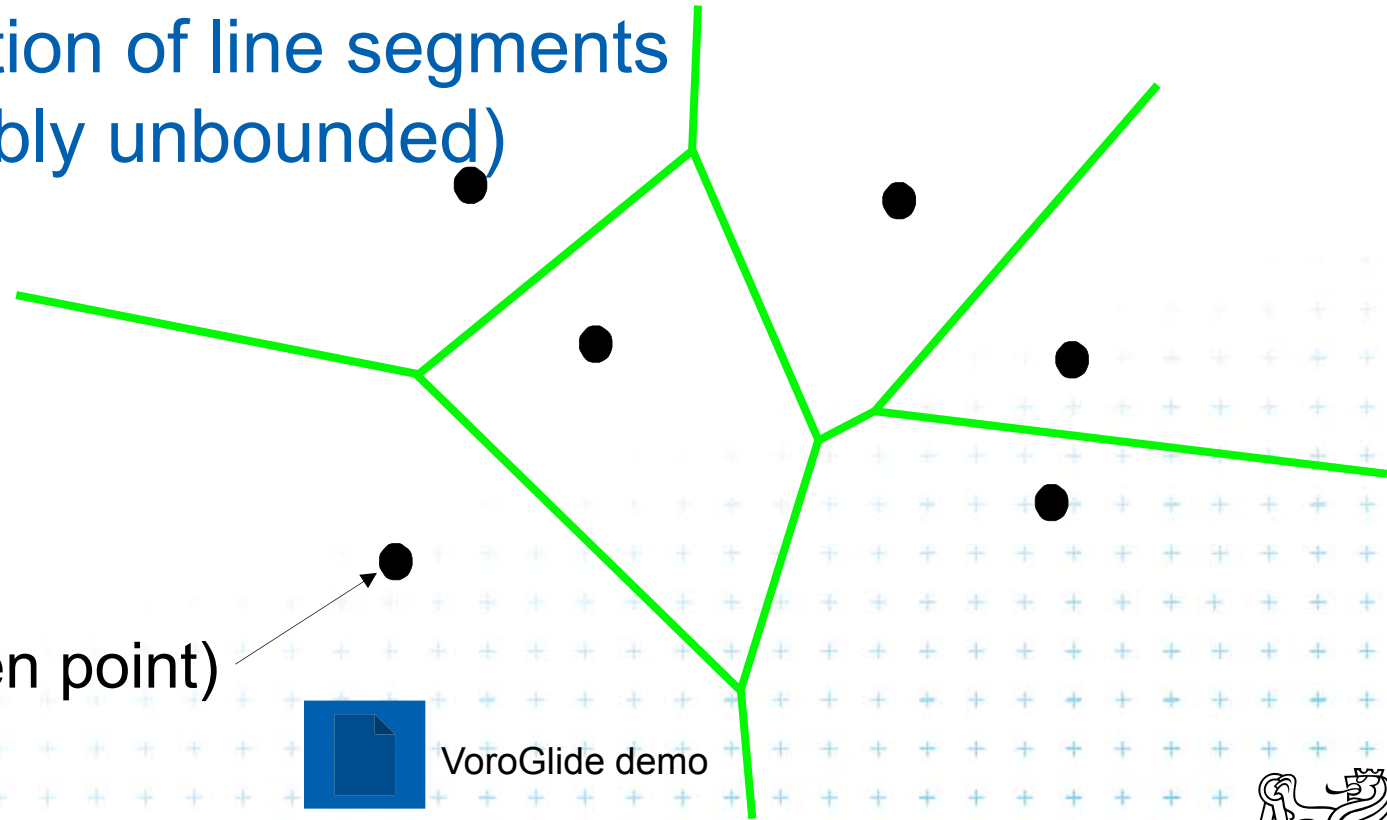
VoroGlide demo



# Voronoi diagram (in plane)

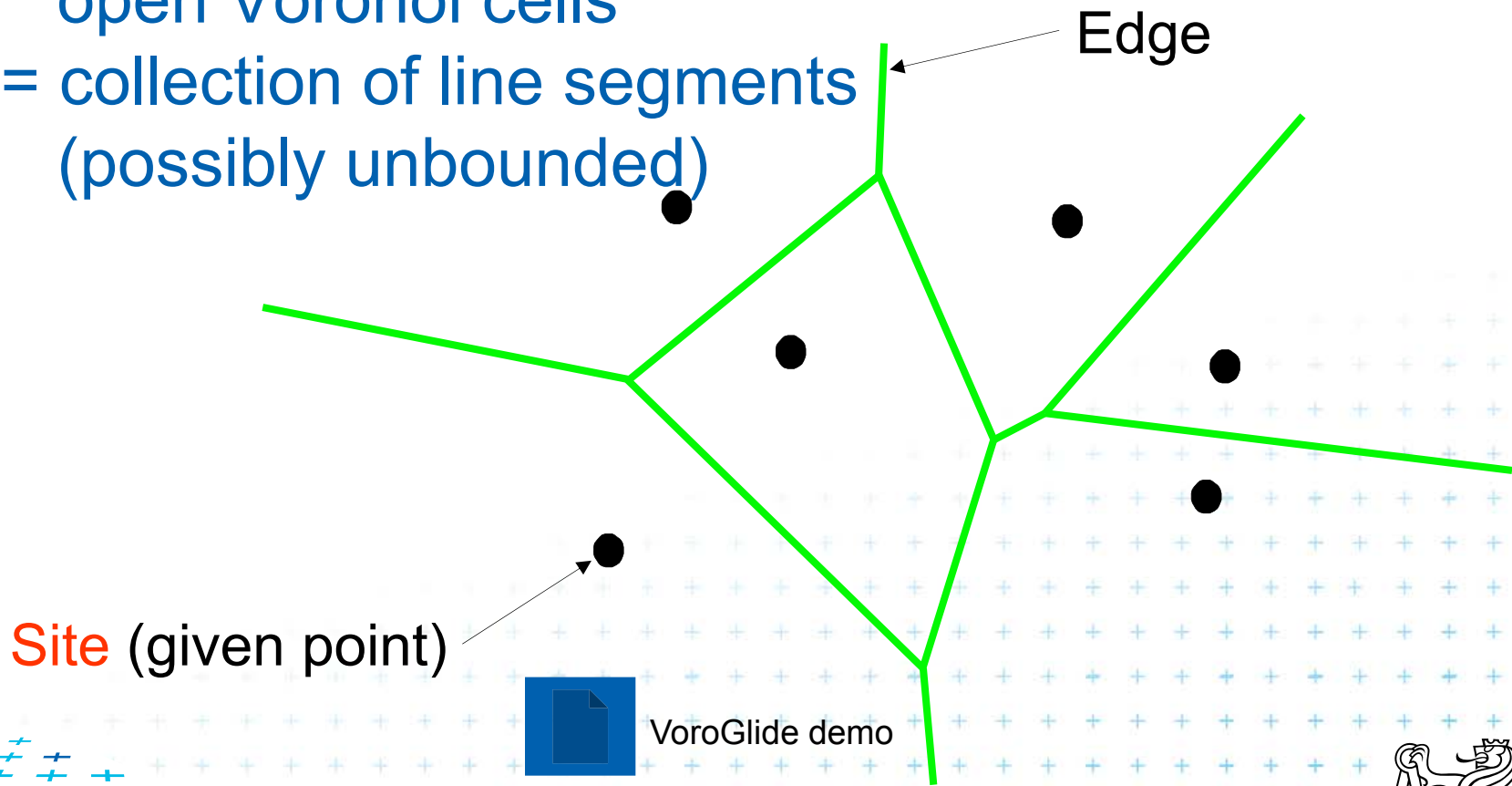
---

- **Voronoi diagram**  $\text{Vor}(P)$  of points  $P$   
= what is left of the plane after removing all the open Voronoi cells  
= collection of line segments (possibly unbounded)



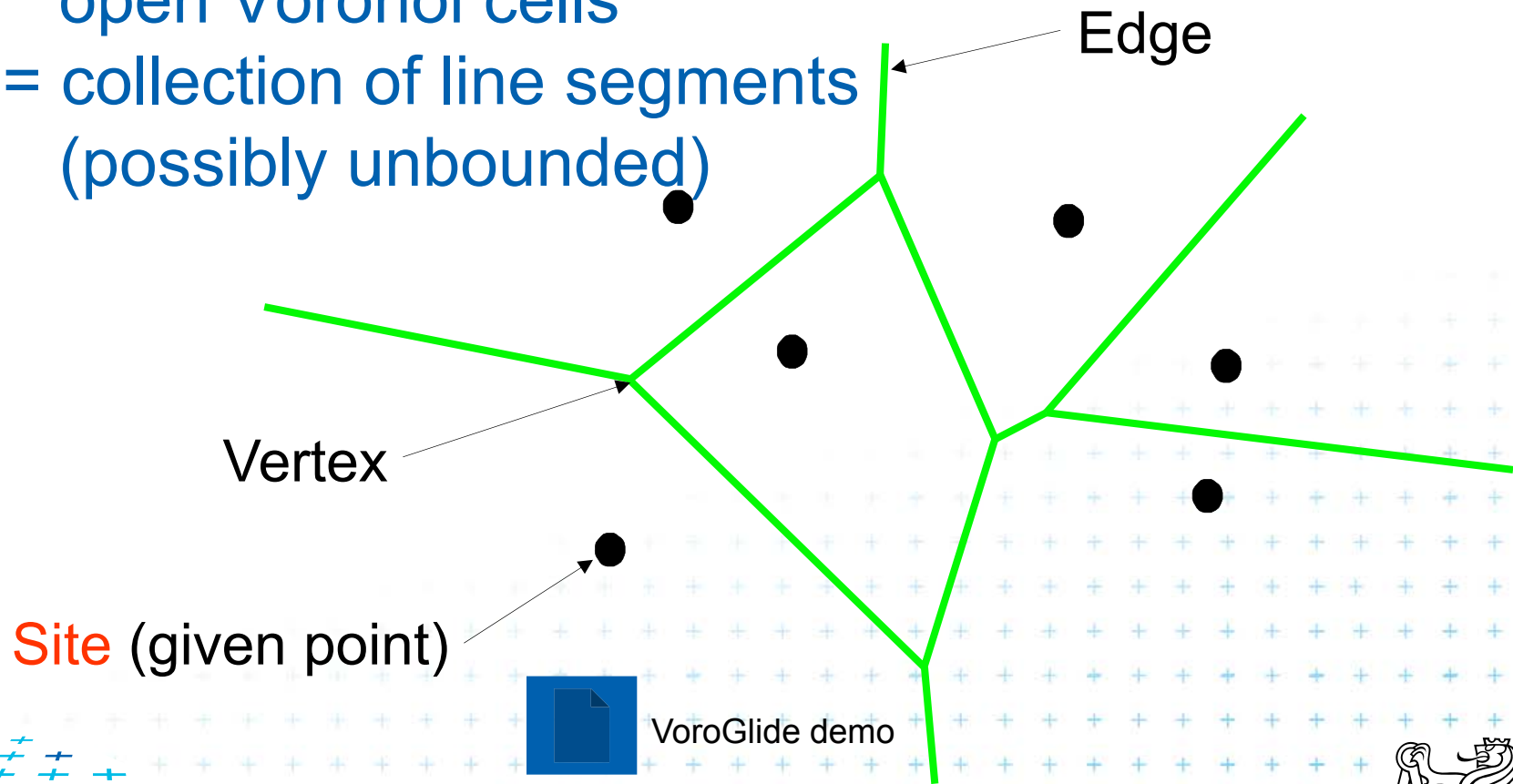
# Voronoi diagram (in plane)

- **Voronoi diagram**  $\text{Vor}(P)$  of points  $P$   
= what is left of the plane after removing all the open Voronoi cells  
= collection of line segments (possibly unbounded)



# Voronoi diagram (in plane)

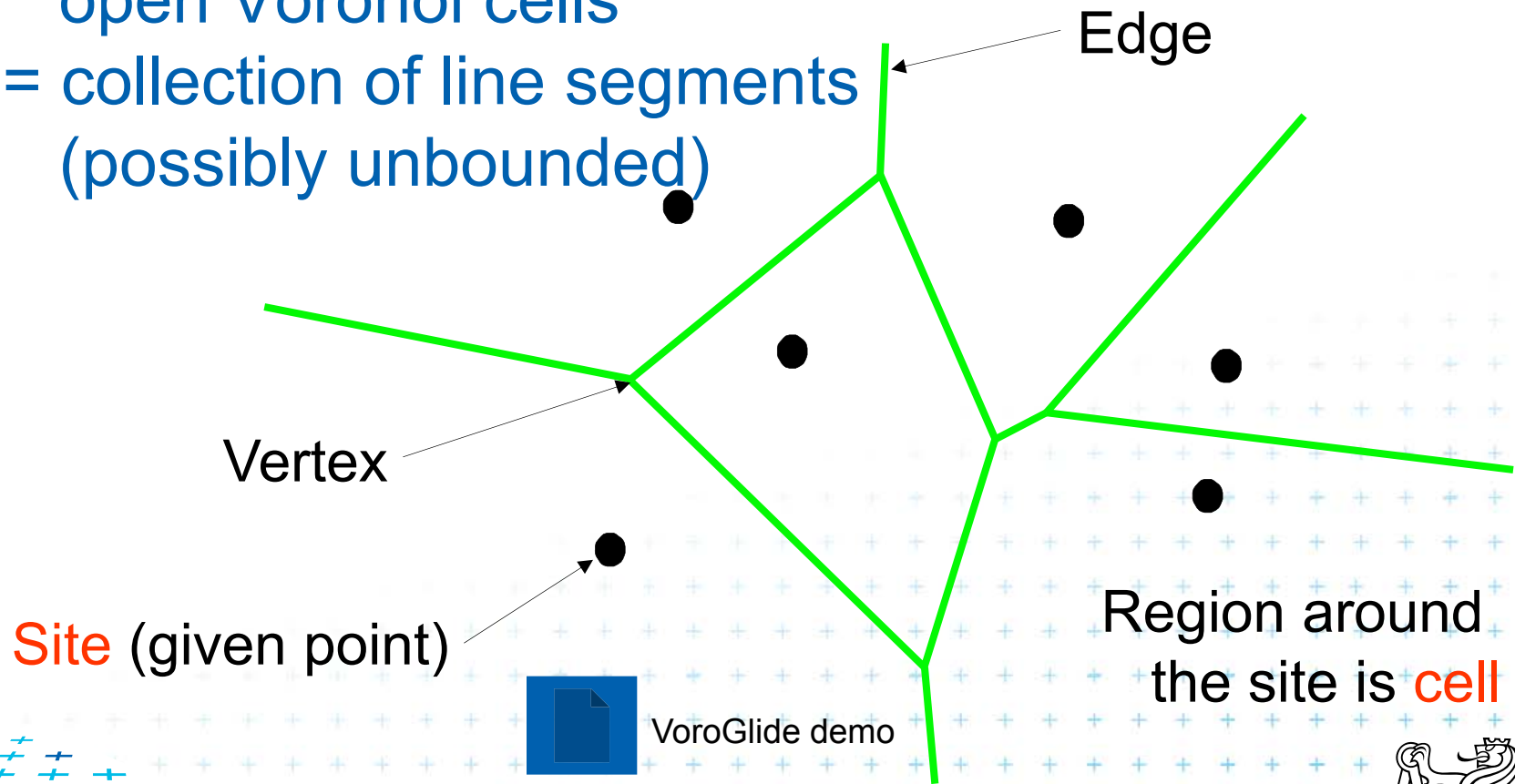
- **Voronoi diagram**  $\text{Vor}(P)$  of points  $P$ 
  - = what is left of the plane after removing all the open Voronoi cells
  - = collection of line segments (possibly unbounded)





# Voronoi diagram (in plane)

- **Voronoi diagram**  $\text{Vor}(P)$  of points  $P$ 
  - = what is left of the plane after removing all the open Voronoi cells
  - = collection of line segments (possibly unbounded)



# Voronoi diagram examples

---

1 point



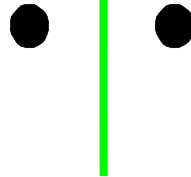
# Voronoi diagram examples

---

1 point



2 points



# Voronoi diagram examples

---

1 point



2 points



3 points



# Voronoi diagram examples

---

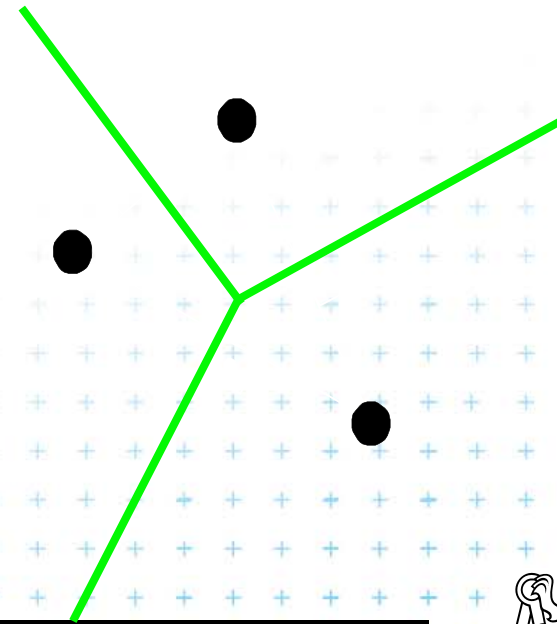
1 point



2 points



3 points



# Voronoi diagram examples

---

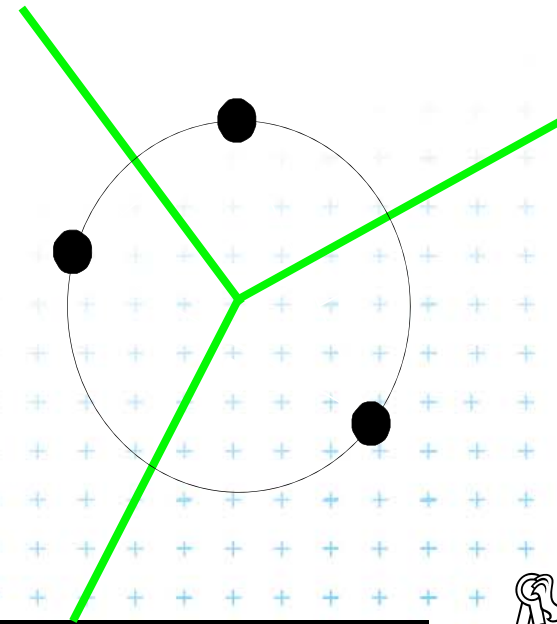
1 point



2 points



3 points



# Voronoi diagram examples

---

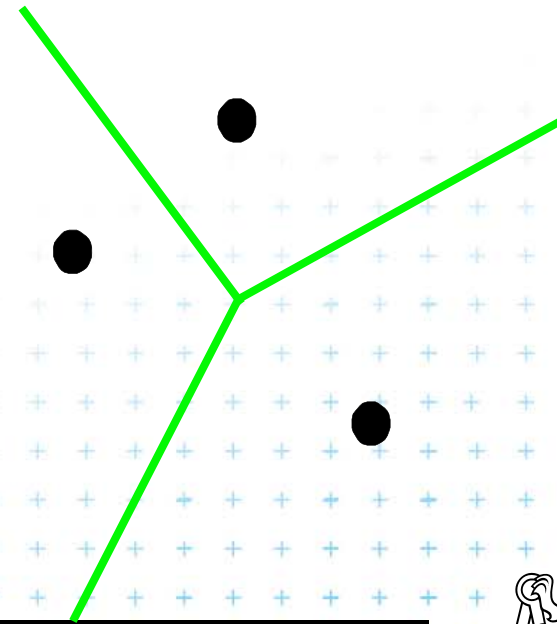
1 point



2 points



3 points



# Voronoi diagram examples

1 point



2 points



3 points

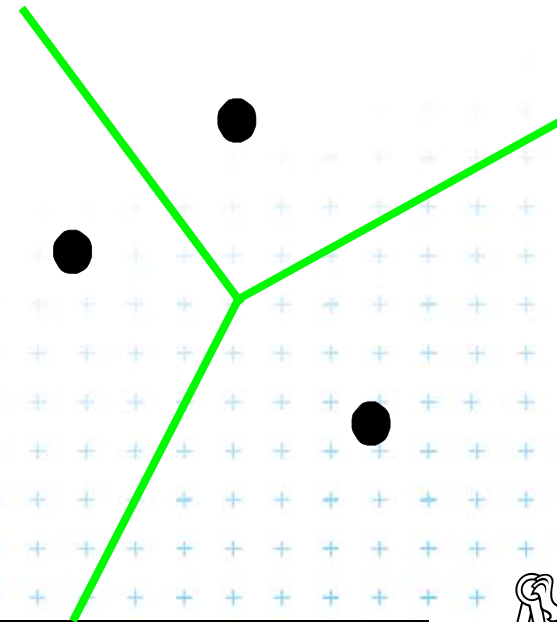


## Cell

- The whole **plain** for 1 point
- **Halfplane** or **strip** for collinear points
- **Convex** (possibly unbounded) polygon

## Edges of VD

- **|| lines** for collinear points
- **Halflines** (for non-collinear CH points)
- **Line segments** (for bounded cells)

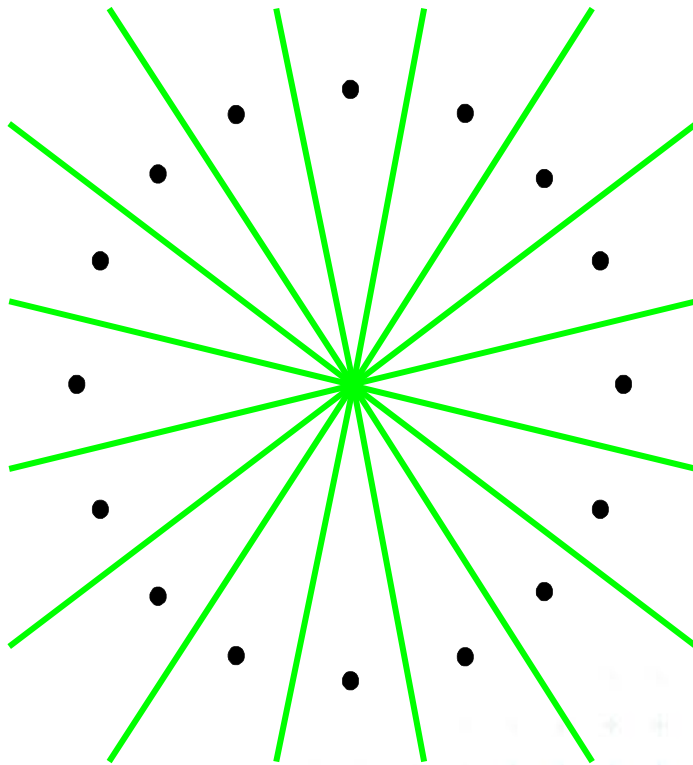




# Voronoi diagram examples

---

16 points



Vertex with  $O(n)$  incident edges

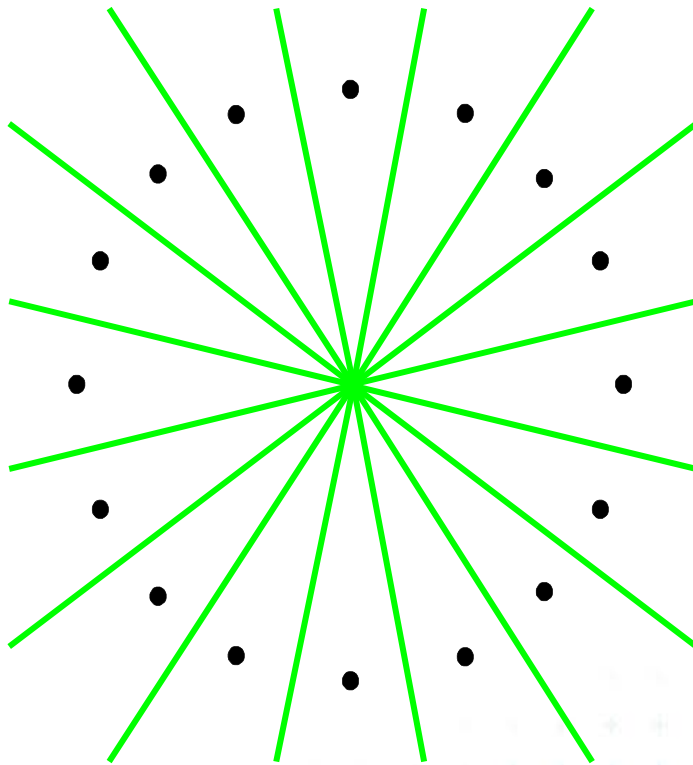
From total  $|n_e| \leq 3n - 6$

[Håkan Jonsson]



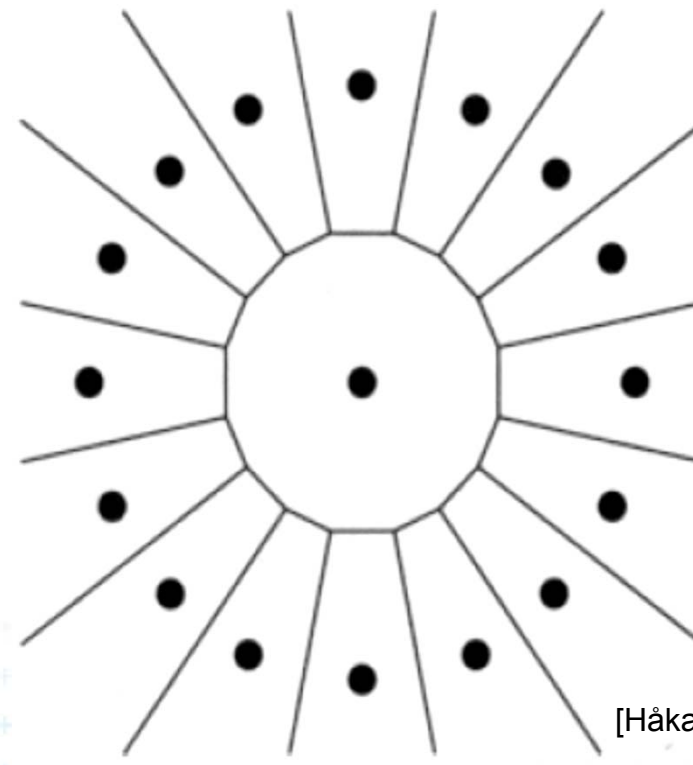
# Voronoi diagram examples

16 points



Vertex with  $O(n)$  incident edges  
From total  $|n_e| \leq 3n - 6$

17 points



Cell with  $O(n)$  vertices  
From total  $|n_v| \leq 2n - 5$

[Håkan Jonsson]



16 <= 42

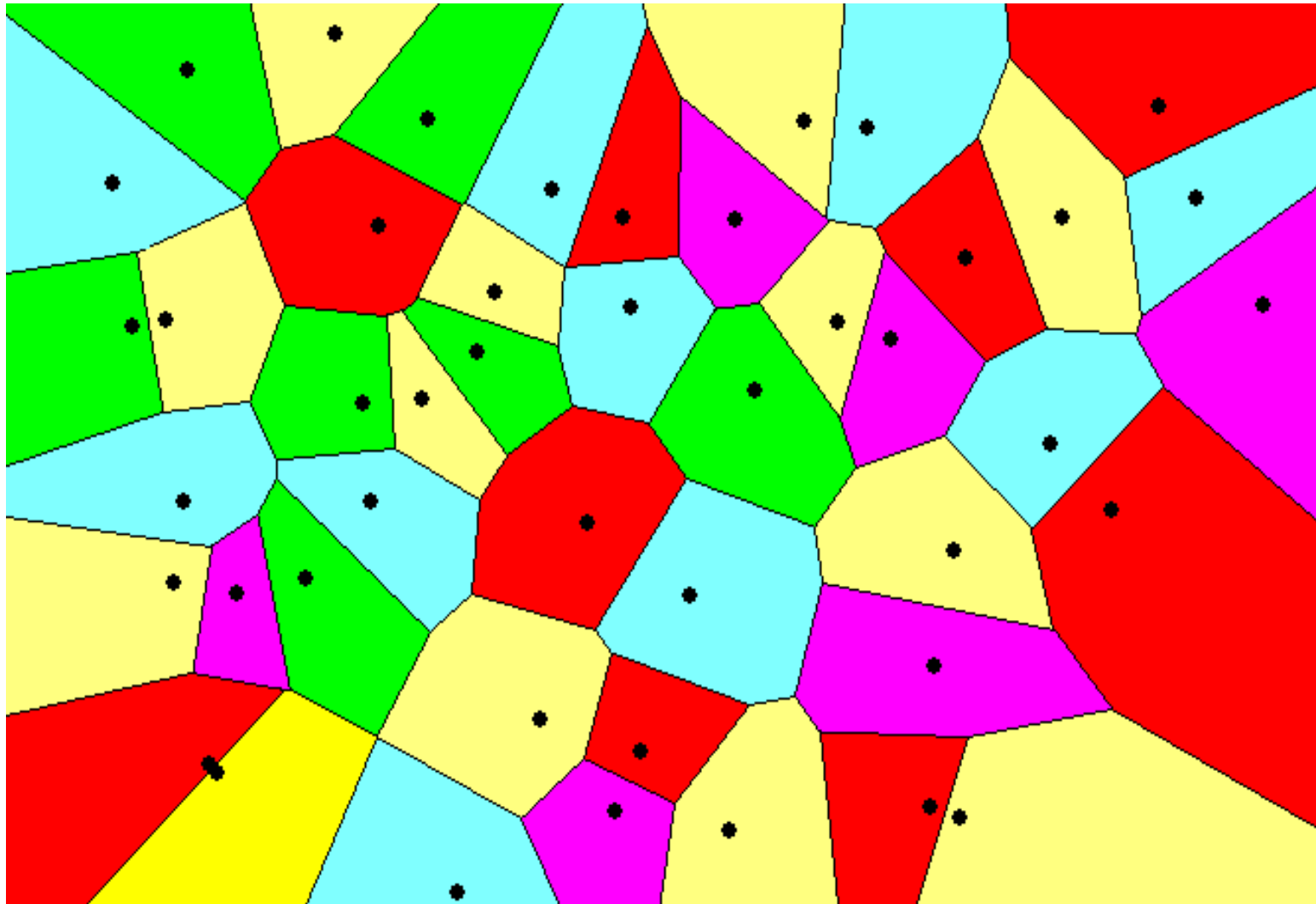
Felkel: Computational geometry

17 <= 29



# Voronoi diagram examples

---



# Voronoi diagram (in plane)

---

= planar graph

- Subdivides plane into  $n$  cells ( $n = \text{num. of input sites } |P|$ )
- Edge = locus of equidistant pairs of points (cells)  
= part of the bisector of these points
- Vertex = center of the circle defined by  $\geq 3$  points  
=> vertices have degree  $\geq 3$
- Number of vertices  $n_v \leq 2n - 5 \Rightarrow O(n)$
- Number of edges  $n_e \leq 3n - 6 \Rightarrow O(n)$   
(only  $O(n)$  from  $O(n^2)$  intersections of bisectors)
- In higher dimensions complexity from  $O(n)$  up to  $O(n^{\lfloor d/2 \rfloor})$
- Unbounded cells belong to sites (points) on convex hull



# Voronoi diagram $O(n)$ complexity derivation

••|• For  $n$  collinear sites:  $n_v = 0 \leq 2n - 5$  both hold  
 $n_e = (n - 1) \leq 3n - 6$

•• For  $n$  non-collinear sites:

- Add extra VD vertex  $v$  in infinity  $m_v = n_n + 1$
- Apply Euler's formula:  $m_v - m_e + m_f = 2$
- Obtain  $(n_v + 1) - n_e + n = 2$   $\left\{ \begin{array}{l} n_e = n_v + n - 1 \\ n_v = n_e - n + 1 \end{array} \right.$
- Every VD edge has 2 vertices Sum of vertex degrees =  $2n_e$
- Every VD vertex has degree  $\geq 3$  Sum of vertex degrees =  $3m_v = 3(n_v + 1)$
- Together  $2n_e \geq 3(n_v + 1)$

$$\begin{aligned} 2n_e &\geq 3(n_v + 1) \\ 2(n_v + n - 1) &\geq 3(n_v + 1) \\ 2n_v + 2n - 2 &\geq 3n_v + 3 \\ n_v &\leq 2n - 5 \end{aligned}$$

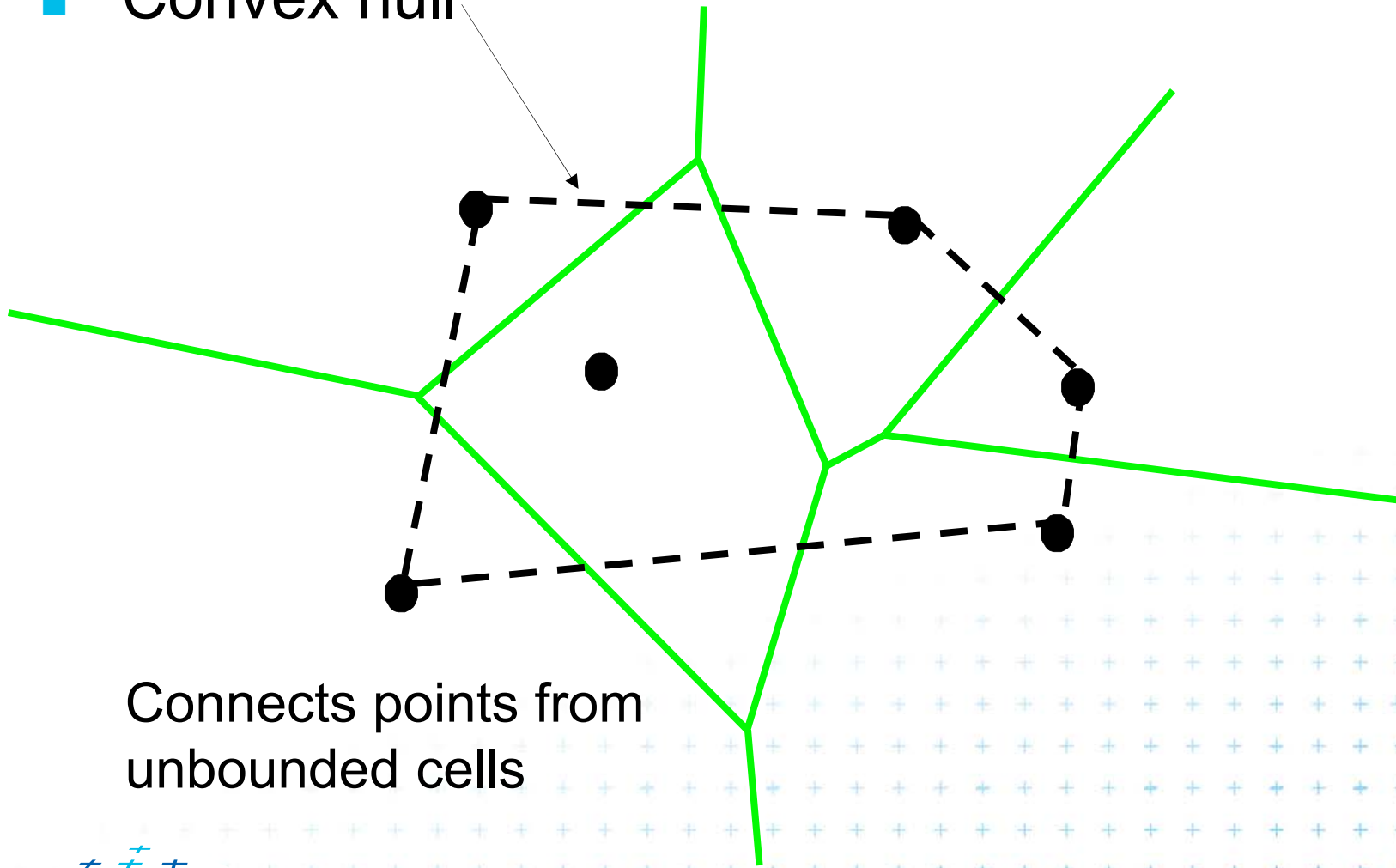
$$\begin{aligned} 2n_e &\geq 3(n_v + 1) \\ 2n_e &\geq 3(n_e - n + 1 + 1) \\ 2n_e &\geq 3n_e - 3n + 6 \\ n_e &\leq 3n - 6 \end{aligned}$$



# Voronoi diagram and convex hull

---

- Convex hull



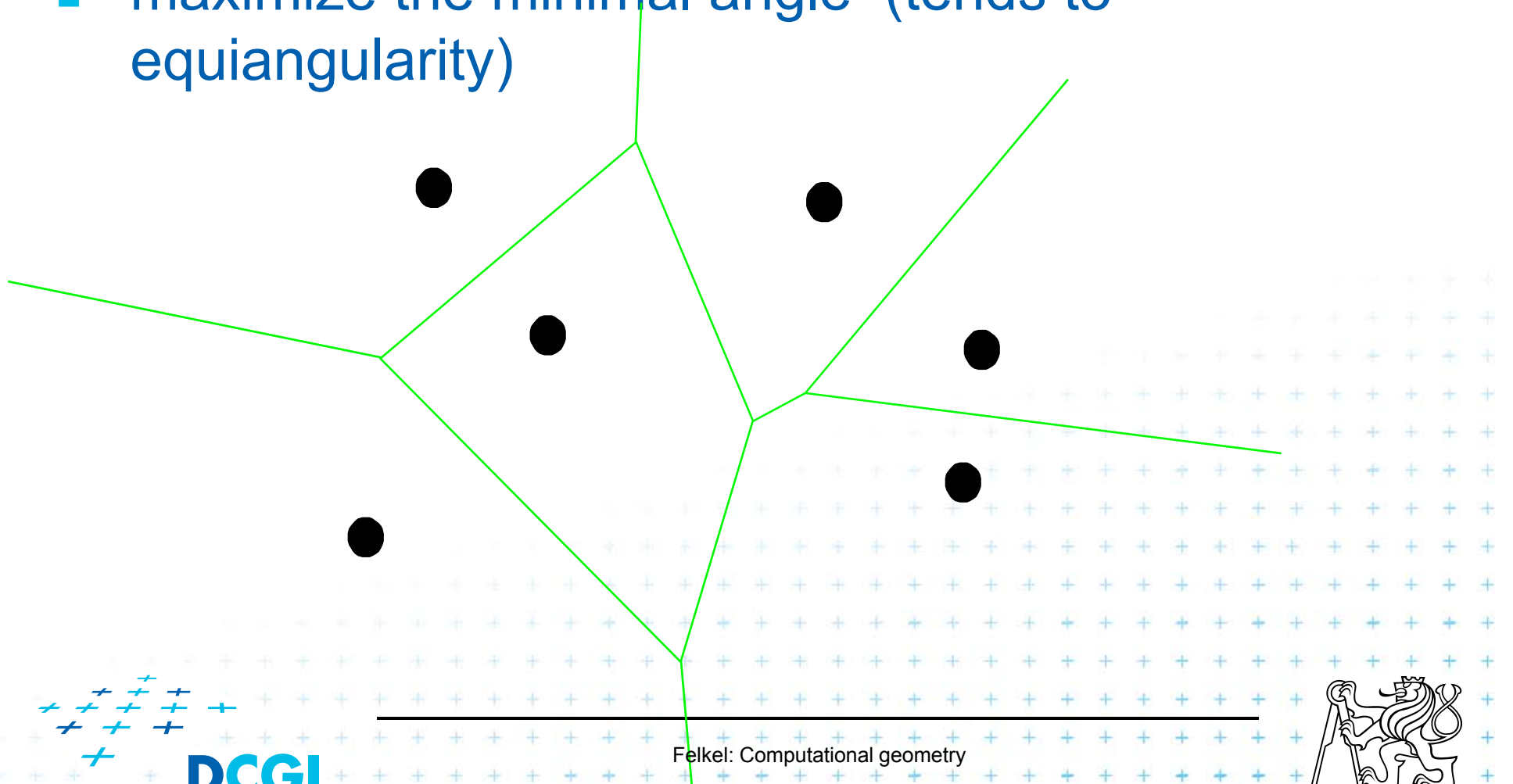
Connects points from unbounded cells



# Delaunay triangulation

---

- point set triangulation (straight line dual to VD)
- maximize the minimal angle (tends to equiangularity)

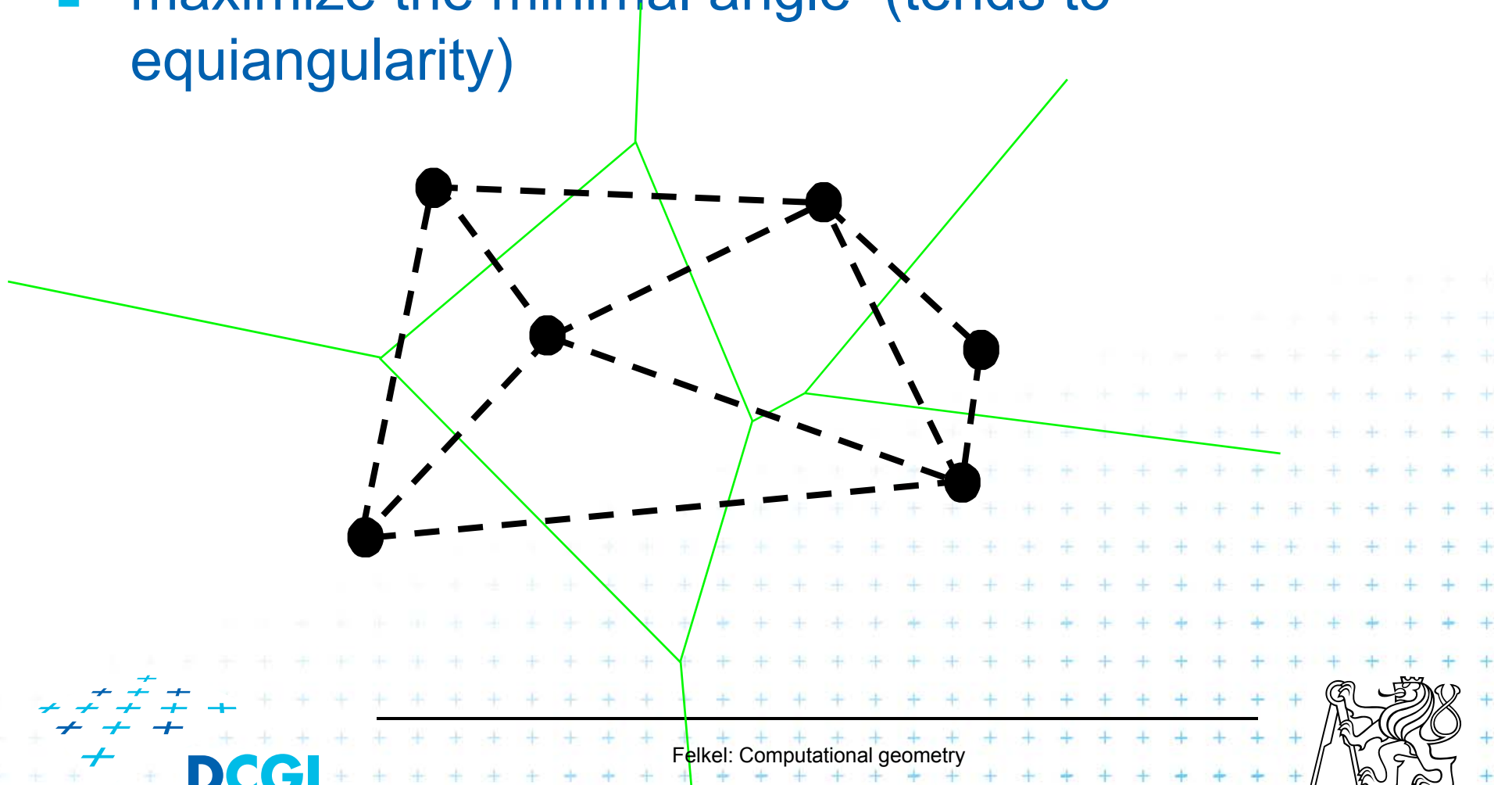




# Delaunay triangulation

---

- point set triangulation (straight line dual to VD)
- maximize the minimal angle (tends to equiangularity)

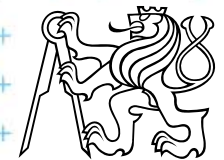
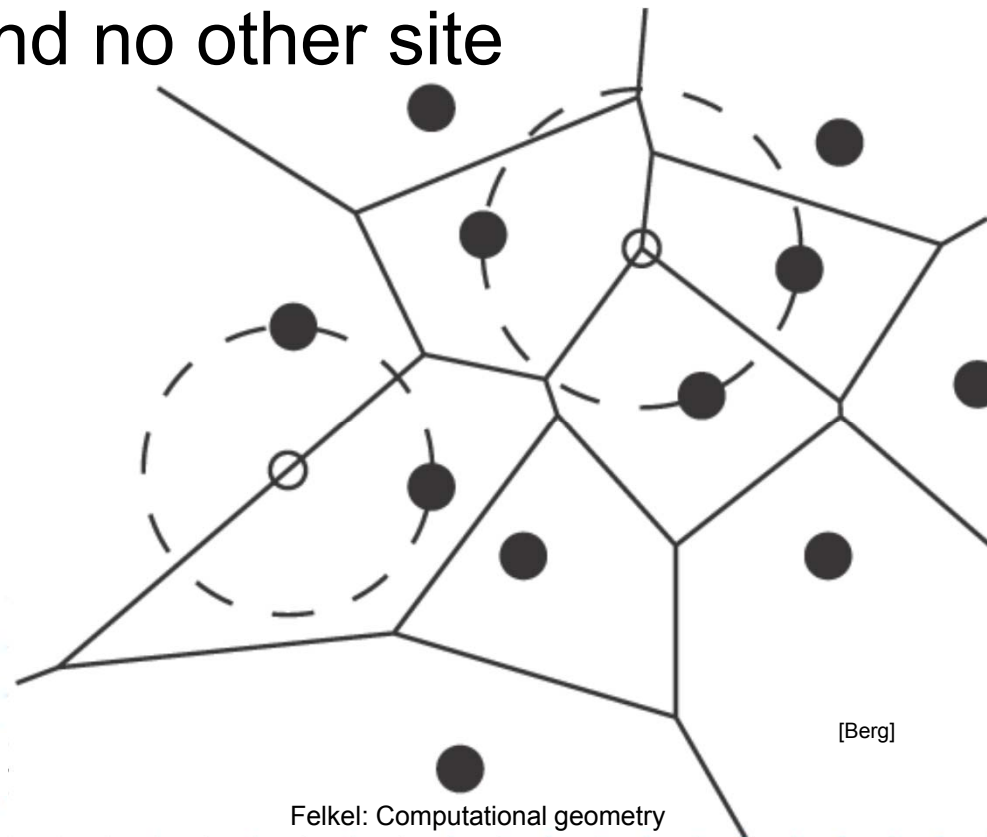
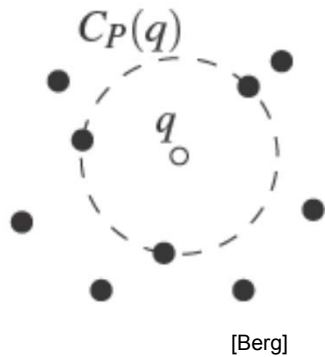




# Edges, vertices and largest empty circles

Largest empty circle  $C_P(q)$  with center in

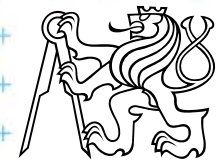
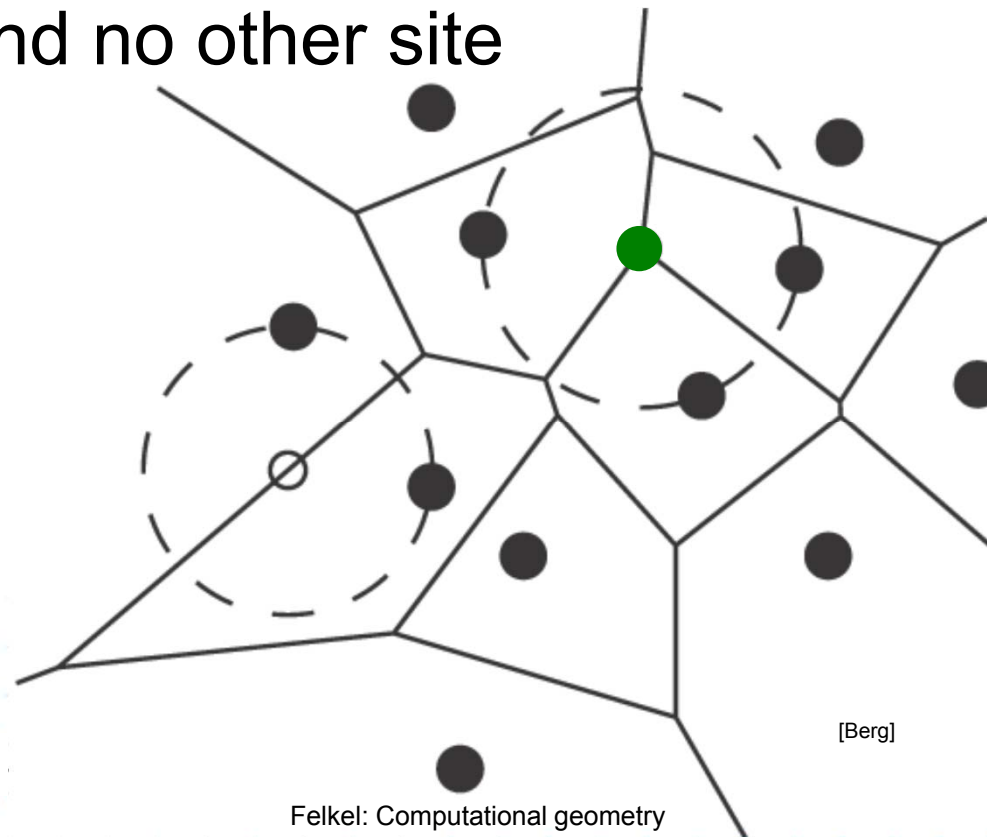
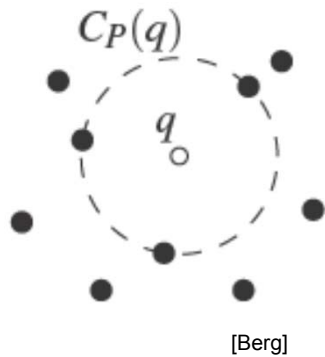
1. In VD **vertex**  $q$ : has 3 or more sites on its boundary
2. On VD **edge**: contains exactly 2 sites on its boundary and no other site



# Edges, vertices and largest empty circles

Largest empty circle  $C_P(q)$  with center in

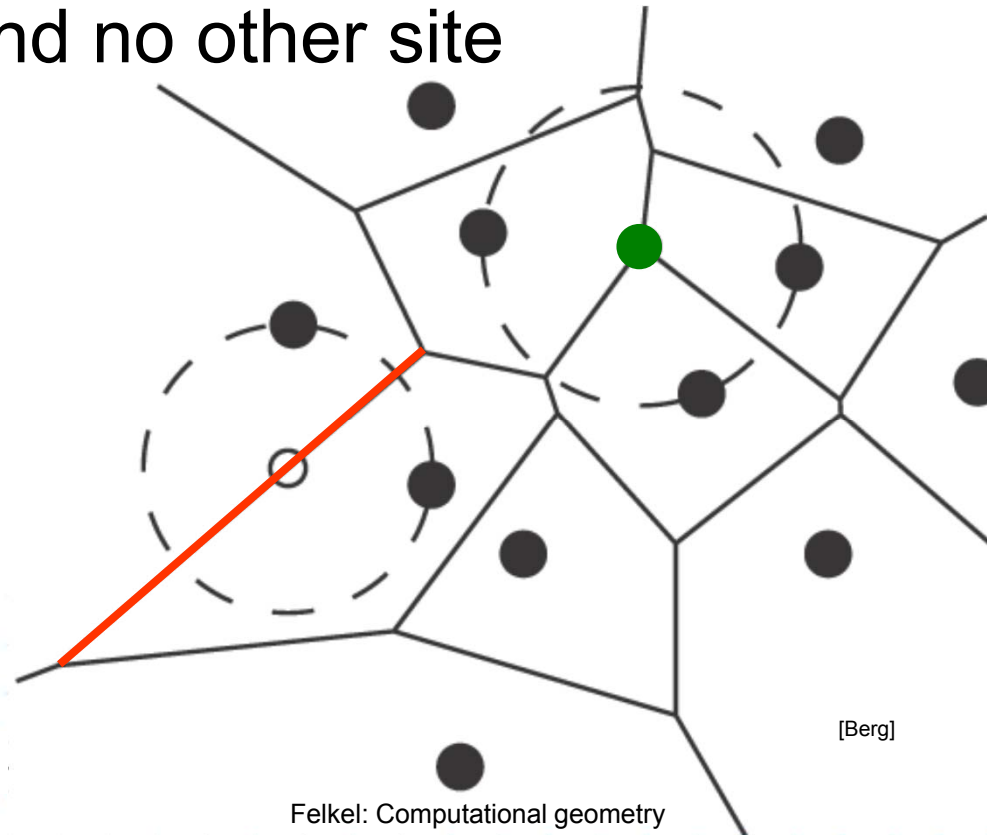
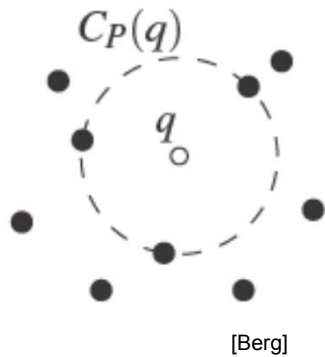
1. In VD **vertex**  $q$ : has 3 or more sites on its boundary
2. On VD **edge**: contains exactly 2 sites on its boundary and no other site



# Edges, vertices and largest empty circles

Largest empty circle  $C_P(q)$  with center in

1. In VD **vertex**  $q$ : has 3 or more sites on its boundary
2. On VD **edge**: contains exactly 2 sites on its boundary and no other site



# Some applications

---

- Nearest neighbor queries in  $\text{Vor}(P)$  of points  $P$ 
  - Point  $q \in P$  ... search sites across the edges around the cell  $q$
  - Point  $q \notin P$  ... point location queries – see Lecture 2 (the cell where point  $q$  falls)
- Facility location (shop or power plant)
  - Largest empty circle (better in Manhattan metric VD)
- Neighbors and Interpolation
  - Interpolate with the nearest neighbor, in 3D: surface reconstruction from points

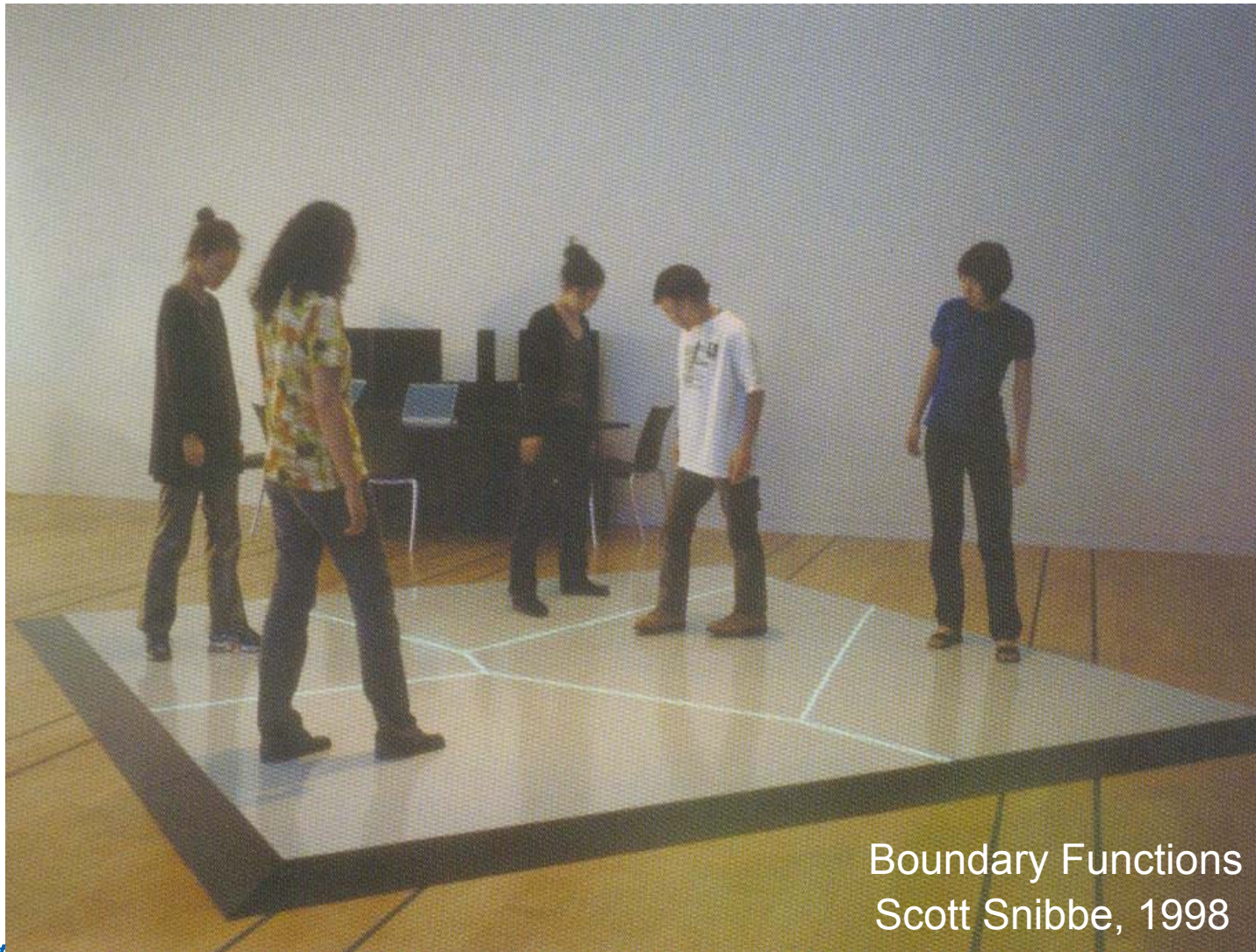
- Art





# Voronoi Art

---



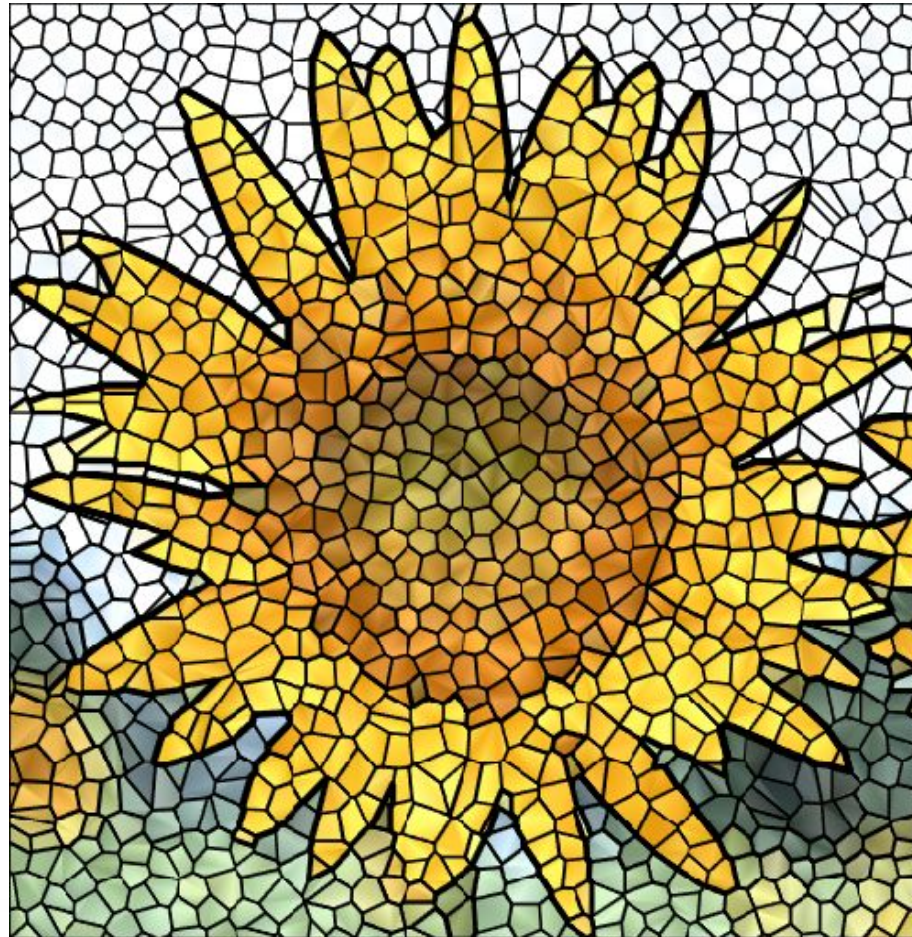
Boundary Functions  
Scott Snibbe, 1998





# Voronoi Art

---



Courtesy [Gold]



# Algorithms in 2D

---

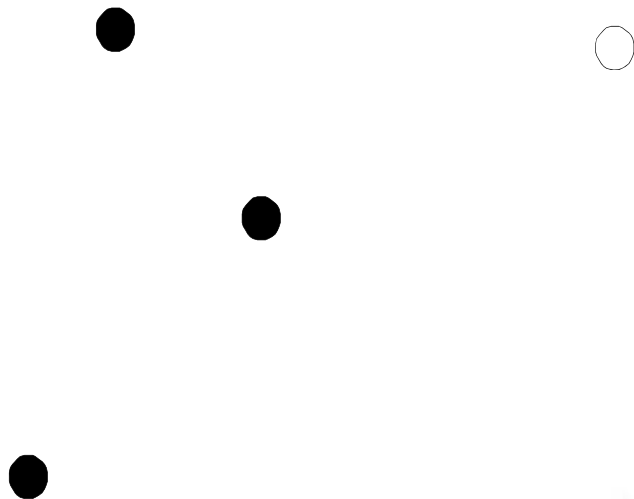
- D&C  $O(n \log n)$
- Fortune's Sweep line  $O(n \log n)$



# Voronoi diagram (VD)

---

## Divide and Conquer method



1. Split points based on x-coord into L and R
2. Recursion on L and R  
1-3 points => return  
>3 points => recursion
3. Merge  $VD_L$  and  $VD_R$ 
  - monotone chain
  - trim intersected edges
  - Add new edges from the chain



**$O(n \log n)$**

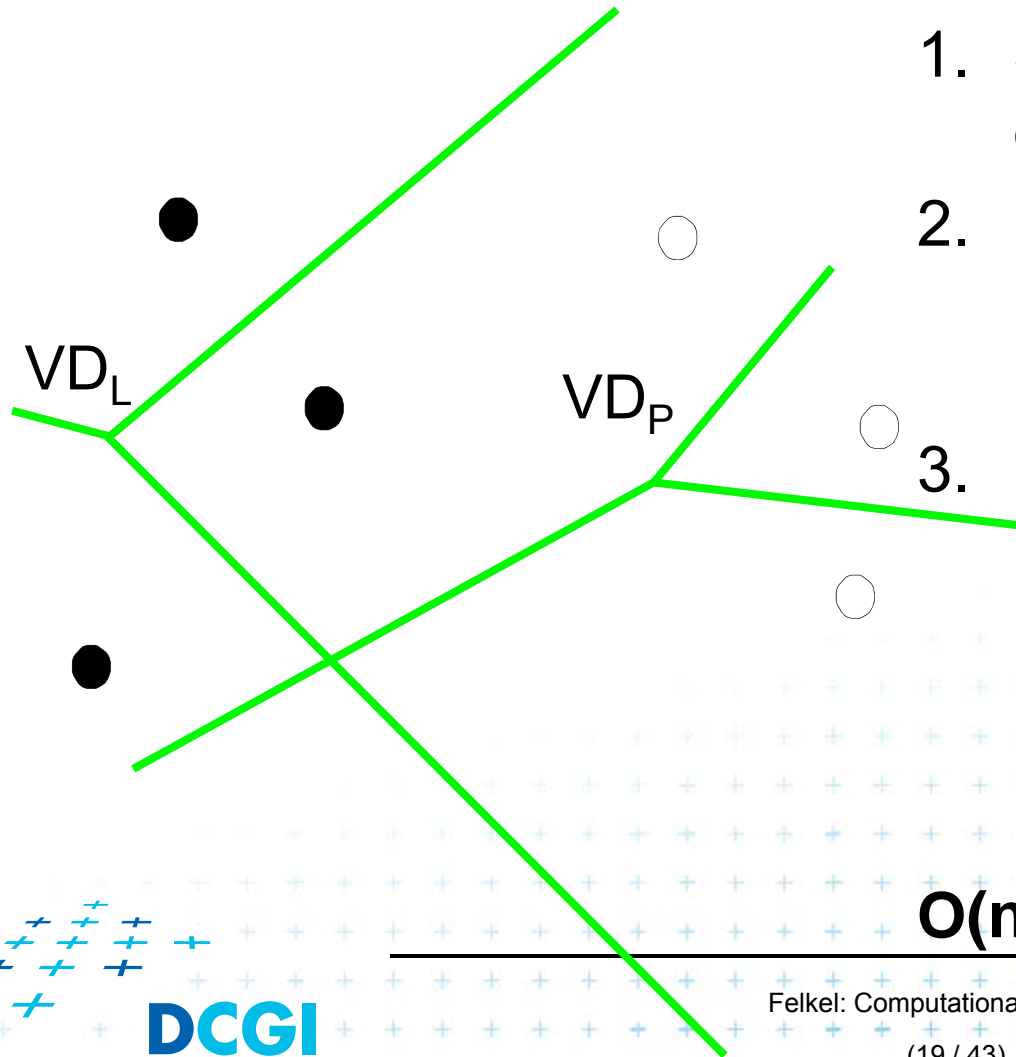




# Voronoi diagram (VD)

---

## Divide and Conquer method



1. Split points based on x-coord into L and R
2. Recursion on L and R  
1-3 points => return  
>3 points => recursion
3. Merge  $VD_L$  and  $VD_R$ 
  - monotone chain
  - trim intersected edges
  - Add new edges from the chain

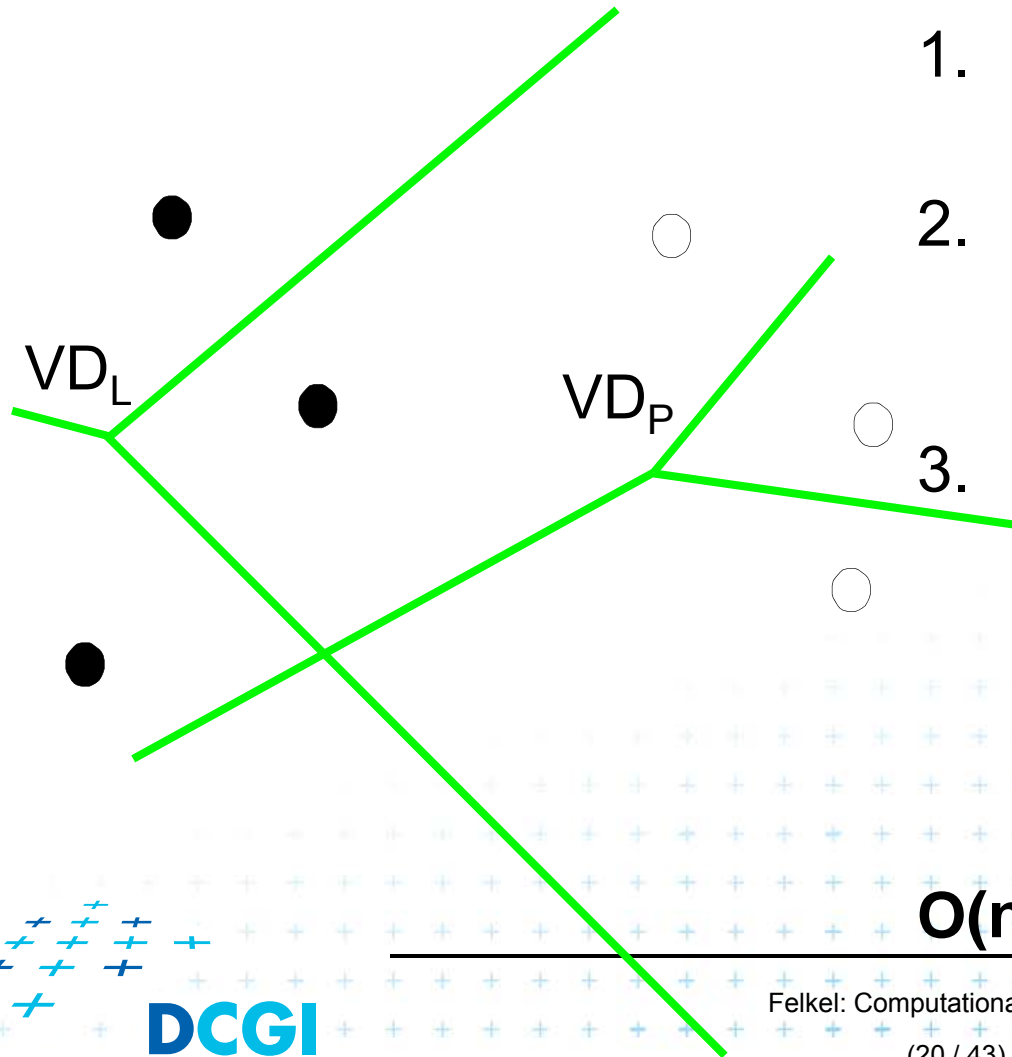
**$O(n \log n)$**



# Voronoi diagram (VD)

---

## Divide and Conquer method



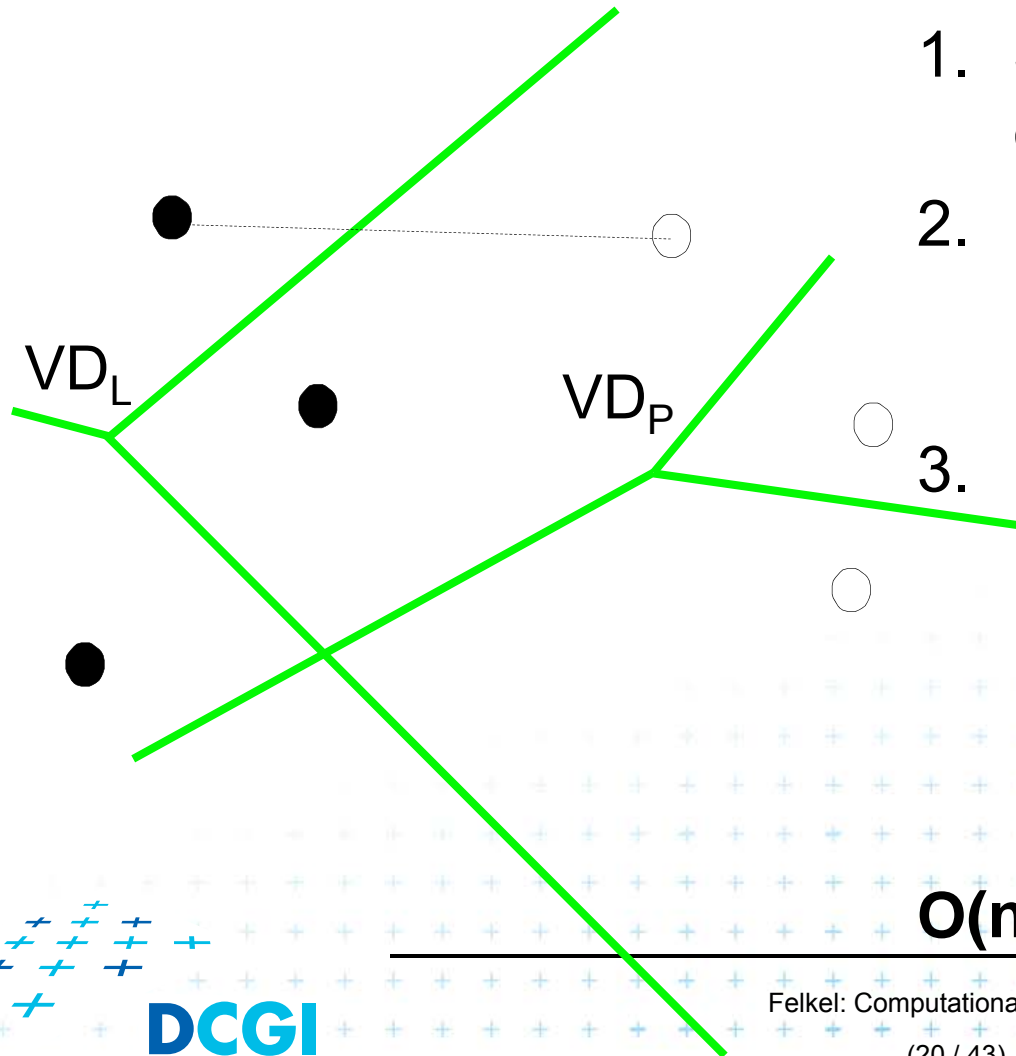
1. Split points based on x-coord into L and R
2. Recursion on L and R  
1-3 points => return  
>3 points => recursion
3. Merge  $VD_L$  and  $VD_R$ 
  - monotone chain
  - trim intersected edges
  - Add new edges from the chain

**$O(n \log n)$**



# Voronoi diagram (VD)

## Divide and Conquer method



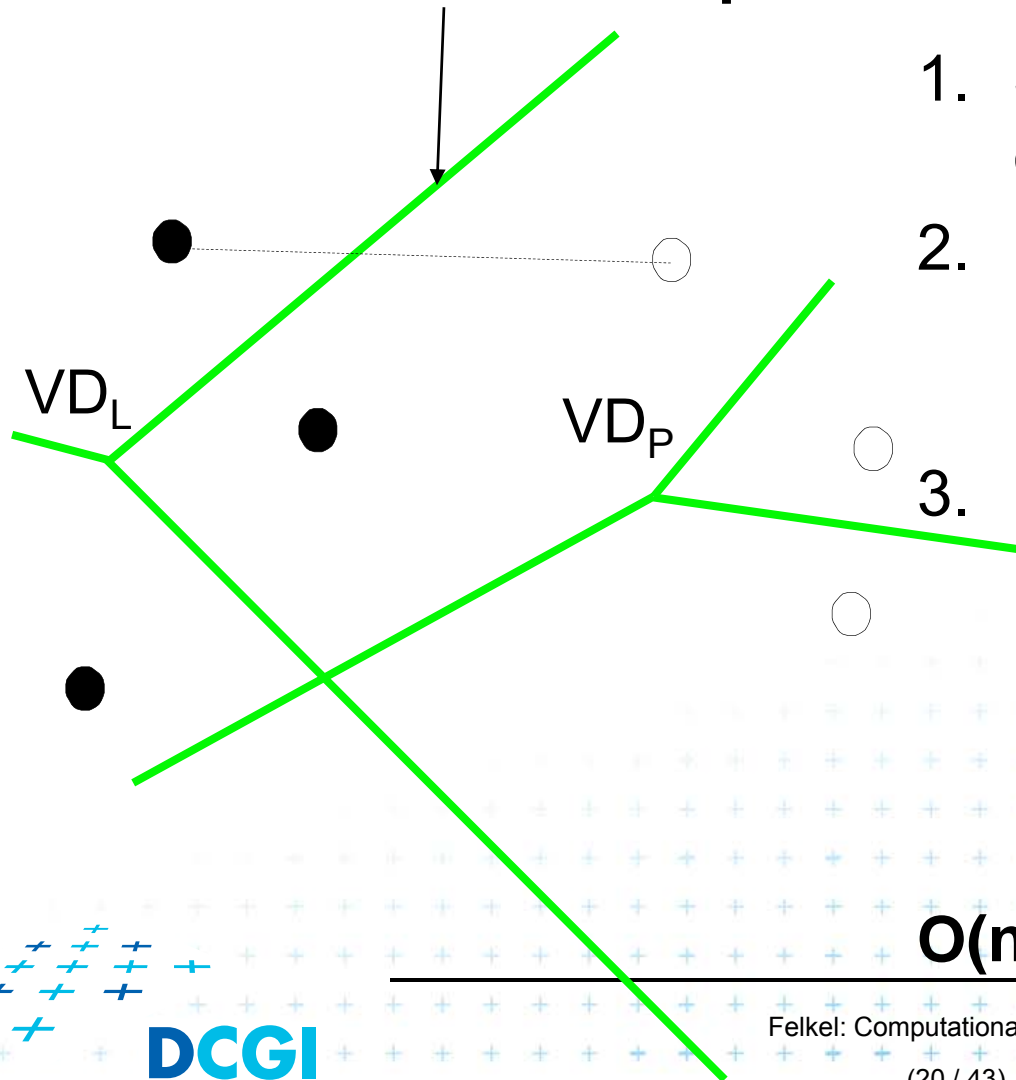
1. Split points based on x-coord into L and R
2. Recursion on L and R  
1-3 points => return  
>3 points => recursion
3. Merge  $VD_L$  and  $VD_R$ 
  - monotone chain
  - trim intersected edges
  - Add new edges from the chain

$O(n \log n)$



# Voronoi diagram (VD)

## Divide and Conquer method



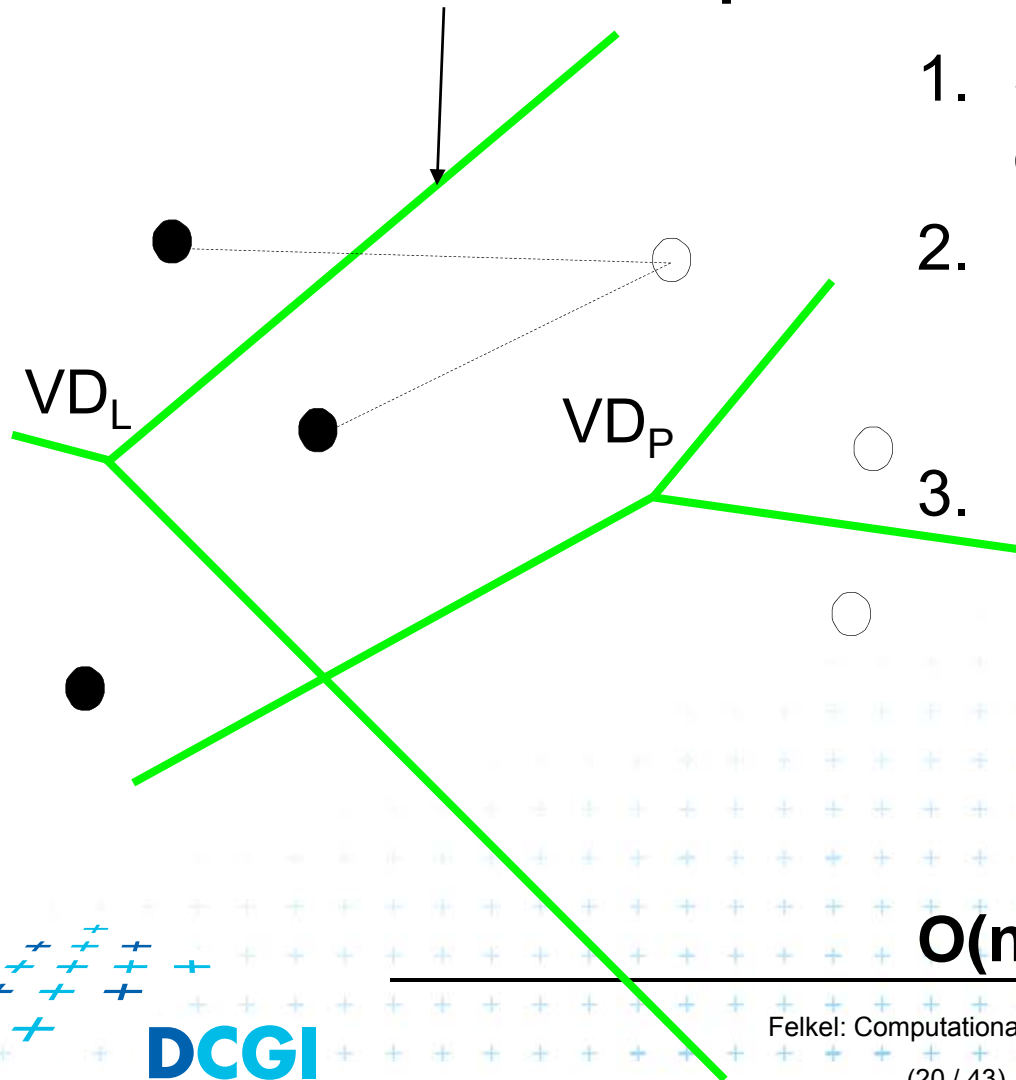
1. Split points based on x-coord into L and R
2. Recursion on L and R  
1-3 points => return  
>3 points => recursion
3. Merge  $VD_L$  and  $VD_R$ 
  - monotone chain
  - trim intersected edges
  - Add new edges from the chain

**$O(n \log n)$**



# Voronoi diagram (VD)

## Divide and Conquer method



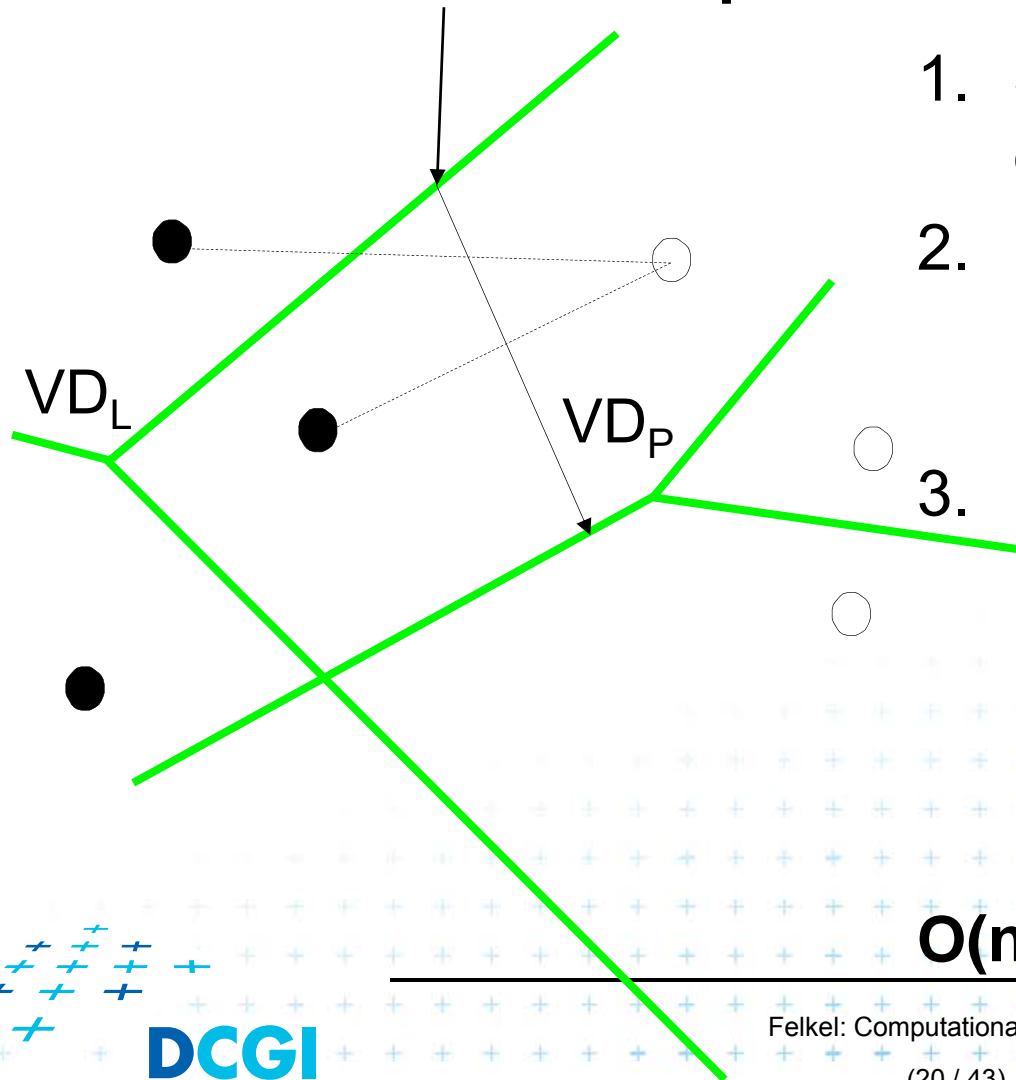
1. Split points based on x-coord into L and R
2. Recursion on L and R  
1-3 points => return  
>3 points => recursion
3. Merge  $VD_L$  and  $VD_R$ 
  - monotone chain
  - trim intersected edges
  - Add new edges from the chain

$O(n \log n)$



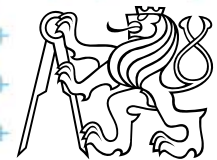
# Voronoi diagram (VD)

## Divide and Conquer method



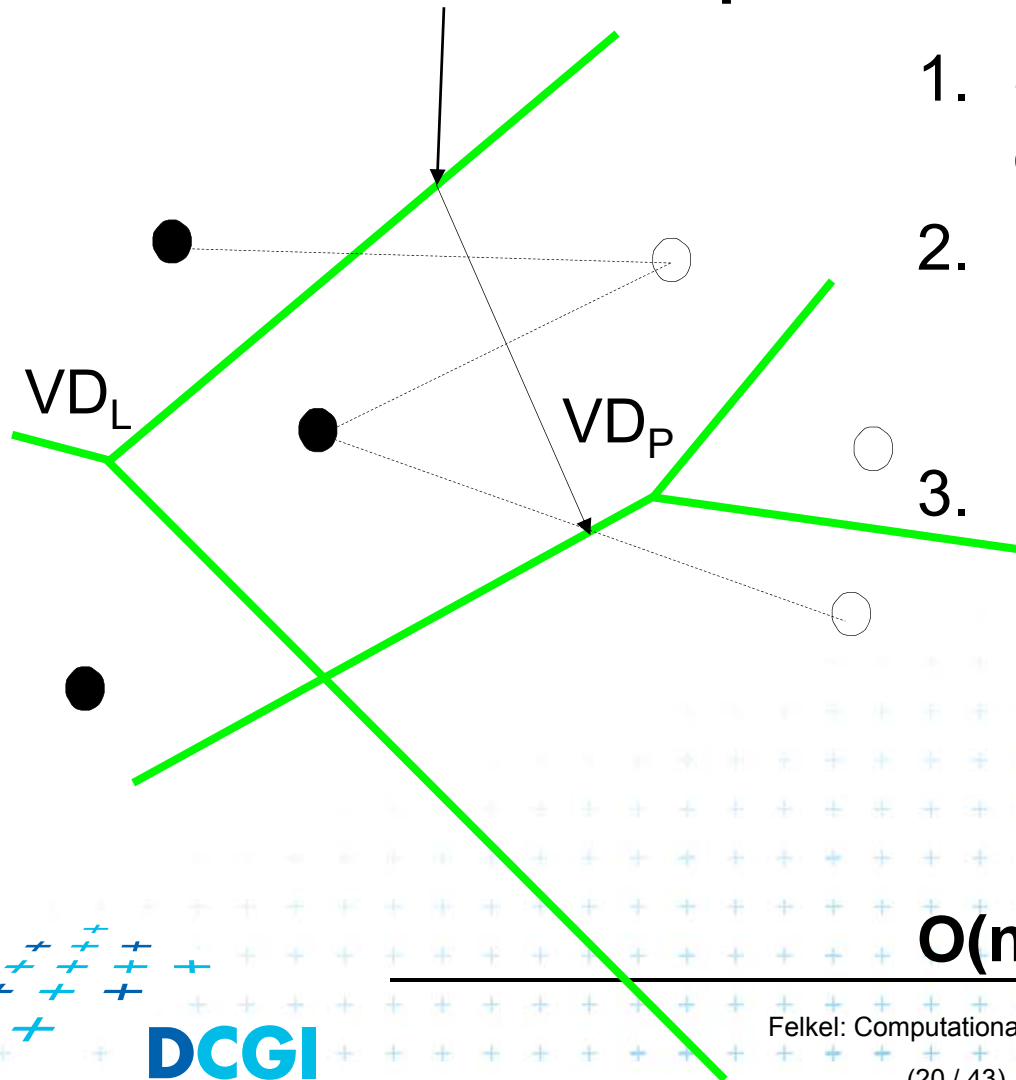
1. Split points based on x-coord into L and R
2. Recursion on L and R  
1-3 points => return  
>3 points => recursion
3. Merge  $VD_L$  and  $VD_R$ 
  - monotone chain
  - trim intersected edges
  - Add new edges from the chain

**$O(n \log n)$**



# Voronoi diagram (VD)

## Divide and Conquer method



1. Split points based on x-coord into L and R
2. Recursion on L and R  
1-3 points => return  
>3 points => recursion
3. Merge  $VD_L$  and  $VD_R$ 
  - monotone chain
  - trim intersected edges
  - Add new edges from the chain

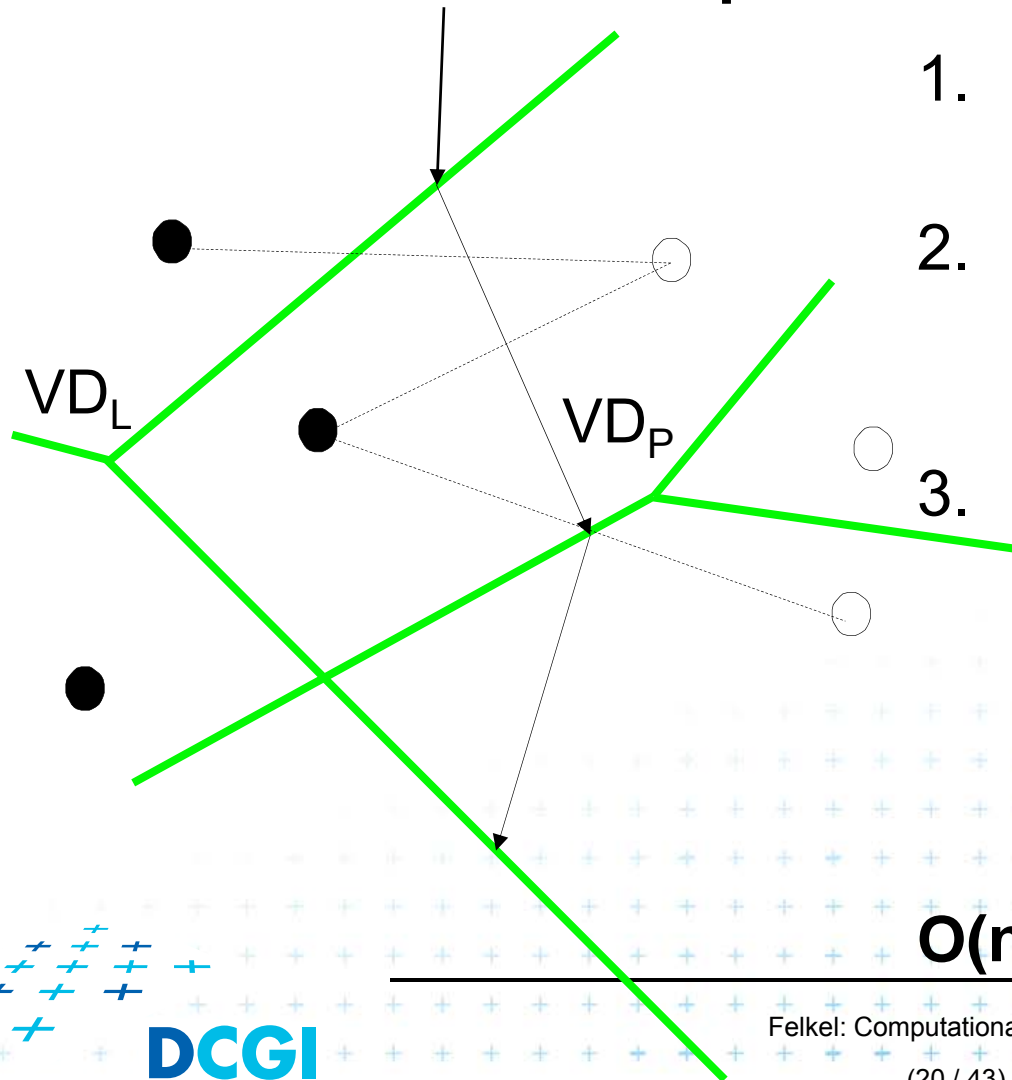
$O(n \log n)$





# Voronoi diagram (VD)

## Divide and Conquer method



1. Split points based on x-coord into L and R
2. Recursion on L and R  
1-3 points => return  
>3 points => recursion
3. Merge  $VD_L$  and  $VD_R$ 
  - monotone chain
  - trim intersected edges
  - Add new edges from the chain

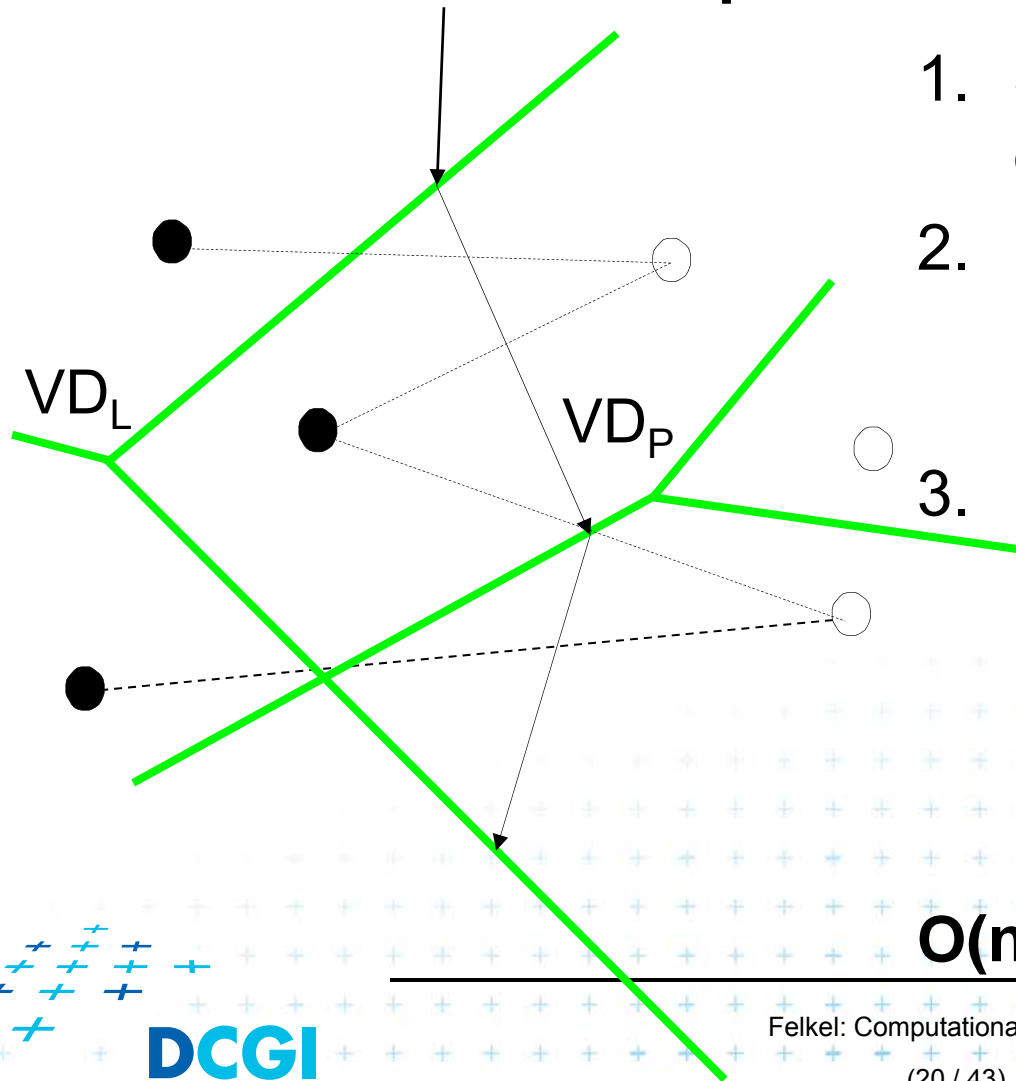
$O(n \log n)$





# Voronoi diagram (VD)

## Divide and Conquer method



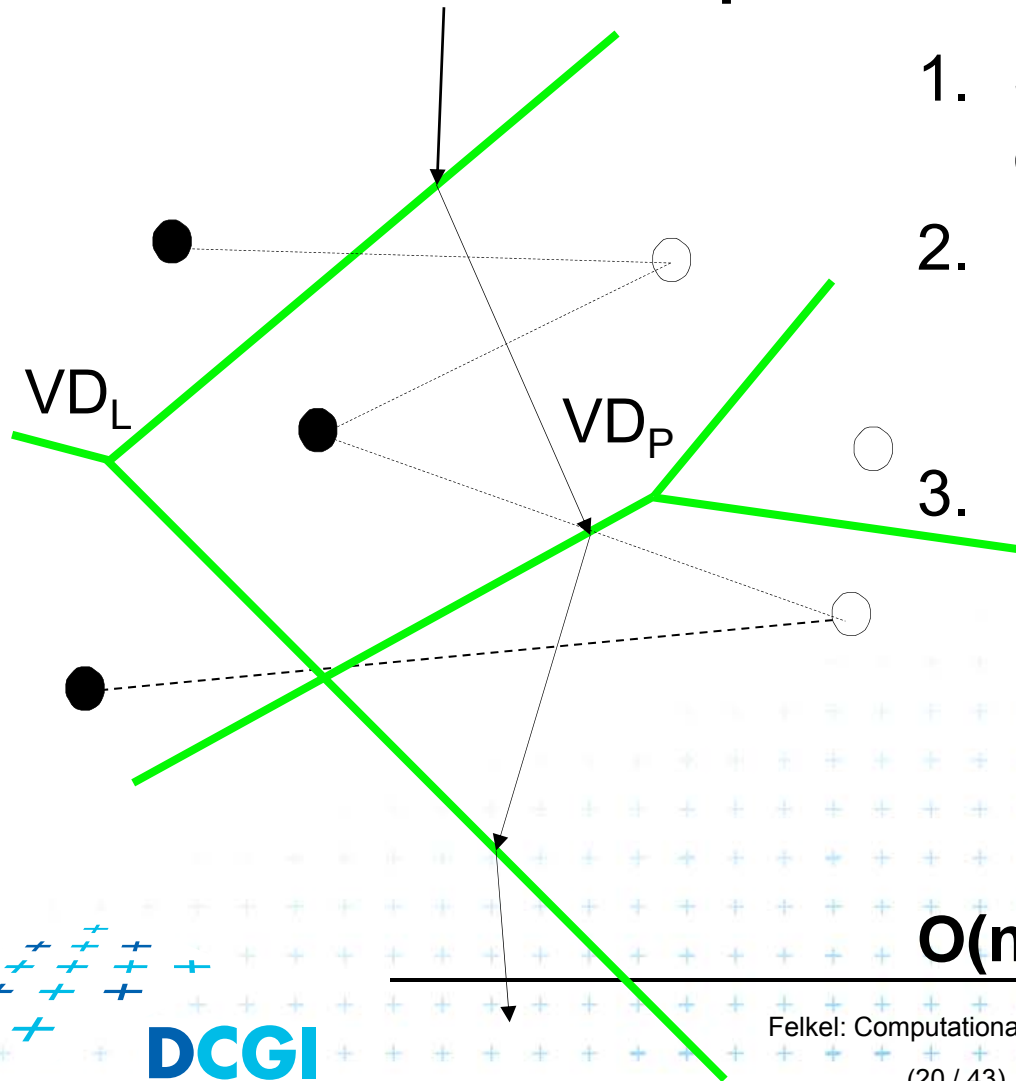
1. Split points based on x-coord into L and R
2. Recursion on L and R  
1-3 points => return  
>3 points => recursion
3. Merge  $VD_L$  and  $VD_R$ 
  - monotone chain
  - trim intersected edges
  - Add new edges from the chain

$O(n \log n)$



# Voronoi diagram (VD)

## Divide and Conquer method



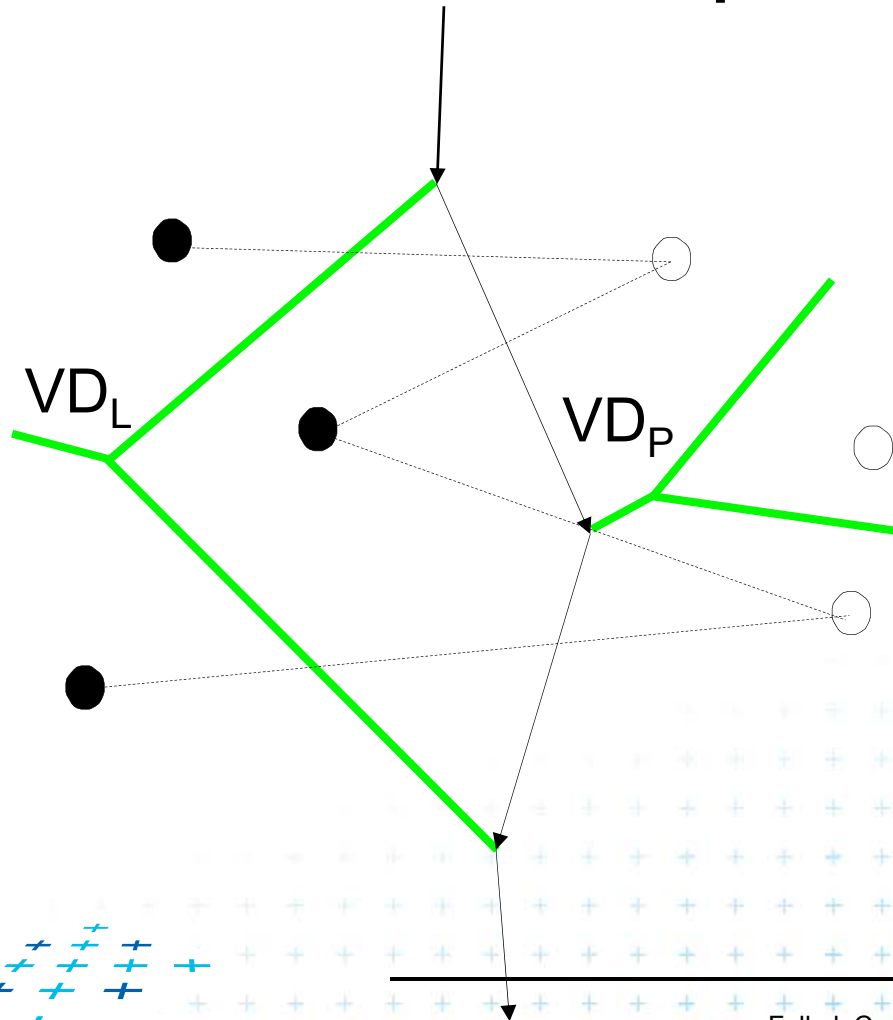
1. Split points based on x-coord into L and R
2. Recursion on L and R  
1-3 points => return  
>3 points => recursion
3. Merge  $VD_L$  and  $VD_R$ 
  - monotone chain
  - trim intersected edges
  - Add new edges from the chain

$O(n \log n)$



# Voronoi diagram (VD)

## Divide and Conquer method



1. Split points based on x-coord into L and R
2. Recursion on L and R  
1-3 points => return  
>3 points => recursion

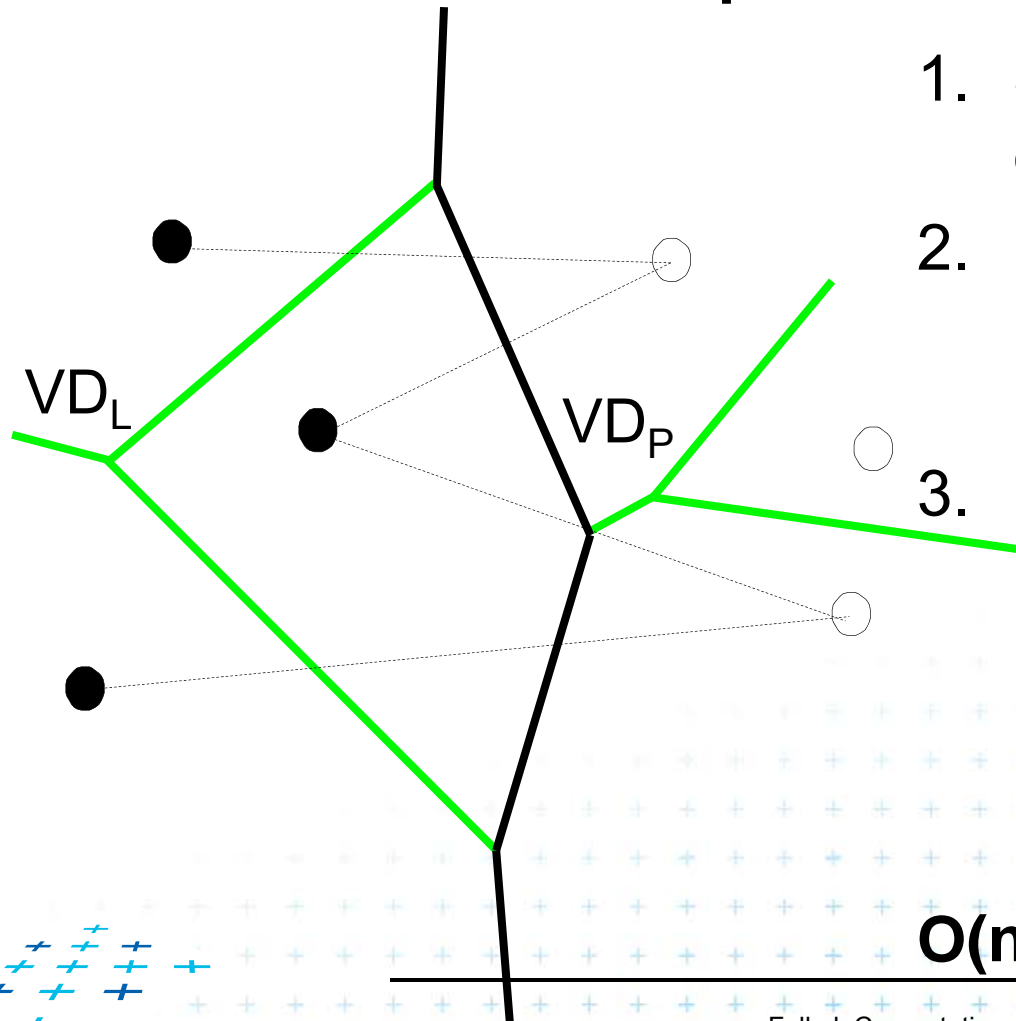
3. Merge  $VD_L$  and  $VD_R$ 
  - monotone chain
  - trim intersected edges
  - Add new edges from the chain

$O(n \log n)$



# Voronoi diagram (VD)

## Divide and Conquer method



1. Split points based on x-coord into L and R

2. Recursion on L and R

1-3 points => return

>3 points => recursion

3. Merge  $VD_L$  and  $VD_R$

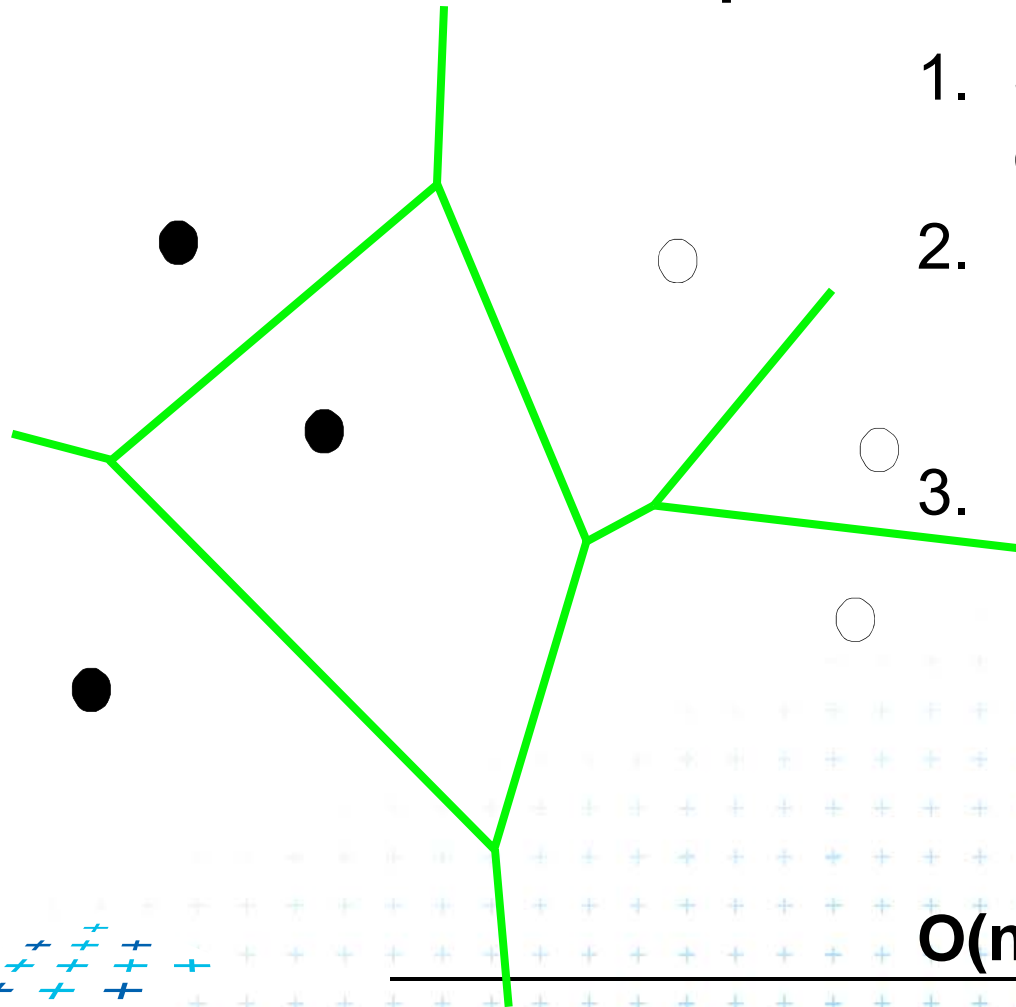
- monotone chain
- trim intersected edges
- Add new edges from the chain

$O(n \log n)$



# Voronoi diagram (VD)

## Divide and Conquer method



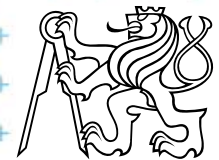
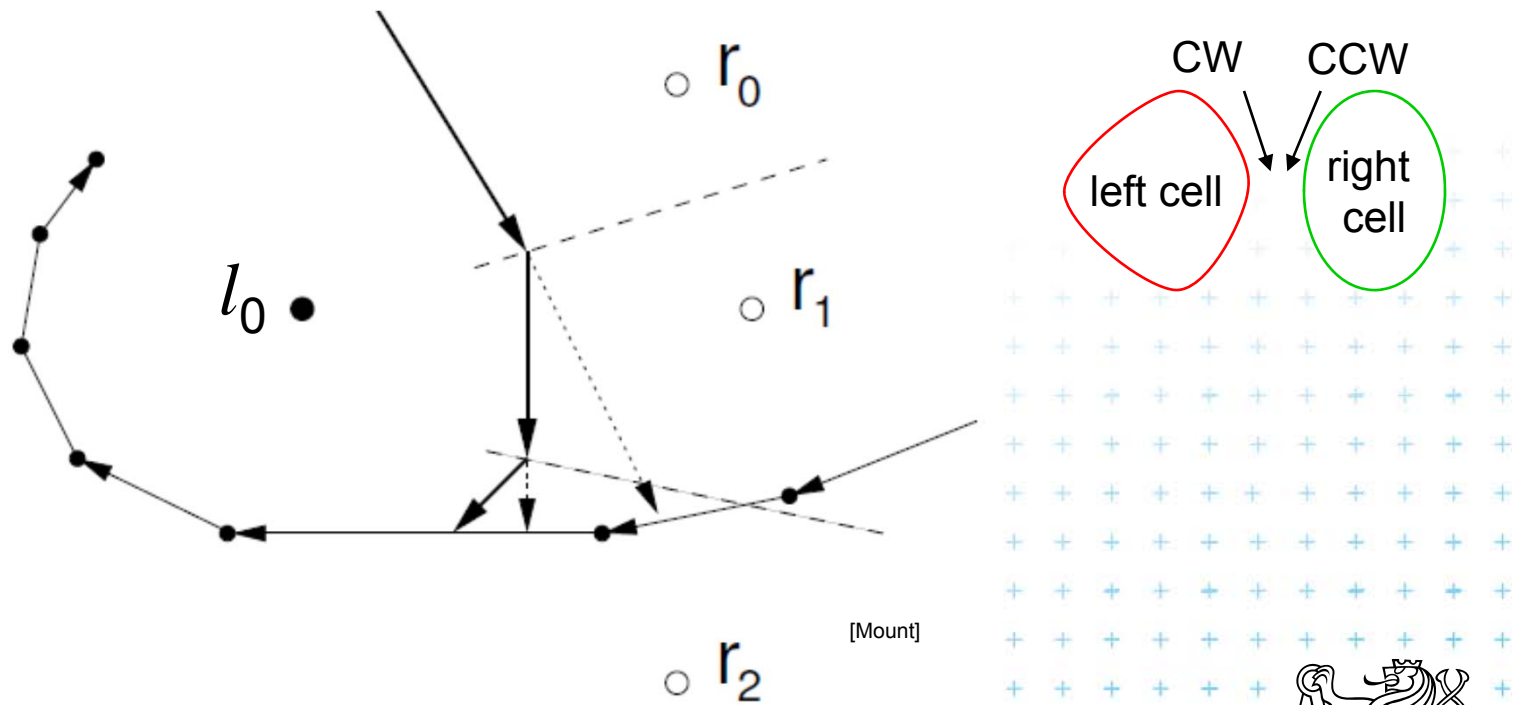
1. Split points based on x-coord into L and R
2. Recursion on L and R  
1-3 points => return  
>3 points => recursion
3. Merge  $VD_L$  and  $VD_R$ 
  - monotone chain
  - trim intersected edges
  - Add new edges from the chain

$O(n \log n)$



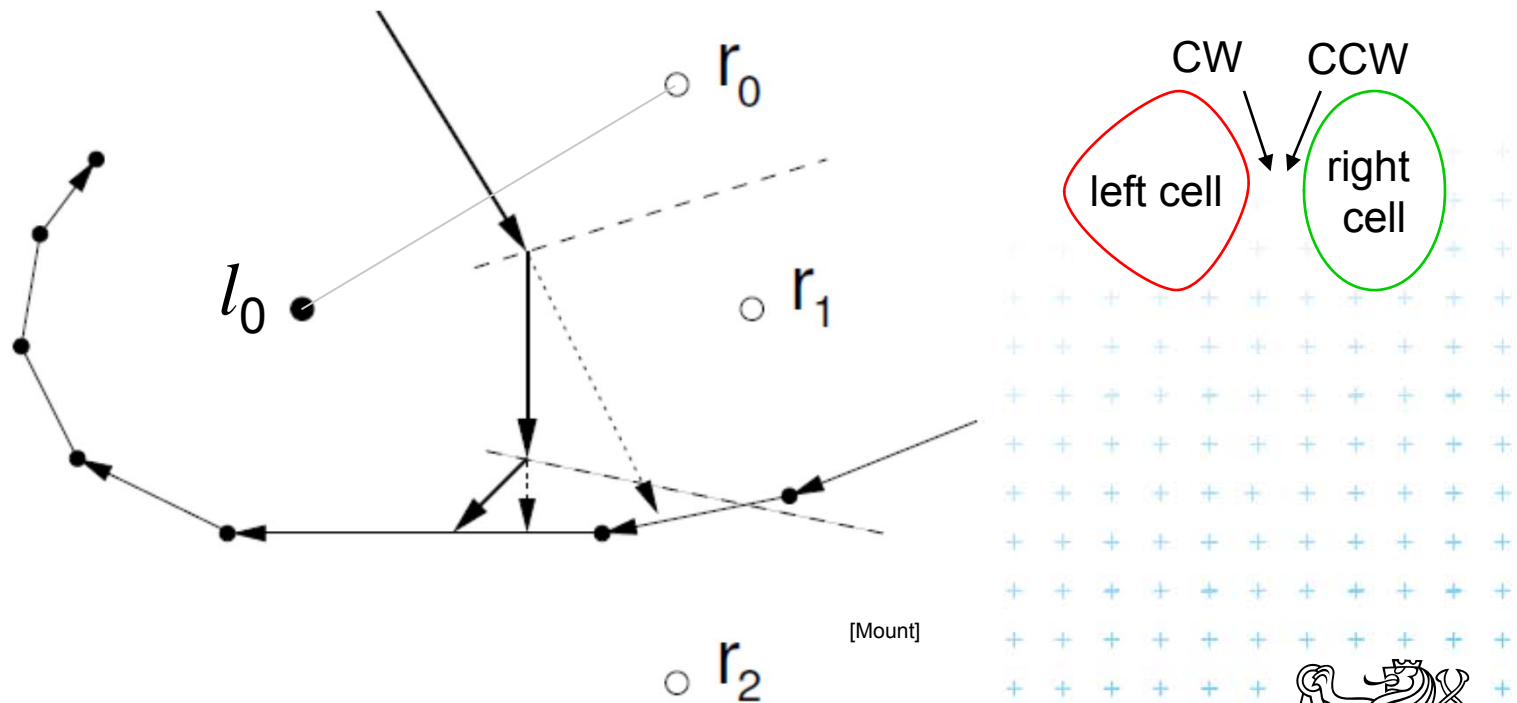
# Monotone chain search in $O(n)$

- Avoid repeated rescanning of cell edges
- Start in the last tested edge of the cell (each edge tested  $\sim$ once)
- In the **left** cell  $l_i$  continue CW, in the **right** cell  $r_i$  go CCW
- Image shows CW search on cell  $l_0$  and CCW on cells  $r_i$ :



# Monotone chain search in $O(n)$

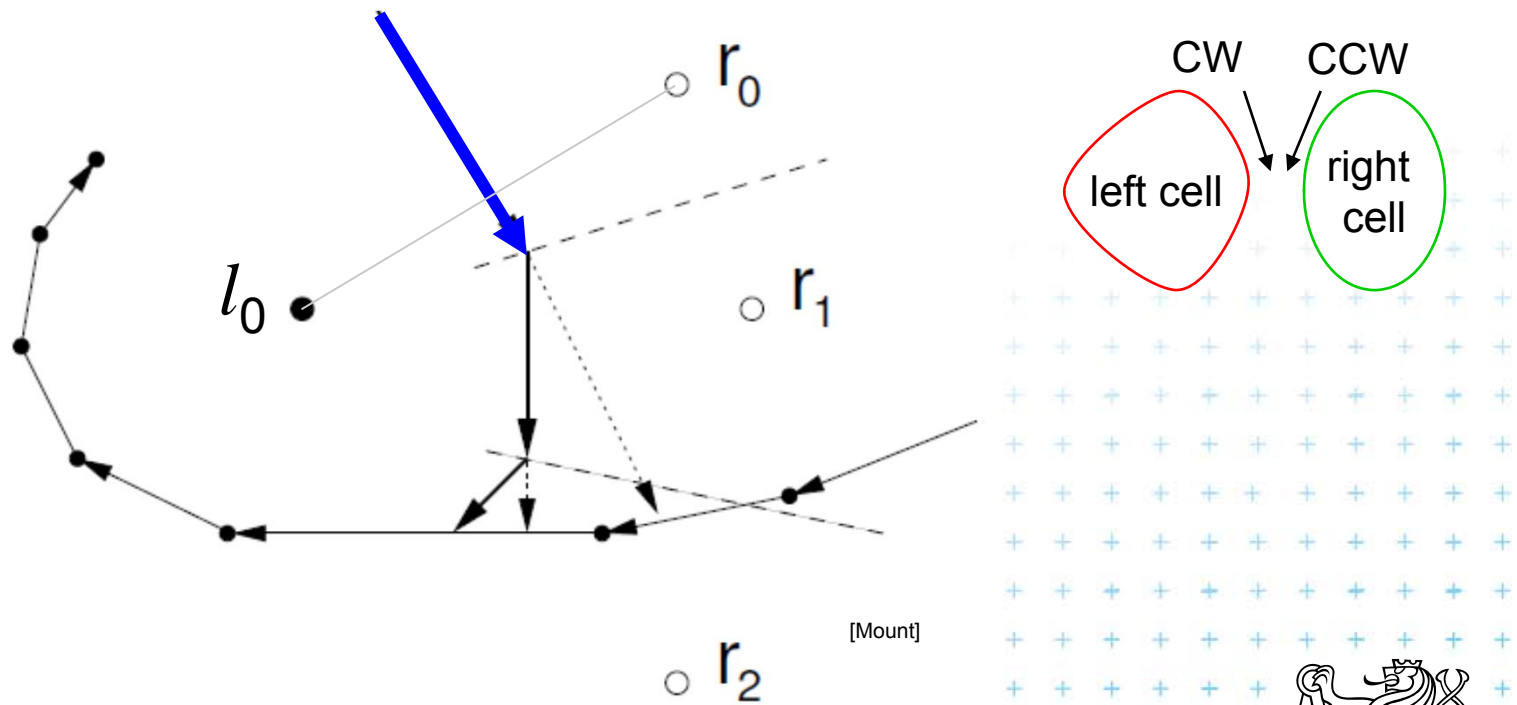
- Avoid repeated rescanning of cell edges
- Start in the last tested edge of the cell (each edge tested  $\sim$ once)
- In the **left** cell  $l_i$  continue CW, in the **right** cell  $r_i$  go CCW
- Image shows CW search on cell  $l_0$  and CCW on cells  $r_i$ :





# Monotone chain search in $O(n)$

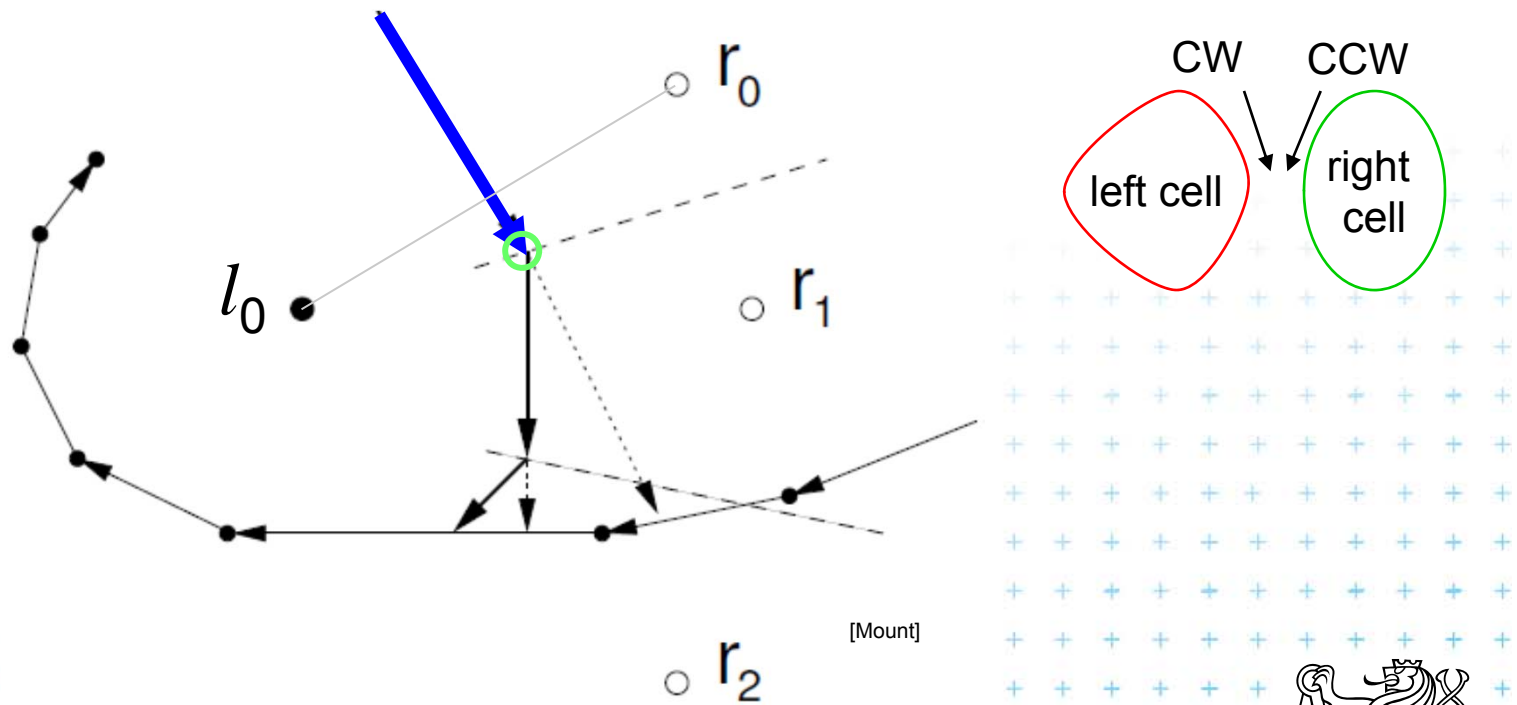
- Avoid repeated rescanning of cell edges
- Start in the last tested edge of the cell (each edge tested  $\sim$ once)
- In the **left** cell  $l_i$  continue CW, in the **right** cell  $r_i$  go CCW
- Image shows CW search on cell  $l_0$  and CCW on cells  $r_i$ :





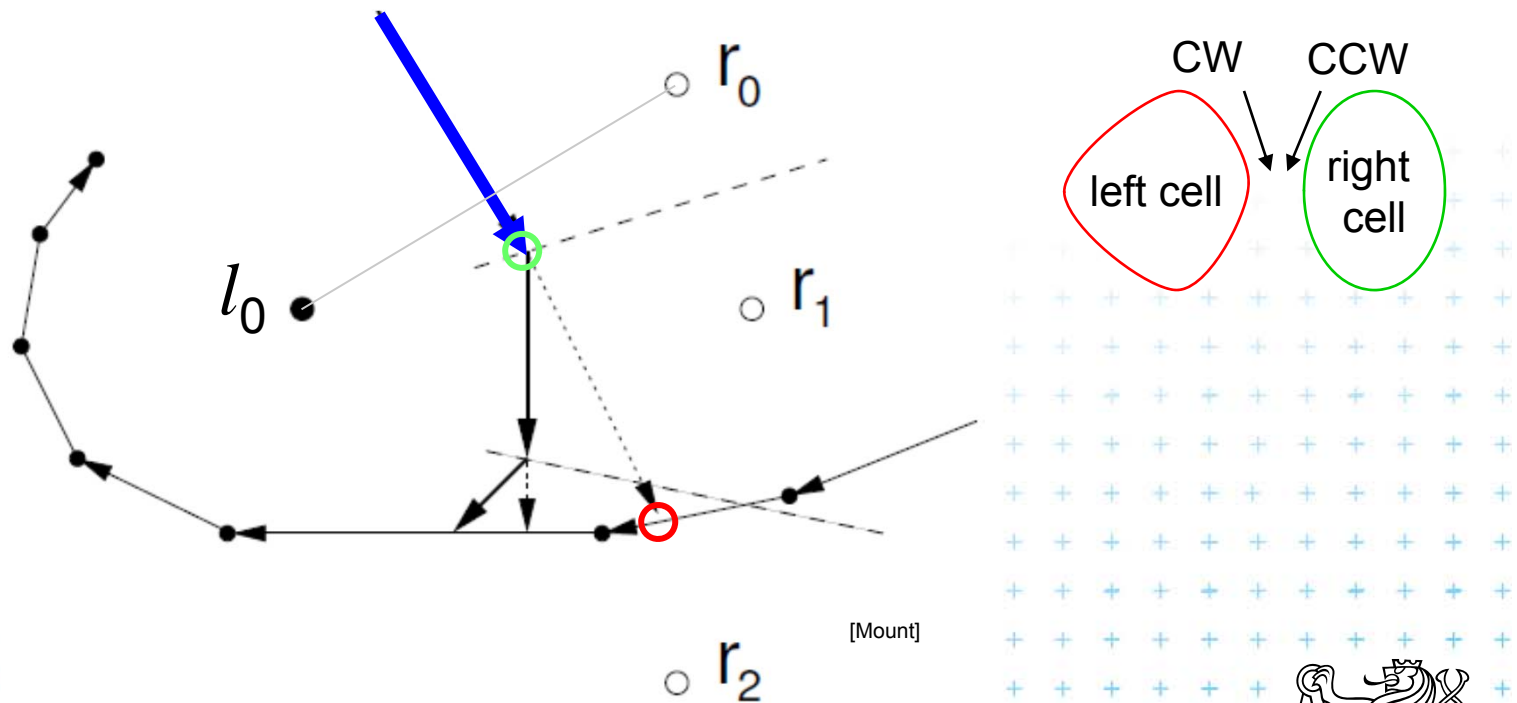
# Monotone chain search in $O(n)$

- Avoid repeated rescanning of cell edges
- Start in the last tested edge of the cell (each edge tested  $\sim$ once)
- In the **left** cell  $l_i$  continue CW, in the **right** cell  $r_i$  go CCW
- Image shows CW search on cell  $l_0$  and CCW on cells  $r_i$ :



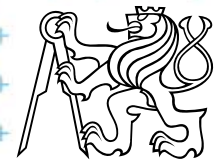
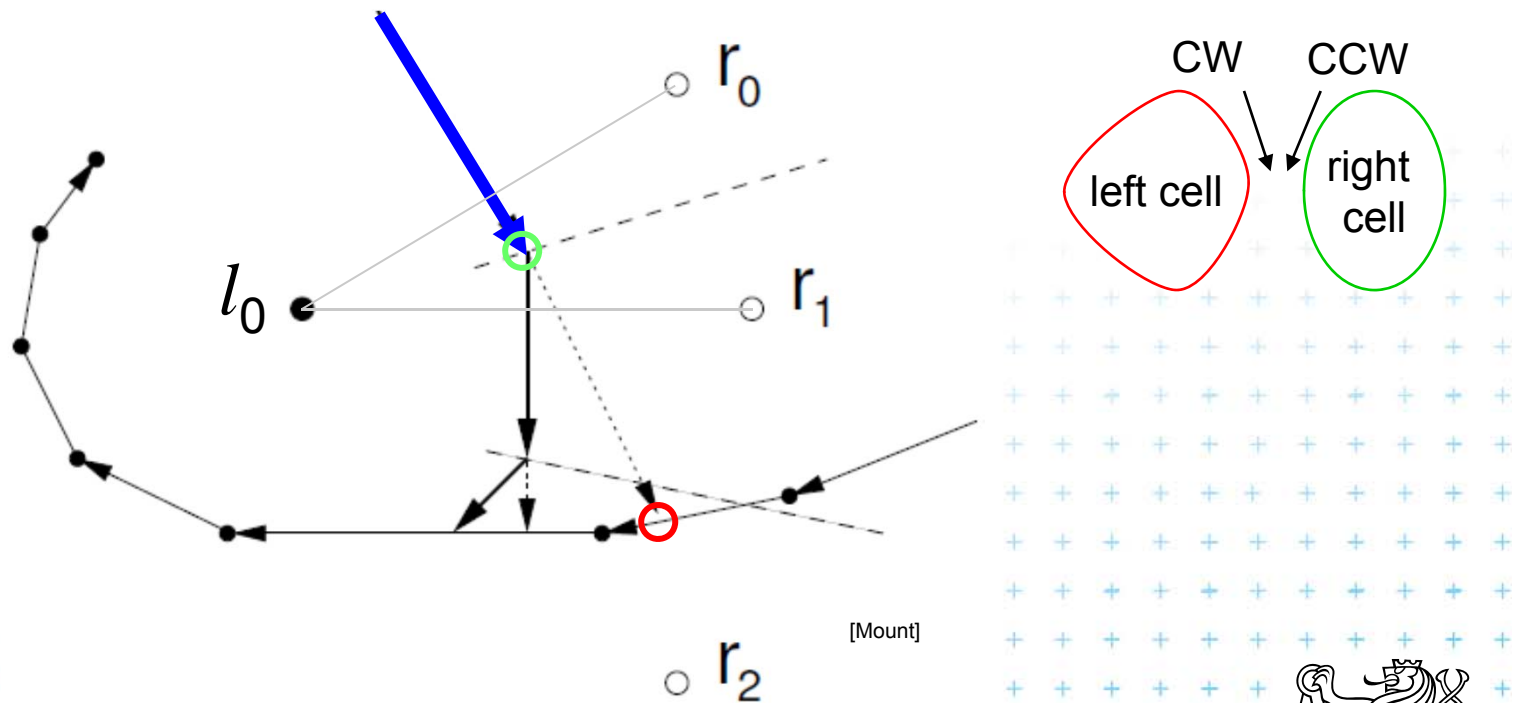
# Monotone chain search in $O(n)$

- Avoid repeated rescanning of cell edges
- Start in the last tested edge of the cell (each edge tested  $\sim$ once)
- In the **left** cell  $l_i$  continue CW, in the **right** cell  $r_i$  go CCW
- Image shows CW search on cell  $l_0$  and CCW on cells  $r_i$ :



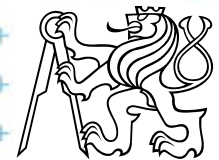
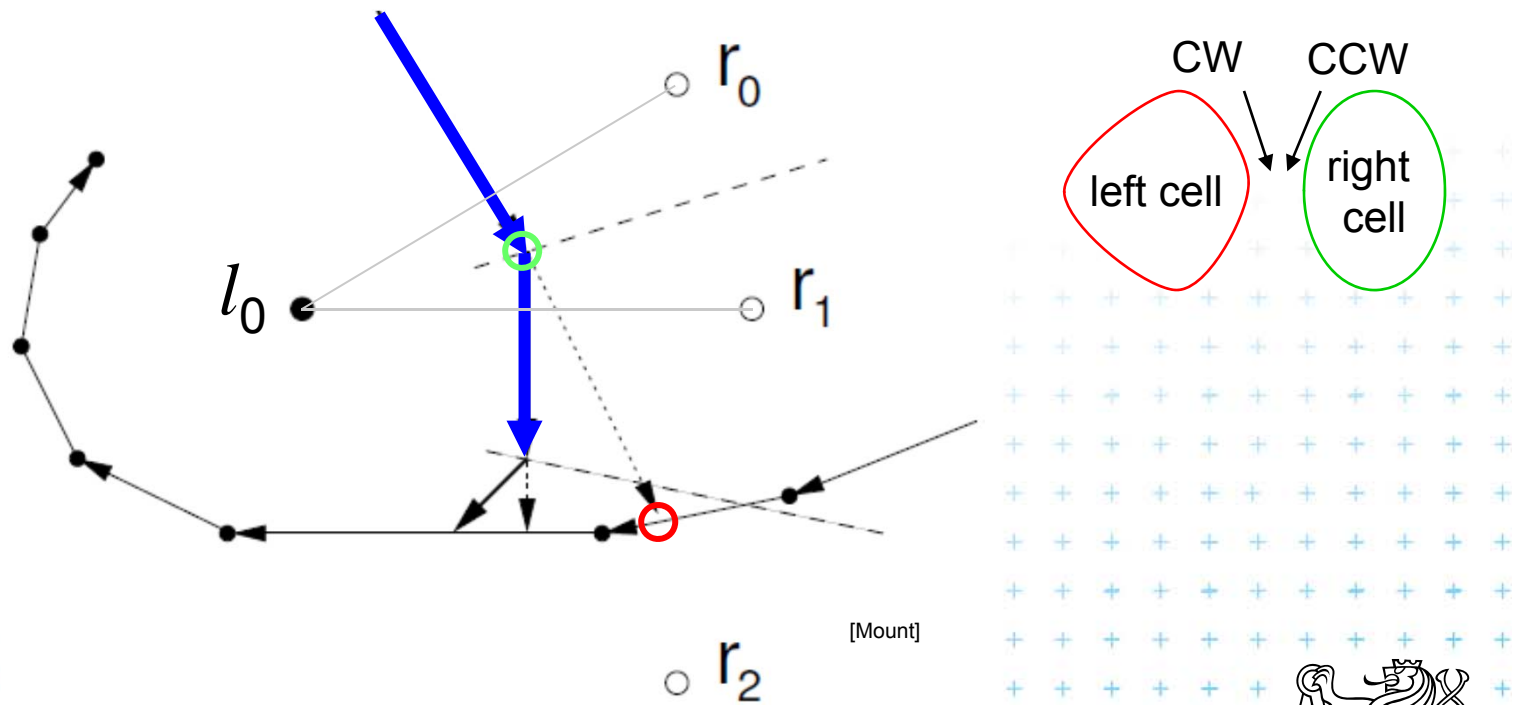
# Monotone chain search in $O(n)$

- Avoid repeated rescanning of cell edges
- Start in the last tested edge of the cell (each edge tested  $\sim$ once)
- In the **left** cell  $l_i$  continue CW, in the **right** cell  $r_i$  go CCW
- Image shows CW search on cell  $l_0$  and CCW on cells  $r_i$ :



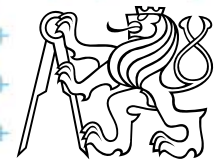
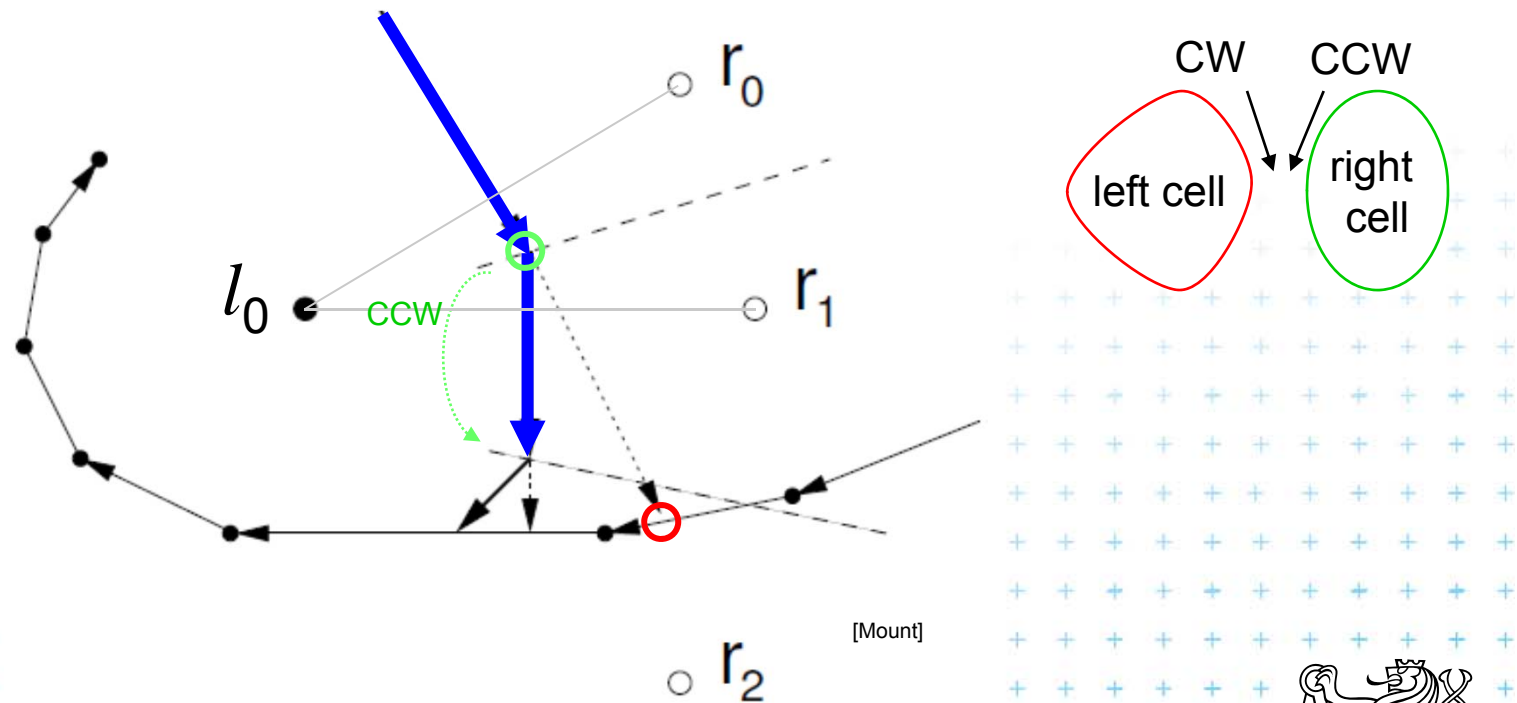
# Monotone chain search in $O(n)$

- Avoid repeated rescanning of cell edges
- Start in the last tested edge of the cell (each edge tested  $\sim$ once)
- In the **left** cell  $l_i$  continue CW, in the **right** cell  $r_i$  go CCW
- Image shows CW search on cell  $l_0$  and CCW on cells  $r_i$ :



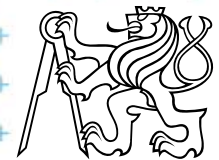
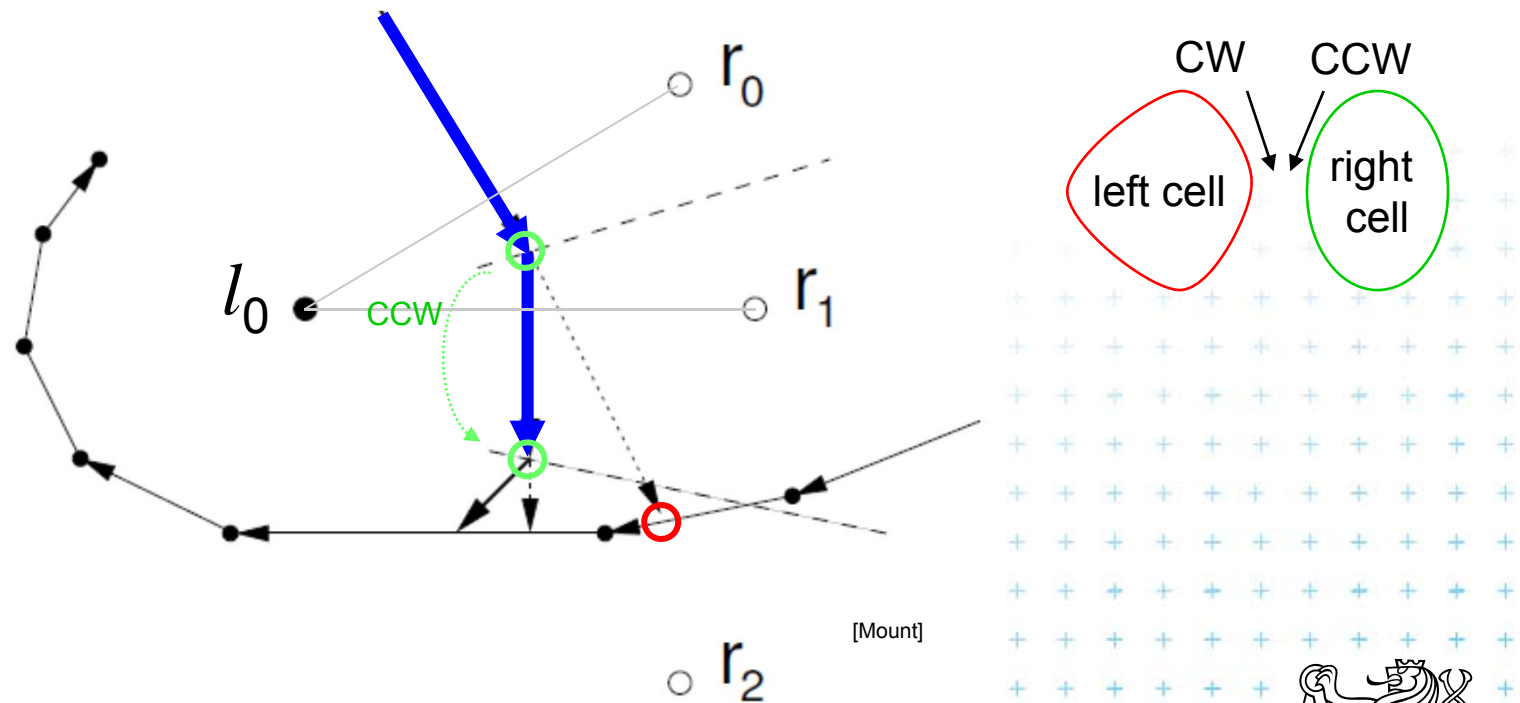
# Monotone chain search in $O(n)$

- Avoid repeated rescanning of cell edges
- Start in the last tested edge of the cell (each edge tested  $\sim$ once)
- In the **left** cell  $l_i$  continue CW, in the **right** cell  $r_i$  go CCW
- Image shows CW search on cell  $l_0$  and CCW on cells  $r_i$ :



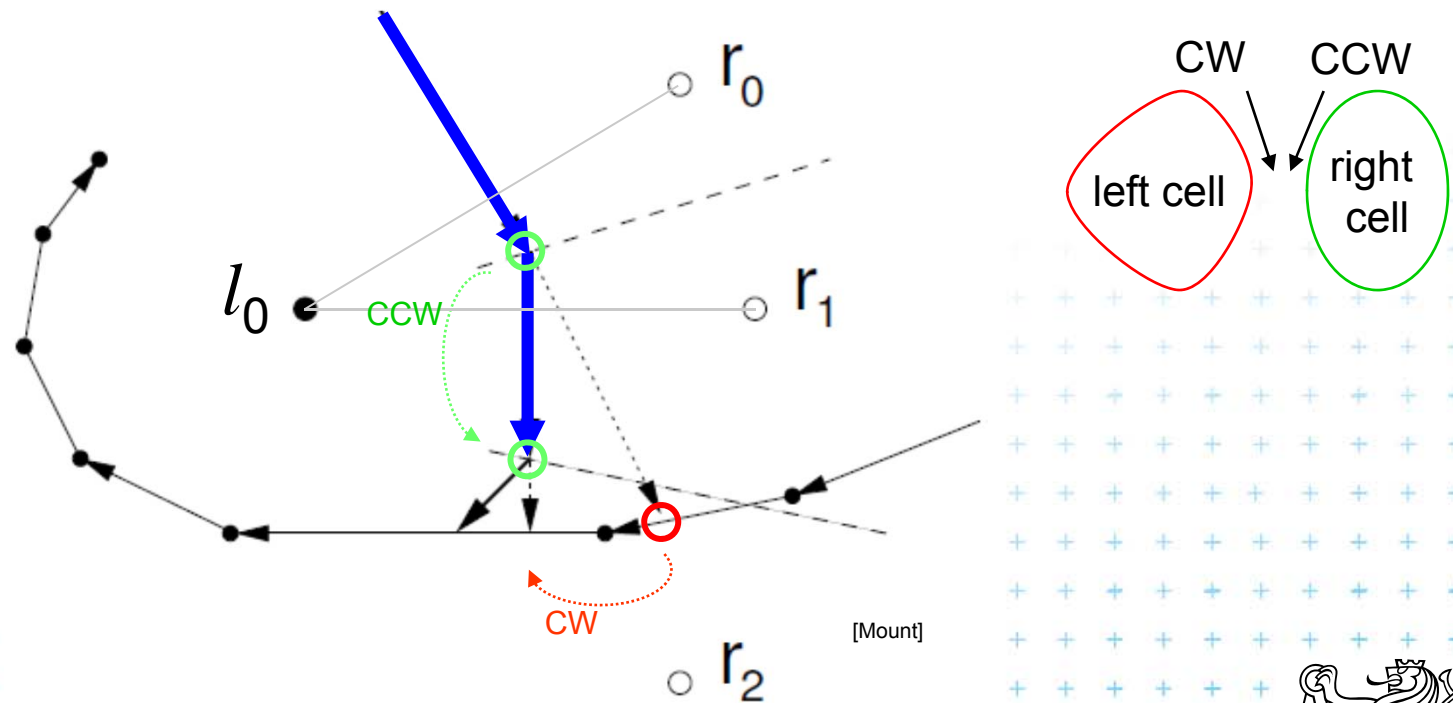
# Monotone chain search in $O(n)$

- Avoid repeated rescanning of cell edges
- Start in the last tested edge of the cell (each edge tested  $\sim$ once)
- In the **left** cell  $l_i$  continue CW, in the **right** cell  $r_i$  go CCW
- Image shows CW search on cell  $l_0$  and CCW on cells  $r_i$ :



# Monotone chain search in $O(n)$

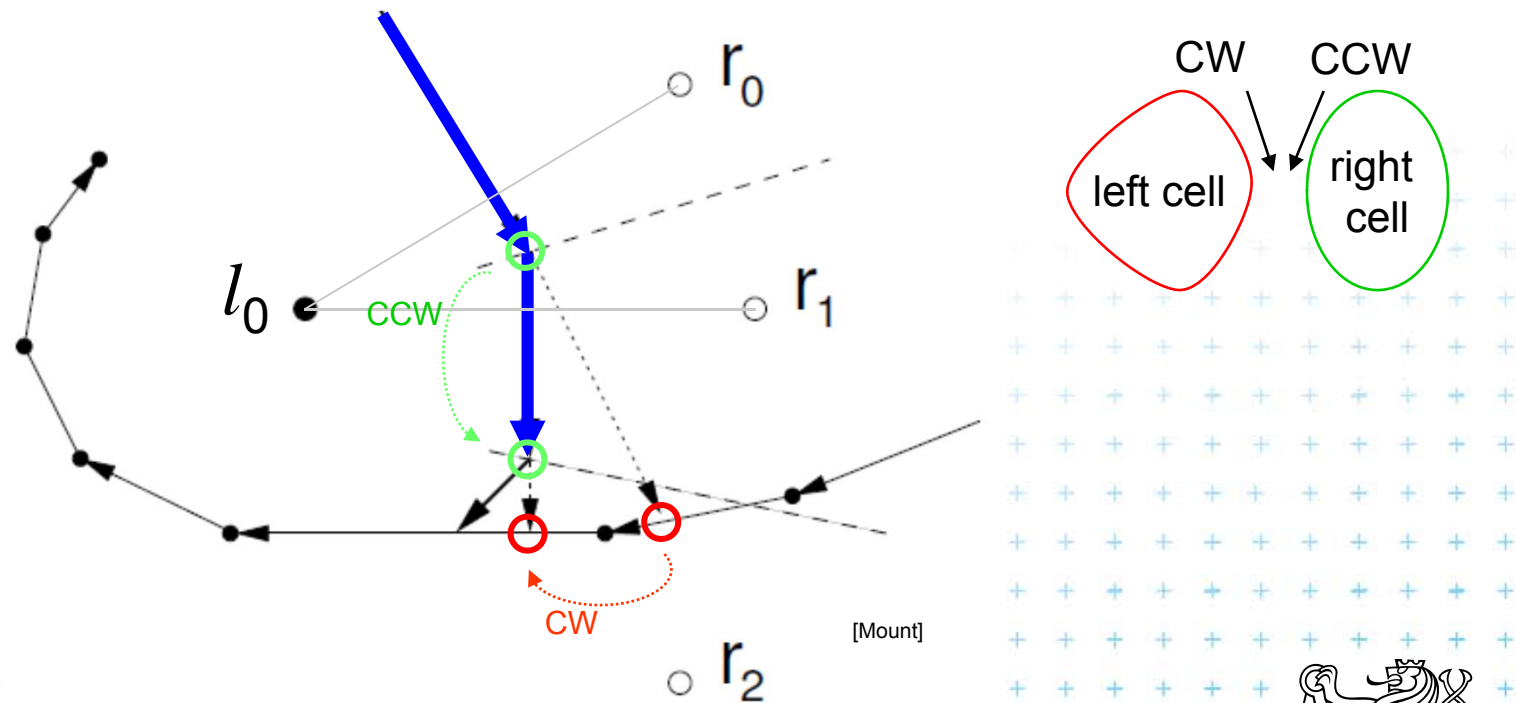
- Avoid repeated rescanning of cell edges
- Start in the last tested edge of the cell (each edge tested  $\sim$ once)
- In the **left** cell  $l_i$  continue CW, in the **right** cell  $r_i$  go CCW
- Image shows CW search on cell  $l_0$  and CCW on cells  $r_i$ :





# Monotone chain search in $O(n)$

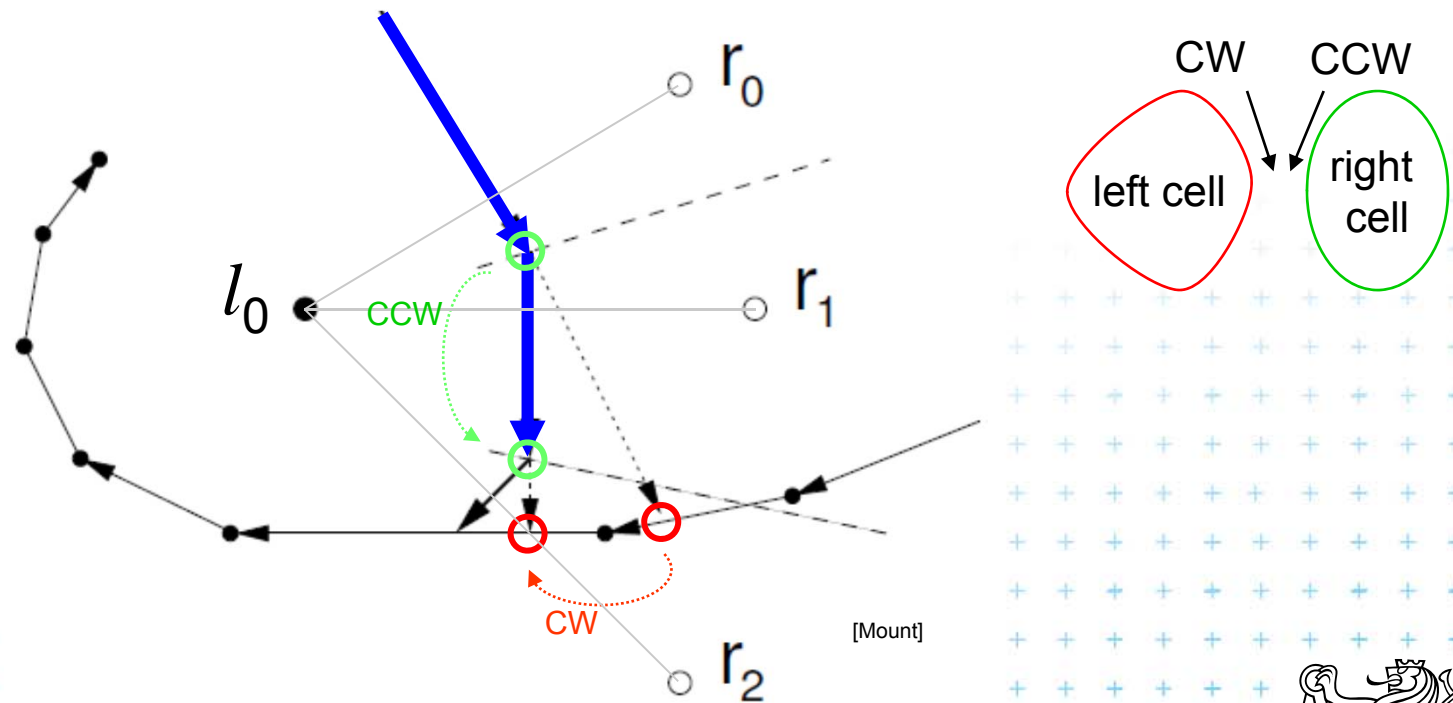
- Avoid repeated rescanning of cell edges
- Start in the last tested edge of the cell (each edge tested  $\sim$ once)
- In the **left** cell  $l_i$  continue CW, in the **right** cell  $r_i$  go CCW
- Image shows CW search on cell  $l_0$  and CCW on cells  $r_i$ :





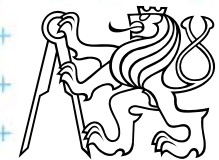
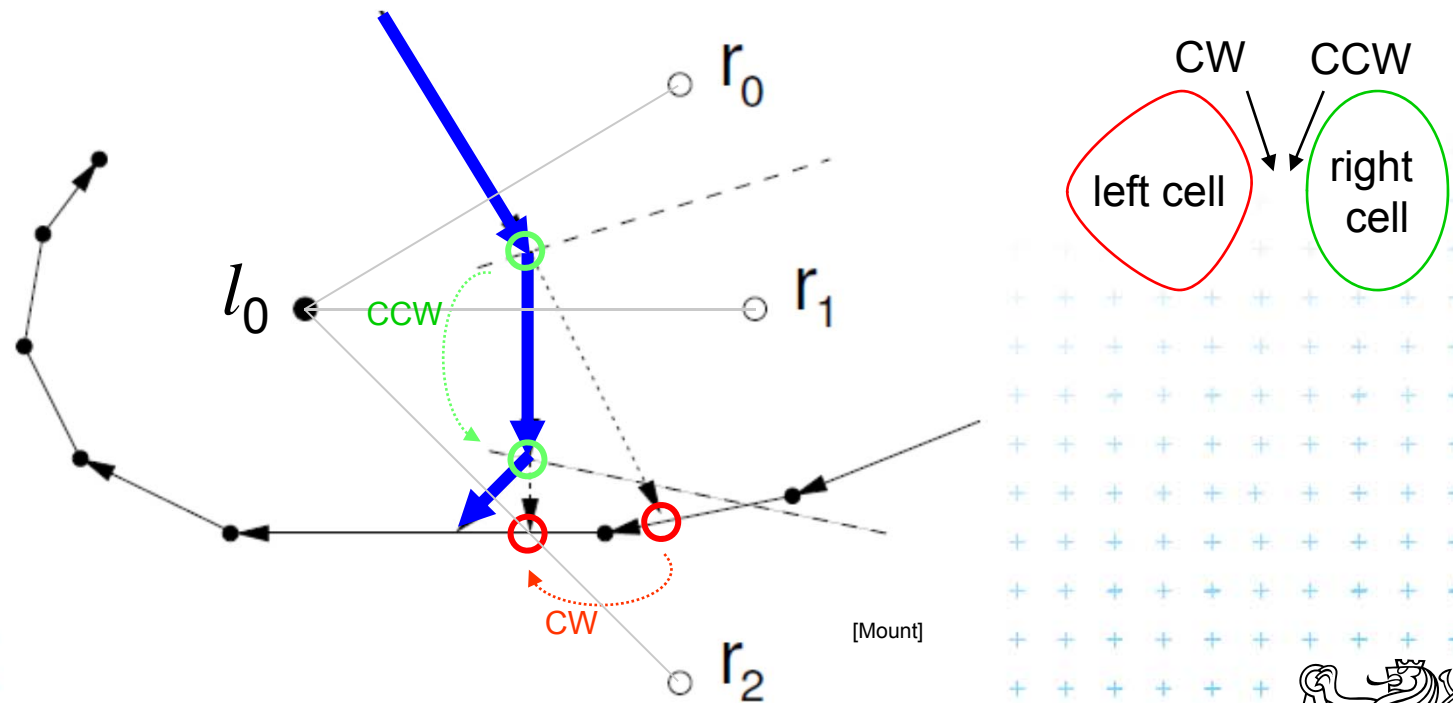
# Monotone chain search in $O(n)$

- Avoid repeated rescanning of cell edges
- Start in the last tested edge of the cell (each edge tested  $\sim$ once)
- In the **left** cell  $l_i$  continue CW, in the **right** cell  $r_i$  go CCW
- Image shows CW search on cell  $l_0$  and CCW on cells  $r_i$ :



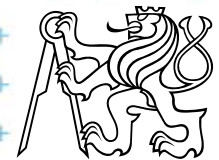
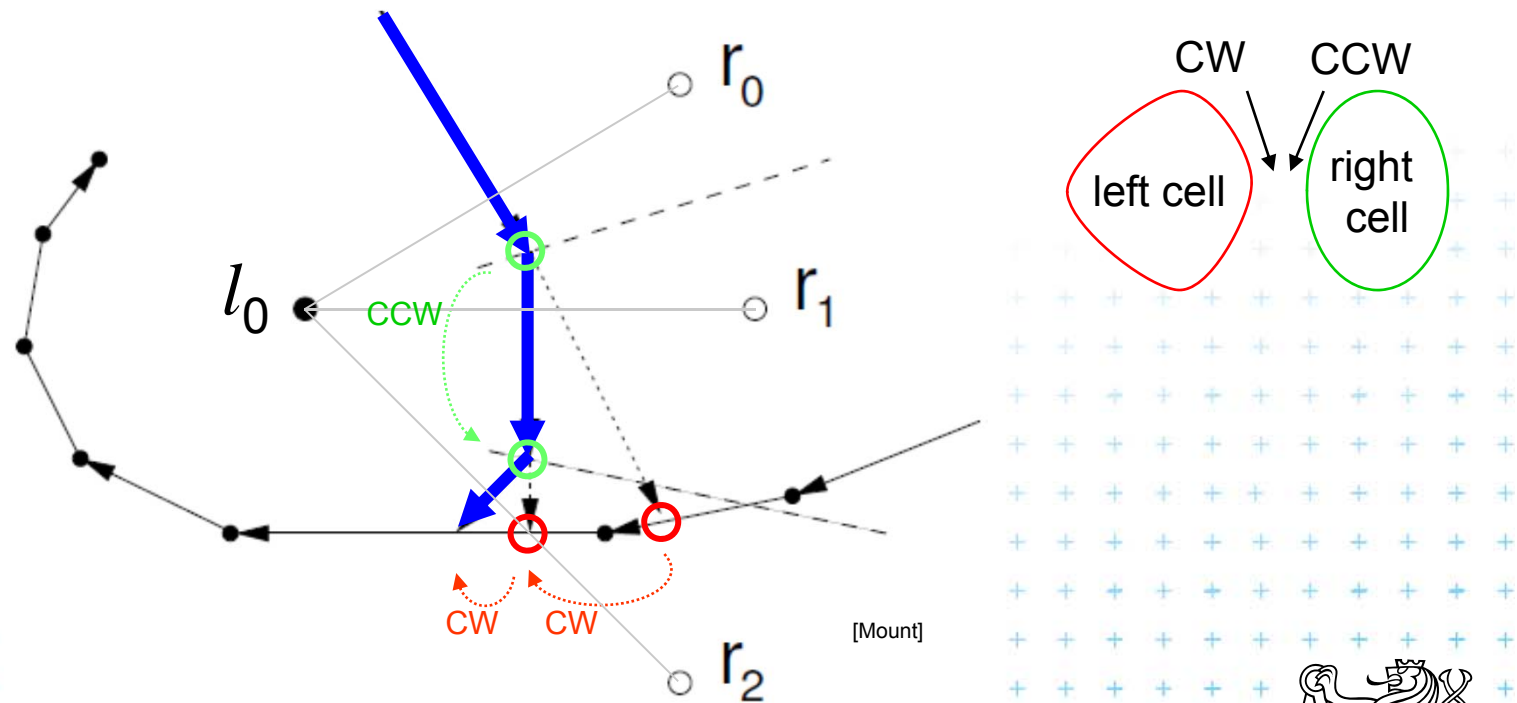
# Monotone chain search in $O(n)$

- Avoid repeated rescanning of cell edges
- Start in the last tested edge of the cell (each edge tested  $\sim$ once)
- In the **left** cell  $l_i$  continue CW, in the **right** cell  $r_i$  go CCW
- Image shows CW search on cell  $l_0$  and CCW on cells  $r_i$ :



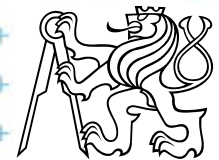
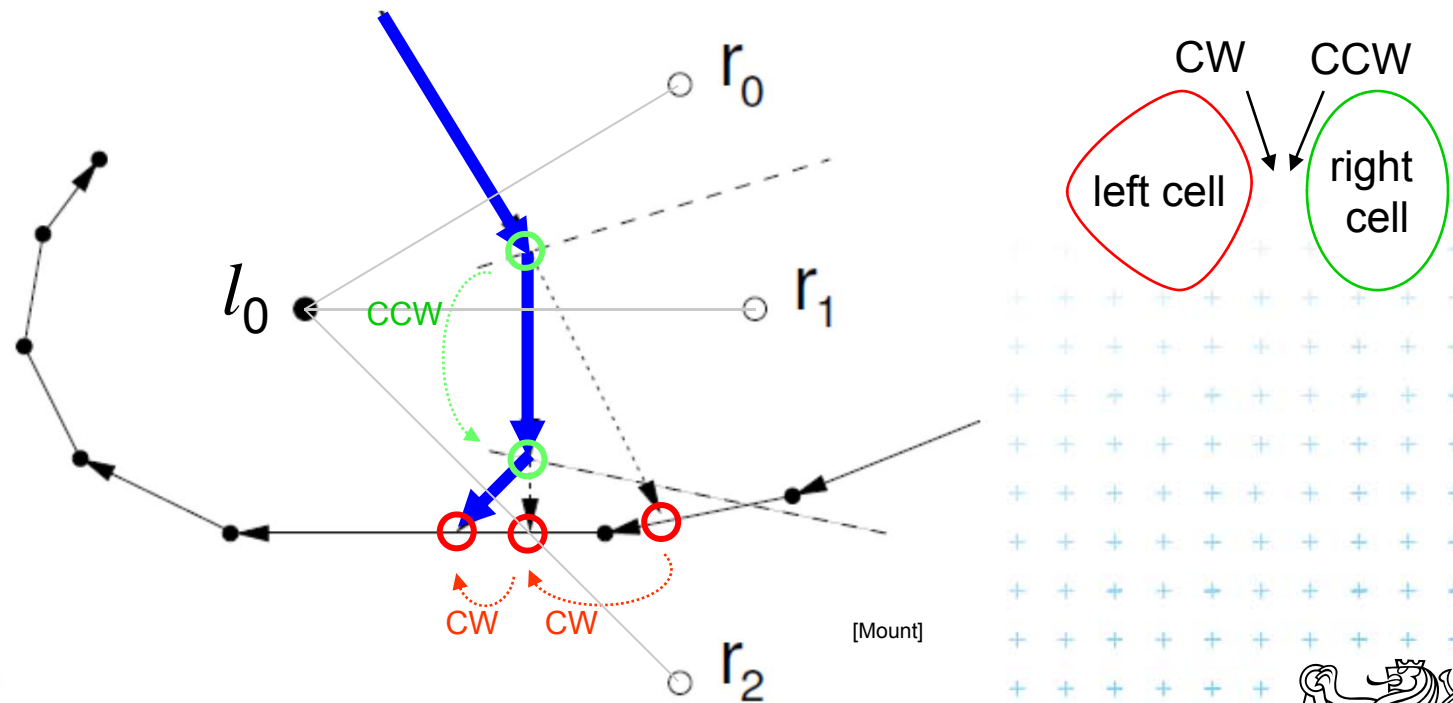
# Monotone chain search in $O(n)$

- Avoid repeated rescanning of cell edges
- Start in the last tested edge of the cell (each edge tested  $\sim$ once)
- In the **left** cell  $l_i$  continue CW, in the **right** cell  $r_i$  go CCW
- Image shows CW search on cell  $l_0$  and CCW on cells  $r_i$ :



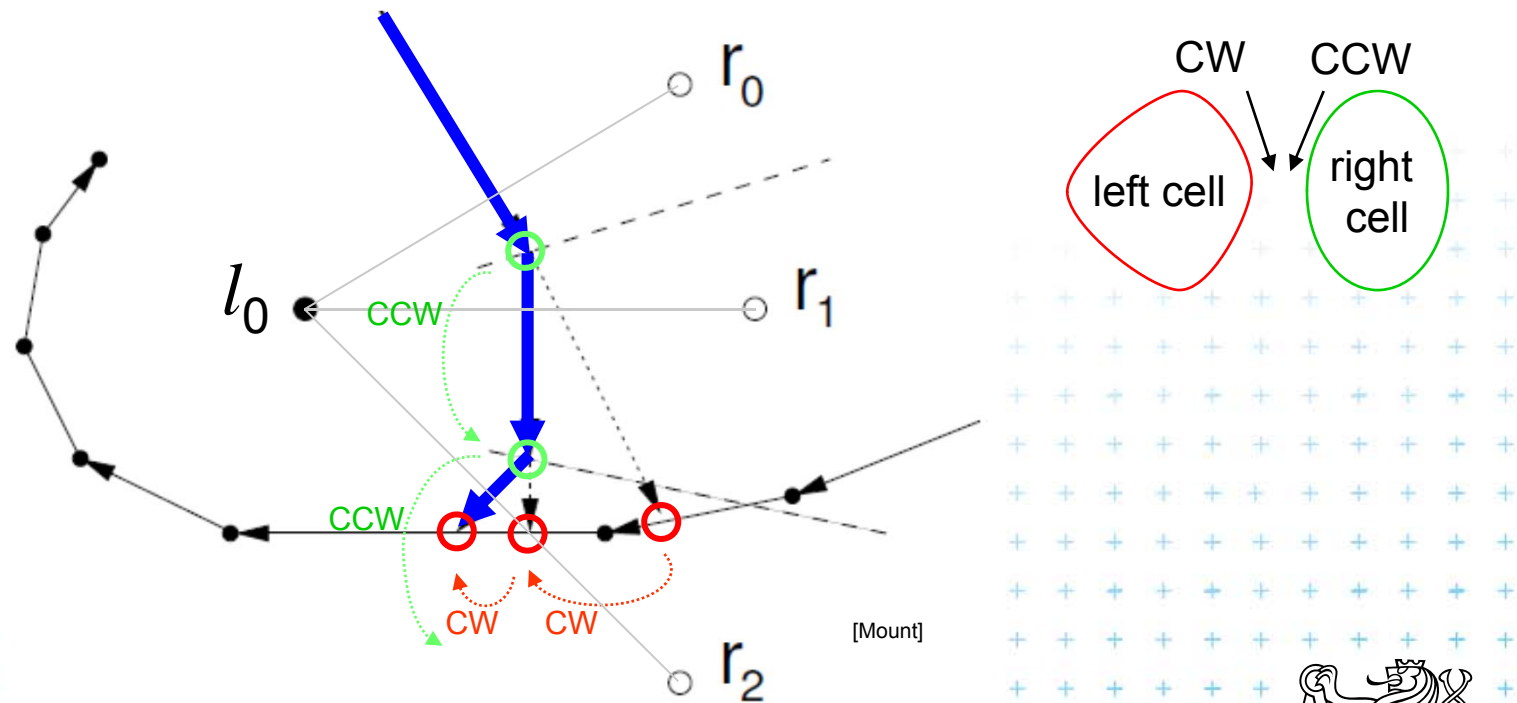
# Monotone chain search in $O(n)$

- Avoid repeated rescanning of cell edges
- Start in the last tested edge of the cell (each edge tested  $\sim$ once)
- In the **left** cell  $l_i$  continue CW, in the **right** cell  $r_i$  go CCW
- Image shows CW search on cell  $l_0$  and CCW on cells  $r_i$ :



# Monotone chain search in $O(n)$

- Avoid repeated rescanning of cell edges
- Start in the last tested edge of the cell (each edge tested  $\sim$ once)
- In the **left** cell  $l_i$  continue CW, in the **right** cell  $r_i$  go CCW
- Image shows CW search on cell  $l_0$  and CCW on cells  $r_i$ :



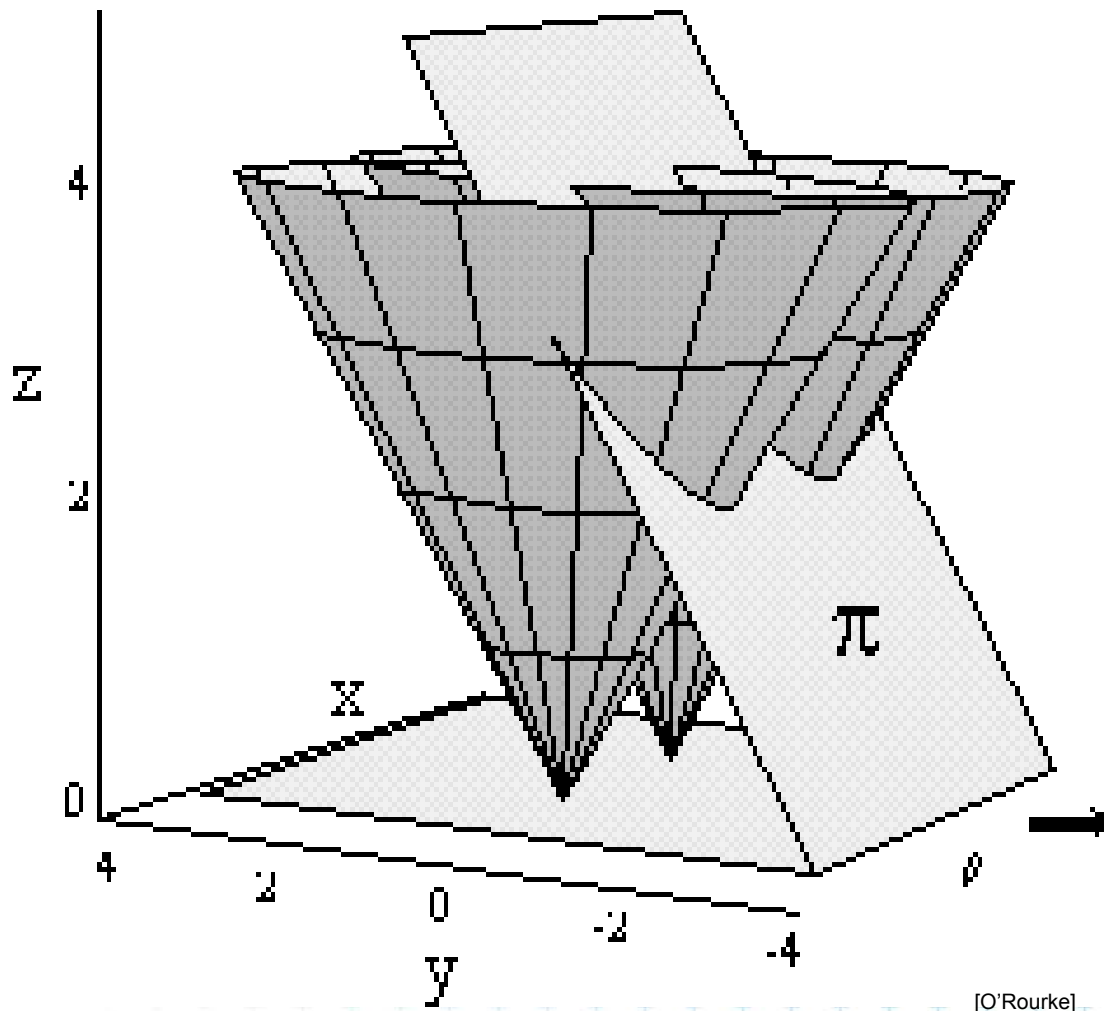
# Divide and Conquer method complexity

---

- Initial sort  $O(n \log n)$
- $O(\log n)$  recursion levels
  - $O(n)$  each merge (chain search, trim, add edges to VD)
- Altogether  $O(n \log n)$



# Fortune's sweep line algorithm – idea in 3D

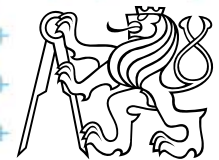


Cones in sites  
Scanning plane  $\pi$   
Both slanted  $45^\circ$

Projection of the intersection to  $xy$ :

- Cone x plane => parabolic arcs
- Cone x cone => edges of VD

[O'Rourke]

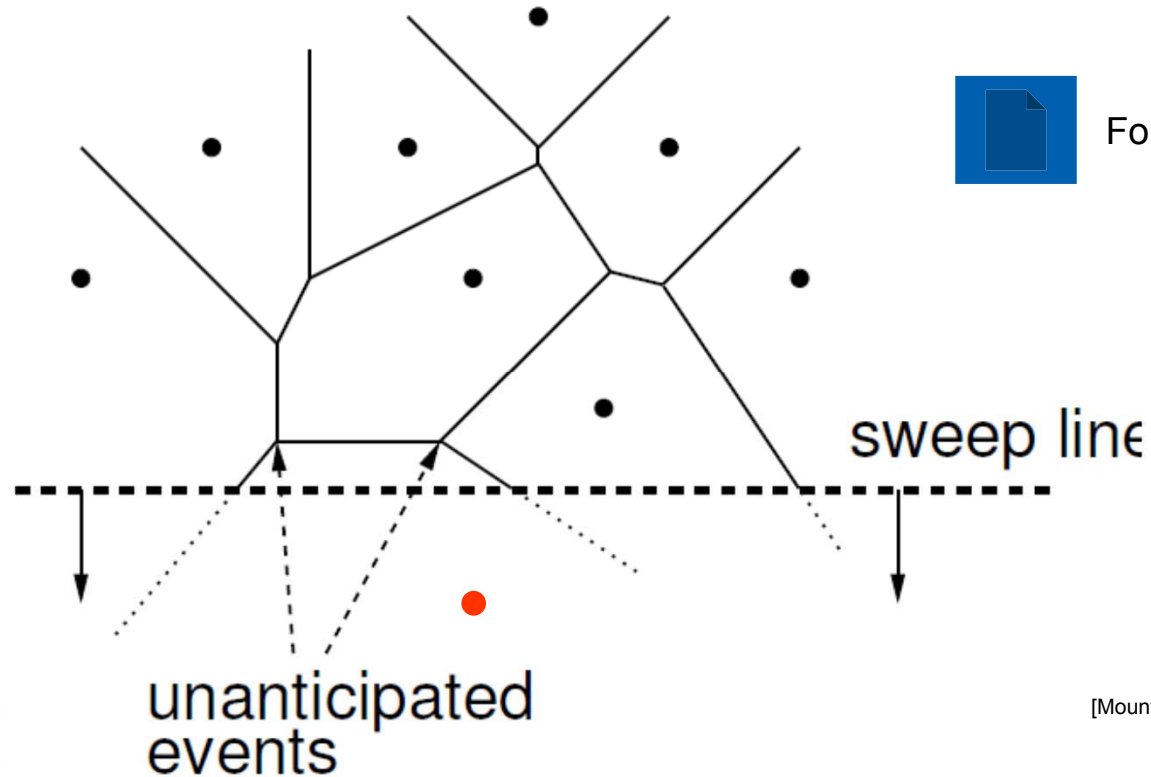




# Fortune's sweep line algorithm

- Differs from “typical” sweep line algorithm
- Unprocessed sites ahead from sweep line may generate Voronoi vertex behind the sweep line

DONE  
-----  
TODO



Fortune's applet

[Mount]

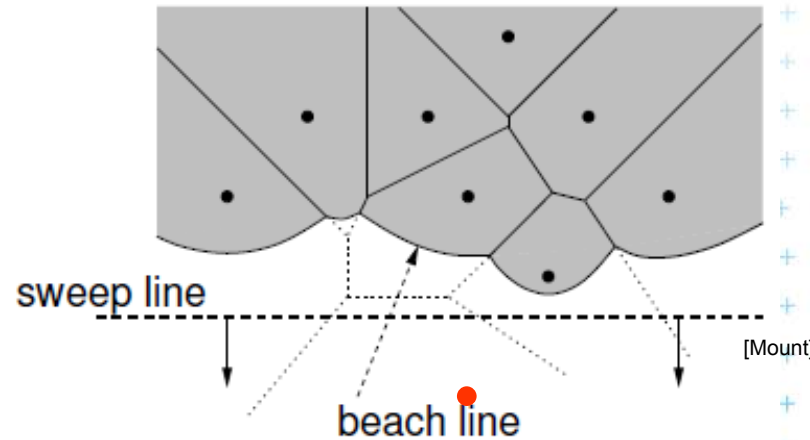
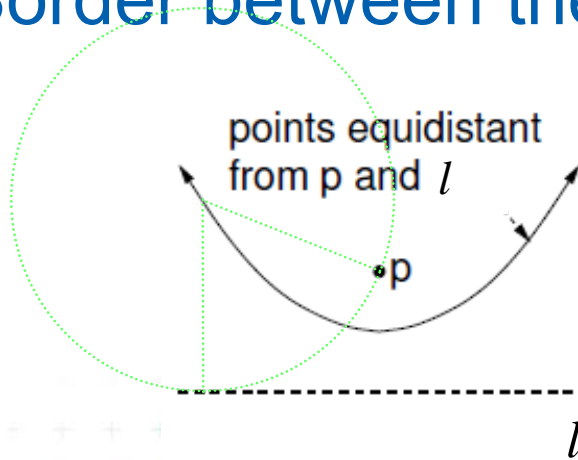




# Fortune's sweep line algorithm idea

DONE  
UNRESOLVED  
TODO

- Subdivide the halfplane above the sweep line  $l$  into 2 regions
  - Points **closer to some site above** than to sweep line  $l$  (solved part)
  - Points **closer to sweep line  $l$**  than any point above (unsolved part – can be changed by sites below  $l$ )
- Border between these 2 regions is a **beach line**



# Sweep line and beach line

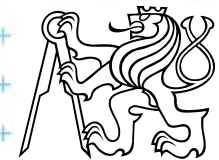
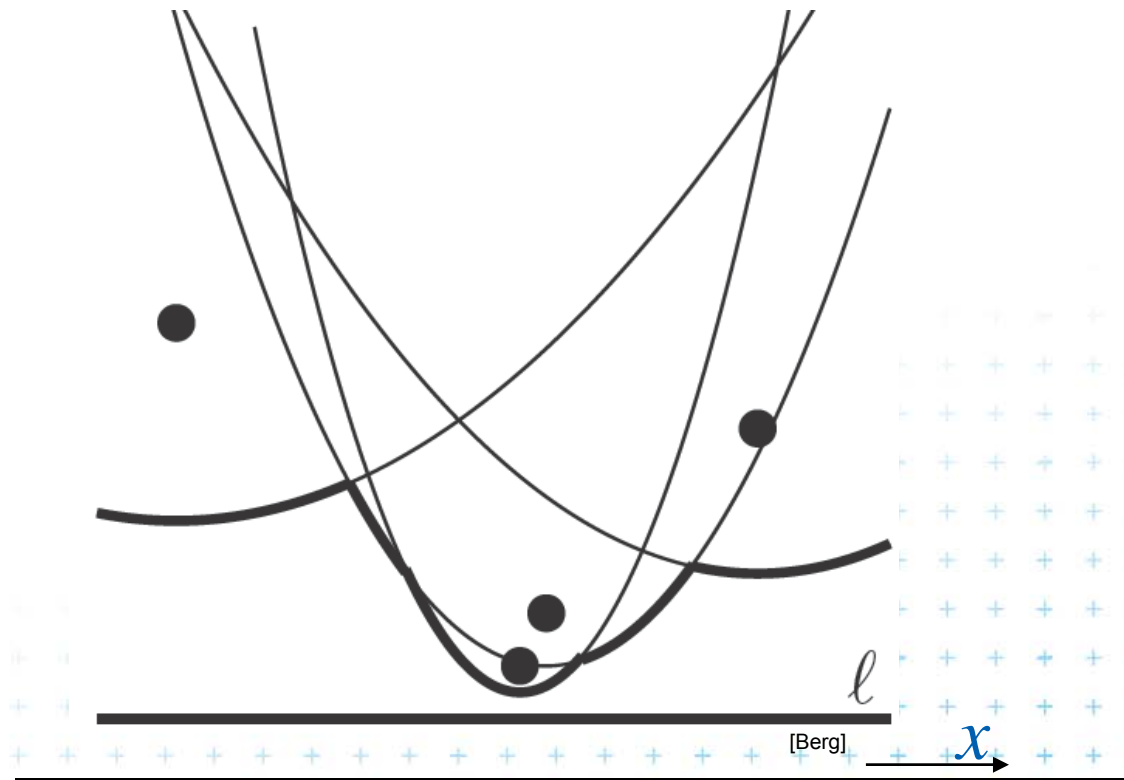
---

- **Straight sweep line  $l$** 
  - Separates processed and unprocessed sites (points)
- **Beach line (Looks like waves rolling up on a beach)**
  - Separates *solved* and *unsolved* regions above sweep line (separates sites above  $l$  that can be changed from sites that cannot be changed by sites below  $l$ )
  - x-monotonic curve made of **parabolic arcs**
  - Follows the sweep line
  - Prevents us from missing unanticipated events until the sweep line encounters the corresponding site



# Beach line

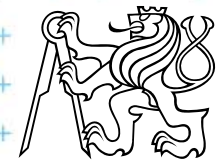
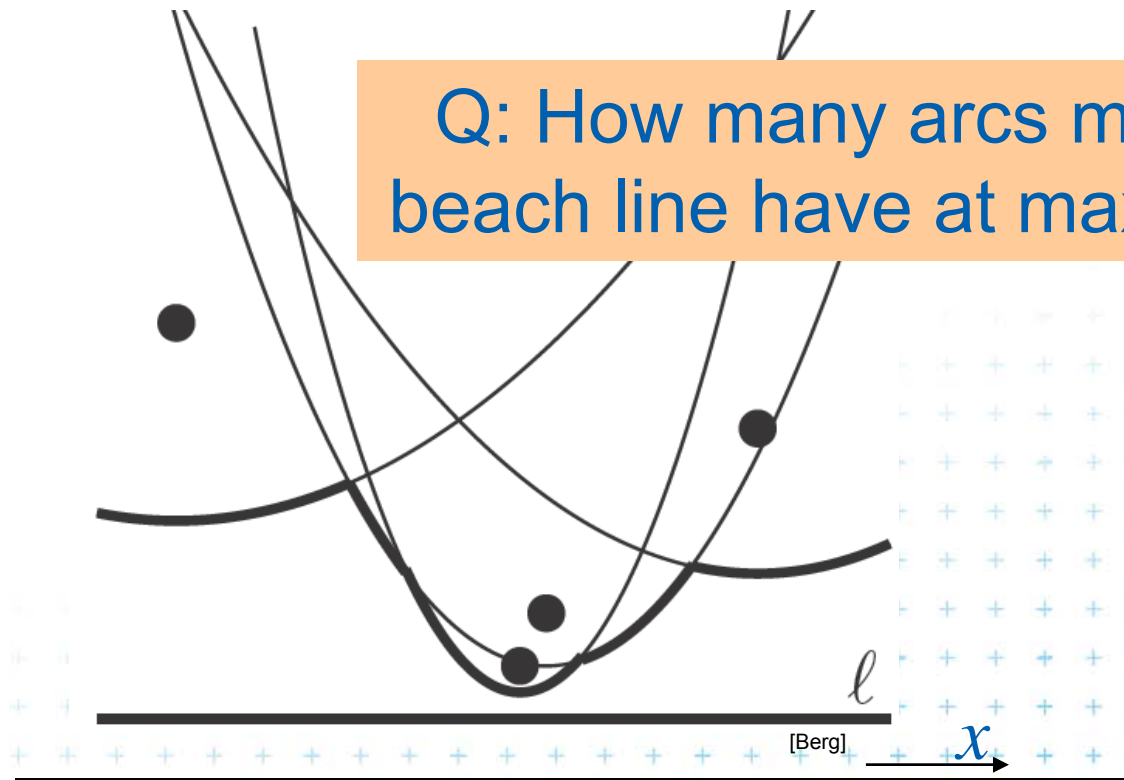
- Every site  $p_i$  above  $l$  defines a complete parabola
- **Beach line** is the function, that passes through the lowest points of all the parabolas (lower envelope)



# Beach line

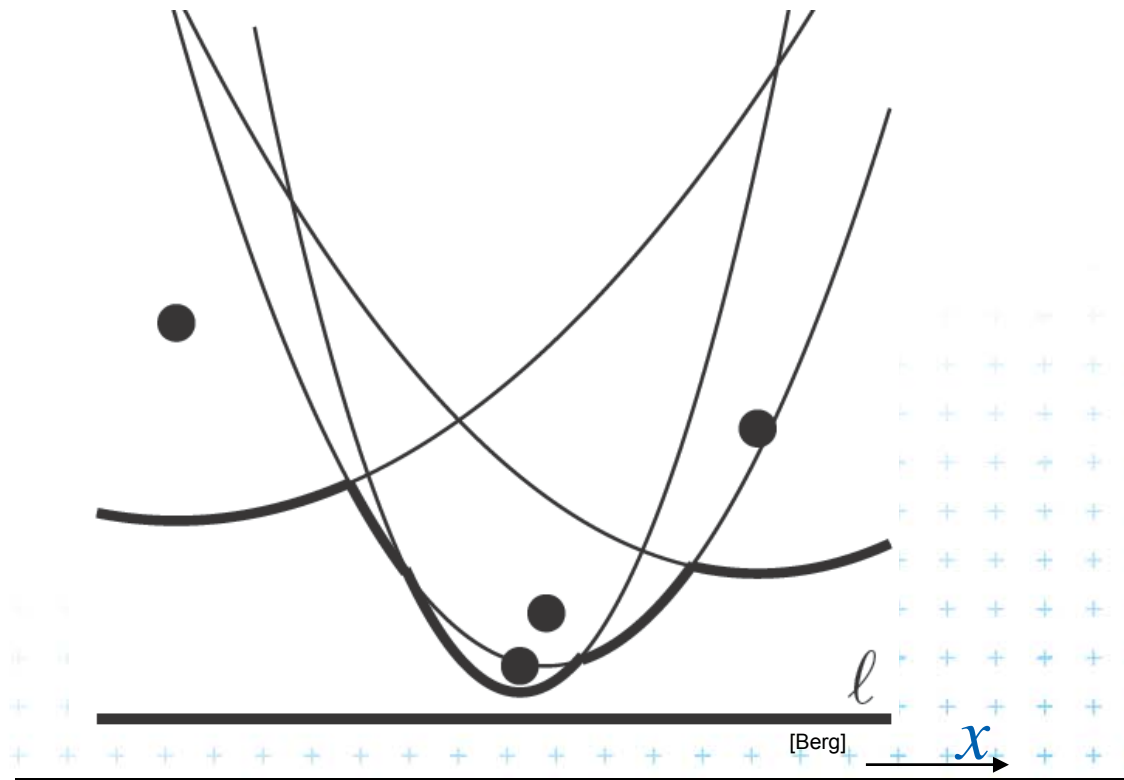
- Every site  $p_i$  above  $l$  defines a complete parabola
- **Beach line** is the function, that passes through the lowest points of all the parabolas (lower envelope)

Q: How many arcs may the beach line have at maximum?



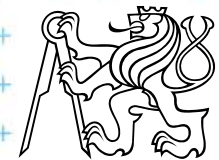
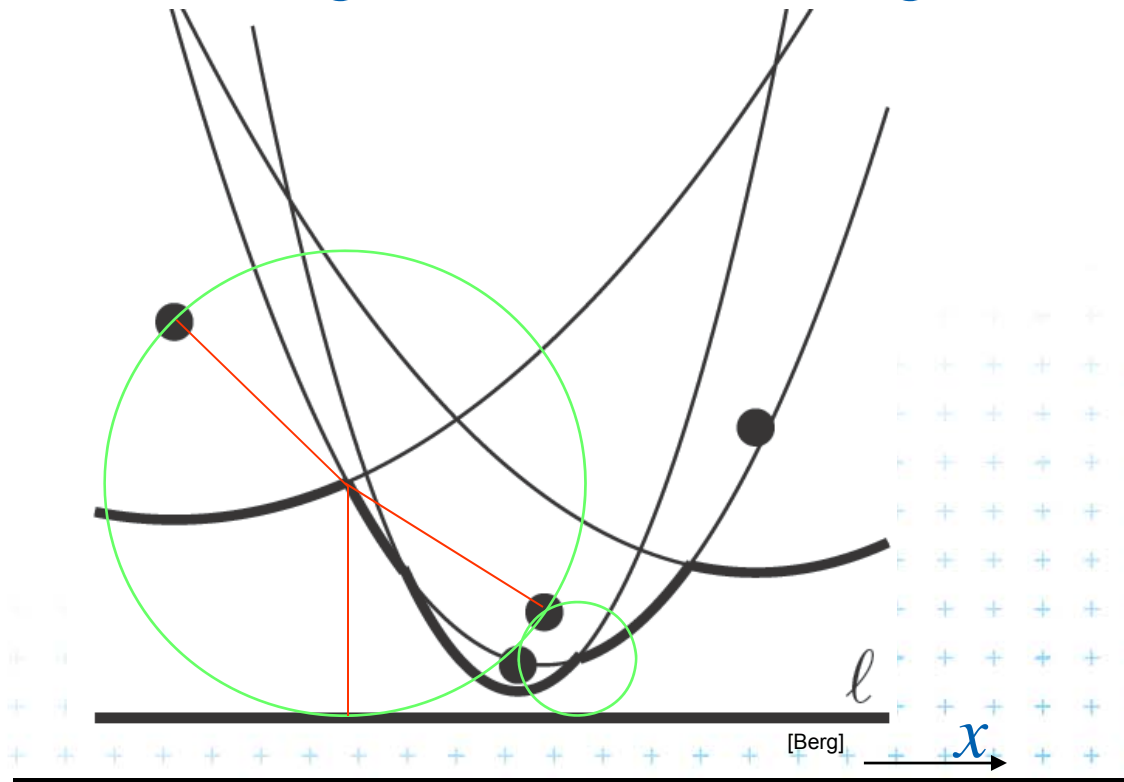
# Beach line

- Every site  $p_i$  above  $l$  defines a complete parabola
- **Beach line** is the function, that passes through the lowest points of all the parabolas (lower envelope)



# Break point (*bod zlomu*)

- = Intersection of two arcs on the beach line
- Equidistant to 2 sites and sweep line  $l$
- Lies on Voronoi edge of the final diagram



# Notes

---

Beach line is **x-monotone**

= every vertical line intersects it in exactly ONE point

Along the beach line

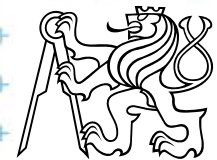
Parabolic arcs are ordered

Breakpoints are ordered

Breakpoints

trace the Voronoi edges

compute their position on the fly from neighboring arcs





# Events

---

What event types exist?



# Events

---

There are two types of events:

- **Site events (SE)**

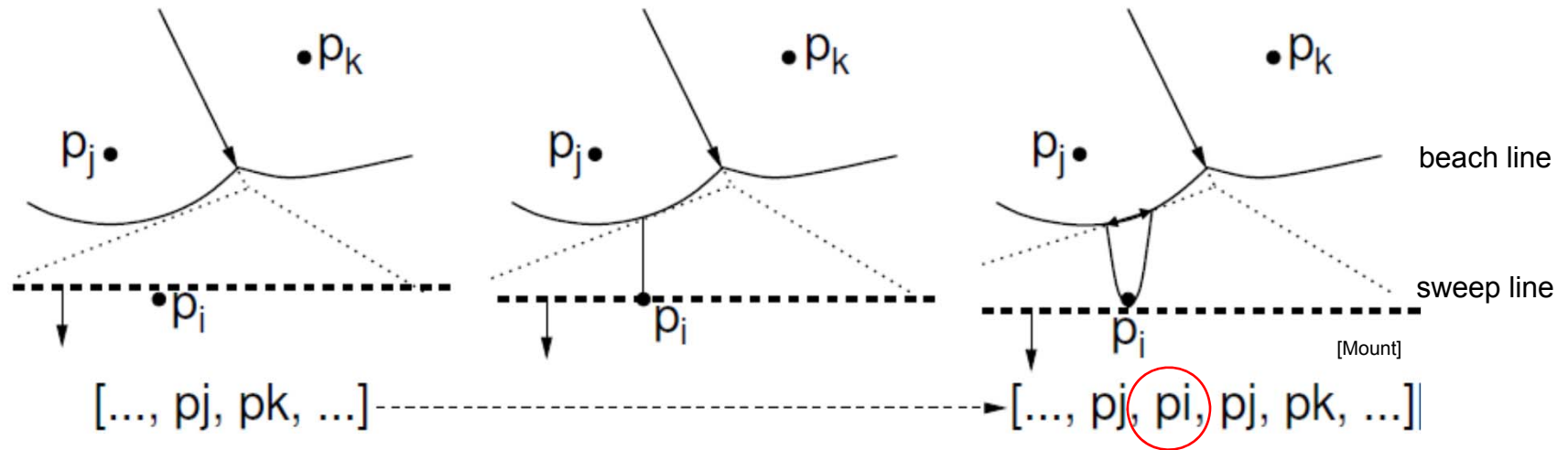
- When the sweep line passes over a new site  $p_i$ ,
  - *new arc* is added to the beach line
  - *new edge fragment* added to the VD.
- All SEs known from the beginning (sites sorted by  $y$ )

- **Voronoi vertex event ([Berg] calls a circle event)**

- When the parabolic *arc shrinks to zero and disappears*, *new Voronoi vertex* is created.
- Created dynamically by the algorithm for **triples or more neighbors on the beach line** (triples changed by both types of events)



# Site event

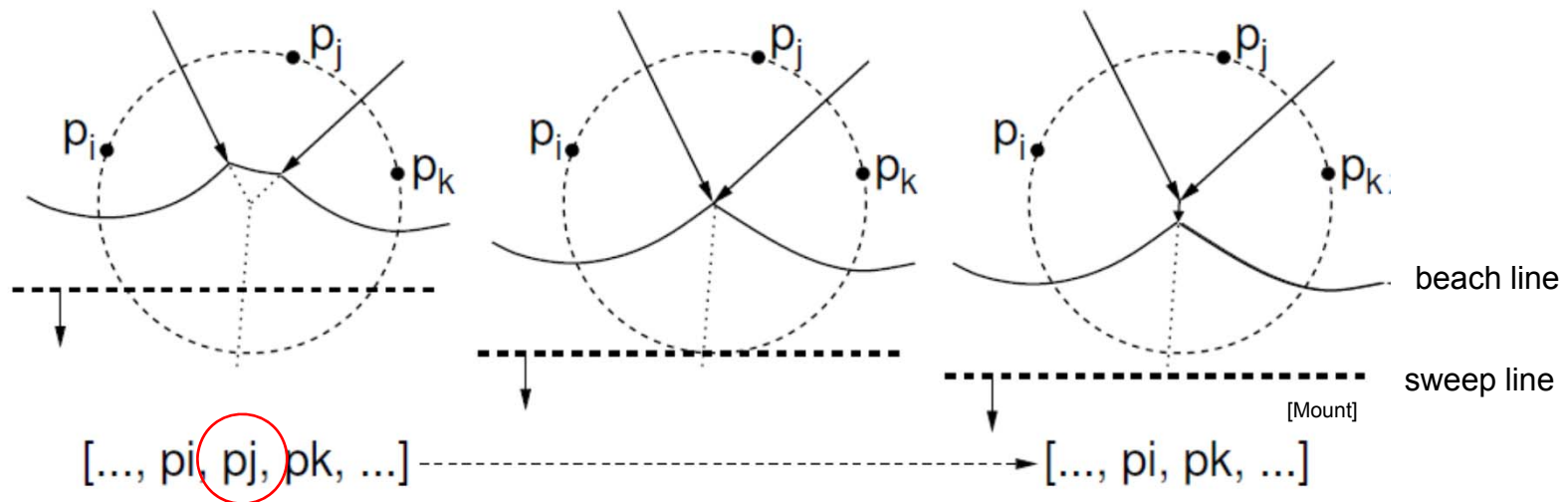


Generated when the **sweep line passes over a site  $p_i$**

- **New parabolic arc** created, it starts as a vertical ray from  $p_i$  to the beach line
- As the sweep line sweeps on, the arc grows wider
- The entry  $\langle \dots, p_j, \dots \rangle$  on the sweep line status is replaced by the triple  $\langle \dots, p_j, p_i, p_j, \dots \rangle$
- **Dangling future VD edge** created on the bisector  $(p_i, p_j)$



# Voronoi vertex event (circle event)



Generated when  $l$  passes the lowest point of a circle

- Sites  $p_i$ ,  $p_j$ ,  $p_k$  appear consecutively on the beach line
- Circumcircle lies partially below the sweep line (Voronoi vertex has not yet been generated)
- This circumcircle contains no point below the sweep line (no future point will block the creation of the vertex)
- Vertex & bisector  $(p_i, p_k)$  created,  $(p_i, p_j)$  &  $(p_j, p_k)$  finished
- One parabolic arc removed from the beach line



# Data structures

---

1. (Partial) Voronoi diagram
2. Beach line data structure  $T$
3. Event queue  $Q$



# Data structures

---

1. (Partial) Voronoi diagram
2. Beach line data structure  $T$
3. Event queue  $Q$

1. VD edges arise during: site event circle event?
2. VD vertices arise during: site event circle event?
3. Site events known from the beginning: yes no?
4. Circle events known from the beginning: yes no?

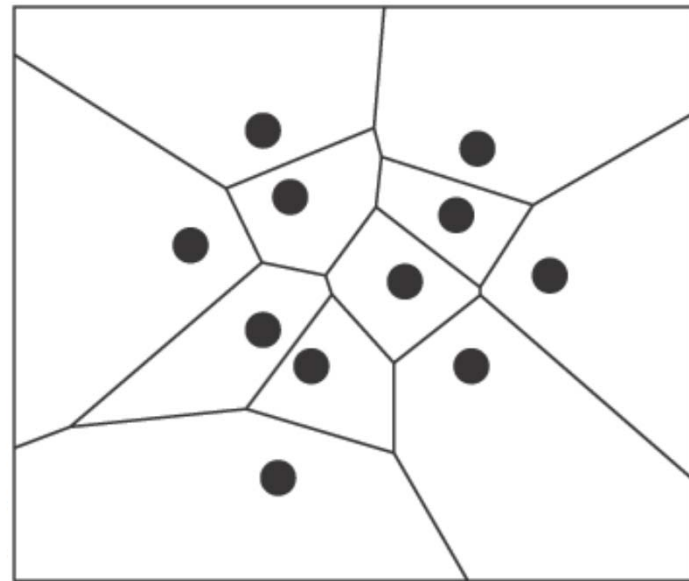


# 1. (Partial) Voronoi diagram data structure

---

Any PSLG data structure, e.g. DCEL (planar straight line graph)

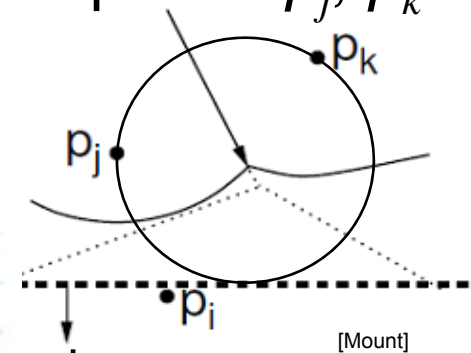
- Stores the VD during the construction
  - Contain unbounded edges
    - **dangling** edges during the construction (managed by the beach line DS) and
    - edges of **unbounded** cells at the end
- => create a bounding box





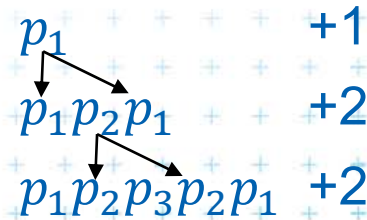
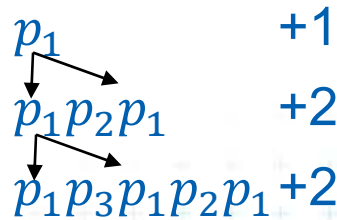
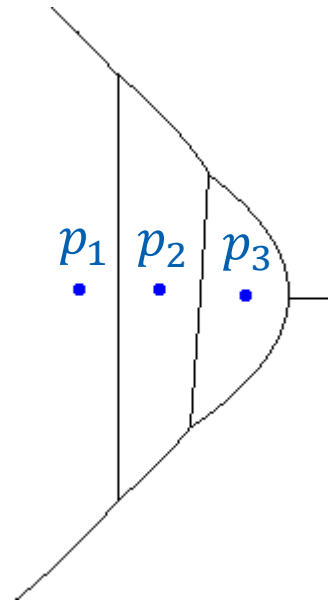
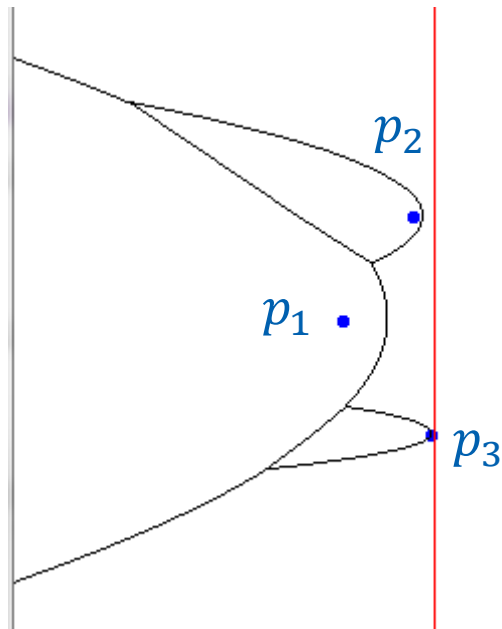
## 2. Beach line tree data structure $T$ – status

- Used to locate the arc directly above a new site
- E.g. Binary tree  $T$ 
  - Leaves - ordered arcs along the beach line (x-monotone)
    - $T$  stores only the sites  $p_i$  in leaves,  $T$  does not store the parabolas
  - Inner tree nodes - breakpoints as ordered pairs  $\langle p_j, p_k \rangle$ 
    - $p_j, p_k$  are neighboring sites
    - Breakpoint position computed on the fly from  $p_j, p_k$  and y-coord of the sweep line
  - Pointers to other two DS
    - In leaves – pointer to event queue, point to node when arc disappears via Voronoi vertex event – if it exists
    - In inner nodes - pointer to (dangling) half-edge in DCEL of VD, that is being traced out by the break point



# Max $2n - 1$ arcs on the beach line

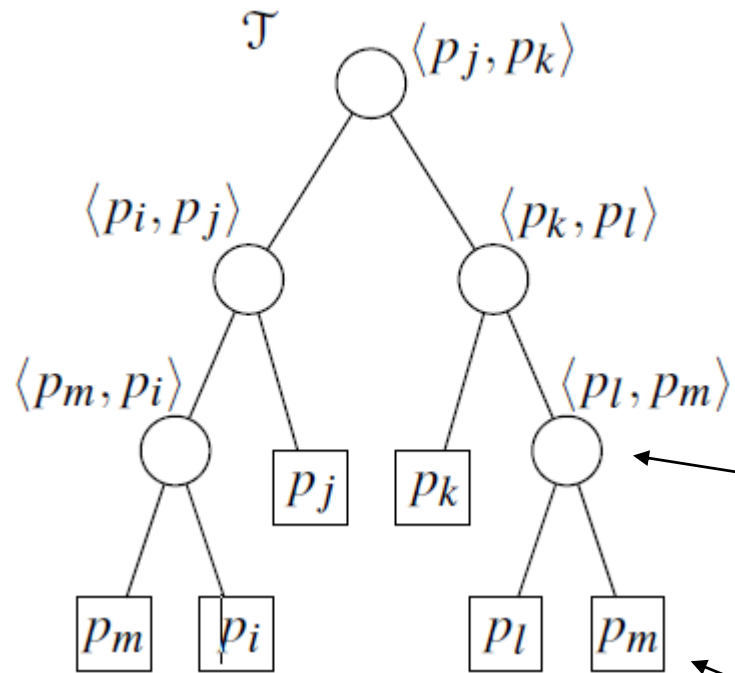
New site splits just one arc



Leaves in T



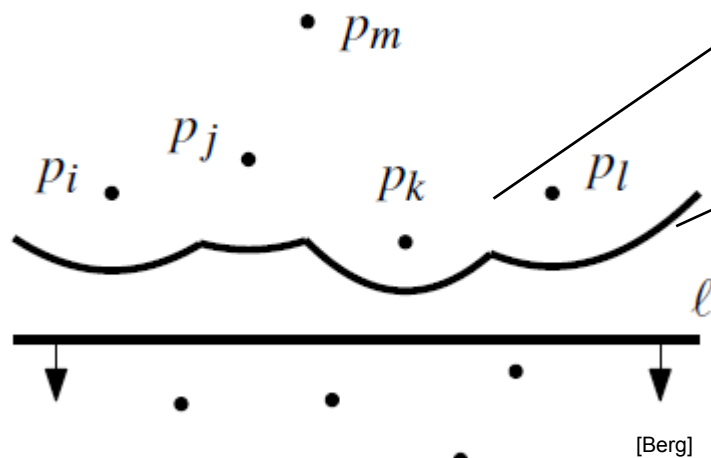
## 2. Beach line tree $T$



x-coord computed on the fly for a given position of the beach line  $l$

Break points  
= inner nodes in  $T$

Arcs = Leaves in  $T$



### 3. Event queue Q

---

- Priority queue, ordered by y-coordinate
- For site event
  - stores the site itself
  - known from the beginning
- For Voronoi vertex event (circle event)
  - stores the **lowest point of the circle**
  - stores also **pointer to the leaf in tree T**  
(represents the **parabolic arc** that will disappear)
  - created by both events, when triples of points become neighbors (possible max three triples for a site)
  - $p_i, p_j, p_k, p_l, p_m$  insert of  $p_k$  can create up to 3 triples and delete up to 2 triples  $(p_i, p_j, p_l)$  and  $(p_j, p_l, p_m)$



# Fortune's algorithm

---

## FortuneVoronoi( $P$ )

*Input:* A set of point sites  $P = \{p_1, p_2, \dots, p_n\}$  in the plane

*Output:* Voronoi diagram  $\text{Vor}(P)$  inside a bounding box in a DCEL struct.

1. Init event queue  $Q$  with all *site events*
2. **while**(  $Q$  not empty) **do**
3. | consider the event with largest  $y$ -coordinate in  $Q$  (next in the queue)
4. | **if**( event is a *site event* at site  $p_i$  )
5. |     **then** HandleSiteEvent( $p_i$ )
6. |     **else** HandleVoroVertexEvent( $p_i$ ), where  $p_i$  is the lowest point  
of the circle causing the event
7. | remove the event from  $Q$
8. Create a bbox and attach half-infinite edges in  $T$  to it in DCEL.
9. Traverse the halfedges in DCEL and  
add cell records and pointers to and from them

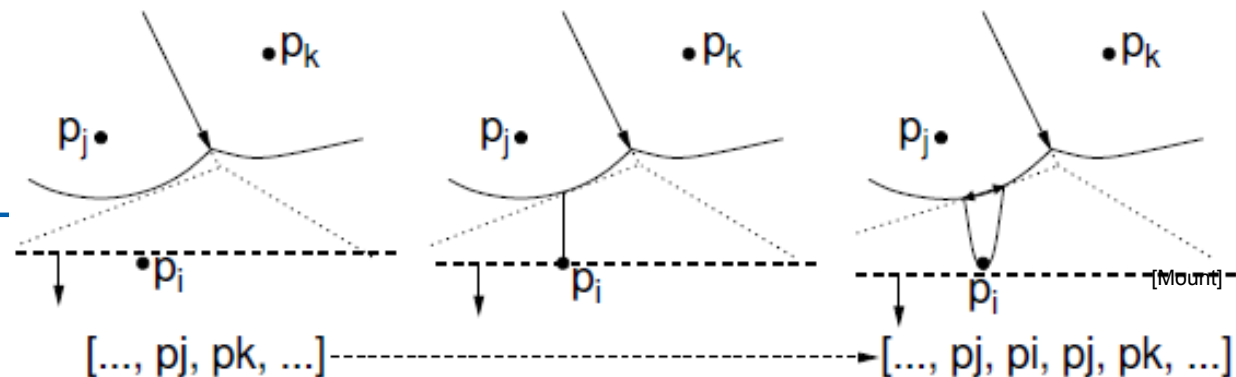


# Handle site event

## HandleSiteEvent( $p_i$ )

Input: event site  $p_i$

Output: updated DCEL



1. Search in  $T$  for arc  $\alpha$  vertically above  $p_i$ . Let  $p_j$  be the corresponding site
2. Apply insert-and-split operation, inserting a new entry of  $p_i$  to the beach line  $T$  (new arc), thus replacing  $\langle \dots, p_j, \dots \rangle$  with  $\langle \dots, p_j, p_i, p_j, \dots \rangle$
3. Create a new (dangling) edge in the Voronoi diagram, which lies on the bisector between  $p_i$  and  $p_j$
4. Neighbors on the beach line changed -> check the neighboring triples of arcs and *insert or delete Voronoi vertex events* (insert only if the circle intersects the sweep line and it is not present yet).

Note: Newly created triple  $p_j, p_i, p_j$  cannot generate a circle event because it only involves two distinct sites.

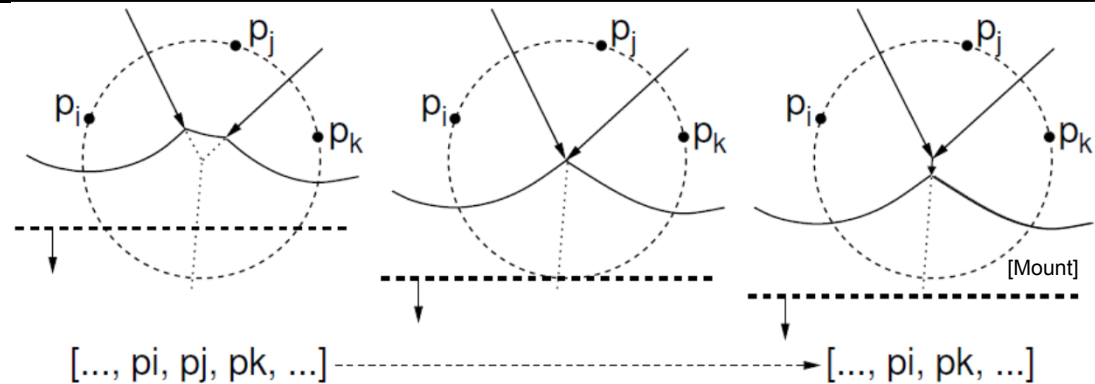


# Handle Voronoi vertex (circle) event

## HandleVoroVertexEvent( $p_j$ )

Input: event site  $p_j$

Output: updated DCEL



Let  $p_i, p_j, p_k$  be the sites that generated this event (from left to right).

1. Delete the entry  $p_j$  from the beach line (thus eliminating its arc  $\alpha$ ), i.e.: Replace a triple  $\langle \dots, p_i, p_j, p_k, \dots \rangle$  with  $\langle \dots, p_i, p_k, \dots \rangle$  in  $T$ .
2. Create a new vertex in the Voronoi diagram (at circumcenter of  $\langle p_i, p_j, p_k \rangle$ ) and join the two Voronoi edges for the bisectors  $\langle p_i, p_j \rangle$  and  $\langle p_j, p_k \rangle$  to this vertex (dangling edges – created in step 3 above).
3. Create a new (dangling) edge for the bisector between  $\langle p_j, p_k \rangle$
4. Delete any Voronoi vertex events (max. three) from  $Q$  that arose from triples involving the arc  $\alpha$  of  $p_j$  and generate (two) new events corresponding to consecutive triples involving  $p_i$ , and  $p_k$ .



# Beach line modification

---

Q: Beach line contains: abcdef

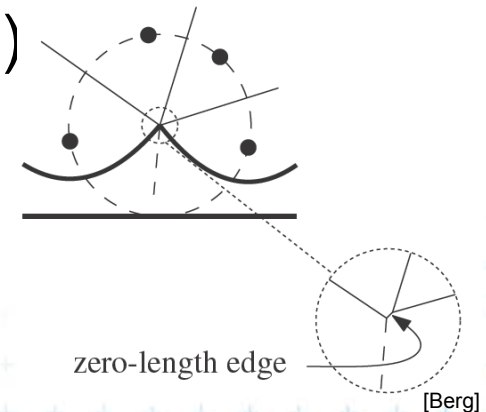
After deleting of **d**, which triples vanish and which triples are added to the beach line?



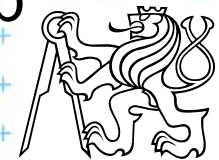
# Handling degeneracies

## Algorithm handles degeneracies correctly

- 2 or more events with the same  $y$ 
  - if  $x$  coords are different, process them in any order
  - if  $x$  coords are the same (cocircular sites) process them in any order, it creates duplicated vertices with zero-length edges, remove them in post processing step



- degeneracies while handling an event
  - Site below a beach line breakpoint
  - Creates circle event on the same position
  - remove zero-length edges in post processing step



# 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, Chapter 7, <http://www.cs.uu.nl/geobook/>

[Mount] David Mount, - *CMSC 754: Computational Geometry, Lecture Notes for Spring 2007*, University of Maryland, Lectures 12 and 29. <http://www.cs.umd.edu/class/spring2007/cmsc754/lectures.shtml>

[Preparata] Preparata, F.P., Shamos, M.I.: *Computational Geometry. An Introduction*. Berlin, Springer-Verlag, 1985. Chapter 5

[VoroGlide] VoroGlide applet:  
<http://www.pi6.fernuni-hagen.de/GeomLab/VoroGlide/>

[Fortune] Fortune's algorithm applet:  
<http://www.personal.kent.edu/~rmuhamma/Compgeometry/MyCG/Voronoi/Fortune/fortune.htm>

[Muhama] <http://www.personal.kent.edu/~rmuhamma/Compgeometry/compgeom.html>

<http://www.personal.kent.edu/~rmuhamma/Compgeometry/MyCG/Voronoi/DivConqVor/divConqVor.htm>

